Lecture Notes in Computer Science 2211
Edited by G. Goos, J. Hartmanis, and J. van Leeuwen

Springer
Berlin
Heidelberg
New York
Barcelona
Hong Kong
London
Milan
Paris
Tokyo

Thomas A. Henzinger
Christoph M. Kirsch (Eds.)

Embedded Software

First International Workshop, EMSOFT 2001
Tahoe City, CA, USA, October 8-10, 2001
Proceedings

Springer

Series Editors

Gerhard Goos, Karlsruhe University, Germany
Juris Hartmanis, Cornell University, NY, USA
Jan van Leeuwen, Utrecht University, The Netherlands

Volume Editors

Thomas A. Henzinger
Christoph M. Kirsch
University of California at Berkeley
Department of Electrical Engineering and Computer Sciences
Berkeley, CA 94720-1770, USA
E-mail: {tah,cm}@eecs.berkeley. edu

Cataloging-in-Publication Data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Embedded software : proceedings ; first international workshop / EMSOFT 2001, Tahoe City, CA, USA, October 8 - 10, 2001. Thomas A. Henzinger ; Christoph M. Kirsch (ed.). - Berlin ; Heidelberg ; New York ; Barcelona ; Hong Kong ; London ; Milan ; Paris ; Tokyo : Springer, 2001
(Lecture notes in computer science ; Vol. 2211)
ISBN 3-540-42673-6

CR Subject Classification (1998):C.3, D.1-4, F.3

ISSN 0302-9743
ISBN 3-540-42673-6 Springer-Verlag Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer-Verlag. Violations are liable for prosecution under the German Copyright Law.

Springer-Verlag Berlin Heidelberg New York
a member of BertelsmannSpringer Science+Business Media GmbH

http://www.springer.de

© Springer-Verlag Berlin Heidelberg 2001
Printed in Germany

Typesetting: Camera-ready by author, data conversion by PTP-Berlin, StefanSossna
Printed on acid-free paper SPIN 10840826 06/3142 5 4 3 2 1 0

Preface

This volume contains the proceedings of EMSOFT 2001, the *First International Workshop on Embedded Software*. The workshop was organized October 8–10, 2001, at Tahoe City, California. The steering committee of the workshop has the following members:

Gérard Berry (Esterel Technologies, France)
Paul Hudak (Yale University, USA)
Hermann Kopetz (Technical University of Vienna, Austria)
Edward Lee (University of California, Berkeley, USA)
Ragunathan Rajkumar (Carnegie Mellon University, USA)
Alberto Sangiovanni-Vincentelli (University of California, Berkeley, USA)
Douglas Schmidt (Defense Advanced Research Projects Agency, USA)
Joseph Sifakis (Verimag Grenoble, France)

The workshop was sponsored jointly by the DARPA Information Technology Office within the MobIES (Model-based Integration of Embedded Systems) program (Dr. Janos Sztipanovits), and by the National Science Foundation (Dr. Helen Gill). The workshop URL is www.emsoft.org.

Embedded software is software that interacts with physical processes. As embedded systems increasingly permeate our daily lives on all levels, from microscopic devices to international networks, the cost-efficient development of reliable embedded software is one of the grand challenges in computer science today. The purpose of the workshop is to bring together researchers in all areas of computer science that are traditionally distinct but relevant to embedded software development, and to incubate a research community in this way. The workshop aims to cover all aspects of the design and implementation of embedded software, including operating systems and middleware, programming languages and compilers, modeling and validation, software engineering and programming methodologies, scheduling and execution time analysis, networking and fault tolerance, as well as application areas, such as embedded control, real-time signal processing, and telecommunications.

All presentations at the workshop were by invitation. The following speakers were invited:

Perry Alexander (University of Kansas, USA)
Rajeev Alur (University of Pennsylvania, USA)
Albert Benveniste (INRIA/IRISA Rennes, France)
Gérard Berry (Esterel Technologies, France)
Manfred Broy (Technical University of Munich, Germany)
Kenneth Butts (Ford Motor Company, USA)
Paul Caspi (Verimag Grenoble, France)
Patrick Cousot (École Normale Supérieure Paris, France)
David Culler (University of California, Berkeley, USA)

Ron Cytron (Washington University, USA)
Luca de Alfaro (University of California, Santa Cruz, USA)
Thomas Henzinger (University of California, Berkeley, USA)
Paul Hudak (Yale University, USA)
Kevin Jeffay (University of North Carolina, USA)
Hermann Kopetz (Technical University of Vienna, Austria)
Edward Lee (University of California, Berkeley, USA)
Sharad Malik (Princeton University, USA)
Krishna Palem (Georgia Institute of Technology, USA)
Wolfgang Pree (University of Constance, Germany)
Ragunathan Rajkumar (Carnegie Mellon University, USA)
Martin Rinard (Massachusetts Institute of Technology, USA)
John Rushby (SRI International, USA)
Alberto Sangiovanni-Vincentelli (University of California, Berkeley, USA)
Shankar Sastry (University of California, Berkeley, USA)
Douglas Schmidt (Defense Advanced Research Projects Agency, USA)
Joseph Sifakis (Verimag Grenoble, France)
John Stankovic (University of Virginia, USA)
Janos Sztipanovits (Vanderbilt University, USA)
Lothar Thiele (ETH Zürich, Switzerland)
Pravin Varaiya (University of California, Berkeley, USA)
Steve Vestal (Honeywell Laboratories, USA)
Reinhard Wilhelm (University of Saarbrücken, Germany)
Niklaus Wirth (ETH Zürich, Switzerland)
Wayne Wolf (Princeton University, USA)

In addition to the invited presentations, there were two panels at the workshop. The first group of panelists was from industry; they discussed the current and future challenges in the industrial development of embedded software. The second group of panelists was from academia; they discussed how the community should organize itself to further both research and education in embedded software.

We are grateful to the steering committee, the invited speakers, the sponsors, and the panelists for making the workshop a success. In addition, we wish to thank Cynthia Ernest, Charlotte Jones, Peggy Kingsley, and Peter Ray for help with the workshop organization.

July 2001

Tom Henzinger
Christoph Kirsch

Table of Contents

Heterogeneous Modeling Support for Embedded Systems Design 1
P. Alexander, C. Kong

Hierarchical Hybrid Modeling of Embedded Systems 14
R. Alur, T. Dang, J. Esposito, R. Fierro, Y. Hur, F. Ivančić, V. Kumar, I. Lee, P. Mishra, G. Pappas, O. Sokolsky

Some Synchronization Issues When Designing Embedded Systems from Components 32
A. Benveniste

Synchronous Programming Techniques for Embedded Systems: Present and Future 50
G. Berry

From Requirements to Validated Embedded Systems 51
M. Broy, O. Slotosch

Usage Scenarios for an Automated Model Compiler 66
K. Butts, D. Bostic, A. Chutinan, J. Cook, B. Milam, Y. Wang

Embedded Control: From Asynchrony to Synchrony and Back 80
P. Caspi

Verification of Embedded Software: Problems and Perspectives 97
P. Cousot, R. Cousot

A Network-Centric Approach to Embedded Software for Tiny Devices 114
D.E. Culler, J. Hill, P. Buonadonna, R. Szewczyk, A. Woo

Storage Allocation for Real-Time, Embedded Systems 131
S.M. Donahue, M.P. Hampton, M. Deters, J.M. Nye, R.K. Cytron, K.M. Kavi

Interface Theories for Component-Based Design 148
L. de Alfaro, T.A. Henzinger

Giotto: A Time-Triggered Language for Embedded Programming 166
T.A. Henzinger, B. Horowitz, C.M. Kirsch

Directions in Functional Programming for Real(-Time) Applications 185
W. Taha, P. Hudak, Z. Wan

Rate-Based Resource Allocation Models for Embedded Systems 204
K. Jeffay, S. Goddard

The Temporal Specification of Interfaces
in Distributed Real-Time Systems 223
H. Kopetz

System-Level Types for Component-Based Design 237
E.A. Lee, Y. Xiong

Embedded Software Implementation Tools
for Fully Programmable Application Specific Systems 254
S. Malik

Compiler Optimizations for Adaptive EPIC Processors 257
K.V. Palem, S. Talla, W.-F. Wong

Embedded Software Market Transformation
through Reusable Frameworks 274
W. Pree, A. Pasetti

An End-to-End Methodology for Building Embedded Systems 287
R. Rajkumar

An Implementation of Scoped Memory for Real-Time Java 289
W.S. Beebee, M. Rinard

Bus Architectures for Safety-Critical Embedded Systems 306
J. Rushby

Using Multiple Levels of Abstractions in Embedded Software Design 324
J.R. Burch, R. Passerone, A.L. Sangiovanni-Vincentelli

Hierarchical Approach for Design of Multi-vehicle Multi-modal Embedded
Software 344
T.J. Koo, J. Liebman, C. Ma, S.S. Sastry

Adaptive and Reflective Middleware
for Distributed Real-Time and Embedded Systems 361
D.C. Schmidt

Modeling Real-Time Systems – Challenges and Work Directions 373
J. Sifakis

VEST – A Toolset for Constructing and
Analyzing Component Based Embedded Systems 390
J.A. Stankovic

Embedded Software: Challenges and Opportunities 403
J. Sztipanovits, G. Karsai

Embedded Software in Network Processors – Models and Algorithms 416
L. Thiele, S. Chakraborty, M. Gries, A. Maxiaguine, J. Greutert

Design of Autonomous, Distributed Systems 435
T. Simsek, P. Varaiya

Formalizing Software Architectures for Embedded Systems 451
P. Binns, S. Vestal

Reliable and Precise WCET Determination for a Real-Life Processor 469
C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, R. Wilhelm

Embedded Systems and Real-Time Programming 486
N. Wirth

Embedded Software for Video 493
W. Wolf

Author Index 503

Heterogeneous Modeling Support for Embedded Systems Design

Perry Alexander and Cindy Kong

The University of Kansas
Information and Telecommunications Technology Center
2291 Irving Hill Rd
Lawrence, KS 66044
{alex,ckong}@ittc.ukans.edu

Abstract. Effective design of embedded computer systems requires considering information from multiple domains in a model-centered approach. Using a model-centered approach, the system designer defines and composes models representing multiple system perspectives throughout the design, implementation and testing process. Rosetta is a heterogeneous systems-level modeling language designed to support specification and analysis of complex, computer-based systems. Rosetta provides a model-centered specification capability that allows specifiers to define and combine system models. Users define models, called facets, and assemble those models to define components and systems using facet composition. Each facet model is written with reference to a domain that defines a vocabulary and semantics for the model definition. To model interaction between specifications from different domains, Rosetta provides an interaction definition mechanism based on institution theory. The Rosetta model-centered specification approach allows systems designers to specify many domains of interest from many perspectives and supports predictive design analysis at the systems-level.

1 Introduction

Systems Engineering is characterized by two important characteristics: (i) the integration of information from multiple, heterogeneous sources; and (ii) working at high levels of abstraction with incomplete information. Heterogeneous information integration must occur when building systems using components from various design domains and when modeling different aspects of the same component. Support for higher abstraction levels is increasingly important due to increases in system complexity and the high-level nature of systems level design. With the advent of complex embedded systems design, engineering of the even the smallest embedded systems becomes a systems engineering task.

Model-centered semantics and languages are necessary to support the design of computer-based embedded systems. Such languages allow the designer to concentrate equally on all heterogeneous aspects of the system being designed. Traditional systems design techniques rely largely on component-based models.

T.A. Henzinger and C.M. Kirsch (Eds.): EMSOFT 2001, LNCS 2211, pp. 1–13, 2001.
© Springer-Verlag Berlin Heidelberg 2001

Specifically, designers center on a structural systems model that parallels its physical architecture and/or the architecture of the organization developing it. Components are identified and connected to define the new system with all component interactions occurring via defined connections. Concentrating solely on a structural model results in the designer potentially ignoring other important models.

As an example, consider the definition of an embedded system involving software processes executing on a CPU, communicating with a collection of peripheral devices. The software architecture defines a collection of software processes communicating with peripheral devices treating the CPU as a resource necessary for performing computation. The hardware architecture defines a collection of peripherals communicating over ports with a CPU. The EMI architecture treats each component as a power spectrum with communication occurring outside the boundaries of the defined interface. In one instance, the CPU is a resource while in another it is a part of the system architecture. In the software and hardware architectures interaction occurs via well-defined interfaces while in the EMI architecture interactions occur around the interface. Multiple perspectives produce multiple design facets that must be reconciled during the design process. Selecting and emphasizing one architecture necessarily makes heterogeneous specification and analysis difficult.

The model-centered approach supports defining, composing and projecting models to define and analyze systems. The designer specifies each system facet using appropriate design semantics and assembles those models to define complete systems and components. The hardware, software and EMI models exist simultaneously and together define several system views. The model-centered approach subsumes the component-centered approach by treating structural models as one specific system model. Furthermore, allowing different semantics for each system model supports representation of information in a manner suitable for that domain. The model-centered approach parallels that of human organizations. Each domain specialist maintains their own models written using domain specific semantics leaving the systems engineer to make tradeoff decisions and integrate information.

Rosetta is an emerging design language whose specific goal is supporting model-centered systems level design. Specifically, Rosetta provides semantic support for defining and combining models from multiple domains using multiple domain semantics. In addition, Rosetta provides support for modeling and analysis at levels of abstraction much higher than current computer-based systems design languages. Its semantics are formally defined and it is highly extensible to support adaptation to emerging systems domains.

2 Facets and Models

The Rosetta language and design methodology are based on defining and composing systems models. In Rosetta, *facets* are the basic unit of design definition. Each facet is a domain specific model of a component or system. To support

heterogeneity in designs, facets use different *domains* to provide vocabulary and semantics specific to a particular domain of interest. Facets are written to define various system aspects and are then assembled to provide complete models of components, component aggregations and systems. For example, an embedded system component may have requirements from functional, power, and packaging domains. Each model uses a different Rosetta domain to provide semantics for definition specific to that domain.

The definition of facets is achieved by: (i) directly defining model properties; (ii) composition previously defined facets; or (iii) projecting a facet into a new domain. Direct definition allows users to choose a specification domain and specify properties that must hold in that domain. Facet composition allows users to select several domain specific models and compose a system model that consists of elements from each facet domain. Finally, projection function allows the user to project a model from one domain into another. The abstract semantics of Rosetta is based upon this specification and combination of models representing information from various design domains.

The facet syntax used for direct definition is designed to be familiar to engineers with experience in existing hardware description languages. Figure 1 shows a specification of the functional aspects of Schmidt trigger. Although the specification is simple, it demonstrates many aspects of the facet specification concept.

```
facet trigger-req(x::in real, y::out bit) is
  s::bit;
begin continuous
  t1: s'= if s=1 then if x >= 0.4 then 1 else 0 endif
                 else if x =< 0.7 then 0 else 1 endif
          endif;
  t2: o@t+10ns = s;
end trigger-req;
```

Fig. 1. Rosetta specification of a Schmidt trigger.

As with traditional computer-based systems description techniques, the trigger specification opens by naming the model (`trigger-req`) and defining a parameter list. A local variable (`s`) is defined to describe the facet state and the begin keyword opens the specification body. Following the begin keyword a domain (`continuous`) for the specification is identified. Finally, two terms (`t1` and `t2`) are defined that describe the behavior of the trigger.

What makes the facet model different than traditional specifications is the identification of the domain. A primarily difficulty in addressing the systems-level design problem is combining information from multiple domains in a single design notation. The domain supports this by allowing each model to reference

a base set of concepts that support definition of information in one particular design area. Thus, designers are not required to specify all aspects of their systems using a single semantic notation. In the trigger specification, the continuous domain provides the concepts of time, instantaneous change, ordering and state. Such concepts are important to continuous time modeling, but may not be important when modeling power consumption or area. Such models reference the constraints domain for defining such information. Figure 2 shows a specification of constraints associated with the Schmidt trigger from Figure 1.

```
facet trigger-perf is
  p :: power;
  h :: heat;
  a :: area;
begin constraint
  c1: p =< 10mW;
  c2: h =< 3mJ;
  c3: a =< 10nm;
end trigger-perf;
```

Fig. 2. Performance constraints for the Schmidt Trigger from Figure 1

The constraints domain provides the trigger specification with concepts of power, heat dissipation, area and other non-functional design requirements. Unlike the continuous domain, the constraints domain has no concept of state or time as such concepts are not needed to describe performance information. However, the constraint domain does define units and quantities useful in specifying performance issues and constraint information.

To define a device that satisfies multiple requirements sets, facet composition operators are provided. The definition in Figure 3 uses *facet conjunction* to define a new Schmidt trigger facet that combines both specification models and asserts that both must hold simultaneously. The declaration defines a new facet called `trigger` and then asserts that the new facet is the conjunction of the trigger requirements and performance models. Effectively, if a trigger is examined, it can be viewed from either a functional or requirements perspective.

```
trigger::facet is trigger-req and trigger-perf;
```

Fig. 3. A single Schmidt trigger defined from the requirements and constraints facets from Figure 1 and Figure 2.

The Rosetta approach, *i.e.* defining and combining models, is important as it reflects how systems engineers currently address their problems. The syntax

makes the semantics approachable, however the real contribution of Rosetta is the semantics of designing and composing of component models. Through the use of domains, users are provided with design vocabulary specific to their domain of expertise rather than forcing the domain expert to work in a language that is to general or otherwise unsuitable for their needs.

The semantics of facet composition and projection are based on mathematical category theory concepts. Formally, the Rosetta facet conjunction operation is a *co-product* of the participating domains. The corresponding projection operations that extract information with respect to a specific domain define a *product* on the original domains. The specifics of the co-product and product operations are defined by the Rosetta interaction construct.

3 Systems and Components

Heterogeneity in systems design not only emerges when defining multiple facets of the same system, but also when describing systems structurally by combining components. VHDL provides a structural definition capability that is mimicked in the Rosetta semantics using a concept called re-labeling. When one facet is included in another, an instance of that facet is created by re-labeling or renaming the facet. Facet parameters are used as ports and channel data types used to model communication between components. Figure 4 shows a simple specification that combines several components to form a transmitter definition. In this example, systems level components are included and combined in an abstract specification. Specifically, an analog to digital converter, modulator and amplifier are used in series to produce a signal suitable for output to an antenna. Both a functional model and power consumption model are shown. When composed, the two facets define a systems level transmitter with a mechanism for calculating power consumption from constituent components. Further, the semantics of facet inclusion maps each power constraint facet to its associated functional facet. This allows power constraints to be automatically mapped to corresponding functional constraints.

Like single components, Rosetta systems may include component models from multiple design domains. Thus, Rosetta provides a mechanism for defining and understanding systems comprised of components from analog, digital, mechanical and optical domains in the same semantic framework. Furthermore, users may define new specification domains to extend Rosetta to address new domains and new modeling techniques for existing domains. Such a capability is extremely important for the future of any potential systems design language.

In addition to the basic facet model, Rosetta also provides a component model where designers may include usage assumptions and analysis requirements. The component model is simply a composition of facets describing each of these system aspects. Associated with each usage assumption and analysis requirement is a justification structure that records why the assumption or requirement has been met. Thus, the user is able to track verification requirements locally with the system requirements being tracked. The locality principle sug-

```
facet tx-structure(i::in real,
                   o::out real) is
  d::bitvector;
  s::real;
begin continuous
  a2d1: a2d(i,d);
  mod1: modulator(d,50Khz,s);
  amp1: amplifier(s,20db,o)l
end tx-structure;
```

```
facet tx-power is
  p::power;
begin constraint
  a2d1: a2d-power;
  mod1: mod-power(50Khz);
  amp1: amp-power(20db);
end tx-structure;
```

```
tx::facet is tx-structure and tx-power;
```

Fig. 4. Example structural Rosetta representation for a simple transmitter system. A functional model, a constraint model and a systems level model are shown.

gests that maintaining verification conditions near systems requirements reminds users to update verification conditions as requirements are modified.

4 Domain Interaction

It is not sufficient to simply model domain specific information in isolation. Cross-domain interaction is the root cause of many systems failures and difficult design problems. Systems level design requires understanding the collective behavior of interconnected components from different design domains, not simply component level behaviors. Further, interaction also occurs between different models at the component level. Rosetta provides methodologies for explicitly modeling and evaluating such interactions by defining how definitions from individual domains affect each other.

There are two fundamental approaches for modeling interaction between design domains in current design practice. The first is to choose a single domain for representing all system models. Analysis tools from that domain are then used to model entire systems. The problem with this approach is that the modeling domain always favors one domain over all others. Designers or tool builders must force information represented quite naturally in one domain into a potentially less suitable representation. The net result is a model that cannot be analyzed or a systems model that ignores certain domains.

The second approach to addressing model interaction is to rely on human observation to detect and analyze interactions between domains. Models are developed and maintained using separate representations and semantics. Human designers are required to constantly observe and predict interactions. This approach is the most common in today's environment and will most probably always exist to some extent. However, Rosetta attempts to provide semantic support for this activity by providing a mechanism for specifying and thus analyzing domain interactions.

Rosetta provides domain and interaction semantic constructs for defining domains and domain interactions respectively. The domain construct simply pro-

vides a mechanism for identifying a facet as a domain model. Syntax is provided in Rosetta to allow users to identify and define domains quickly, but their semantics is the same as a facet. When a facet includes a domain, it extends that domain to represent a specific component or system model. When facets using domains involved in an interaction specification are included in a definition, the interaction specification is automatically included.

Using Rosetta's reflection capabilities, a *projection function* defines how information from one domain can be transformed into another. Formally, terms in one facet definition are treated as axioms and transformed into axioms associated with another domain. Because Rosetta facets are first-class items, no special semantics is used to define such functions. The elements of the facet definition are accessed using pre-defined meta-functions and transformed as appropriate.

An *interaction* defines when specifications in one domain imply specifications in another by defining two projection functions. An interaction is a facet that accepts two domains as parameters. It then instantiates the projection function `M__I` twice to define mappings between domains. Two facets from the interacting domains are accepted as arguments and the function evaluates to a set of terms defining theorems that must be added to the second facet to represent the interaction. Thus, $\texttt{M__I}(\texttt{f}_1, \texttt{f}_2) :: \texttt{set}(\texttt{term})$ returns a collection of terms that represents the impact of $\texttt{f}_1$ on $\texttt{f}_2$. Adding those terms to $\texttt{f}_2$ defines a new facet taking into consideration the results of the interaction. Rosetta's interaction definition mechanism supports pair-wise facet interaction by extension. However, using Rosetta's reflective meta-functions allows the designer to specify interactions any manner desired.

Figure 5 shows an example interaction definition defining interaction between the state-based and logic domains. The interaction states that any term that is true in the logic domain is also true in all states in the state-based domain. Specifically, terms from the logic domain are invariant in the state-based domain. Although simple, this interaction allowed detection of a constraint violation in the context of an aircraft redesign problem.

```
interaction and(f::state-based,g::logic) is
begin interaction
  t1::M__I(f::logic,g::state-based)::set(term) is
      {ran(t::M__terms(f) | ran(s::g.S | t@s))};
  t2::M__I(g::state-based,f::logic)::set(term) is
      {sel(t::M__terms(g) | forall(s::g.S | t@s))};
end and;
```

Fig. 5. Interaction between the logic and state-based domains

While the interaction models when and where information from one domain impacts another under composition, the projection extracts one model from

another. This projection operation directly supports meta-modeling allowing the extraction of domain specific models from model composition results.

Together, interactions and projections support powerful model composition and extraction capabilities. Given two facets `F` and `G`, the composition of the models can be specified as `F` and `G`. Using the interaction operation, the resulting facet includes not only information from the original facets, but also information resulting from model interaction. Using projection operations on `F` and `G` back into the original domains results in models `F'` and `G'` that are expressed in the original domains. This result is exceptionally important as the resulting models are presented to the designer in the vocabulary of the original domain. Thus, the designer can deal with the results of the interaction without detailed knowledge of the other interacting models.

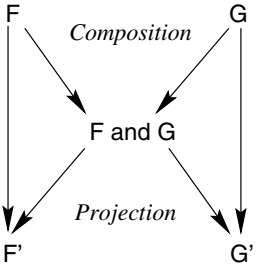


Fig. 6. Morphism relationships between `F` and `G` under composition and projection.

A concrete example of the successful application of composition and projection to a real engineering problem involves redesign of a mechanical actuator associated with a jet aircraft. In this particular problem, a control surface exhibited flutter characteristics and required stiffening of the hydraulic actuator driving the control surface. After a series of trade-off decisions, the cylinder was bored out to provide more fluid flow and thus more power. Although this corrected the flutter problem during high speed performance, it created a new problem at low speed where more power is required to drive the actuator.

Rosetta models were constructed to describe this problem and the interaction described in Figure 5 was used to generate a new analysis model that accounted for constraints in the functional analysis process. A functional model of the actuator was composed with a constraints model indicating the power consumption constraints. The projection operation was then used to generate a new analysis model in the functional domain. In this situation, the evaluation model used Matlab to generate a simulation of the increased power. With the constraint included, engineers attempting to solve the flutter problem immediately recognize the power problem associated with low speed operation. Although this particular example exhibits systems-level design in the mechanical domain, it is a concrete

example of how interactions and projections together assist in the systems design activity.

The semantics of domain interactions is based on institution theory [7]. Formally, an institution defines a relationship between formal systems that allows information from one formal system to be moved to another. A thorough discussion of interaction semantics is beyond this paper. It is sufficient to view an interaction as a relationship between theorems in one formal system and theorems in another. It is important to note that the institution model in conjunction with the category theoretic model is limited in the case where the composition generates a contradiction. In such cases projections are not well-defined. However, such situations are important to detect during the modeling process.

Figure 7 shows a collection of currently defined or emerging Rosetta specification domains. Solid arrows represent interactions where all information from the source domain can be represented in the target domain. Such interactions are referred to are homomorphism and define an inheritance hierarchy. Dashed arrows represent a sampling of interactions that are not homomorphic, but have been used successfully to model interactions between such domains.

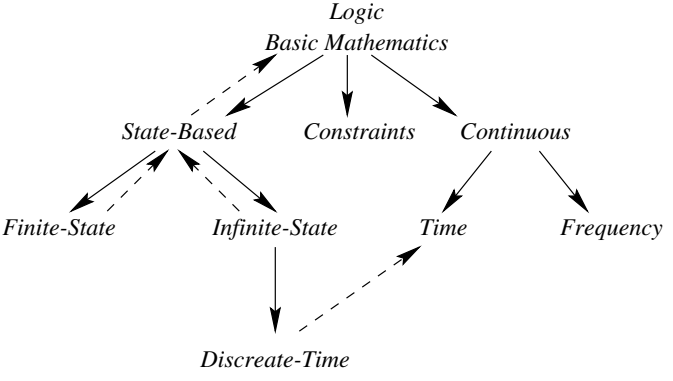


Fig. 7. Currently defined or working domains and selected interactions.

Rosetta currently provides semantics for mathematical specifications, state-based (finite and infinite), discrete time, and continuous time specification. Support for specification of testing requirements, constraints, mechanical systems, and Fourier domains are currently being developed. Interactions are defined for many supported domains with others under development. It is anticipated and intended that Rosetta users will write specialized domains on top of the basic set of computation and time model domains. For example, a digital specification domain is planned using the discrete time domain as a semantic basis. Thus, the basic domain set provides a basis for constructing discipline-specific domain models.

5 Semantic Issues

The base Rosetta language reflects requirements defined by the systems-on-chip and embedded systems communities. The Semiconductor Industry Council's Systems Level Design Language Committee initially defined a set of core requirements that include:

- Precise, formal semantics
- Explicit representation of non-functional constraints
- Support for analysis activities other than simulation
- Representation of heterogeneous system components and models

In support of these requirements, Rosetta is declarative and formally defined. No guarantee is made concerning the executability of specifications although an executable subset is being identified. Rosetta facet terms are simultaneously true with no concept of order. As a reflective language, all Rosetta specification constructs are first-class items in that they can be treated as data structures. Thus, functions can be used to define types, facets and other specification constructs. Although this significantly complicates type checking, it allows for powerful specification capabilities. The type system and expression semantics are loosely based on Hehner's Universal Algebra [8], the PVS type system [12], and the Haskell [9] function semantics. Type correctness is in general not decidable due to the first-class nature of types and functions.

All Rosetta functions are pure and side-effect free, but are not referentially transparent. The decision to make Rosetta specifications functional comes from a language requirement for a formal semantics and observation of VHDL usage and VHDL signal semantics. At first this may seem odd, but consider a classical VHDL signal assignment such as:

```
s <= inc(s) after 4ns;
```

where the signal `s` is updated with an new value `4ns` from the current time. The signal assignment operator effectively identifies when the next state occurs (`4ns` in the future) and what the value of the signal is in that new state (`inc(s)`). In the Rosetta continuous time domain, a similar expression has the form:

```
s@t+4ns = inc(s);
```

The semantics of the model of computation are exposed in the temporal expression (`t+4ns`) rather than hidden in the signal assignment syntax. An assertion that the value of `s` in the next state is made by equating it to the value of `inc(s)`. Defining `inc(s)` as a pure function is quite natural while providing support for such useful techniques as formal analysis and partial evaluation.

It is important to note that Rosetta's language semantics is defined independently from its concrete syntax. Although a facet does have a defined syntactic representation, its semantics was defined initially and syntax developed later to address usability and readability issues. Other syntactic representations are not precluded and are being encouraged.

6 Related Work

Model Integrated Computing (MIC) [11] activities ongoing at Vanderbilt University represent an approach similar to that employed by the Rosetta semantics. Multiple system models are maintained and transformed into heterogeneous tools for analysis using meta-modeling techniques similar to Rosetta projections.

The Advanced Design Environment Prototyping Tool (ADEPT) [10] developed at The University of Virginia represents a tool that considers multiple design perspectives. ADEPT takes an approach opposite to Rosetta by translating models into a common semantic language. ADEPT is highly successful in its application domain because of limited domain scope. Rosetta avoids the unified modeling language approach because it's domain of application is larger than that of ADEPT.

Model-based co-design surrounding the DEVS specification system [13] allows accounting for different aspects of a system design. Multiple perspectives including performance constraints are considered in modeling and evaluation a system prior to implementation. The DEVS work is centered on co-design decisions relating to hardware/software tradeoff decisions. Simulation is the primary means for evaluating specifications. In general, DEVS provides excellent co-design support, but is intentionally more application specific than Rosetta.

The Ptolemy [1] framework and its successor Ptolemy II are heterogeneous design environments for rapid prototyping of digital signal processing systems. Like Rosetta they are extensible by design and incorporate many of the heterogeneous modeling capabilities proposed here. Ptolemy II introduces the concept of polymorphic domains that provide an inheritance mechanism similar to Rosetta's interaction semantics. The major differences are the application domain and what heterogeneity means in that domain. Specifically, Ptolemy addresses data-flow systems where the major source of heterogeneity results from different time models used in specification domains. Ptolemy has proven highly successful implying that similar techniques may be applicable in the formal analysis domain.

A theoretical framework similar to Rosetta's facets and facet interaction is viewpoints [2,3]. Viewed abstractly, a viewpoint and a facet are identical in their intended purpose. Both allow the systems engineer to specify one system model. Like facets, multiple viewpoints are simultaneously valid for a system and must be mutually consistent. The viewpoints approach provides an excellent mechanism for requirements modeling in a heterogeneous environment. The primary difference being that viewpoints are manipulated informally using traditional requirements engineering techniques. Viewpoints are a cognitive structure while facets in Rosetta are mathematical descriptions.

Mechanisms for managing multiple, heterogeneous viewpoints are represented by work including knowledge-based critiquing [6] and parallel development [5]. These techniques are quite effective in managing and resolving interaction between specification domains. Unlike the proposed approach they are largely informal and heuristic in nature.

Use of categories to describe morphisms between system representations is an established technique documented by Erig and Mahr [4] and applied the synthesis of software systems in the KIDS [14] and SpecWare environments, among others. Institution theory is proposed by Goguen [7] as a mechanism for defining systems of logics.

7 Current Status

The Rosetta language and approach have been evaluated in several different domains and applications. The primary driver for Rosetta is the systems-on-chip domain where engineers have recognized the significance of systems-level design problems in their computer-based systems. In addition to the actuator design example, models have been constructed for satellite communications links, transmitter/receiver pairs, simple digital circuits, and DSP systems, and several systems-on-chip examples. Current research is investigating interactions between the information domain (software) and the physical domain examining resource requirements and impact of software behavior on such system aspects as packaging and safety.

Commercial applications are beginning with an IP reuse tool and test vector generation tool currently available in beta form. The IP reuse tool uses Rosetta's formal semantics to provide high assurance retrieval. Component specifications are formally compared with problem specifications to determine if the component can successfully be reused to solve the problem. Test vector generation system uses Rosetta requirements and domains for defining test cases and test vectors to automatically generate test vectors from requirements. Test scenarios are generated from requirements and abstract vectors generated to cover each test scenario. These abstract vectors are then converted automatically to formats suitable for simulation systems such as VHDL, Verilog and C++.

To assure usage across domains and throughout the design lifecycle, Rosetta is vendor and tool independent while providing support for importing and exporting models from various CAD languages and tools. An interchange format has been developed that is compatible with the MoML interface description language developed by University of California at Berkeley. The Rosetta semantics and language are being jointly developed by the Accellera CAD Standards organization's Systems Level Design Language Committee, Titan (formerly Intermetrics), and The University of Kansas Information and Telecommunications Technology Center with guidance and support from The European Chips and Systems Initiative, The Semiconductor Industry Council, DARPA, and The United States Air Force. An Accellera standardization committee has been formed and a draft standard is being prepared.

Rosetta provides modeling capabilities needed to support systems-level design that allow modeling at high levels of abstraction, modeling heterogeneous systems, and support for model composition. By allowing the user to define and compose models, Rosetta allows flexibility and heterogeneity in design modeling and supports meaningful integration of models representing all aspects of sys-

tems design. Furthermore, Rosetta's formal semantics provide an unambiguous means for defining and composing models. Finally, the default Rosetta syntax provides a user-friendly mechanism designed to be comfortable to today's existing hardware description language user base.

The draft Rosetta language description, a prototype Java-based language parser, and a collection of beta-level tools are freely available on the Rosetta web page at `http://www.sldl.org`.

References

[1] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *Int. Journal of Computer Simulation*, 4:155–182, April 1994.

[2] S. Easterbrook. Domain modeling with hiearchies of alternative viewpoints. In *Proceedings of the First International Symposium on Requiremetns Engineering (RE-93)*, San Diego, CA, January 1993.

[3] S. Easterbrook and B. Nuseibeh. Managing inconsistencies in evolving specifications. In *Proceedings of the Second IEEE International Symposium on Requirements Engineering (RE-95)*, pages 48–55, York, UK, April 1995. IEEE Press.

[4] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specifications 1: Equations and Initial Semantics*. EATCS Mongraphs on Theoretical Computer Science. Springer–Verlag, Berlin, 1985.

[5] M. Feather. The evolution of composite specifications. In *Proceedings of the* 4^{th} *IEEE International Workshop on Software Specification and Design*, Monterey, CA, April 1987. IEEE Press.

[6] S. Fickas and P. Nagarajan. Being suspious: Critiquing problem specifications. In *Proceedings of The Seventh Conference on Artificial Inteligence AAAI 88*, St. Paul, MN, July 1988. AAAI.

[7] J. Goguen. Parameterized Programming. *IEEE Transactions on Software Engineering*, SE-10(5):528–543, 1984.

[8] Eric C. R. Hehner. *A Practical Theory of Programming*. Texts and Monographs in Computer Science. Springer-Verlag, New York, 1993.

[9] P. Hudak. *The Haskell School of Expression*. Cambridge University Press, 2000.

[10] S. Kumar, R. Klenke, J. Aylor, B. Johnson, R. Williams, and R. Waxman. Adept: A unified system level modeling design environment. In *Proceedings of The First Annual RASSP Conference*, pages 114–123, Arlington, VA, August 1994. DARPA.

[11] A. Misra, G. Karsai, J. Sztipanovits, A. Ledeczi, and M. Moore. A model-integrated infomration system for increasing throughput in discrete manufacturing. In *Proceedings of The 1997 Conference and Workshop on Engineering of Computer Based Systems*, pages 203–210, Montery, CA, March 1997. IEEE Press.

[12] S. Owre, N. Shankar, J. Rushby, and Stringer-Calvert D. W. J. *PVS System Guide*. SRI International Computer Science Laboratory, 333 Ravenswood Ave, Menlo Park, CA 94025, 2.3 edition, September 1999.

[13] S. Schulz, J. Rozenblit, M. Mrva, and K. Buchenrieder. Model-based codesign. *IEEE Computer*, 31(8):60–67, August 1998.

[14] Douglas R. Smith. KIDS: A Semiautomatic Program Development System. *IEEE Transactions on Software Engineering*, 16(9):1024–1043, 1990.

Hierarchical Hybrid Modeling of Embedded Systems*

R. Alur, T. Dang, J. Esposito, R. Fierro, Y. Hur, F. Ivančić, V. Kumar, I. Lee, P. Mishra, G. Pappas, and O. Sokolsky

University of Pennsylvania
http://www.seas.upenn.edu/hybrid/

Abstract. This paper describes the modeling language CHARON for modular design of interacting hybrid systems. The language allows specification of architectural as well as behavioral hierarchy, and discrete as well as continuous activities. The modular structure of the language is not merely syntactic, but is exploited by analysis tools, and is supported by a formal semantics with an accompanying compositional theory of refinement. We illustrate the benefits of CHARON in design of embedded control software using examples from automated highways concerning vehicle coordination.

1 Introduction

An embedded system typically consists of a collection of digital programs that interact with each other and with an analog environment. Examples of embedded systems include manufacturing controllers, automotive controllers, engine controllers, avionic systems, medical devices, micro-electromechanical systems, and robots. As computing tasks performed by embedded devices become more sophisticated, the need for a sound discipline for writing embedded software becomes more apparent (c.f [23]). Model-based design paradigm, with its promise for greater design automation and formal guarantees of reliability, is particularly attractive given the following trends.

Software Design Notations. Modern object-oriented design paradigms such as *Unified Modeling Language* (UML) allow specification of the architecture and control at high levels of abstraction in a modular fashion, and bear great promise as a solution to managing the complexity at all stages of the software design cycle [7]. There are emerging tools such as RationalRose (see `www.rational.com`) that support modeling, simulation, and code generation, and are increasingly becoming popular in domains such as automotive software and avionics.

Control Engineering. Traditionally control engineers have used tools for continuous differential equations such as MATLAB (see `www.mathworks.com`) for modeling of the plant behavior, for deriving and optimizing control laws, and for validating functionality and performance of the model through analysis and

* Supported by DARPA MoBIES grant F33615-00-C-1707

T.A. Henzinger and C.M. Kirsch (Eds.): EMSOFT 2001, LNCS 2211, pp. 14–31, 2001.
© Springer-Verlag Berlin Heidelberg 2001

simulation. Tools such as SIMULINK recently augmented the continuous modeling with state-machine-based modeling of discrete control.

Formal Verification Tools. Model checking is emerging as an effective technique for debugging of high-level models (see [10] for a survey). Model checkers such as SMV [26] and SPIN [20] have been successful in revealing subtle errors in cache coherency protocols in multiprocessors and communication protocols in computer networks. In recent years, the model checking paradigm has been successfully extended to models with continuous variables leading to tools such as UPPAAL [22], HYTECH [18], and CheckMate [8].

This paper describes our modeling language, CHARON, that is suitable for high-level specification of interacting embedded systems. We proceed to discuss the three distinguishing aspects of CHARON.

Hybrid Modeling. Traditionally, control theory and related engineering disciplines, have addressed the problem of designing robust control laws to ensure optimal performance of processes with continuous dynamics. This approach to system design largely ignores the problem of implementing control laws as a piece of software and issues related to concurrency and communication. Computer science and software engineering, on the other hand, have an entirely discrete view of the world, which abstracts from the physical characteristics of the environment to which the software is reacting to, and is typically unable to guarantee safety and/or performance of the embedded device as a whole. An embedded system consisting of sensors, actuators, plant, and control software is best viewed as a *hybrid* system. The relevance of hybrid modeling has been demonstrated in various applications such as coordinating robot systems [2], automobiles [6], aircrafts [29], and chemical process control systems [13].

Early formal models for hybrid systems include phase transition systems [25] and hybrid automata [1]. While modularity in hybrid specifications has been addressed in languages such as hybrid I/O automata [24], CHARON allows richer specifications. Discrete updates in CHARON are specified by *guarded actions* labeling transitions connecting the modes. Some of the variables in CHARON can be declared *analog*, and they flow continuously during continuous updates that model passage of time. The evolution of analog variables can be constrained in three ways: *differential* constraints (e.g. by equations such as $\dot{x} = f(x, u)$), *algebraic* constraints (e.g. by equations such as $y = g(x, u)$), and *invariants* (e.g. $|x - y| \leq \varepsilon$) which limit the allowed durations of flows.

Hierarchical Modeling. Modern software design paradigms promote *hierarchy* as one of the key constructs for structuring complex specifications. We are concerned with two distinct notions of hierarchy. In *architectural hierarchy*, a system with a collection of communicating agents is constructed by parallel composition of atomic agents, and in *behavioral hierarchy*, the behavior of an individual agent is described by hierarchical sequential composition. The former hierarchy is present in almost all concurrency formalisms, and the latter, while present in all block-structured programming languages, was introduced for state-machine-based modeling in STATECHARTS [17].

In CHARON, the building block for describing the system architecture is an *agent* that communicates with its environment via shared variables. The language supports the operations of *composition* of agents to model concurrency, *hiding* of variables to restrict sharing of information, and *instantiation* of agents to support reuse. The building block for describing flow of control inside an atomic agent is a *mode*. A mode is basically a hierarchical state machine, that is, a mode can have submodes and transitions connecting them. Variables can be declared locally inside any mode with standard scoping rules for visibility. Modes can be connected to each other only via well-defined entry and exit points. We allow *sharing* of modes so that the same mode definition can be instantiated in multiple contexts. To support *exceptions*, the language allows group transitions from default exit points that are applicable to all enclosing modes, and to support *history retention*, the language allows default entry transitions that restore the local state within a mode from the most recent exit.

Compositional Semantics. Formal semantics leads to definitions of *semantic* equivalence (or refinement) of specifications based on their observable behaviors, and compositional means that semantics of a component can be constructed from the semantics of its subcomponents. Such formal compositional semantics is a cornerstone of concurrency frameworks such as CSP [19] and CCS [27], and is a prerequisite for developing modular reasoning principles such as compositional model checking and systematic design principles such as stepwise refinement. The global nature of time makes it challenging to define semantics of hybrid components in a modular fashion. For rich hierarchical specifications, features such as as group transitions, exceptions, and history retention, cause additional difficulties.

CHARON supports observational trace semantics for both modes and agents [4]. The key result is that the set of traces of a mode can be constructed from the traces of its submodes. This result leads to a compositional notion of refinement for modes. Suppose we obtain an implementation design I from a specification design S simply by locally replacing some submode N in S by a submode M. Then, to show I refines S, it suffices to show that M refines N.

Overview. The remaining paper is organized as follows. In Section 2, we present the features of the language CHARON, and in Section 3 we describe the formal semantics and accompanying compositional refinement calculus. We use examples from the automotive experimental platform of the DARPA's MoBIES program for illustrative purposes. Section 4 gives a summary of the design toolkit, and we conclude in Section 5 with pointers to ongoing research on formal analysis.

2 Modeling Language

2.1 Agents and Architectural Hierarchy

We present an example from the MoBIES Automotive Open Experimental Platform (OEP) to illustrate the features of CHARON. Figures 1, 2, and 3 are CHARON agent diagrams illustrating the architectural hierarchy of a team of two vehicles.

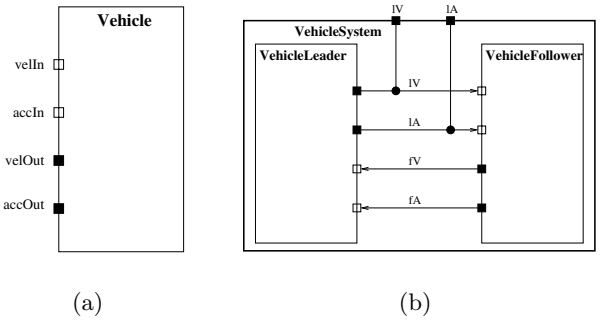


Fig. 1. The Leader-Follower Vehicle System

A single vehicle is represented by the agent `Vehicle`, shown in Figure 1(a). Each agent has a well-defined interface which consists of its typed input and output variables. In the case of `Vehicle`, `velIn` and `accIn` are the input variables, and `velOut` and `accOut` are the outputs. Formally, an *agent* consists of a set of variables V, a set of initial states, and a set of modes TM. The set V is partitioned into *local* variables V_l and *global* variables V_g; global variables are further partitioned into input and output variables. Type correct assignments of values to variables are called valuations and denoted Q_V. The set of initial states $I \subseteq Q_V$ specifies possible initializations of the variables of the agent. The modes, described in more detail below, collectively define the behavior of the agent. An *atomic* agent has a single top-level mode. *Composite* agents have many top-level modes and are constructed from other agents as described below.

Figure 1(b) illustrates the three operations defined on agents. It shows a composite agent `VehicleSystem` that contains two instances of the agent `Vehicle` *composed* in parallel. The parallel agents execute concurrently and communicate through shared variables. To enable communication between the two vehicles, global variables are *renamed*. For example, `velIn` of the follower agent and `velOut` of the leader agent are both renamed to `lV`. Finally, the communication between the vehicles can be *hidden* from the outside world. In our example, only the leader's outputs `lV` and `lA` are the outputs of the composite system. The composite agent is written in CHARON syntax as below:[1]

```
agent VehicleSystem {
 write analog real lV, lA;
 private analog real fV, fA;
 agent VehicleLeader= Vehicle[velIn,velOut,accIn,accOut := fV,lV,fA,lA];
 agent VehicleFollower= Vehicle[velIn,velOut,accIn,accOut := lV,fV,lA,fA];}
```

[1] CHARON also allows parameterized definitions. For instance, the initial position of the vehicle can be defined as a parameter within the agent `Vehicle`, and can be assigned to different values in the two instances `VehicleLeader` and `VehicleFollower`.

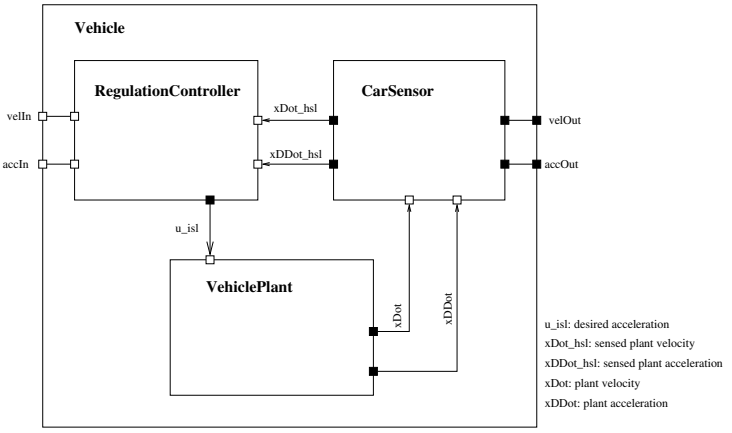


Fig. 2. The Vehicle Agent

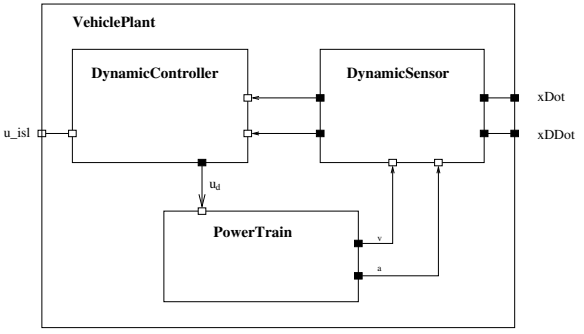


Fig. 3. The VehiclePlant Agent

The agent `Vehicle` itself has a hierarchical structure. Figure 2 illustrates the overall vehicle architecture, which comprises the regulation controller, the car sensor, and the vehicle plant. The higher level regulation controller handles data from the car sensor or other vehicles and generates a desired acceleration to the vehicle plant. The lower level dynamics controller equipped in the vehicle plant controls actual vehicle dynamics such as throttling and braking. The vehicle plant is composed of the dynamic controller, the dynamic sensor, and the powertrain. Figure 3 describes the vehicle plant. The dynamic controller maps the control command `u` onto a desired throttle position or brake command `u_d`. The sign of `u_d` will define two submodes of the powertrain: acceleration and brake. We show how the behavior of an agent is modeled in the next section.

2.2 Modes and Behavioral Hierarchy

Modes represent behavioral hierarchy in the system design. The behavior of each atomic agent is described by a mode, which corresponds to a single thread of

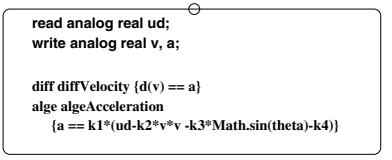


Fig. 4. The Behavior of the Agent `PowerTrain`

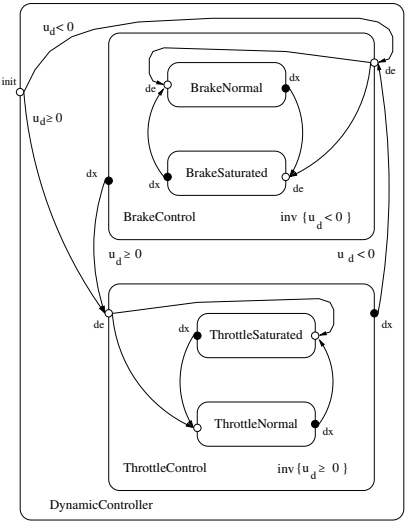


Fig. 5. The Behavior of the Agent `DynamicController`

control. At the lowest level of the behavioral hierarchy are atomic modes. They describe continuous behaviors. For example, Figure 4 illustrates the behavior of the agent `PowerTrain`. There is a differential constraint `diffVelocity` that asserts the relationship between velocity v and acceleration a: $\dot{v} = a$, and an algebraic constraint `algeAcceleration` for the acceleration, relating it to the current speed v, the control input u_d, and the road grade θ: $a = k_1 \cdot (u_d - k_2 \cdot v^2 - k_3 \cdot \sin(\theta) - k_4)$ for some constants $k_1, \ldots, k_4$.

Composite modes contain a number of submodes. During execution, a composite mode performs discrete transitions, switching between its submodes. Each (sub)mode has a well-defined data interface consisting of typed global variables used for sharing state information, and also a well-defined control interface consisting of entry and exit points that are used to connect modes modes via transitions. For example, the behavior of the agent `DynamicController` is captured by the mode shown in Figure 5. Depending on the sign of the input variable u_d that represents the acceleration desired by the higher-level controller, the dynamic controller applies either brake or throttle, represented by the submodes `BrakeControl` and `ThrottleControl`, respectively. The condition for staying in the `ThrottleControl` mode is captured by the invariant $u_d \geq 0$. When the sign

of u_d changes to negative, the invariant is violated and the controller is forced to take a transition to `BrakeControl`. Note that the mode `ThrottleControl` also has internal structure: acceleration may be normal, when a larger desired acceleration translates in more torque supplied by the engine, or saturated, when the limit of the engine capacity has been reached. The transition from `ThrottleControl` to `BrakeControl` happens regardless of which submode of `ThrottleControl` was active at that time, interrupting any lower-level behaviors.

Formally, a mode M consists of a set of submodes SM, a set of variables V, a set of *entry control points* E, a set of *exit control points* X, a set of transitions T, and a set of constraints $Cons$. As in agents, variables are partitioned into global and local variables. For the submodes of M, we require that each global variable of a submode is a variable (either global or local) of M. This induces a natural scoping rule for variables in a hierarchy of modes: a variable introduced as local in a mode is accessible in all its submodes but not in any other mode. Every mode has two distinguished control points, called default entry (de) and exit (dx) points. They are used to represent such high-level behavioral notions as interrupts and exceptions, which will be discussed in more detail in the following section.

The set $Cons$ of constraints contains constraints of three kinds. An *invariant* specifies when a mode can be active. Continuous trajectories of a variable x can be given by either an algebraic constraint A_x, which defines the set of admissible values for x in terms of values of other variables, or by a differential constraint D_x, which defines the admissible variables for the first derivative of x with respect to time.

Transitions of a mode M can be classified into *entry transitions*, which connect an entry point of M with an entry point of one of its submodes, *exit transitions*, connecting exit points of submodes to exit points of M, and internal transitions that lead from an exit point of a submode to an entry point of another submode. Every transition has a *guard*, which is a predicate over the valuations of mode variables that tells when the transition can be executed. When a transition occurs, it executes a sequence of assignments, changing values of the mode variables. A transition that originates at a default exit point of a submode is called a *group transition* of that submode. A group transition can be executed to interrupt the execution of the submode.

In CHARON, transitions and constraints can refer to externally defined Java classes, thus allowing rich discrete and continuous specifications.

3 Formal Semantics and Compositional Refinement

We proceed to define a compositional formal semantics for CHARON. First, the operational semantics of modes and agents makes the notion of executing a CHARON model precise, and can be used, say, by a simulator. Second, we define an observational semantics for modes and agents. The observational semantics hides the details about internal structure, and retains only the information about

inputs and outputs. Informally, the observational semantics consists of the static interface (such as the global variables and entry/exit points) and dynamic interface consisting of the *traces*, that is, sequences of updates to global variables. Third, for modularity, we show that our semantics is compositional. This means that the set of traces of a component can be defined from the set of traces of its subcomponents. Intuitively, this means that the observational semantics captures *all* the information that is needed to determine how a component interacts with its environment. Finally, we define a notion of refinement (or equivalence) for modes/agents. This allows us, for instance, to relate different models of the agent `PowerTrain`. We can establish that the abstract (simplified) version of powertrain refines the detailed version, and then, to analyze the system of vehicles, use the abstract version instead of the detailed one. The compositional rules about refinement form the basis for analysis in a system with multiple components, each with a simplified and a detailed model.

3.1 Formal Semantics of Modes

Intuitive semantics. Before presenting the semantics formally, we give the intuition for mode executions. A mode can engage in discrete or continuous behavior. During an execution, the mode and its environment either take turns making discrete steps or take a continuous step together. Discrete and continuous steps of the mode alternate. During a continuous step, the mode follows a continuous trajectory that satisfies the constraints of the mode. In addition, the set of possible trajectories may be restricted by the environment of the mode. In particular, when the mode invariant is violated, the mode must terminate its continuous step and take one of its outgoing transitions. A discrete step of the mode is a finite sequence of discrete steps of the submodes and enabled transitions of the mode itself. A discrete step begins in the current state of the mode and ends when it reaches an exit point or when the mode decides to yield control to the environment and lets it make the choice of the next step. Technically, when the mode ends its discrete step in one of its submodes, it returns control to the environment via its default exit point. The closure construction, described below, ensures that the mode can yield control at appropriate moments, and that the discrete control state of the mode is restored when the environment schedules the next discrete step.

Preemption. An execution of a mode can be preempted by a *group* transition. A group transition of a mode originates at the default exit of the mode. During any discrete step of the mode, control can be transferred to the default exit and an enabled group transition can be selected. There is no priority between the transitions of a mode and its group transitions. When an execution of a mode is preempted, the control state of the mode is recorded in a special *history* variable, a new local variable that we introduce into every mode. Then, when the mode is entered through the default entry point next time, the control state of the mode is restored according to the history variable.

The history variable and active submodes. In order to record the location of discrete control during executions, we introduce a new local variable

h into each mode that has submodes. The history variable h of a mode M has the names of the submodes of M as values, or a special value ϵ that is used to denote that the mode does not have control. A submode N of M is called *active* when the history variable of M has the value N.

Flows. To precisely define continuous trajectories of a mode, we introduce the notion of a *flow*. A flow for a set V of variables is a differentiable function f from a closed interval of non-negative reals $[0, \delta]$ to Q_V. We refer to δ as the *duration* of the flow. We denote a set of flows for V as $\mathcal{F}_V$.

Syntactic restrictions on modes. In order to ensure that the semantics of a mode is well-defined, we impose several restrictions on mode structure. First, we assume that the set of differential and algebraic constraints in a mode always has a non-empty set of flows that satisfy them. This is needed to ensure that the set of behaviors of a mode is non-empty. Furthermore, we require that the mode cannot be blocked at any of its non-default control points. This means that the disjunction of all guards originating from a control point evaluates to `true`.

State of a mode. We define the state of a mode in terms of all variables of the mode and its submodes, including the local variables on all levels. We use V_* for the set of all variables. The set of local variables of a mode together with the local variables of the submodes are called the private variables and is denoted as V_p.

The state of a mode M is a pair (c, s), where c is the location of discrete control in the mode and $s \in Q_{M.V_*}$. Whenever the mode has control, it resides in one of its control points, that is, $c \in M.C$. Given a state (c, s) of M, we refer to c as the *control state* of M and to s as the *data state* of M.

Closure of a mode. Closure construction is a technical device to allow the mode to interrupt its execution and to maintain its history variable. Transitions of the mode are modified to update the history variable h after a transition is executed. Each entry or internal transition assigns the name of the destination mode to h, and exit transitions assign ϵ to h. In addition, default entry and exit transitions are added to the set of transitions of the mode. These default transitions do not affect the history variable and allow us to interrupt an execution and then resume it later from the same point.

The default entry and exit transitions are added in the following way. For each submode N of M, the closure adds a default exit transition from $N.dx$ to $M.dx$. This transition does not change any variables of the mode and is always enabled. Default entry transitions are used to restore the local control state of M. A default entry transition that leads from a default entry of M to the default entry of a submode N is enabled if $h = N$. Furthermore, we make sure that the default entry transitions do not interfere with regular entry transitions originating from *de*. The closure changes each such transition so that it is enabled only if $h = \epsilon$. The closure construction for the mode `DynamicController` introduced in Section 2 is illustrated in Figure 6.

Operational semantics. An operational view of a closed mode M with the set of variables V consists of a *continuous* relation R^C and, for each pair $c_1 \in E$, $c_2 \in X$, a *discrete* relation $R^D_{c_1,c_2}$.

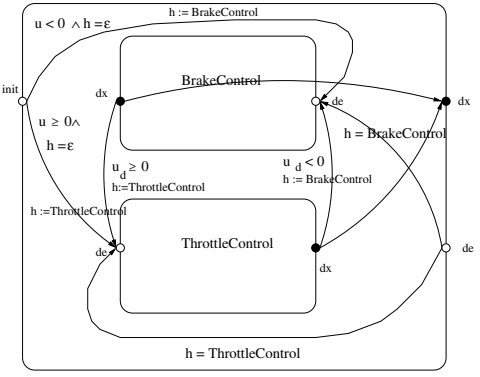


Fig. 6. Closed modes

The relation $R^C \subseteq Q_V \times \mathcal{F}_V$ gives, for every data state of the mode, the set of flows from this state. By definition, if the control state of the mode is not at dx, the set of flows for the state is empty. R^C is obtained from the constraints of a mode and relations $SM.R^C$ of its submodes. Given a data state s of a mode M, $(s, f) \in R^C$ *iff* f satisfies the constraints of M and, if N is the active submode at s, (s, f), restricted to the global variables of N, belongs to $N.R^C$.

The relation $R^D_{e,x}$, for each entry point e and exit point x of a mode, comprises *macro-steps* of a mode starting at e and ending at x. A macro step consists of a sequence of *micro-steps*. Each micro-step is either a transition of the mode or a macro-step of one of its submodes. Given the relations $R^D_{e,x}$ of the submodes of M, a *micro-execution* of a mode M is a sequence of the form $(e_0, s_0), (e_1, s_1), \ldots, (e_n, s_n)$ such that every (e_i, s_i) is a state of M and for even i, $((e_i, s_i), (e_{i+1}, s_{i+1}))$ is a transition of M, while for odd i, (s_i, s_{i+1}) is a macro-step of one of the submodes of M. Given such a micro execution of M with $e_0 = e \in E$ and $e_n = x \in X$, we have $(s_0, s_n) \in R^D_{e,x}$. To illustrate the notion of macro-steps, consider the closed mode `DynamicController` from Figure 6. Considering only the control points, we have the following micro-execution when $u > 0$ and $u_c <$ *maxThrottle*: *init*, *ThrottleNormal.de*, *ThrottleNormal.dx*, dx. For every u, u_c satisfying the above inequalities this micro-execution gives us a macro-step in $R^D_{init,dx}$.

The *operational semantics* of the mode M consists of its control points $E \cup X$, its variables V and relations R^C and $R^D_{e,x}$. The operational semantics of a mode defines a transition system $\mathcal{R}$ over the states of the mode. We write $(e_1, s_1) \xrightarrow{o} (e_2, s_2)$ if $(s_1, s_2) \in R^D_{e_1,e_2}$, and $(dx, s_1) \xrightarrow{f} (dx, s_2)$ if $(s_1, f) \in R^C$, where f is defined on the interval $[0, t]$ and $f(t) = s_2$. We extend $\mathcal{R}$ to include *environment* steps. An environment step begins at an exit point of the mode and ends at an entry point. It represents changes to the global variables of the mode by other components while the mode is inactive. Private variables of the mode are unaffected by environment steps. Thus there is an environment step $(x, s) \xrightarrow{\varepsilon} (e, t)$ whenever $x \in X$, $e \in E$, and $s[V_p] = t[V_p]$. We let λ range over $\mathcal{F}_V \cup \{o, \varepsilon\}$. An *execution* of a mode is now a path through the graph of $\mathcal{R}$:

$$(e_0, s_0) \xrightarrow{\lambda_1} (e_1, s_1) \xrightarrow{\lambda_2} \ldots \xrightarrow{\lambda_n} (e_n, s_n).$$

Trace semantics. To be able to define a refinement relation between modes, we consider a trace semantics for modes. A *trace* of the mode is a projection of its executions onto the global variables of the mode. The *trace semantics* for M is given by its control points E and X, its global variables V_g, and its set of its traces L_M.

In defining compositional and hierarchical semantics, one has to decide, what details of the behavior of lower-level components are observable at higher levels. In our approach, the effect of a discrete step that updates only local variables of a mode is not observable by its environment, but stoppage of time introduced by such a step *is* observable. For example, consider two systems, one of which is always idle, while the other updates a local variable every second. These two systems are different, since the second one does not have flows more than one second long. Defining a modular semantics in a way that such distinction is not made seems much more difficult.

3.2 Trace Semantics for Agents

An execution of an agent follows a trajectory, which starts in one of the initial states and is a sequence of flows interleaved with discrete updates to the variables of the agent. An execution of A is constructed from the relations R^C and R^D of its top-level mode. For a fixed initial state s_0, each mode $M \in TM$ starts out in the state $(init_M, s_M)$, where $init_M$ is the non-default entry point of M and $s_0[M.V] = s_M$. Note that as long as there is a mode M whose control state is at $init_M$, no continuous steps are possible. However, any discrete step of such a mode will come from $R^D_{init_M,dx}$ and bring the control state of M to dx. Therefore, any execution of an agent $A = \langle TM, V, I \rangle$ with $|TM| = k$ will start with exactly k discrete initialization steps. At that point, every top-level mode of A will be at its default exit point, allowing an alternation of continuous steps from R^C and discrete steps from $R^D_{de,dx}$. The choice of a continuous step involving all modes or a discrete step in one of the modes is left to the environment. Before each discrete step, there is an environment step, which takes the control point of the chosen mode from dx to de and leaves all the private variables of all top-level modes intact. After that, a discrete step of the chosen mode happens, bringing control back to dx. Thus, an execution of A with $|TM| = k$ is a sequence $s_0 \xrightarrow{o} s_1 \xrightarrow{o} \ldots s_k \xrightarrow{\lambda_1} s_{k+1} \xrightarrow{\lambda_2} \ldots$ such that

- The first k steps are discrete and initialize the top-level modes of A.
- for every $i \geq k$, one of the following holds:
  - the i^{th} step is a continuous step, in which every mode takes part, or
  - the i^{th} step is a discrete environment step, or
  - the i^{th} step is a discrete step by one of the modes and the private variables of all other modes are unchanged.

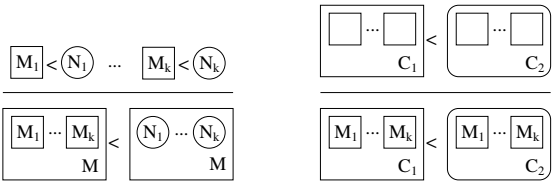


Fig. 7. Compositionality rules for modes

Note that environment steps in agents and in modes are different. In an agent, an environment step may contain only discrete steps, since all agents participate in every continuous step. The environment of a mode can engage in a number of continuous steps while the mode is inactive.

A trace of an agent A is an execution of A, projected onto the set of its global variables. The denotational semantics of an agent consists of its set of global variables V_g and its set of traces L_A.

Trace semantics for modes and agents can be related to each other in an obvious way. Given an atomic agent A whose behavior is given by a mode M, we can obtain a trace of A by taking a trace of M and erasing the information about the control points from it.

3.3 Compositionality Results

We show that our semantics is compositional for both modes and agents. First, the set of traces of a mode can be computed from the definition of the mode itself and the semantics of its submodes. Second, the set of traces of a composite agent can be computed from the semantics of its sub-agents.

Mode Refinement. The trace semantics leads to a natural notion of refinement between modes: a mode M refines N if it has the same global variables and control points, and every trace of M is a trace of N. A mode M and a mode N are said to be *compatible* if $M.V_g = N.V_g$, $M.E{=}N.E$ and $M.X{=}N.X$. Given two compatible modes M and N, M refines N, denoted $M{\preceq}N$, if $L_M{\subseteq}L_N$.

The refinement operator is compositional with respect to the encapsulation. If, for each submode N_i of M there is a mode N_i' such that $N_i \preceq N_i'$, then we have that $M \preceq M'$, where M' is obtained from M by replacing every N_i with N_i'. The refinement rule is explained visually in Figure 7, left.

A second refinement rule is defined for contexts of modes. Informally, if we consider a submode N within a mode M, the remaining submodes of M and the transitions of M can be viewed as an environment or *mode context* for N.

As with modes, refinement of contexts is also defined by language inclusion and is also compositional. If a context C_1 refines another context C_2, then inserting modes $M_1, \ldots, M_k$ into the two contexts preserves the refinement property. A visual representation of this rule is shown in Figure 7, right. Precise statements of the results can be found in [4].

Compositionality of agents. An agent is, in essence, a set of top level modes that interleave their discrete transitions and synchronize their flows. The

compositionality results for modes lift in a natural way to agents too. The operations on agents are compositional with respect to refinement. An agent A and an agent B are said to be *compatible* if $A.V_g = B.V_g$. Agent A refines a compatible agent B, denoted $A \preceq B$, if $L_A \subseteq L_B$. Given compatible agents such that $A \preceq B, A_1 \preceq B_1$ and $A_2 \preceq B_2$, let $V_1 = \{x_1, \ldots, x_n\}, V_2 = \{y_1, \ldots, y_n\}$ be indexed sets of variables with $V_1 \subseteq A.V$ and let $V_h \subseteq A.V$. Then $A \backslash \{V_h\} \preceq B \backslash \{V_h\}, A[V_1 := V_2] \preceq B[V_1 := V_2]$ and $A_1 || A_2 \preceq B_1 || B_2$

4 The CHARON Toolkit

In this section we describe the CHARON toolkit. Written in Java, the toolkit features an easy-to use graphical user interface, with support for syntax-directed text editing, a visual input language, a powerful type-checker, simulation and a plotter to display simulation traces. The CHARON GUI uses some components from the model checker JMOCHA [3], and the plotter uses a package from the modeling tool PTOLEMY [12].

The editor windows highlight the CHARON language keywords and comments. *Parsing on the fly* can be enabled or disabled. In case of an error while typing, the first erroneous token will be highlighted in red. Further, a pop up window can be enabled that tells the user what the editor expects next. Clicking one of the pop up options, the associated text is automatically inserted at the current cursor position. This allows the user not only to correct almost all syntactic errors at typing but also to learn the CHARON language.

The CHARON toolkit also includes a visual input language capability. It allows the user to draw agent and mode definitions in a hierarchical way, as shown in Figures 1 – 5. The interpreter of the visual input translates the specification into text-based CHARON source code using an intermediate XML-based representation. The visual input tool is depicted in Figure 8.

Once an edited and saved CHARON language file exists, the user can simulate the hybrid system. In this case the CHARON toolkit calls the parser and the type checker. If there are no syntactic errors, it generates a *project context* that is displayed in a separate project window that appears on the left hand side of the desktop, as shown in Figure 9.

The project window displays the internal representation of CHARON in a convenient tree format. Each node in the tree may be expanded or collapsed by clicking it. The internal representation tree consists of two nodes: `agents` and `modes`. They are initially collected from the associated CHARON file.

A CHARON specification describes how a hybrid system behaves over the course of time. CHARON's simulator provides a means to visualize a possible behavior of the system. This information can be used for debugging or simply for understanding in detail the behavior of the given hybrid system description.

The simulation methodology used in the CHARON toolkit, which is depicted in Figure 10, resembles concepts in code generation from a specification. As CHARON allows to write external Java source code the simulator needs to be an executable Java program. CHARON has a set of Java files that represent a core

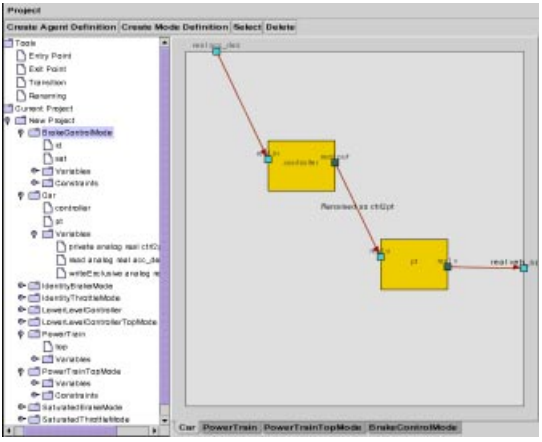


Fig. 8. The visual input tool of CHARON: The agent car is defined as the composition of two other agents. The arrows depict variable renamings.

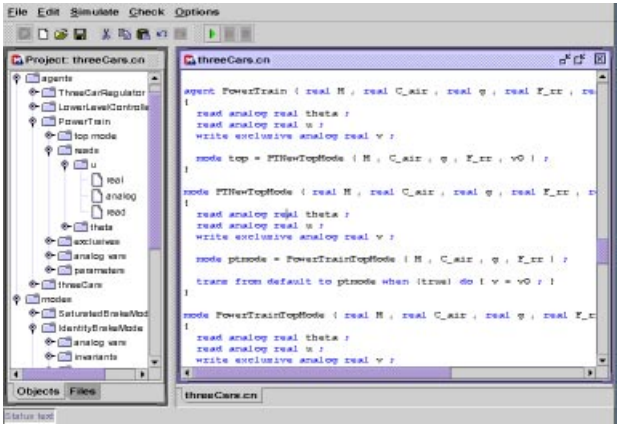


Fig. 9. The editor frame on the right hand side of the CHARON desktop and the corresponding project frame on the left

simulator. Given a CHARON file, Java files are automatically generated which represent a Java interpretation of the CHARON specification of a hybrid system. They are used in conjunction with the predefined simulator core files and the external Java source code to produce a simulation trace.

The CHARON plotter allows the visualization of a simulation trace generated by the simulator. It draws the value of all selected variables using various colors with respect to time. It also highlights the time that selected transitions have been taken. A screen-shot of the plotter is given in Figure 11.

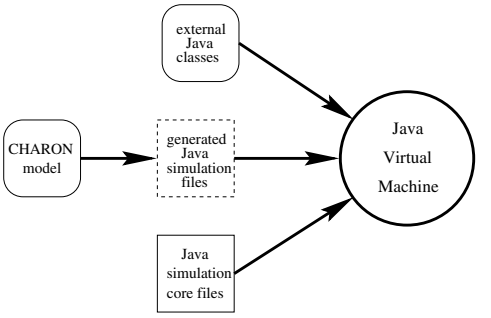


Fig. 10. The simulation methodology of CHARON

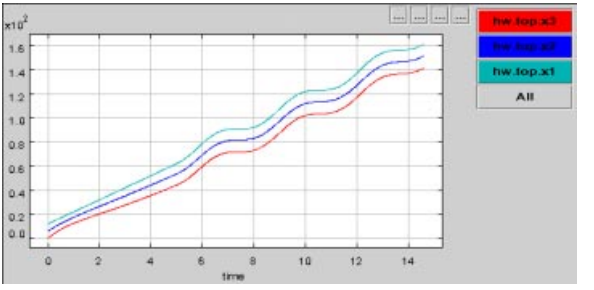


Fig. 11. A plot of a simulation trace of a three car vehicle-to-vehicle coordination model. The three graphs represent the x-coordinates of the three respective cars.

More information on the CHARON toolkit, along with a preliminary release, is available at `www.cis.upenn.edu/mobies/charon/`.

5 Research in Formal Analysis

Since CHARON models have a precise semantics, they can be subjected to a variety of analyses. In this final section we will give a brief overview of our ongoing research efforts in formal analysis methods for hybrid systems. These include new techniques for accurate and efficient simulation, reachability analysis to detect violations of safety requirements, and abstraction methods for enhancing the applicability of analysis techniques.

Accurate event detection. The problem of accurately detecting and localizing the occurrence of transitions when simulating hybrid systems has received an increased amount of attention in recent years. This is partially motivated by the observation that the commonly used technique of simply checking the value of the guards at integration points can cause the simulator not to detect enabling of transitions. It has been shown that such inaccuracies can lead to

grossly inaccurate simulations due to the discontinuous nature of hybrid systems. We have developed a method [15] which is guaranteed to detect enabling of all transitions. Our method has the advantage of being the only method which can properly detect such events when they occur in the neighborhood of model singularities. We select our step size in such a way as to steer the system toward the event surface without overshooting it.

Multirate simulation. Many systems, especially hierarchical ones, naturally evolve on different time scales. For example the center of mass of an automobile may be accelerating relatively slow compared to the rate at which the crank shaft angle changes; yet, the evolution of the two are intimately coupled. Despite this disparity, traditional numerical integration methods force all coupled differential equations to be integrated using the same step size. The idea behind multi-rate integration method is to use larger step sizes for the slow changing sets of differential equations and smaller step sizes for the differential equations evolving on the fast time scale. Such a strategy increases efficiency without compromising accuracy. To implement such a schme we need to show how to accommodate coupling between the sets of fast and slow equations when they are integrated asynchronously and how to schedule the order of integration. In [14] we resolve these issues and introduce a multirate algorithm well suited to hybrid system simulation.

Distributed Simulation. Another way for simulation speed-up is to utilize more computing resources in a multi-processing platform by exploiting the inherent modularity of systems described in CHARON. Each agent of CHARON has different dynamics and thus does not have to be integrated at the same speed. The challenge is to determine an effective scheme for communication of the values of the shared variables among the various agents simulated on different processing units.

Requiem for reachability analysis. Formal verification of safety requirements of hybrid systems requires the computation of reachable states of continuous systems. Requiem is a Mathematica notebook which, given a nilpotent linear differential equation and a set of initial conditions, symbolically computes the set of reachable states exactly. Given various classes of linear differential equations and semi-algebraic sets of initial conditions, the computation of reachable sets can be posed as a quantifier elimination problem in the decidable theory of the reals as an ordered field. Given a nilpotent system and a set defined by polynomial inequalities, Requiem automatically generates the quantifier elimination problem and invokes the quantifier elimination package in Mathematica 4.0. If the computation terminates, it returns a quantifier free formula describing the reachable set. More details can be found in [21]. The entire package is available for free at `www.seas.upenn.edu/hybrid/requiem.html`.

Predicate Abstraction. Abstraction is emerging as the key to formal verification as a means of reducing the size of the system to be analyzed [11]. The main obstacle towards an effective application of model checking to hybrid systems is the complexity of the reachability procedures which require expensive computations over sets of states. For analysis purposes, it is often useful to ab-

stract a system in a way that preserves the properties being analyzed while hiding the details that are of no interest [5]. We build upon notion of predicate abstraction [28] for formal analysis of hybrid systems. Using a set of boolean predicates, that are crucial with respect to the property to be verified, we construct a finite partition of the state space of the hybrid system. The states of the abstracted system correspond to truth assignments to the set of predicates. By using conservative reachability approximations we guarantee that if the property holds in the abstracted system, then it also holds in the concrete system represented by the hybrid system. Conversely, if the property does not hold in the abstract system, then it may or may not hold in the concrete system. In the latter case, the concrete system will be checked against the counter-example found during the model checking of the abstract system. If a trace corresponding to the counter-example is feasible in the concrete system we have established the property to be false. This procedure can also help discover new predicates that can be used to refine the abstraction, as it is suggested in [9] for analysis of discrete systems. The combination of this method for finding new predicates and our abstraction procedure thus provides an effective way to apply a finite state model checking approach to hybrid systems.

Multi-robot coordination. We develop a hybrid system framework and the software architecture for the deployment of multiple autonomous robots in an unstructured and unknown environment with applications ranging from scouting and reconnaissance, to search and rescue and manipulation tasks. Our software framework allows a modular and hierarchical approach to programming deliberative and reactive behaviors in autonomous operation. Formal definitions for sequential composition, hierarchical composition, and parallel composition allow the bottom-up development of complex software systems. We demonstrate the algorithms and software on an experimental testbed that involves a group of car-like robots using a single omni-directional camera as a sensor without explicit use of odometry. More information can be found in [2,16].

References

1. R. Alur, C. Courcoubetis, N. Halbwachs, T.A. Henzinger, P. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138:3–34, 1995.
2. R. Alur, A. Das, J. Esposito, R. Fierro, Y. Hur, G. Grudic, V. Kumar, I. Lee, J. P. Ostrowski, G. Pappas, J. Southall, J. Spletzer, and C. J. Taylor. A framework and architecture for multirobot coordination. In *Proc. ISER00, Seventh Intl. Symp. on Experimental Robotics*, pages 289–299, 2000.
3. R. Alur, L. de Alfaro, R. Grosu, T.A. Henzinger, M. Kang, R. Majumdar, F. Mang, C.M. Kirsch, and B.Y. Wang. MOCHA: A model checking tool that exploits design structure. In *Proc. 23rd Intl. Conf. on Software Engineering*, pages 835–836, 2001.
4. R. Alur, R. Grosu, I. Lee, and O. Sokolsky. Compositional refinement for hierarchical hybrid systems. In *Hybrid Systems : Computation and Control*, LNCS 2034, pages 33–48, 2001.
5. R. Alur, T. Henzinger, G. Lafferriere, and G. Pappas. Discrete abstractions of hybrid systems. *Proceedings of the IEEE*, 88(7):971–984, July 2000.

6. A. Balluchi, L. Benvenuti, M. Di Benedetto, C. Pinello, and A. Sangiovanni-Vicentelli. Automotive engine control and hybrid systems: Challenges and opportunities. *Proceedings of the IEEE*, 88(7):888–912, July 2000.
7. G. Booch, I. Jacobson, and J. Rumbaugh. *Unified Modeling Language User Guide.* Addison Wesley, 1997.
8. A. Chutinan and B. Krogh. Verification of polyhedral-invariant hybrid automata using polygonal flow pipe approximations. In *Hybrid Systems : Computation and Control*, LNCS 1569, 1999.
9. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, pages 154–169, 2000.
10. E.M. Clarke and R.P. Kurshan. Computer-aided verification. *IEEE Spectrum*, 33(6):61–67, 1996.
11. S. Das, D. Dill, and S. Park. Experience with predicate abstraction. In *Computer Aided Verification, 11th Intl. Conf.*, LNCS 1633, pages 160–171, 1999.
12. J. Davis, M. Goel, C. Hylands, B. Kienhuis, E.A. Lee, J. Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Reekie, N. Smyth, J. Tsay, and Y. Xiong. Overview of the Ptolemy project. Technical Report UCB/ERL M99/37, 1999.
13. S. Engell, S. Kowalewski, C. Schulz, and O. Stursberg. Continuous-discrete interactions in chemical processing plants. *Proc. of the IEEE*, 88(7):1050–1068, 2000.
14. J. Esposito and V. Kumar. Efficient dynamic simulation of robotic systems with hierarchy. In *Intl. Conf. on Robotics and Automation*, pages 2818–2823, 2001.
15. J. Esposito, V. Kumar, and G. Pappas. Accurate event detection for simulating hybrid systems. In *Hybrid Systems : Computation and Control*, LNCS 2034, pages 204–217, 2001.
16. R. Fierro, A. Das, V. Kumar, and J. P. Ostrowski. Hybrid control of formations of robots. *Proc. Int. Conf. Robot. Automat.*, pages 157–162, 2001.
17. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.
18. T.A. Henzinger, P. Ho, and H. Wong-Toi. HyTech: the next generation. In *Proc. TACAS'95*, LNCS 1019, pages 41–71, 1995.
19. C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.
20. G.J. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, 23(5):279–295, 1997.
21. G. Lafferriere, G. Pappas, and S. Yovine. Symbolic reachability computation for families of linear vector fields. *Journal of Symbolic Computation*, 2001.
22. K. Larsen, P. Pettersson, and W. Yi. Uppaal in a nutshell. *Springer Intl. Journal of Software Tools for Technology Transfer*, 1, 1997.
23. E.A. Lee. What's ahead for embedded software. *IEEE Computer*, pages 18–26, September 2000.
24. N. Lynch, R. Segala, F. Vaandrager, and H. Weinberg. Hybrid I/O automata. In *Hybrid Systems III: Verification and Control*, LNCS 1066, pages 496–510, 1996.
25. O. Maler, Z. Manna, and A. Pnueli. From timed to hybrid systems. In *Real-Time: Theory in Practice, REX Workshop*, LNCS 600, pages 447–484, 1991.
26. K. McMillan. *Symbolic model checking: an approach to the state explosion problem.* Kluwer Academic Publishers, 1993.
27. R. Milner. *A Calculus of Communicating Systems.* LNCS 92, 1980.
28. S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *Proc. 9th Intl. Conf. on Computer Aided Verification*, LNCS 1254, 1997.
29. C. Tomlin, G. Pappas, and S. Sastry. Conflict resolution for air traffic management: A study in multi-agent hybrid systems. *IEEE Trans. Automatic Control*, 43(4):509–521, 1998.

Some Synchronization Issues When Designing Embedded Systems from Components

Albert Benveniste[1]

Inria / Irisa, Campus de Beaulieu, 35042 Rennes, France ;
Albert.Benveniste@inria.fr.
http://www.irisa.fr/sigma2/benveniste/home.html

Abstract. This paper is sort of a confession. Issues of synchrony, asynchrony, and synchronization, arise frequently in designing embedded systems from components, like everyone I knew this for quite a long time. But it is only recently that it went aware of the *diversity* of such issues, depending on the context. The aim of this paper is to show and explain this diversity by looking at three areas where systems design is of interest, namely : 1/ building software or hardware architectures composed of components interacting asynchronously, 2/ synchronous hardware design from IP's, and 3/ designing distributed real-time control systems. A large part of this paper relies on other people's work, I indicate appropriate references in each case.

1 Think Synchronously, Act Asynchronously

This is a widely used engineering practice in several important areas where embedded systems are designed.

Real-time control systems such as encountered in automobiles or even more in aeronautics, interact continuously with a physical system for the purpose of achieving some desired behaviour. The underlying physical system for control has its own dynamics. This dynamics is typically approximately modeled by some partial differential equation, or, more approximately, by some system of differential equations. As examples, think of car or aircraft engine combustion, car of aircraft dynamics, and so on. The important fact is that this dynamics is global and bound to physical time and space.

Analog control design consists in building another dynamical system to be put in closed loop with the physical one, in order to achieve a desired response to both external stimuli from the operator (the requests to be satisfied), and disturbances from the environment (against which robustness is required). Computer control, which has replaced the traditional analog control, aims at providing the same service, with a drastic improvement in flexibility and intelligence, but at the price of digitalization.

In doing his job, the designer handles global state and time, he thinks globally. Said in computer science terms, *the designer thinks "synchronously"*. The important fact to know about control engineering practice is that *the designer*

T.A. Henzinger and C.M. Kirsch (Eds.): EMSOFT 2001, LNCS 2211, pp. 32–49, 2001.
© Springer-Verlag Berlin Heidelberg 2001

is aware that the model he uses is only approximate. And this approximation occurs in both *time* and *space*:

- in *time*: sensing devices are only approximately synchronized, applying the computed control takes time, in other words, the (distributed) computer system acts with some (limited) asynchrony;
- in *space*: measurements are noisy, disturbances from the environment can be poorly known, models used are approximate (sometimes very coarsely so).

Until quite recently, the second problem (approximation in space) has overcomed the first one. The reason is that continuous control and regulation was the dominating task, for which approximation in time was solved using well established concepts from control such as that of *phase margin*[1]. However, real-time control systems have become much more complex by involving mode changes, protection, and more generally supervision and human-machine interaction. This results in a complex combination of continuous and discrete control, popularized recently as *hybrid* systems.

It turns out that the classical approach of control engineering to deal with approximation in time no longer suffices for the more complex embedded control systems. The lesson, first noticed and analyzed by Paul Caspi in [6], is that

> the synchrony/asynchrony issue, in distributed real-time embedded control systems, is motivated by the requested robustness against approximation in time.

In modern high performance hardware such as prevalent in embedded systems today, the combination of request for higher performance (small size, low power, high throughput and low latency), and low cost, calls for a design of systems from available components without paying (too much) for the corresponding overhead. Here we quote some arguments from L.P. Carloni, K.L. McMillan, and A.L. Sangiovanni-Vincentelli [5].

In hardware, while the number of gates reachable in a cycle will not change significantly and the on-chip bandwidth that wires provide will continue to grow, the percentage of the die reachable within one clock cycle will decrease dramatically: we will soon reach a point where more gates can be fit on a chip than can communicate in one clock cycle. In particular it has been predicted that for deep sub-micron designs a signal will need more than ten clock cycles to traverse the entire chip area. Also, it has been estimated that only a fraction of the chip area between 0.4% and 1.4% will be reachable in one clock cycle. This calls for a fine and efficient tuning of placement, routing, pipelining, and retiming. However, since precise data on wire-lengths are available only late in the design process, several costly re-designs become necessary to change the placement or the speed of the chip modules while satisfying performances and functionality

[1] When designing a PID feedback loop, the designer provides robust control by ensuring some *gain margin* (to be robust against uncertainty in the gain and damping of the systems poles), and some *phase margin* (to be robust against uncertainty in the phase of the systems poles, or, equivalently, to be robust against various delays that can result from both the sensoring and computer equipments).

constraints. Even worse, predefined Intellectual Properties (IP) cannot take into account such data, as these are inherently bound to the global system design.

Still, low cost design of hardware calls for a design methodology that maintains the inherent simplicity of synchronous design, and yet does not suffer from the "interconnect-delay problem". The idea is that, at early stage of the design, both IP's and the system can be regarded as completely synchronous, i.e., just as a set of modules that communicate by means of channels having "zero-delay", i.e., a delay neglectible with respect to the period of the common "virtual" clock signal. And it is wished that, at later stages of the design where actual real clocks are used, with precise frequency and phase, proper retiming, pipelining, and resynchronization be automatically taken care of. In this way, successive redesigns are performed at the early design stage, where the simple synchronous model is assumed, and costly manual re-design phases are avoided. Note that the resulting hardware is still synchronous, we do not consider here the more advanced "asynchronous" technologies, in which the global synchronization of the different clocks is not maintained throughout the die. To summarize,

> the synchrony/asynchrony issue, in synchronous hardware systems design from IP's, consists in providing assistance for the design of proper retiming, pipelining, and resynchronization, which result from the "interconnect-delay problem".

Design of software systems from components is the prevalent approach, e.g., in telecommunications systems today. The migration, from a network dominant architecture to a service dominant one, results in an increase in Quality of Service (QoS) requirements. QoS is more and more taken care of by processing at the edges of the network, not within its backbone. The increase in versatility of services calls for far more dynamical protocol suites and frequent updates or upgrades of protocols. Also, software radio is a central technology for the 4th generation telecommunications. Thanks to ultra wide band, high performance, A/D converters, used in the front end of receivers, entire communication protocols can migrate to a fully digital technology. This makes it possible to dynamically and adaptively change the protocols at a receiver station. In the whole, this trend calls for drastic improvements in network management middleware and corresponding software architectures, in order to facilitate the reuse of predefined components. The essence of this point of view is that the systems software is made of Java/C/C++ *distributed* components interacting asynchronously via threading mechanisms, OS primitives, or software bus architectures.

The explosion of resulting middleware typically results in a huge overhead at run time. One of the techniques envisioned to face this is the combination of reflexive architectures [10][11] and specialization of pre-defined generic components (both off-line and on-line) [12][13].

However alternative approaches can be considered. While the above fully asynchronous point of view is perfectly valid when the major issue is that of resource sharing by different components which otherwise do not interact, it is no longer efficient at handling software or SW/HW systems, implementing, e.g., software radio techniques. In the latter case, the different components (filter banks, channel/source codes, protocols, schedulers) must effectively cooperate

at delivering the overall expected function. Once more, the synchronous point of view can drastically simplify the early stages of the design, by providing easier specification, debugging and verification, and code optimization. Also, this point of view offers a smooth transition from a signal processing oriented style of specification, down to the system deployment.

The typical architecture is parallel and distributed (in the small), and involves a combination of DSP's, ASIC's, micro-processors, and communication buses. The resulting model of target architecture is Globally Asynchronous, Locally Synchronous (GALS). GALS architecture share some features with asynchronous ones: no global clock drives it, and no global state is available. For this and similar cases,

> the synchrony/asynchrony issue consists in providing assistance for a correct-by-construction migration from a synchronous specification to a GALS type of architecture; here, correct-by-construction means that the specified *logical* function is implemented, and some "soft" real-time *performance requirements* are satisfied.

The rest of this paper is devoted to discussing how the above three informal problems can be or have been formalized.

2 Distributed Real-Time Control Systems [6]

Here we must start from physics, since controlling a physical plant or device is, after all, the duty. Regardless of issues of digitalization and measurements, we shall consider the ideal case in which the physical system in consideration stays in closed loop with some analog feedback, and is subject to mode changes and reconfigurations. The adequate model for this is a *hybrid dynamical system.* One possible general model is the following:

$$\begin{cases} \dot{x} = f(x, v, t, q) \\ y = h(x, v, t, q) \end{cases} \tag{1}$$

$$\alpha \;=\; a(x, u) \tag{2}$$

$$\alpha \neq \alpha_- \;\Rightarrow\; T(q, \alpha, q') \tag{3}$$

Equations (1) constitute the continuous part of the model. It is a dynamical system having $t \in \mathbf{R}$ as real time, $v \in \mathbf{R}^m$ as disturbance from the environment, $x \in \mathbf{R}^n$ as continuous state, $y \in \mathbf{R}^p$ as continuous output, and $q \in Q$ belongs to some finite alphabet of discrete states, figuring the current mode — mode $q(t)$ is piecewise constant in time. Hence dynamics (1) is mode dependent. Equation (2) generates mode changes, it is driven by some control u (say, from the operator), and function a takes its values in some finite alphabet A of mode change requests. Thus α is a piecewise constant signal, which we regard as continuous from the right, i.e., $\alpha(t) = \lim_{s \searrow t} \alpha(s)$ holds, its left limits are denoted by $\alpha_-(t) := \lim_{s \nearrow t} \alpha(s)$, and a mode change request occurs when $\alpha(t) \neq \alpha_-(t)$. This mode change request results in a mode change according to formula (3), where $T(q, \alpha, q')$ is the transition relation of a finite state automaton.

Formulas (1,2,3) can be regarded as a generic model of hybrid component, and such models can be connected by either synchronizing their mode change automata, or connecting some continuous outputs of a component to corresponding continuous inputs in another component.

Now assuming that available techniques from modern robust control theory and engineering can deal with the continuous part, let us focus on the discrete part. It consists of the two formulas (2,3), actually the key equation for our discussion is (2).

This equation summarizes the transformation, from the pair (x, u) consisting of the states and stimuli received from the environment, to the mode change request signal α. Interpret equation (3) as the discrete control implemented within the computer, then equation (2) abstracts the complex process of 1/ performing distributed measurements, digitalization, and possibly thresholding, and 2/ collecting these partial informations through the communication bus to compute change mode request. This can be summarized as in figure 1. In this

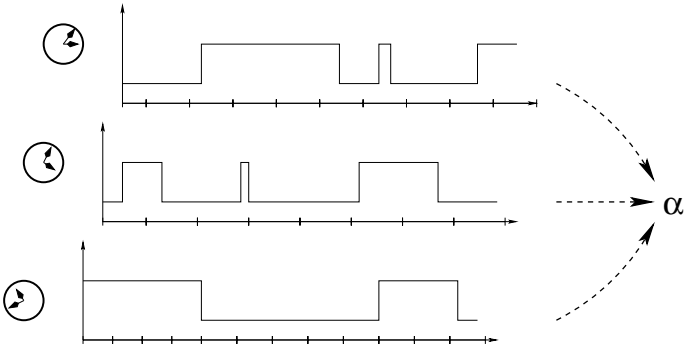


Fig. 1. Distributed sensing of logical signals, and their combination.

figure, we show three different piecewise constant signals, they result from some thresholding mechanism associated to the distributed sensors. Each sensor has its own independent clock to drive A/D conversion, and the clocks are loosely synchronized (their sampling frequencies are nearly the same, but their phases are different and subject to jitter). The phases include transmission delay from sensor to computer. Clearly, for a given tuple of analog sensor records, the combination of their different *sampled* versions may differ in general, due to slight deviations occurring within each clock and nondeterminism in communication delays. Hence the computation of α can yield different results, depending on the clock drifts and jitters, and communication delays. The request is that *the behaviour of the purely discrete part* (3) *should not depend on such variations* (by this we mean that the resulting sequence of successive discrete states should remain the same). Otherwise, spurious, unexpected, behaviours of the discrete controller may arise. And these spurious behaviours would not be encompassed

by the purely "synchronous" abstract model, and therefore could not be exhibited by model checking its single discrete part. The lesson is that

> robustness against uncertainties with respect to physical sampling time, communication delays, and timing uncertainties in RTOS primitives, is the very question in distributed real-time control systems.

We refer the reader to [7] for possible solutions to ensure such a robustness.

3 Synchronous Hardware Design from IP's [5]

To my mind, the approach I discuss here, which is due to [5], consists in adapting classical ideas from Leiserson's retiming techniques for hardware, to the more recent concept of IP. Let me briefly explain how I see this.

3.1 The Folklore of Retiming

A simple model of synchronous hardware consists of a data-flow graph $\mathcal{G}$, i.e., a DAG whose vertices figure variables, and branches figure dependencies. For u, v two variables, and k the integer time index:

- draw a branch $u \to v$ if v_k uses u_k for its computation, and
- draw a branch $u \stackrel{n}{\longrightarrow} v$ if the kth occurrence, v_k, of v uses the n-shifted version u_{k-n} of u, for its computation. Call integer index n the *weight* of the branch.

Note that $u \to v$ can be seen as a short hand for $u \stackrel{0}{\longrightarrow} v$. The weight of a path in the graph is simply the sum of the weights of its successive branches. A graph is called *well formed* if every circuit has a stricly positive weight. We consider only well formed graphs, for simplicity. Now the question is: *find modifications of $\mathcal{G}$ that will not change its semantics.* We claim that any combination of the following primitive changes has the desired property:

(a) Moving latches around. Let v be a variable. Denote by $u_1, \ldots, u_p$ the variables of $\mathcal{G}$ such that $u_i \stackrel{n_i}{\longrightarrow} v, i = 1, \ldots, p$ are branches of $\mathcal{G}$, and denote by $w_1, \ldots, w_q$ the variables of $\mathcal{G}$ such that $v \stackrel{n_j}{\longrightarrow} w_j, j = 1, \ldots, q$ are branches of $\mathcal{G}$. Associate to v a positive or negative integer $m \in \mathbf{Z}$. Using m, modify $\mathcal{G}$ by performing the following changes:

$$\begin{array}{l} u_i \stackrel{n_i}{\longrightarrow} v \mapsto u_i \stackrel{n_i - m}{\longrightarrow} v \\ v \stackrel{n_j}{\longrightarrow} w_j \mapsto v \stackrel{n_j + m}{\longrightarrow} w_j \end{array} \tag{4}$$

meaning that we pick m latches from each ingoing branch of v, and move them to each outgoing branch of it. Clearly, this shifts v by m ticks, but this does not change the map

$$(u_i, i = 1, \ldots, p) \mapsto (w_j, j = 1, \ldots, q)$$

We consider that shifting v in time is acceptable, as long as its sequence of successive values remains the same. Hence we regard transformation (4) as a semantic preserving transformation. Of course, for (4) to be feasible, we need that m satisfies the following condition (there are no "negative latches"):

$$\forall i = 1, \ldots, p \,:\, n_i - m \geq 0 \text{ and } \forall j = 1, \ldots, q \,:\, n_j + m \geq 0\,. \qquad (5)$$

One way of satisfying condition (5) is to provide input signals with additional latches. This only increases the overall latency of the circuit, but otherwise does not change its behaviour. If enough input latches are added, then satisfying (5) is eventually possible.

(b) Upsampling. The successive occurrences of variable v are denoted by $v_k, k = 1, 2, \ldots$ Now, pick $J > 1$ an integer, and set

$$v'_l := \textbf{ if } l = kJ \textbf{ then } v_k \textbf{ else } \tau\,,$$

where τ is a special value, called *non informative,* not belonging to any useful domain, τ is to be interpreted as a don't care. Ignoring don't cares in v' yields again v. If we perform this globally using the same integer J at all vertices of the data-flow graph $\mathcal{G}$, we preserve the semantics (just ignore don't cares, as said before).

Now, applying successively transformations of types (a) or (b) in any order, still preserves the semantics: the different signals are preserved, up to a shift, and ignoring don't cares yields again the original signals. Playing with this allows to adjust throughput and latency of the circuit. This is classical stuff, very useful indeed to synthesize pipelining.

Note that transformation (b) can be generalized by allowing a "variable upsampling": there is no need that informative values in a signal be separated by a fixed number J of don't cares. The only thing which matters is that don't cares should be inserted simultaneously, for every variable. Then ignoring don't cares yields again the original signals. Therefore, the upsampling can be, so to say, modified "on-line" while the circuit is running.

Well, assume we wish to implement the function specified by the original $\mathcal{G}$ (I mean prior to any transformation (a) or (b)), in the form of a hardware block being part of a larger circuit. Then, as argued in section 1, input wires may be shifted in time, due to "interconnect-delay problem". If this shift is fixed, then we can use transformations (a) and (b) to compensate for such a mismatch. However, it may be the case that this interconnect-delay can vary with time, just because routing can be data dependent, dynamically. Then, we need to resource to the generalized rule (b) to solve the problem.

This is fine if we have the environment of the considered block at hand. But we are still not done if such an information is not available. Unfortunately, this is precisely the case for IP's, since they are designed prior to their possible reuse, so their future environments are not known! The nice idea of [5] is *to design a priori the considered block in such a way that it adjusts automatically to any interconnect-delay, on-line.* This is called latency insensitive design, we briefly explain it hereafter.

3.2 The Models Used

There are basically two models of interest :

strictly synchronous : According to this model, there is some given finite set V of variables, and a *state* x assigns an effective value to each variable $v \in V$. A *strictly synchronous behaviour* is a sequence $s = x_1, x_2, \ldots$ of states. A *strictly synchronous process* is a set of strictly synchronous behaviours. A *strictly synchronous signal* is the sequence of values $s_v = v(x_1), v(x_2), \ldots$, for $v \in V$ given. Hence all signals are indexed by the same totally ordered set of integers $\mathbf{N} = \{1, 2, \ldots\}$ (or some finite prefix of it). Hence all behaviours are *synchronous* and are tagged by the same clock, this is why I use the term "strictly" synchronous. In practice, strictly synchronous processes are specified using a set of legal *strictly synchronous reactions* R, and therefore strictly synchronous processes take the form

$$P \;=\; R^{\omega}\,,$$

where superscript "$.^{\omega}$" denotes unbounded iteration. Composition is defined as the intersection of the set of behaviours, it is performed by taking the conjunction of reactions :

$$P \parallel P' \;:=\; P \cap P' = (R \wedge R')^{\omega}\,. \tag{6}$$

This is the classical mathematical framework used in synchronous hardware modelling. It raises problems when using it as a framework for IP design, as it does not comply with retiming.

synchronous : Here the model is the same as in the previous case, but every domain of data is enlarged with some *non-informative* value, denoted by the special symbol τ. A τ value is to be interpreted as a don't care. Values different from τ are called *informative,* since they correspond to actual, useful values for use in computing the outputs of the circuit. Besides this, things are as before : a *state* x assigns an informative or non-informative value to each state variable $v \in V$, where V is some given finite set of variables. A *synchronous behaviour* is a sequence of states. A *synchronous process* is a set of synchronous behaviours. A *synchronous signal* is the sequence of informative or non-informative values $s_v = v(x_1), v(x_2), \ldots$, for $v \in V$ given. And composition is performed as in (6). Hence, strictly synchronous processes are just synchronous processes involving only informative values.

3.3 The Problems

The idea is that we wish to implement a strictly synchronous specification P by means of a synchronous process P^{ℓ}, insensitive to latency. Then P^{ℓ} is the version of P for use as an IP block. For this the following problems need to be addressed.

Problem 1 : how to model that a synchronous process P^ℓ implements a strictly synchronous specification P, while being insensitive to latency ? The picture is that valuations of variables travel on wires of the design, and this causes latency and may require pipelining. Such latency may take several clock cycles, and can differ for different variables (since different wires are used). How to reflect this in the model ? For $v \in V$, pick some signal

$$s_v = v(x_1), v(x_2), v(x_3), \ldots \in P\,. \tag{7}$$

To reflect a wire-dependent latency, the same signal, observed later on along a wire, has (for example) the form

$$s_v^\ell = \tau, v(x_1), \tau, v(x_2), \tau, \tau, v(x_3), \ldots \tag{8}$$

i.e., τ symbols can be inserted at arbitrary places of the original signal s_v. This is the mechanism of *stalling* a signal. Conversely, for s_v^ℓ as in (8), its *strict* counterpart is given in (7). For v a variable, denote by

$$\chi_v : s_v^\ell \longmapsto s_v$$

the map giving the strict version of a stalled signal, and denote by χ_V the tuple composed of the χ_v where v ranges over the considered set V of variables. For P^ℓ a process having V as set of variables, its image $\chi_V(P^\ell)$ by χ_V is called the *strict process* corresponding to P^ℓ.

A process P^ℓ is called *patient* iff it satisfies the following : for all $s \in P^\ell$, all input signal s_i of s, and all instant k, there exists another behaviour $stall(s) \in P^\ell$, whose i-signal coincides with s_i before instant k, has a τ-event at k, and can be further stalled after k. It is clear (but tedious verifying) that the i/o-connection of patient processes is patient. The intuition is that a patient process can still offer the desired reaction to an input event if this input is delayed.

Let us formalize this some more. A *singleton buffer* is any process which possesses two variables v_i, v_o (standing for "input" and "output"), and has the identity process $s_{v_o} := s_{v_i}$ as corresponding strict process [2]. A *buffer* is simply the parallel composition of finitely many singleton buffers involving disjoint sets of variables (i.e., having no interaction). It has V_i, V_o, two copies of the same set V, as input and output variables ; for each $v \in V$, the pair (v_i, v_o) is related via a singleton buffer. In the following theorem, P^ℓ and Q^ℓ are two processes having disjoint sets of variables, communicating by means of a buffer having V_i, V_o as sets of input and output variables (V_i is shared by P^ℓ and the buffer, and V_o is shared by the buffer and P^ℓ).

Theorem 1 ([5]). *If P^ℓ and Q^ℓ are* patient *processes, and B, B' are two buffers as stated before, then*

$$\chi_V(P^\ell \,\|\, B \,\|\, Q^\ell) = \chi_V(P^\ell \,\|\, B' \,\|\, Q^\ell) \;=\; P \,\|\, Q\,,$$

where P, Q are the strict processes corresponding to P^ℓ and Q^ℓ.

Setting for instance $B' = Id$ (the identity buffer), theorem 1 implies in particular that inserting a buffer does not change the corresponding strict process.

[2] Here I simplify by accepting unbounded buffers. A more precise analysis is performed in [5].

Problem 2: for P a strict process, how to build a *patient* process P^ℓ such that $\chi_V(P^\ell) = P$? Here I deviate from [5] and rather propose a solution formulated as an extension of the classical technique of subsection 3.1. The idea is to interleave communication rounds and adaptive retiming, and to perform actual computations only when corresponding functions are provided with informative values for all their input variables. We detail this next.

Our starting point is the data-flow graph $\mathcal{G}$ serving as our original specification. Hence $\mathcal{G}$ implements a strict process and is supposed to process only informative values, it is defined as in subsection 3.1. To encompass the interaction with the environment, we enlarge $\mathcal{G}$ with additional branches of the form

$$\longrightarrow u \tag{9}$$

where $u \in V^{\mathrm{i}}$ (the set of input variables), and the missing vertex at the origin of the branch refers to the environment.

To each variable v of $\mathcal{G}$ we associate a weight as in (4), we denote it by m_v. The tuple of all m_v, for v ranging over V, is denoted by m_V. The moving of latches is equivalently encoded by our set of weights m_V, and we must restrict ourselves to sets of weights satisfying the feasibility condition (5). As an initial value for m_V we set

$$\forall v \in V : \; m_v := 0\,. \tag{10}$$

Hence our original data structure to model the circuit is the pair $(\mathcal{G}, \mathbf{0})$ consisting of $\mathcal{G}$ together with the zero weighting (10). Then the pair $(\mathcal{G}, m_V)$ is updated on-line, at each reaction, according to the following protocol.

As a simple case, suppose that all input wires of $\mathcal{G}$ receive informative values for the first round. Then the reaction can proceed as specificed by $\mathcal{G}$ directly, and the circuit can wait for a second set of input values. This was the trivial case. The interesting case is of course when at least one input wire offers a non-informative value τ for the first reaction. Assume this occurs exactly for one single $u \in V^{\mathrm{i}}$. What can we do?

The key idea is to model the reception of a non informative value on input wire u via the insertion of a *negative* delay in the corresponding input branch of $\mathcal{G}$:

$$\text{update } \mathcal{G} \;:\; [\,\overset{0}{\longrightarrow} u\,] \shortmid\!\longrightarrow [\,\overset{-1}{\longrightarrow} u\,] \tag{11}$$

Of course this negative delay can be equivalently implemented by buffering all informative values received at the other input wires, for proper resynchronization at the next reaction with the considered late input wire u.

But we can be more clever indeed. Denote by

$$u^{\rightarrow}$$

the subset of the variables $v \in V$ which instantaneously depend on input u, i.e., there exists a path from u to v in $\mathcal{G}$ having zero weight. Then we update the set of weights m_V as follows:

$$\begin{array}{l} \forall v \in u^{\rightarrow} \;:\; m_v := m_v - 1 \\ \forall v \notin u^{\rightarrow} \;:\; m_v := m_v \end{array} \tag{12}$$

Using the weights (12) we apply rule (4) of retiming. Having $m_v = -1$ for $v \in u^{\rightarrow}$ results in: 1/ assuming the availability of one latch at the output wires belonging to $u^{\rightarrow}$, and, 2/ moving these latches backward until a variable not belonging to $u^{\rightarrow}$ is reached. In passing we have compensated the negative delay in front of u by a positive one, therefore making the whole synchronization correct. As a result, variables not belonging to $u^{\rightarrow}$ are evaluated immediately, and the evaluation of other variables is delayed, hence τ values are not effectively processed. This results in an eager evaluation scheme implementing the original, strict, specification.

Generalizing this idea results in the following protocol for updating on-line the pair $(\mathcal{G}, m_V)$ at each reaction, where notation $s_u(x)$ denotes the value of input variable u for the considered behaviour, at the current reaction:

$$\text{update } \mathcal{G}: \quad s_u(x) = \tau \;\Rightarrow\; [\xrightarrow{n} u] \shortmid\longrightarrow [\xrightarrow{n-1} u] \tag{13}$$

$$\text{update } m_V: \begin{cases} \forall v \in U_\tau^{\rightarrow} \;:\; m_v := m_v - 1 \\ \forall v \notin U_\tau^{\rightarrow} \;:\; m_v := m_v \\ \quad \text{where} \;:\; U_\tau^{\rightarrow} = \bigcup_{u\,:\,s_u(x)=\tau} u^{\rightarrow} \end{cases} \tag{14}$$

One particular case of interest for this protocol is that of a *stuttering* or *stalling* reaction, in which all input wires receive a non informative value. Then a negative delay is added at each input wire, which is compensated by backward propagating, from each output wire, a (positive) delay. Of course we better do nothing and simply skip this reaction! In general, this protocol is applied by suitable adaptive routing of the data through a predefined set of latches. This can be done at the chosen level of granularity (here I assume the finest level, namely one node of the graph $\mathcal{G}$ per function, but *one node per IP* is probably a more realistic choice, further analyzed in [5]).

Comparing this solution to the one proposed in [5], our adaptive retiming protocol corresponds to the *equalizers* of [5], and the use of stuttering or stalling reactions correspond to the *stallable* processes of [5]. The latter study is more precise, however, since it takes into account the request of resourcing only to a bounded set of latches.

4 Building Software or Hardware GALS Architectures [2][3][4]

Here we discuss a different situation. We wish to assist the mapping from synchronous specifications to GALS implementations. Here we insist on *logical* correctness, meaning that each synchronous component should have exactly the *same* set of behaviours, be it composed synchronously (as in the specification) or *a*synchronously with the other synchronous components (as in the GALS implementation architecture). As said before, logical correctness is the driving requirement, real-time requirements are soft ones, and are generally ancillary.

Thus the requirements share some similarity with those of section 3. But there are significant differences: our GALS model is not "globally ticked", and does not have a global state. The problem considered in this section is in fact an extension of the one considered in section 3.

4.1 The Models Used

Here we consider two *different* models:

synchronous : Processes progress via an infinite sequence of *reactions*:

$$P \;=\; R^{\omega}$$

where R denotes the family of possible *reactions,* a reaction is a transition relation relating successive states. Within a state, some events can be *absent,* modelled by the occurrence of a special, non informative symbol τ, also denoted by $\perp$ following the notations of [2][3][4]. And the control can use the absence of these events as a viable information. Parallel composition is given by taking the pairwise conjunction of associated reactions, whenever they are composable:

$$P_1 \| P_2 \;=\; (R_1 \wedge R_2)^{\omega}$$

This results in having

$$P_1 \| P_2 = P_1 \cap P_2$$

This model coincides with the synchronous model of section 3.

***a*synchronous** : Reactions cannot be observed any more, no global clock exists. Instead a *behaviour* is a tuple of *signals,* and each individual signal is a totally ordered sequence of (informative) events. A process P^a is a set of behaviours. "Absence" cannot be sensed, and has therefore no meaning. Composition occurs by means of unifying each individual signal shared between two processes:

$$P_1^a \|_a P_2^a := P_1^a \cap P_2^a$$

This models in particular the communications via asynchronous unbounded FIFOs. More generally, a reliable communication medium is assumed, in which messages are not lost, and for each individual channel, messages are sent and received in the same order, but no global synchronization is assumed.

In our study we consider in particular the following two different architectures:

synchronous architecture : see above. We use it for the specification stage.

GALS **architecture** : a network of synchronous processes, interconnected by asynchronous communications (as defined above). We use it as deployment architecture.

And the central issue is: *what do we preserve when deploying a synchronous specification on a* GALS *architecture?*

4.2 The Fundamental Problems

Before stating the problems, let me say that [2][3][4] assumes that the considered synchronous processes are *stuttering invariant,* meaning that in each state x it is always possible to keep "silent", by not changing the state and not emitting

anything (i.e. emitted variables have the value $\perp$). Stuttering invariance is a key notion when considering open systems, since it is always possible, for a system, to be subsequently combined with another one which will be active while the considered one stutters. Note that suttering invariance has some similarity with "stallability" introduced in [5] and discussed in section 4.1.

Problem 1: what if a synchronous block receives its data from an asynchronous environment? Focus on a synchronous block within a GALS architecture, it receives its inputs as a tuple of (non synchronized) signals. Since some events can be absent in a given state, it can be the case that some signal will not be involved in a given reaction. But since the environment is asynchronous, this information is not provided by the environment. In other words, the environment does not offer to the synchronous block the correct model for its input stimuli. In general this will drastically affect the semantics of the block. However, some particular synchronous blocks are robust against this type of difficulty. How to formalize this?

Let P be such a block. Let $\sigma = x_0, x_1, x_2, \ldots$ be a behaviour of P, i.e., a sequence of states compliant with the reactions of P. Let V be the (finite) set of state variables of P, each state is a valuation of all $v \in V$, including the possible special status *absent,* which is denoted by the special symbol $\perp$ in [2][3][4], the valuation of v at state x is written $v(x)$. Hence we can write equivalently

$$\begin{aligned} \sigma &= (v(x_0))_{v\in V}, (v(x_1))_{v\in V}, (v(x_2))_{v\in V}, \ldots \\ &= (v(x_0), v(x_1), v(x_2), \ldots)_{v\in V} \\ &=_{\text{def}} (\sigma_v)_{v\in V} \end{aligned}$$

Now, for each separate v, remove the $\perp$ from the sequence $\sigma_v = v(x_0), v(x_1), v(x_2), \ldots$, this yields a (strict) *signal*

$$s_v \ =_{\text{def}} \ s_v(0), s_v(1), s_v(2), \ldots$$

where $s_v(0)$ is the first non $\perp$ term in σ_v and so on. Finally we set

$$\sigma^a \ =_{\text{def}} \ (s_v)_{v\in V}$$

The so defined map $\sigma \mapsto \sigma^a$ takes a synchronous behaviour, and returns a uniquely defined asynchronous one. This results in a map

$$P \longmapsto P^a$$

defining the *desynchronization* P^a, of P. Clearly, the map $\sigma \mapsto \sigma^a$ is not one-to-one, and thus it is not invertible. However, we have shown in [2][3][4] the first fundamental result that

$$\begin{aligned} &\text{if } P \text{ satisfies a special condition called } \textit{endochrony,} \text{ then} \\ &\forall \sigma^a \in P^a \text{ there exists a unique } \sigma \in P \text{ such that } \sigma \mapsto \sigma^a \text{ holds!} \end{aligned} \tag{15}$$

This means that, by knowing the formula defining reaction R such that $P = R^\omega$, we can uniquely reconstruct a synchronous behaviour, from observing its desynchronized version. In addition, it is shown in [2][3][4] that this reconstruction

can be perfomed *on-line* meaning that each continuation of a prefix of σ^a yields a corresponding continuation for the corresponding prefix of σ. The important point about result (15) is that endochrony can be model checked on the reaction R defining the synchronous process P. Also,

$$\text{any } P \text{ can be given a } \textit{wrapper } W \text{ making } P \| W \text{ endochronous.} \tag{16}$$

How can we use (15) to solve problem 1 ? Let E be the model of the environment, it is an asynchronous process according to our above definition. Hence we need to formalize what it means having "P interacting with E" since they do not belong to the same world. The only possible formal meaning is

$$P^a \parallel_a E$$

Hence having P^a interacting with E results in an asynchronous behaviour $\sigma^a \in P^a$, but using (15) we can reconstruct uniquely its synchronous counterpart $\sigma \in P$. So this solves problem 1.

However considering problem 1 is not enough, since it only deals with a single synchronous block interacting with its asynchronous environment, it does not consider the problem of mapping a synchronous network of synchronous blocks onto a GALS architecture.

Problem 2 : what if we deploy a synchronous network of synchronous blocks onto a GALS architecture ? Consider the simple case of a network of two blocks P and Q. Since our communication media behave like a set of FIFOs, one per each signal sent from one block to the other, we perfectly know what the desynchronized behaviours of our deployed system will be:

$$P^a \parallel_a Q^a$$

There is not need for inserting any particular explicit model for the communication medium, since by definition $\parallel_a$-communication preserve each individual asynchronous signal (but not their global synchronization!). In fact, Q^a will be the asynchronous environment for P^a and vice-versa.

Now, if P is endochronous, then, by having solved problem 1 we can uniquely recover a synchronous behaviour σ for P, from observing an asynchronous behaviour σ^a for P^a as produced by $P^a \parallel_a Q^a$.

But we are still not happy: it can be the case that there exist some asynchronous behaviour σ^a for P^a produced by $P^a \parallel_a Q^a$, which *cannot be obtained* by desynchronizing the synchronous behaviours of $P \parallel Q$. In fact we only know in general that

$$(P \parallel Q)^a \subseteq (P^a \parallel_a Q^a) \tag{17}$$

However, we have shown in [2][3][4] the second fundamental result that

$$\begin{array}{l}\text{if } (P, Q) \text{ satisfies a special condition called } \textit{isochrony,} \\ \text{then equality in (17) indeed holds !}\end{array} \tag{18}$$

The nice thing about isochrony is that it is *compositional*: if P_1, P_2, P_3 are pairwise isochronous, then $((P_1 \| P_2), P_3)$ is an isochronous pair, so we can refer to an *isochronous network* of synchronous processes. Also, isochrony enjoys additional useful properties listed in [2][3][4]. Again, the condition of isochrony can be model checked on the pair of reactions associated to the pair (P, Q), and

$$\begin{array}{l}\text{any pair } (P,Q) \text{ can be given } \textit{wrappers } (W_P, W_Q)\\ \text{making } (P\|W_P, Q\|W_Q) \text{ an isochronous pair.}\end{array} \tag{19}$$

Just a few additional words about the condition of isochrony, which is of interest per se. Synchronous composition $P\|Q$ is achieved by considering the conjunction

$$R_P \ \wedge \ R_Q$$

of corresponding reactions. In taking this conjunction of relations, we ask in particular that common variables have identical status present/absent in both components, in the considered reaction. Assume we relax this latter requirement by simply asking that the two reactions should only agree on effective values of common variables, *when they are both present.* This means that a given variable can be freely present in one component but absent in the other. This defines a "weakly synchronous" conjunction of reactions, we denote it by

$$R_P \ \wedge_a \ R_Q$$

In general, $R_P \ \wedge_a \ R_Q$ has more legal reactions than $R_P \ \wedge \ R_Q$. It turns out that the isochrony condition for the pair (P, Q) writes:

$$(R_P \ \wedge \ R_Q) \ \equiv \ (R_P \ \wedge_a \ R_Q)$$

4.3 A Sketch of the Resulting Methodology

How can we use (15) and (18) for a correct deployment on a GALS architecture? Well, consider a synchronous network of synchronous processes

$$P_1 \parallel P_2 \parallel \ldots \parallel P_K$$

such that

(GALS$_1$) ***: each*** P_k ***is endochronous, and***
(GALS$_2$) ***: the*** $P_k, k = 1, \ldots, K$ ***form an isochronous network.***

Using condition (GALS$_2$), we get

$$P_1^a \parallel_a (P_2^a \parallel_a \ldots \parallel_a P_K^a) \ = \ (P_1 \parallel P_2 \parallel \ldots \parallel P_K)^a$$

Hence every asynchronous behaviour σ_1^a of P_1^a produced by its interaction with the rest of the asynchronous network $(P_2^a \parallel_a \ldots \parallel_a P_K^a)$ is a desynchronized version of a synchronous behaviour of P_1 produced by its interaction with the rest of the synchronous network. Hence the asynchronous communication does not add spurious asynchronous behaviour. Next, by (GALS$_1$), we can reconstruct on-line this unique synchronous behaviour σ_1, from σ_1^a. Hence,

Theorem 2. *For $P_1 \parallel P_2 \parallel \ldots \parallel P_K$ a synchronous network, assume the deployment is simply performed by using an asynchronous mode of communication between the different blocks. If the network satisfies conditions* (GALS_1) *and* (GALS_2)*, then the original synchronous semantics of each individual block of the deployed* GALS *architecture is preserved (of course the global synchronous semantics is not preserved).*

To summarize, *a synchronous network satisfying conditions* (GALS_1) *and* (GALS_2) *is the right model for a* GALS*–targetable design,* and we have a correct-by-construction deployment technique for GALS architectures. The method consists in preparing the design to satisfy (GALS_1) and (GALS_2) by adding the proper wrappers, and then performing bruteforce desynchronization as stated in theorem 2.

5 Discussion

Of course there are similarities:

- The general concern is the same, namely to formalize the construction of modular architectures by providing a correct by construction methodology.
- The role and use, in sections 3 and 4, of special symbols e.g., $\perp$ and τ is similar [3]. Stuttering invariant processes of [2][3][4] very much correspond to stallable processes of [5]. The mechanism of stalling in [5] is reminiscent of the desynchronization mechanism of [2][3][4] (τ's are wildly inserted in individual signals in the former case, whereas synchronization barriers of reactions are lost in the latter). Also, system distribution in real-time control systems (section 2) also causes nondeterministic delay.
- The use of equalizers in [5] resembles the special protocol used in [2][3][4] to reconstruct σ from σ^a, see the detailed papers. Both involve a certain type of "blocking" or "delaying" reactions until necessary information is made available. It turns out that using "thick" events in combination with the so-called *confirmation* mechanism proposed by Caspi, performs the same for distributed real-time control systems, but in an implicit way.

Now there are differences:

- The work of [2][3][4] was intended to target architectures in which *no global clock nor state* is available, this differs from [5] in which synchronized hardware was considered, therefore providing global clock and state. Worse, for distributed real-time control systems, handling discrete events is not enough and the problem does not reduce to the only preserving of the discrete time/discrete event semantics.
- The approach of [5] is suited to synchronized hardware, it is not applicable to distributed software architectures in which communication is offered by some OS or middleware services. In [2][3][4] we were rather targeting either

[3] The use of "absent" or "non-informative" $\perp$ symbol was considered important, from the very beginning, when developing the SIGNAL synchronous language [1][9].

pure SW or mixed HW/SW. For HW, the techniques of [2][3][4] still apply, a variation of it has been used by the SIGNAL team with Paul Le Guernic and colleagues [8].

– In [5], the early spec for an IP is in fact the strict process it implements, there every signal has only informative events and an identical clock. In [2][3][4], the early spec is "multiclock" from the beginning, we distinguish between "running signals" and, say, "interruptions" and consider they are tagged by *different* clocks — they are not synchronous in the sense of [5]. Again, the early spec for distributed real-time control systems is much more subtle, as it already involves *robustness requirements,* from the very beginning, against variations of phase and delay in the {system + controller} dynamics, but these robustness requirements are considered mainly for the continuous control part. Then the very question is to make sure that distributing the computer control architecture still guarantees that the above robustness is maintained, including the discrete control part.

To conclude, "think synchronously – act asynchronously" emerges as a common paradigm for the design of embedded systems from components. We have discussed three different cases implementing it (the case of real-time control is still currently under finalization by Paul Caspi & friends). In some sense, one can see deep similarities between these different cases, and I can imagine that knowing the techniques used therein would allow one to find the proper adaptation for yet another, slightly different situation. Topologists or dynamicists are challenged to understand, capture, and formulate these diverse situations within a common mathematical framework, this challenge is certainly non trivial, given the diversity of the different mathematical approaches today.

Acknowledgements. *I am indebted to many people, who have gently stimulated my quest for the truth. Paul Caspi has consistently warned me that the desynchronization techniques of section 4 would not yield sensible results for distributed* real-time *control systems, although importing these techniques was formally feasible ; he convinced me just-in-time for the preparation of this paper. I thought synchronization would not be an issue for systems equipped with a global clock, and I am grateful to Luca Carloni, Dave Mc Millan, and Alberto Sangiovanni Vincentelli for demonstrating the converse to me.*

References

1. A. Benveniste and P. Le Guernic. "Hybrid dynamical systems theory and the Signal language." *IEEE Trans. on Automatic Control,* AC-35(5): 535-546, 1990.
2. A. Benveniste, B. Caillaud, and P. Le Guernic. Compositionality in dataflow synchronous languages : specification & distributed code generation. *Information and Computation*, 163, 125-171 (2000).
3. A. Benveniste, B. Caillaud, and P. Le Guernic. From synchrony to asynchrony. In J.C.M. Baeten and S. Mauw, editors, *CONCUR'99, Concurrency Theory, 10th International Conference*, volume 1664 of *Lecture Notes in Computer Science*, pages 162–177. Springer, August 1999.

4. A. Benveniste, B. Caillaud, P. Le Guernic, and J.P. Talpin. Desynchronization of synchronous programs : summary of results. Preprint, available at http://www.irisa.fr/sigma2/benveniste/pub/B_al99.html
5. L.P. Carloni, K.L. McMillan, and A.L. Sangiovanni-Vincentelli. The theory of latency insensitive design. Submitted for publication, 2001.
6. P. Caspi and R. Salem. Threshold and Bounded-Delay Voting in Critical Control Systems. *Proceedings of Formal Techniques in Real-Time and Fault-Tolerant Systems*, September 2000, Joseph Mathai Ed., 68–81, Lecture Notes in Computer Science, vol 1926.
7. P. Caspi. Embedded control: from asynchrony to synchrony and back. This volume.
8. A. Kountouris and C. Wolinski. A method for the generation of HDL code at the RTL level from a high-level formal specification language. In *Proc. of MWSCAS'97,* IEEE Computer Society Press, Sacramento, Aug. 1997.
9. P. Le Guernic and T. Gautier. Data-flow to von Neumann : the SIGNAL approach. In *Advanced topics in data-flow computing,* J-L. Gaudiot, L. Bic Eds., 413–438, Prentice Hall, 1991.
10. P. Maes. Concepts and Experiments in Computational Reflection. In *Proceedings of OOPSLA'87,* Vol. 22 of ACM SIGPLAN Notices, ACM Press, 1987.
11. J. McAffer. Meta-Level Architecture Support for Distributed Objects. In *Proceedings of Reflection 96,* G. Kiczales (ed), San Francisco, USA, March 1996.
12. G. Muller, R. Marlet, E.N. Volanschi, C. Consel, C. Pu, and A. Goel. Fast, Optimized Sun RPC using Automatic Program Specialization. In *Proceedings ICDCS'98,* Amsterdam, May 1998.
13. C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, K. Zhang. Optimistic Incremental Specialization: Streamlining a Commercial Operating System. In *Proceedings 15th ACM Symposium on Operating Systems Principles,* Copper Mountain Resort, Colorado, USA, December 1995.

Synchronous Programming Techniques for Embedded Systems: Present and Future

Gérard Berry

Chief Scientific Officer
Esterel Technologies
Gerard.Berry@esterel-technologies.com
www.esterel-technologies.com

Abstract. The synchronous programming methodology for reactive systems has been developed since the beginning of the 80's and it is currently used in numerous critical embedded systems projetcs. The methodolody is based on a strong mathematical framework that rigorously established the semantics of synchronous formalisms. The main industrial tools are Scade (Telelogic), Esterel Studio (Esterel Technologies), and Sildex (TNI). Current applications concern avionics, transportation systems, energy, telecom and wireless systems, etc. We present the general synchronous framework and the formalisms tailored to data-flow or control-flow dominated applications. We discuss the main issues in efficient code generation and formal property verification (safety, timing, etc.). We discuss the strength and limits of the basic synchronous paradigms, and present new ways of designing globally asynchronous / locally synchronous (GALS) systems.

T.A. Henzinger and C.M. Kirsch (Eds.): EMSOFT 2001, LNCS 2211, p. 50, 2001.
© Springer-Verlag Berlin Heidelberg 2001

From Requirements to Validated Embedded Systems

Manfred Broy[1] and Oscar Slotosch[2]

[1]Institut für Informatik, Technische Universität München
D-80290 München, Germany
http://wwwbroy.informatik.tu-muenchen.de/~broy/
[2]Validas Model Validation AG
www.validas.de

Abstract. We outline a practical modular method for the development of embedded systems from requirements and its comprehensive support by a CASE tool. Our focal point is on model-oriented development, multiview modelling, and validation. We present a method with a comprehensive tool support to develop models systematically from requirements and discuss various validation techniques by means of a car seat example. The resulting method is industrial strength, but nevertheless based on a solid scientific foundation and mathematical theory.

1. Introduction

Embedded systems are often applied in critical applications with a wide variety of often nonprofessional users. These systems have on one hand to meet the users' anticipations and capabilities and at the same time to be extremely reliable. In many applications such as in avionics applications over the last decades and in automotive applications currently and even more over the future years we observe an increase in system and requirements complexity. This is due to the fact that embedded systems of today and tomorrow have to provide deeper, more sophisticated support integrating several services. Stand-alone embedded systems of the past start to exchange information, cooperate, get connected, and form complex integrated distributed systems. Typically adaptive man machine interfaces that provide flexible access to several services of a hardware/software system gain importance. This requires better engineering methods and more comprehensive tool support to be able to cope with the higher complexity and the required system quality. Typical issues from software engineering of large distributed systems gain more and more importance.

In addition, more and more hardware/software systems include interfaces to external systems and thus are integrated into overall networks. Hence the classical design techniques from control theory and real time system design have to be extended by modern techniques from network software engineering.

As a result the area of embedded system is more and more dominated by software. The software determines the functionality and quality of the systems. Software processes dominate the development costs.

As result we see the subsequent key activities in the development of embedded software: advanced requirements engineering, component oriented architectures, and

T.A. Henzinger and C.M. Kirsch (Eds.): EMSOFT 2001, LNCS 2211, pp. 51-65, 2001.
© Springer-Verlag Berlin Heidelberg 2001

total quality assurance. This is only conceivable by adequate models, a rigorously scientifically based development method and comprehensive tool support.

Requirements engineering is a key activity in software and embedded system development for major quality attributes such as functionality, usability, and costs. It is the perhaps the most complex and certainly most crucial part in the development of a software system. It determines the functionality of the system, its user interface and in so far the costs of the development process. In particular, it provides the bridge between the application domain and the software solutions.

A software product can hardly be better than its requirements. The requirements establish the quality and usability of the system and dominate its development process. The functionality of a system is fixed in the requirements engineering process. The system requirements determine the functionality and thus its usability, complexity, and the costs of the development. It is a commonly acknowledged fact that errors in system and software design are the more expensive to correct the later they are discovered. This justifies the vital role requirements engineering and validation takes in system and software development.

Simple and tractable models of embedded systems are crucial for the systematic specification, design, and implementation of highly distributed interactive software in applications typical for reactive control systems, telecommunication, or computer networking. Several mathematical and logical models for such systems have been proposed so far. However, often they are too complex for practical applicability since they are based on sophisticated theories or they do not fulfill essential methodological requirements such as modularity or appropriate techniques for abstraction [3].

Based on these considerations we have developed a comprehensive theory [2, 5] under the name Focus that provides a flexible comprehensive system model, forms of composition, specification and refinement. For the system model a logical specification and development approach is worked out in all details. Based on Focus we have developed the CASE tool AutoFocus [6].

We give an overview on our approach in this paper. None of the issues can be treated in much detail. However, we give references to the extensive literature. Our paper is structured as follows. We start with an overview of the development process (Section 2) and close an important gap in this process by presenting a method that allows deriving models from requirements (Section 4). Section 3 contains the description of an example (Car Seat) that is modeled (Section 5) and validated (Section 6). Section 7 gives a number of concluding remarks.

2. The Development Process and Its Foundation

Our development process has several phases (analysis, design, implementation, and test). It is based on models. Hence two development phases are necessary: modelling and code generation. Code generation is performed completely automatically. An advantage of the model based development is that models can be validated better than text or code. Especially if the models have a formal basis essential properties can be verified. Furthermore the formal basis allows to validate the system (i.e. generate code and test sequences).

The process and the formal basis are described in [4]. For every description technique and every validation technique an extended theory is available. The tool AutoFocus uses essentially the following models:

- Data type declarations
- Component box diagrams
- System structure diagrams
- Extended event traces
- State transition diagrams

For each of these elements a comprehensive theory has been developed and published. It comprises the theory of black box behaviors of systems, their composition, the theory of state machines, the theory of extended event traces (message sequence charts) and a theory of refinement. The system model includes time aspects.

In this section (and Section 4) we concentrate on the early phases, i.e. the process of structuring requirements and building models. Design, validation, and code generation are not described here.

The development process starts with requirements analysis. The next step is the design. Usually there is a big gap between analysis and design. This gap is crucial if code is implemented manually, as well as if modeling tools are used. Many modern modeling tools (Together, Statemate, Matlab, Rose, …) provide therefore interfaces to requirements management tools (e.g. DOORS). This allows to link requirements to models. Using these links the requirements can be followed (traced) into the models and the generated code.

However there is still a gap in the process because all analyzed tools require to export the model into the requirements management tool, before the links between the requirements can be set (in the requirements tool). Therefore the model has to be build without any help from the requirements. Furthermore it is a lot of work to set all links manually.

Therefore we propose (in Section 4) a method that allows to derive model frames from requirements. If the requirements are detailed large parts of the model can be generated. This method also generates links from the generated models to the requirements.

3. Example: Car Seat

We illustrate the process with the development of a car seat. In this section we present the requirements. In the following sections we describe the model and show how the model can be used for validation. The example is a simplified view of the Premium Car Seat that has been the subject of the development competition of the OMER II (Object oriented Modeling of Embedded Systems) workshop. During this workshop Validas AG has used the tools AutoFocus and their validation suite and has been awarded.

The car seat (see Figure 1a) has a motor build in to move the seat (forward and backward) to a desired position. The user has a panel (see Figure 1b) that allows to move the seat and to store the current positions into two memory place.

Fig. 1a. Car Seat

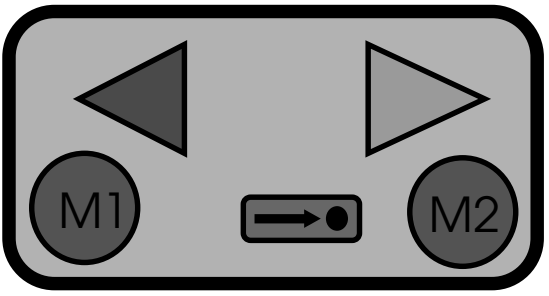


Fig. 1b. Control Panel

A motor moves the seat (forward/backward using the left/right triangular button) to a desired position. The user has a panel (see Figure 1b) that allows to move the seat and to store the current positions into two memory positions (M1 and M2), and to move towards the stored positions.

The process of storing the current position is as follows:

1. press the store button (in the middle)
2. release the store button
3. press a memory button (M1 or M2)
4. release the memory button

The seat motor is a simple electric motor (see Figure 2) to move the seat. The motor by itself recognizes the following commands: left / right / stop. The motor does not stop at a desired position, but it has a hall sensor that issues a signal for every turn of the motor axis. The maximum movement from the motor (from front to back) is 250 turns.

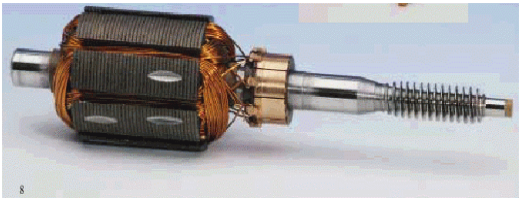


Fig. 2. Motor with Hall Sensor

4. Model Based Requirements Engineering

Model based requirements engineering is the integration of requirements engineering and model based software development. This integration allows to develop models from requirements and to link the requirements into the later development phases. Requirements influence essentially the design of the system. Moreover, they can be used for deriving test cases. Requirements tracing allows to trace the effect of requirements into the system.

In this section we first give an overview over methods for classifying requirements. After a general note on the formalization of requirements we present a general method that uses classifications of requirements to develop models from requirements and to link the models automatically to the requirements.

4.1. Classifying Requirements

Users describe a problem they want to be solved with **user requirements**. The engineer suggests to build a system to solve the problem and describes the solution using **system requirements**. Usually both user requirements and system requirements are structured. In general, however, these structures differ.

Besides these classes that stem from the development process there are other classifications, more related to the specific problems and solutions.

Requirements always are directed towards certain quality aspects of the software such as its behavior, its development process or its product properties. This leads to a way to classify requirements into the following three basic classes:

1. **Functional requirements (logical requirements):** Properties of the software system related to its behavior, its offered services, and its performance. These are all properties that are relevant for the active usage of the system, including risks associated with the operation of the system.
2. **Product requirements:** Properties of the software system related to its representation and the way of its realization and distribution. These are all properties that are relevant for making the software operational, distributing it, or using it in a specific technical environment.
3. **Process requirements:** Properties of the software development process. These are all properties that are relevant for developing the software.

An essential basis for all kinds of requirements is the understanding of the application domain and its properties. Here the fundamental notions, structures and properties of the application domain have to be discovered, documented and analyzed. This yields a further basic class of data relevant to requirements engineering.

4. **Domain analysis:** Properties and structures of the application domain.

Domain analysis is an important part of requirements engineering, although it does not directly lead to the formulation of requirements but forms its basis. Similar to other documented requirements, the documented properties of the application domain are only an *image* (or a *model*) of the real domain. A thorough analysis is necessary to ensure that this image is sufficiently faithful to reality w.r.t. the system under development. In the context of embedded systems the result of domain analysis is also referred to as the *environment assumptions*.

Let us give an example for classifying requirements. Typically safety and reliability requirements belong to the class of functional requirements, because safety requirements are relevant for the usage of the system. Reliability and safety requirements are therefore not listed as a basic class explicitly here, because they are derived from functional requirements and the domain model.

We characterize these basic classes further by identifying subclasses of them:

1. **Functional requirements**
   - **Functional and behavioral requirements** determine the behavior of a software system and the services offered by it.
   - **Performance requirements** define operational demands with respect to resources such as storage or computation time including, for instance, needed the accuracy of computed results.
   - **Timing requirements** constraint the frequency or computation time after which results must become available.
   - **Reliability and safety requirements** result from the potential hazards caused by malfunctions of the considered function. Typically they define constraints on the correctness or availability of the function.
     - **Quality requirements** describe the quality attributes of a software system. They comprise availability, acceptable failure rates and reliability, for instance. Depending on the kind of system under development, test coverage criteria or, e.g. in case of complex hardware/software systems, classical quality attributes like mean-time-between-failure or mean time availability may be applicable.
     - **Failure detection and monitoring requirements** define which kinds of failures must be detected or to what extent correct functionality must be monitored.
     - **Partitioning requirements** determine which level of independence must be provided by given functional units. The aim of such requirements is to ensure that a faulty component does not corrupt others as well.
2. **Product requirements**
   - **Product infrastructure requirements and (project specific) standards** constrain the products of the software development process, e.g. define which implementation technology (Java, EJB, ...) must be used or which naming conventions have to applied in the source code or which standard components have to be re-used.
   - **Hardware-related requirements** define memory size constraints and interface characteristics, such as protocols and input and output frequencies.
3. **Process requirements**
   - **Certification requirements** determine the application of a certain development process, or specific documentation and planning steps. One source of such requirements are certification procedures necessary for the product under development, e.g. in the context of software in airborne systems.
   - **Process infrastructure requirements and (project specific) standards** define, e.g., in which way the results of development activities are to be documented and which development tools must be applied.

In the context of embedded systems, hardware-related requirements can often also be functional requirements. Furthermore, some hardware-related requirements may be a

direct result from domain analysis, such that they can be regarded as a subclass of domain assumptions.

Note that the given requirement (sub)classes are not entirely orthogonal. For instance, safety requirements may strongly influence process requirements or partitioning requirements. In fact, the classification also reflects dependencies between requirement (sub)classes. Usually, the strongest influence on a subclass in our classification stems from its superclass and from the other subclasses in this superclass. Safety requirements are an exception here. They are primarily coined by the system's offered services and the domain assumptions, i.e. domain assumptions play a vital role here, which is not reflected in the classification.

To a large extent the listed requirement subclasses are taken from certification standards for aircraft systems [11,12,13]. Nevertheless, their relevance is not limited to safety critical systems. In particular, timing and performance requirements also play a role outside the embedded systems field.

The given classification is primarily based on the technical content of the requirements. Other classification criteria are the stages in the development process where the requirements appear and influence design decisions, or the system units they affect (hardware, software, physical environment, human operators). This explains that derived requirements, i.e. requirements resulting from other (higher-level) requirements and from implementation decisions that possibly have further implications on high-level requirements, do not occur in the classification.

The presented classification is not solely of academic interest. It leads to a guide line for decomposing and structuring informal unstructured requirements systematically into different classes of requirements. This encompasses a step towards a process of more technical requirements engineering making informal requirements operational.

4.2. A Note on Models in Requirements Engineering

As indicated above, functional requirements and environment assumptions are those that can be described in terms of (formal) system models. Documentation of the other requirement classes is usually restricted to informal notations such as structured text or statements in terms of the application domain. Nevertheless, these requirements also influence the formal models. Performance and timing requirements have an intermediate position here, because some of them may be described within appropriate models. Models that are amenable to some sort of computer aided analysis (model checking, prototyping, simulation, test, ...) can be particularly helpful to clarify and refine informal customer requirements. However, they are only one element supporting requirements engineering. The handling of textual requirements remains important as well.

Assuming an incremental development process with model based requirements specifications, non-functional requirements influence the models which are built during system or software development. This indicates that functional and non-functional requirements influence the models and are documented or *implemented* (in the case of executable models) within them. Vice versa the models lead to derived

functional and non-functional requirements. While development progresses more and more requirements are implemented within the model. For non-functional requirements the bar does not vanish completely, since non-functional requirements, like requirements concerning the documentation of the development process, are hardly reflected in models.

4.3. Deriving Models from Requirements

In this section we present a short overview of a method for deriving model structures from requirements. The complete method is described in [1]. The method does not restrict the user to use specific requirements classifications and it is not restricted to specific types of models.

The method consists of the following steps

1. identify requirements (assign them Ids)
2. classify requirements (using tags)
3. sort (and link) requirements according to their tags
4. select the system requirements
5. select a modeling language with model types
6. assign model types (as tags) to the requirements
7. generate model elements (and links) from the requirements with model tags
8. refine the models using appropriate modeling tools

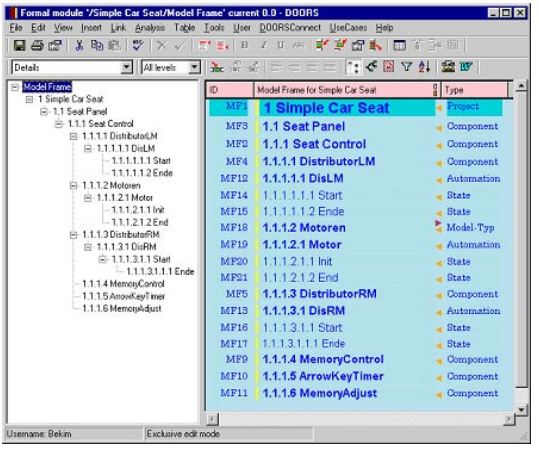


Fig. 3. Structured Car Requirements in DOORS

An important aspect is the hierarchy: hierarchic system requirements can be translated into hierarchic models (if the target modeling language support hierarchy). The method described in [1] also contains a reuse concept of types and instances that is not described here.

In our example the modeling language is that of AutoFocus, some model types are components, ports, states, values, types (see Section 5), and we use the tool DOORS to structure, classify and link the requirements (see Figure 3). For example the user requirement *Press store button* is structured into the following system requirements:

- There is a store button
- There are events
- Store button process the event pressed (`SPressed`)

The model tags of these requirements are component, type, and value. There is a forward-connection from DOORS to AutoFocus, such that a model frame can be generated from these tagged requirements. The model frame is refined using the AutoFocus modeling tool. The resulting model is described in the next section.

5. Modeling

From the structured requirements we derived a structure of the system. The Doors-Export into AutoFocus-models generates a system structure frame from the requirements. The structure of the systems is described using the System Structure Diagrams (SSD) of AutoFocus (see Figure 4). This view shows that the seat is receiving inputs from the environment via a channel called Panel. The values on this channels are of the type Events. The seat has two subcomponents: `SeatController` and `Motor`.

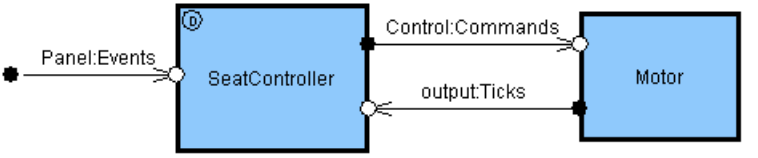


Fig. 4. Structure of the Seat

The used types are defined using the Data Type Definitions (DTDs) as follows:

```
data Events = FwdPressed | FwdReleased | …;
data Commands = Left | Right | Stop;
data Ticks = Turn;
data Signal = Present;
const MaxMove = 250;
```

DTDs also allow parameters and the definition of functions in the style of ML or Haskell. The structure of the `SeatController` is refined with another SSD (see Figure 5).

The structure reflects the architecture of the `SeatController`: It consists of a component `Buttons` that transforms the button events to constant signals and a `Controller` component. The `Controller` processes the button signals and the ticks from the motor. The Controller is an atomic component and uses local variables to describe the behavior.

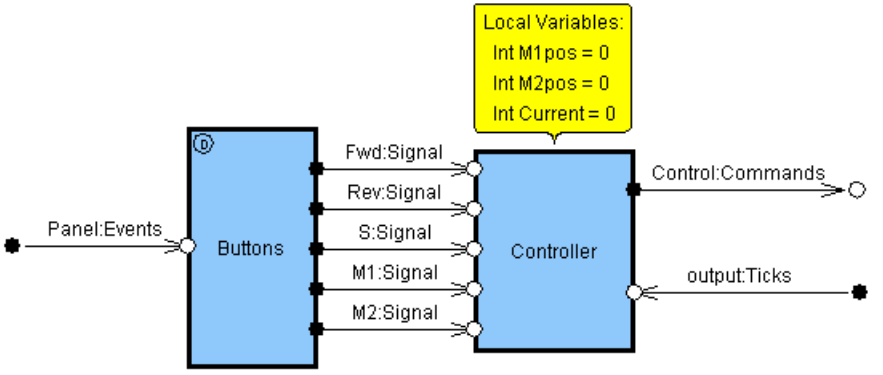


Fig. 5. Structure of `SeatController`

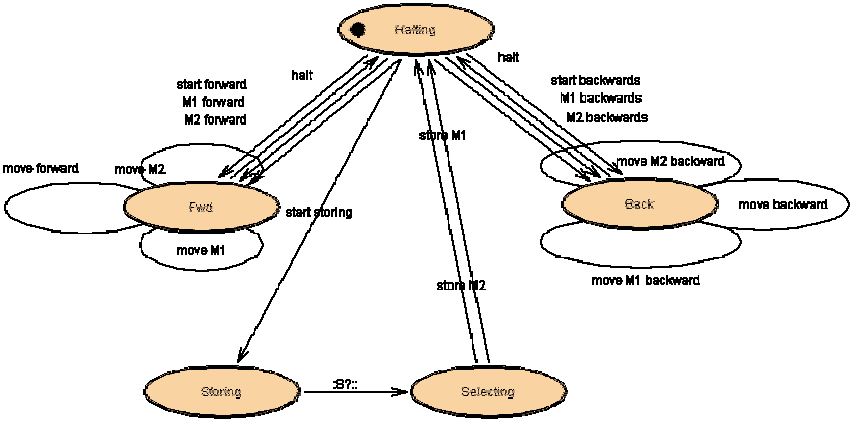


Fig. 6. Behavior of Component `Controller`

The behavior of the `Controller` component is defined using with the State Transition Diagram (STD) in Figure 6. It shows the different states of the system and their transitions.

The labels of the transitions are assigned with a description of the behavior using input and output patterns, preconditions and variable updates of the local variables. For example the transition `move forward` in state `Fwd` has the following semantic attributes:

- Input-Pattern: `forward?Present; output?Turn`
- Precondition: `Current<MaxMove`

- Output-Pattern: `Control!Right`
- Update-Action: `Current=Current+1`

The description of the behavior of the component refers only to the interface of the component and to the local variables. This allows us to reuse one STD for several components, for example the Buttons Component is modeled using five button components, each waiting for different `BUTTONPressed` and `BUTTONReleased` events of the button. They all use the same STD. Parameterization is modeled using the local variables in the SSDs.

The dynamic behavior of the system can be specified using Extended Event Traces (EETs). They describe the messages on the channels of the SSDs. EETs can be used to model requirements, simulation protocols, test cases or (counter) examples that show how certain (undesired) states of the system can be reached. The example EET in Figure 6 describes the interaction of the subcomponents of `SeatController` in the case the forward Button has been pressed.

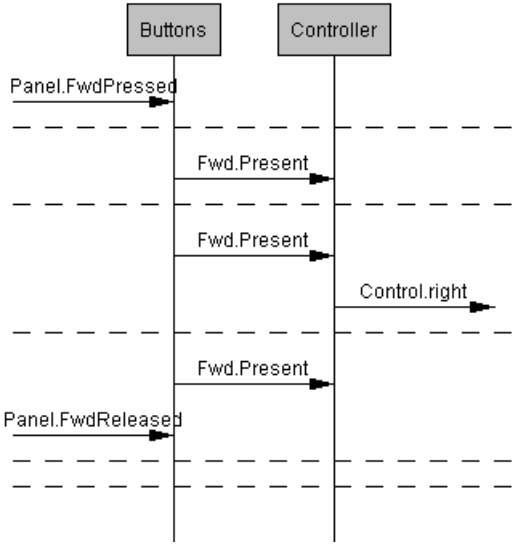


Fig. 7. EET: Simulation Trace of `SeatController` with input `FwdPressed`

The dashed lines delimit time intervals (time ticks). All messages between two ticks occur simultaneously. This temporal denotation allows for testing real-time systems.

Using AutoFocus the model can be checked for syntactic consistency and interactive simulation of the models is supported. Furthermore a C code generator is integrated. AutoFocus offers an XML-like interface to the Validas Validator that can be used to validate the system.

6. Validation Techniques

There are many different validation techniques available, some are more formal, and some are less formal and easier to use. All techniques rely on the same formal model of the system.

- Consistency checking (using OCL)
- Type checking
- Determinism checking
- Model checking
- Bounded model checking
- Abstraction techniques
- Theorem proving
- Prototyping
- Code generation (Java, C)
- Simulation
- Generation of test sequences
- Execution of tests (Generation of test drivers)

The Validas Validator tool is a validation framework around the models of AutoFocus. It has many been built during the project Quest [9] for the German Bundesamt für Sicherheit in der Informationstechnik (BSI).

The tool architecture integrates other powerful tools like SMV, Eclipse, SATO,VSE II, CTE, Mucke. Other tools can be connected easily. The connections to these tools are described separately in [7, 8, 9,10]. In this paper we describe other useful features of the validator: the determinism checker and the generation of test sequences.

The determinism checker of the validator checks for possible non-determinism in the models. The checker compares pairs of transitions that departs from the same state. The checker can be applied to states, automata, components or complete systems. It compares all selected pairs of transitions. There are two variants of these checks available: one compares only the patterns of the transitions and check if they match. For example the pattern `in?n` and `in?` are disjoint, while `in?n` and `in?True` overlap. Overlapping patterns are a source for possible non-determinism. If the model uses preconditions the pure pattern check might be too coarse, i.e. it might find too many situations, for example `n=False;in?n` and `in?True`. In this case the second variant of the determinism check is finer. It enumerates all variables in the preconditions (in this case n) and evaluates the preconditions for all possible values of the variables and compares the results. In this case (in?True, and in?False). This variant uses a (configurable) value MaxInt to enumerate integer domains. Floating values and recursive user defined types cannot be checked with this check.

Both checks are local, i.e. no reachability analysis in the system is necessary. Therefore these checks can be applied to larger systems efficiently. Determinism checking is especially useful for synchronous models, where several events can occur at the same time. In the car seat example we discovered that the controller behaves non-deterministically if several buttons of the panel are pressed at the same time. This is a valuable hint.

The models are independent of the programming language. Different code can be generated (C, Java, Prolog, Ada). In the premium car seat example the model was very large. 96 components where required to control the seat (with six motors and various functionalities). Due to the very simple semantics of the AutoFocus, and the high degree of reuse in the model the resulting C code for the complete system had only 28KB size and is very fast, such that it can run on low-cost micro controllers common in automotive applications.

Testing is the main validation technique for embedded systems. We use the models to generate test sequences. The graphical representations of test sequences are EETs. An additional, textual representation (test data format) can be used for the generation of test drivers, or (if the code is generated using our generators) the generated code can process the test data format directly.

There are many possibilities to generate test sequences (EETs). The simplest way is to generate them using the simulation (with manual input). Such test sequences are protocols of the simulation. Many modern tools have this feature. More difficult is to search automatically for inputs that cover a test purpose. For example, a transition shall be executed, or a certain output shall be produced.

The transition checker [10] is a feature of the validator that allows selecting an arbitrary transition and automatically generates input values that drive the system into a state that the transition is executed. The check is based on bounded model checking, i.e. it requires a finite model and to enter a maximum length (search depth) for the sequence. The transition checker can also search for input for fixed transition sequences. The transition checker respects all aspects of the model, and therefore all generated sequences fit to the model. This makes the checker more powerful than other generations (for example the transition tour) that base only the graphical structure of the model.

Constraint solving techniques are not restricted to finite domains and can also be applied to search in the model for test sequences. Figure 8 shows a generated test sequence that tests if a value (different from 0) can be stored into M2. This sequence can be generated using model checking with the invariant `[](M2pos=0)`. The sequence is a counter example for this property. For model checking this property abstraction can be used for the domain of integers. Constraint solving techniques find the same test sequence, but require more interactions: the constraints have to be formulated such that the search is directed towards the solution. The advantage of the constraint method is that it also works on large and infinite domains.

Of course the validation techniques can be used to improve the model. For example property `[](Events=FwdPressed => <>Controller.@=Fwd)` is false because the forward button is not checked during backward movement or during storing of values. This bug can be removed by changing the model.

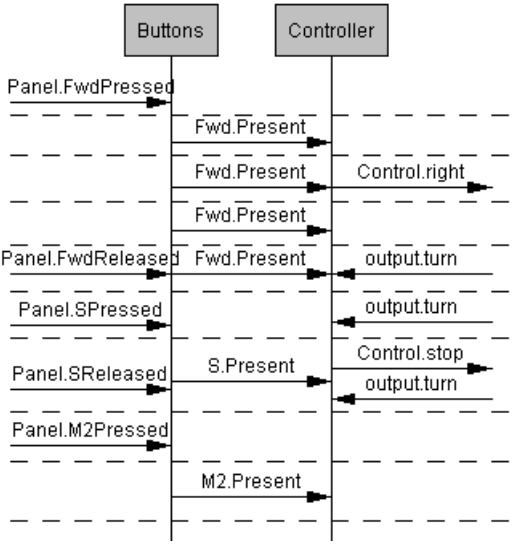


Fig. 8. Generated Test Sequence

7. Conclusion

Embedded systems are an important application domain for software engineering. Their proper formal and methodological treatment is of major importance due to the high demands on reliability, usability and integration into the systems. For a systematic development standards are needed such as a systematic development process and product models as well as reference architectures.

It is the purpose of this paper to give an overview and to outline the general issues of the engineering for embedded systems. The process of requirements engineering is decisive for the quality of service and usability of the system. For embedded systems it is typically a part of the systems engineering. It has to pay attention to all aspects of the system behavior and therefore needs models that go beyond the classical models for hardware or software systems.

According to the model structure, model complexity and the desired quality level different validation techniques can be applied. Several case studies (a SmartCard [8], the OMER II car seat, ...) have shown that validation is possible and that abstraction techniques are necessary to check complex and large models with high security requirements.

For our approach and its extended tool support the foundation by a well-worked out model with a full-fledged theory (see [5]) is essential. Although the tool AutoFocus is based only a simplified subtheory with synchronous time, the availability of the theory made it possible to design and validate the methods integrated into the tool. Moreover, the theory gives guidelines for the developer.

Acknowledgment. It is a pleasure to thank Peter Braun and Bekim Bajraktari and Jan Philipps for useful discussions and comments on previous versions of this paper.

References

1. B. Bajraktari: Modelbasiertes Requirements Tracing. Master thesis, TU München, 2001
2. M. Broy: Compositional Refinement of Interactive Systems. DIGITAL Systems Research Center, SRC 89, 1992. Also in: Journal of the ACM, Volume 44, No. 6 (Nov. 1997), 850-891
3. M. Broy. Requirements engineering for embedded systems. In *Proc. of FemSys'97*, 1997.
4. M. Broy, O. Slotosch Enriching the Software Development Process by Formal Methods, *Proceedings of FM-Trends 98*, LNCS 1641
5. M. Broy, K. Stølen: Specification and Development of Interactive Systems: FOCUS Focus on Streams, Interfaces, and Refinement. Springer 2001
6. F. Huber, S. Molterer, A. Rausch, B. Schätz, M. Sihling, O. Slotosch: Tool supported Specification and Simulation of Distributed Systems, *Proceedings of International Symposium on Software Engineering for Parallel and Distributed Systems,* 1998
7. J. Philipps, O. Slotosch: The Quest for Correct Systems: Model Checking of Diagramms and Datatypes, P*roceedings of Asia Pacific Software Engineering Conference 1999,* 449-458,
8. A. Pretschner, O. Slotosch, H. Lötzbeyer, E. Aiglstorfer, S. Kriebel: Model Based Testing for Real: The Inhouse Card Case Study in Proc. 6th Intl. Workshop on Formal Methods for Industrial Critical Systems (FMICS01), Paris, July 2001
9. O. Slotosch: Quest: Overview over the project, *Proceedings of FM-Trends 98*, 1998 LNCS 1641: 346-350
10. G. Wimmel, A. Pretschner, O. Slotosch: Specification Based Test Sequence Generation with Propositional Logic, *Journal on Software Testing Verification and Reliablilty* (to appear)
11. SAE. Certification considerations for highly-integrated or complex aircraft systems. ARP 4754, 1996.
12. RTCA Inc., EUROCAE. Design assurance guidance for airborne electronic hardware. Working draft #17, SC-180 / ED-80, 1999. RTCA Inc., EUROCAE. Software considerations in airborne systems and equipment certification. DO-178B / ED-12B, 1992.
13. IEEE Std 830 -1993: IEEE Recommended Practice for Software Requirements Specification. IEEE, 1994.

Usage Scenarios for an Automated Model Compiler

Ken Butts[1], Dave Bostic[1], Alongkrit Chutinan[2], Jeffrey Cook[1], Bill Milam[1], and Yanxin Wang[1]

[1]Ford Research Laboratory, Dearborn, MI, USA
{kbutts1, dbostic, jcook2, wmilam, ywang} @ford.com
[2]Emmeskay, Inc., Plymouth, MI, USA
alongkrit@emmeskay.com

Abstract. This paper is meant to motivate tools and methods research in the field of model-based embedded software development. In particular, we include usage scenarios to describe how an automated model assembler called a *model compiler* could support automotive embedded control systems development. We describe some desired characteristics and features of the envisioned model compiler and place particular emphasis on support for model compatibility checking. Finally, we describe characteristics of model components that are commonly used in practice.

1 Introduction

The automotive manufacturers' need to reduce cycle times and costs has resulted in a shift toward model-based design and development. Such processes offer the opportunity to generate and validate robust designs with minimal expenditures on physical prototypes. However, early adopters of model-based development often find that model creation is a time consuming and resource intensive task [1], especially when a large number of system and vehicle level models is required to support typical vehicle and technology variations.

We believe that, just as automotive vehicles are efficiently assembled from mass-produced parts and sub-assemblies, it should be possible to cultivate an environment in which subsystem models can be automatically validated for compatibility and assembled into system or vehicle models for model-based design and development. We strive for such a modeling environment to facilitate model reuse over product life cycles, across product families, and across development organizations. Experience has shown that, given sufficient attention to model compatibility, significant reuse can improve quality, reduce development cost, and reduce time-to-market. See [2] for a case history in applying software factory approaches to industrial control software development.

The automated model assembler, or *model compiler*, is the subject of this paper. A model compiler is a tool that automatically composes a model from a set of sub-models and an architectural description of the arrangement of the sub-models. It will also ensure full connectivity of all control flow and data flow signals between sub-

T.A. Henzinger and C.M. Kirsch (Eds.): EMSOFT 2001, LNCS 2211, pp. 66-79, 2001.
© Springer-Verlag Berlin Heidelberg 2001

models, proper sequencing of sub-models, and compatibility of sub-models. In the following we use the term *component* to mean an element, whether part, subsystem, system, or vehicle, of a composition set that is submitted to the model compiler for assembly. To convey the scale and complexity of a typical assembly level model, let us consider a modeling environment used for energy analysis [1]. This model is composed of nine major systems; seven of these include controller, sensor, and actuator subsystems. There are forty-five locations in the environment where the user can select [3] between 129 component offerings. Within this framework, analysts can readily evaluate various vehicle technologies by assembling the components into alternative configurations.

Another interesting example is the powertrain control application. Here 218 algorithm model libraries, each with 3 to 30 component variations, are used to service 130 vehicle applications in the product-line. A typical vehicle application is composed from 75 to 105 components and requires more than 2000 signal-flow interconnections between the components. Ultimately, we would like to be able to link these two example environments by managing the vehicle specific powertrain control application as a single (albeit complex) component in the energy analysis environment; hence the need for automation. We believe the model compiler will be used recursively to assemble complex components and then to create systems of such component assemblies.

We describe our notion of a model compiler in section 2 and emphasize component compatibility analysis in section 3. Our application focus for this work is the on-board electronic control system domain, including the powertrain, chassis, and energy management systems. Given this domain, we describe how a model compiler can be used to support system component selection and requirements generation, facilitate system evaluation studies, and support large-scale software development factories in section 4. We conclude with a call for research collaboration in section 5.

2 Model Compilers

In order to understand the concept of a model compiler, we must first set the context within which one would use a model compiler and define what kinds of models are involved. While we feel the model compiler concept has a broad range of applications and should help to ease the use of many types of models, our scope for this paper is restricted to the Matlab®, Simulink®, Stateflow®[1] modeling domain.

Consider the powertrain control example mentioned in the introduction. Using median numbers, there are 218 different elements. Each of those has a median of 16 variants leading to 3,488 possible components. From this domain we need to select 80, which would make up a median application. If we assume 25 input/output ports per component we now have 2000 connections to make to create a fully composed application model. We need to ensure that the signal characteristics actually make

[1] Matlab, Simulink, and Stateflow are Registered Trademarks of The MathWorks, Inc., Natick, MA.

sense for each of these port connections. A simple signal name match does not necessarily imply that the connection is valid. We also need to ensure that the format of the signal matches at both functional and non-functional levels. At the functional level, we have to ascertain that the two signals are of compatible type: for example both should be scalar types or vector types of the same size. Non-functional characteristics that must be checked include verifying that the units of the two connections are correct. If, for example, an input-output pair attempts to connect "radians per second" to "degrees Kelvin", an obvious (non-functional) error can be detected and flagged to the user. If however, we are trying to match radians per second and revolutions per minute, the possibility exists to insert a conversion to correct the units mismatch. A warning would then be given to the user that this less onerous mismatch has occurred, but the units may interact correctly.

The model compiler requires the existence of component data dictionaries because the internal behavior of the model component may not be exposed. In the Matlab®, Simulink®, Stateflow® world, this is true if the component is an S-Function using C or Fortran. In the case where the component contains yet more Simulink® or Stateflow® components it is possible to build directed graph representations of the internal data and control flow of each component in order to look at optimal sequencing of components in a particular subsystem. The result of this analysis might be a Stateflow® component that deliberately sequences the components to ensure the proper sequence of operations.

Now that we have presented some of the tasks the model compiler has to accomplish, how do we envision it working? There are two necessary items that must be defined in order for the idea to work. First we need to define what constitutes a component. Second, we need to define how we select components and how we describe the hierarchical organization of the model.

2.1 Model Component Attributes

Some of the key component properties, which can be derived from a model compiler compliant component model, are:

- Sample Time (if the controller models are designed to be fixed step.)
- Solver (if one is used.)
- Set of inputs
  - Data dictionary definition for each input
    - Name.
    - Type.
- Set of outputs
  - Data dictionary definition for each output
    - Name.
    - Type.
    - Source component.

- Set of workspace parameters (typically, calibration parameters)
  - Data dictionary definition for each parameter
    - Name.
    - Type.

Once we have this information for each component to be composed, we can begin to determine whether the given components actually can be composed.

2.2 Assembly Model Structural Specification

One solution to the problem of model organization may be to combine the model component concept with a *formal language* [4] that describes the architecture of the model created by the components. Consider a simple example based on three simple components as shown in figure 1. Each component, X, Y, and Z, has some number of inputs and at least one output.

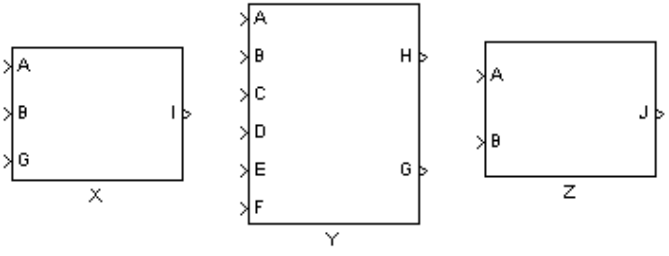


Fig. 1. Simple composition example

Each input and output is assumed to be of a scalar type. In addition, each input and output is named with a signal name. For our example these are just simple single character names, in a real model however they might be more descriptive names such as RPM, Coolant Temperature or Throttle Position. Now, define a component to have a set of inputs and a set of outputs and walk through the following scenario.

Accept, for the moment that the statement: Q { X,Y,Z} means that a component Q is to be created with members X, Y and Z. (Q might look a lot like figure 1.) The set of all inputs to Q can be found by taking the union of the sets of inputs from the three components:

$$X.in = \{A,B,G\}$$
$$Y.in = \{A,B,C,D,E,F\}$$
$$Z.in = \{A,B\}$$
$$Q.in = X.in \bigcup Y.in \bigcup Z.in = \{A,B,C,D,E,F,G\}$$

So *Q.in* represents the set of all inputs to Q. The same operation can be used to determine the set of outputs for Q. Figure 1 shows that the input G to component X is

actually an output from Y. This can be determined by finding the intersection of *Q.in* and *Q.out*.

$$Q.out = X.out \bigcup Y.out \bigcup Z.out = \{G, H, I, J\}$$
$$Q.inner = Q.out \bigcap Q.in = \{G\}$$

So *Q.inner* represents the set of signals that originate and terminate inside Q. Another reasonable action might be to remove G as an element of *Q.in* and *Q.out*. If G originates within Q, it certainly should not also be a member of *Q.in*. It may, or may not, also be a member of *Q.out*. G could be removed from *Q.out* by default. Of course it may be possible that G is used by another component, which is a peer of Q. In that case, the model compiler would need the ability to search a model to find instances of G that are a match in both name and type to the G that is needed.

Our example demonstrates that, by applying simple operations on component attributes, we can 'wire together' the components to form either larger groups of components or a full model. As these functions are recursive, it is now possible to define a language to support this task of combining model components according to some rules. The use of rules implies that errors can be identified and flagged. This would be similar to the function of a High Level Language (e.g. ANSI C) compiler that does matching between prototype function declarations and instantiations of the function call. This leads to the idea that the model compiler will consist of two processes: one process will parse an architecture language that describes relationships between components. The other will parse the components to create mappings of relationships. The architecture language then becomes a control language to the output generator that will take the parsed components and create a new component as directed by the architectural description. The model compiler should also perform analysis on the components to determine if the components being combined are compatible.

3 Compatibility Relationships

The model compiler must ensure compatibility among the components being composed into the overall target model. There are two levels of compatibility that the model compiler must consider. Locally, as described in the previous section, the model compiler must ensure interoperability of the components in the same subsystem. Globally, the model compiler must ensure that only components that are designed to operate together are integrated in the same model. The following subsections describe the nature of these compatibility requirements in greater detail.

3.1 Structural Compatibility

The model compiler must first ensure the interoperability of the components at each Simulink® subsystem level. While the model compiler may not be able to fully

determine whether the components in the subsystem are interoperable semantically, it can at least ensure structural compatibility of components in the following areas.

Signal Type. When connecting an output signal of a component to an input signal of another component, the model compiler must ensure that each input signal is of a type compatible to that of its source output signal. The type information for each signal may describe both *static* and *dynamic* [5][6] properties of the signals. Examples of static properties of a signal are data type (real, integer, boolean, trigger, enable, function call, etc.), units, and dimension. Dynamic properties describe how the signal evolves with time or how it is updated possibly both in simulation and real hardware. Examples of these properties are:

- Continuous-time vs. discrete-time (with sampling rate.)
- Data transfer protocols such as Controller Area Network, Time Triggered Protocol, or direct connection (signal values are always identical at both ends of the connection at any time.)

The signal compatibility may be defined in terms of the so-called *lossless convertibility* [5][6] where the output signal can be converted into the format of the input signal without loss of information. For example, an integer output can be converted into a real input. A discrete-time signal with can be up-sampled to a higher sampling rate provided that the new sampling rate is a multiple of the original sampling rate. The model compiler must ensure compatibilities of both static and dynamic properties of the signals being connected and notify the user when type incompatibility is detected.

Simulation Properties. Simulation properties may be specified for each component in the subsystem. Examples of such properties are ODE solver type, fixed or variable time step size, and tolerances. These properties essentially specify the preferred execution method for the component. The model compiler must ensure that the simulation properties for all components in the model are compatible. This assurance may be provided, again, through the concept of lossless convertibility similar to the one discussed in the previous discussion. For example, a model component that uses a fixed step ODE solver can be simulated with another component that uses a fixed step solver with a smaller time step, provided that the larger time step is a multiple of the smaller time step. The aggregate component now uses the smaller time step for the solver, i.e. the most general simulation property among all components is used. If no common simulation property can be found that all components can be converted into, the model compiler must notify the user of the conflict.

3.2 Component Compatibility

We discuss structural component compatibility earlier in this section. Here we attempt to address functional component compatibility. Our basic idea is to capture and exploit the knowledge that model components have been designed to satisfy certain product requirements or that they have previously been used successfully with other components. To refine this idea we introduce the concept of a *model domain*. Model

domains are loose classifications that serve to sort model components and other domains according to some functional attribute. A *Customer Vehicle Character* domain could be used to classify vehicle level customer attributes such as *green* (environmentally friendly) and *fun-to-drive*. *Application* domains corresponding to actual automotive products (e.g. *2.0L, lean burn engine*) will be useful in tracking previously validated model assemblies. A *System* domain hierarchy could be used to replicate product system hierarchies (e.g. *Electronic Throttle Control*) to aid in model component organization and selection. It is often useful to aggregate model components that are considered alternatives in some sense. For example, models representing two types of electronic throttle control (ETC) sensors should reference the *ETC sensors* attribute of the *Component Choices* domain. Similarly, a *Parameter Choices* domain could aggregate model parameter choices for a single model component. We present usage scenarios associated with these notions of model domains in the next section.

To identify compatibility we assign *Member of* relationships between domains and components. If two components do not have a common domain attribute (reachable by tracing *Member of* relationships) then they are potentially incompatible and their combined usage must be validated. For example, the usage of two components that trace only to *green* and *fun-to-drive* respectively should be investigated since there is likely to be a conflict. Moreover, two elements that are members of the same choice domain attribute are, by construction, incompatible for a given instance. The *Member of* relationship can be specialized to the *Necessary member of* relationship when a fixed relationship exists (e.g. an *ETC system* always contains *ETC Controller*, *ETC Actuator*, and *ETC Sensors* components*.)* This specialization admits completeness compatibility analysis.

A component manager will be tasked with assigning domain membership relationships to each model component and its associated parameterization sets when it is checked into the model compiler database. A model component may be a member of multiple domains. The accuracy of the domain compatibility checking largely depends on the level of detail that is included in the domain models and the maintenance of the membership relationships. It may not be practical to model the domain structure to the fullest detail in general. Nevertheless, one can benefit from this type of compatibility checking. Even with partial domain information, the model compiler can still lessen the chance for the modeler to incorporate nonconforming components into the same model.

4 Usage Scenarios

In this section we describe how the concepts presented in the previous sections can be applied to automotive control system development. The powertrain challenge models in the Model-Based Integration of Embedded Software project [7] are representative of the types of model components that could be assembled into automotive system models. We have plant components that model the behavior of the physical system under control. In this case, the engine, transmission, vehicle, sensors, and actuators are managed as components whose hybrid-dynamics [8] evolve in the continuous and

crankshaft-position domains. The signal-flows between these components represent physical quantities such as pressures, torques, and velocities. Controller-function models form a second component type. The dynamics of the controller-function components are event-driven. Thus the controller-function component has two interface types: the signal-flow for representing inter-function signal-flows and the function-call event for passing initiation directives from function to function. The third component type is the scheduler type that serves as the control-flow interface between the physical and controller function types. The scheduler observes the physical system and starts threads of initiation directives in the controller functions controller.

Figure 2 shows how the three basic components interact in a control system model. Note that the controller-input models are a special controller-function (event-driven) type that input continuous physical signal-flows and output event-driven controller-function signal-flows. Conversely, the controller-output models are a special plant type that input controller-function signal-flows and output physical signal-flows.

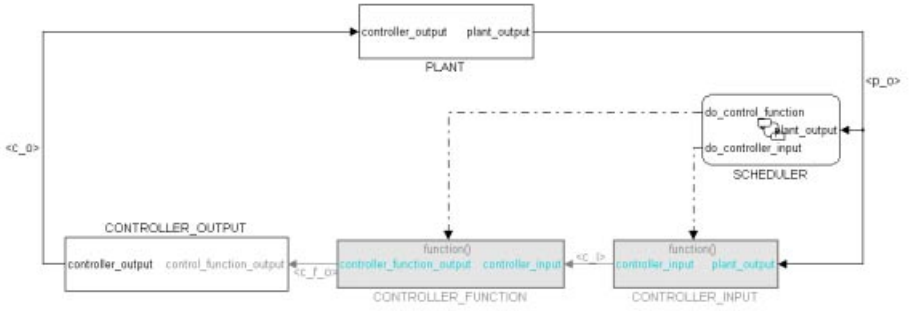


Fig. 2. Component model types

Let us consider the task of developing an embedded control system for a new vehicle application. Given the appropriate level of abstraction, the engineering processes and corresponding model applications are similar for the powertrain, chassis, and energy management electronic control domains. The remainder of this section describes the role of the model compiler in an advanced electronic controller development process.

4.1 System Definition

Once the requirements for a vehicle are known, a system must be assembled to satisfy those requirements. As discussed in the introduction, component reuse can have a significant impact on quality, development cost, and time-to-market. However in practice, we experience three difficulties with component-based development at the system definition stage. First, how does the system designer sift possibly thousands of component models to identify the appropriate set of components for this system? Second, how does the system designer know that the identified set of components is consistent in terms of interface and system architecture? Third, how does the system

designer know to request new component development? And what are these new component's requirements in terms of interface and system architecture?

Sifting the Components (Creating an Assembly Model Build List.) Let us show by example that the previously described notions of component compatibility are applicable here. A vehicle team is charged with developing a *green* and *fun-to-drive* vehicle and the system designer decides to specify a *2.0L, lean-burn engine* to accomplish this task. The control designer typically assembles a system model from previously used components to validate this design choice.

Consider the (incomplete) component relationship model shown in figure 3. This model contains valuable information that can guide the creation of an appropriate assembly model build list. By selecting the *2.0L, lean-burn engine* from the *Application* domain the control designer can readily see that this engine is a plausible choice for a *green* vehicle because it is a member of the *low emissions* and *low fuel usage* attributes. Moreover, the designer learns that this engine must employ an *engine management system (EMS)*. The designer could choose to include the optional *electronic throttle control (ETC)* system since it further contributes to *low emissions* and has been previously applied in the *2.0L, lean-burn engine* application. Thus the control designer can sift through the relevant application domains and their component membership relationships to create a complete build list for the assembly model of interest.

We remark here on an implicit requirement of the model compiler technology. Please notice that the component relationship model includes the ability to create component and parameter choice relationships. As described in [3], there are several design scenarios in which the control designer may wish to iteratively switch between predefined choice selections without incurring the cost or system architecture modifications associated with a system model re-compile. Thus we require that the model compiler and Large Scale Model [3] model management technologies be complementary.

Analyzing the build list. Once the control designer has generated the assembly model build list, the analysis engine of the model compiler can be used to check the quality of the build list specification. We envision that the following validation analyses will be particularly useful:

- That the assembled model will satisfy the subsystem compatibilities described in section 3.1.
- That all inputs are either generated by a component model or defined as assembly model inputs.
- That all unused output signals are either terminated or defined as assembly model outputs.
- That all component compatibility relationships are satisfied or explicitly over-ridden.
- That the build list is complete in the sense that all "necessary member of" relationships are satisfied.
- That the build list is unambiguous in the sense that component models with choices employ only one component selection per location.

- That the specified hierarchical structure follows (user-defined) architecture rules, for example:
  - Plant component models are grouped with other plant components.
  - Controller-function component models are grouped with other controller-function components.
  - One and only one scheduler component model is associated with each controller-function group.

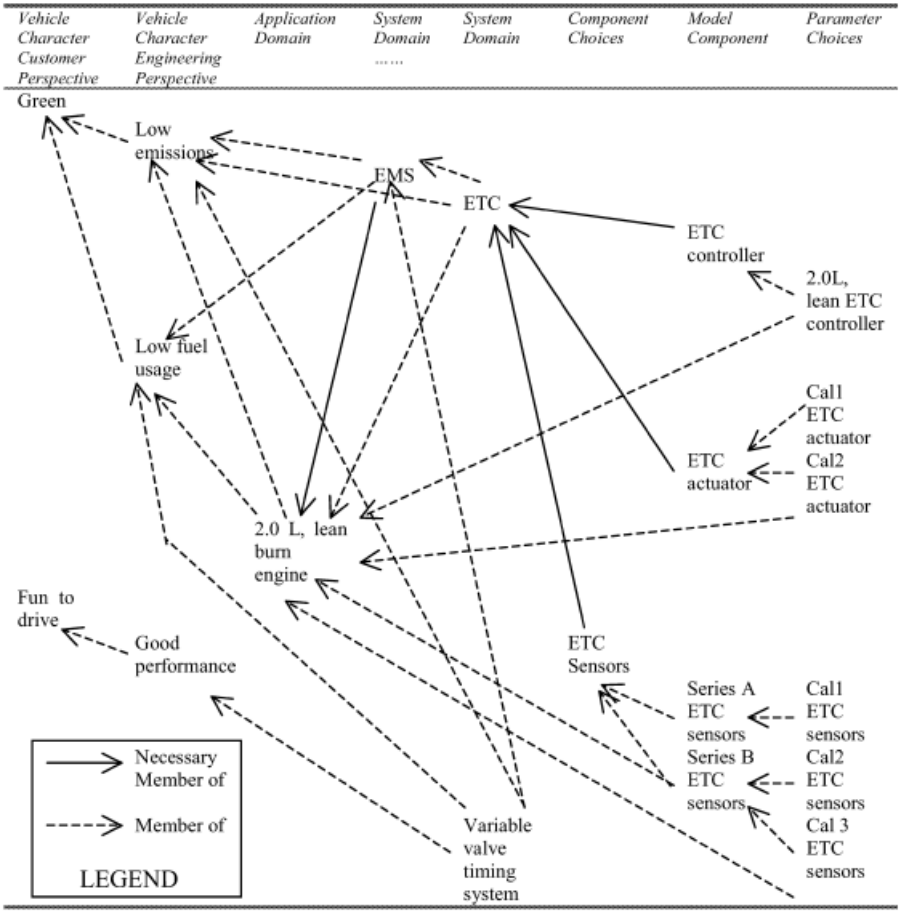


Fig. 3. Component Relationship Model

Generating New Component Requirements. Let us continue the example from section 4.1. Assume that inadequate *fun-to-drive* characteristics are generated from the assembled vehicle system model that includes the *2.0L, lean-burn engine* with *electronic throttle control* system. Under investigation, the component relationship

model (figure 3) will show that the *variable valve timing (VVT) system* is a member of *good performance*, *low fuel usage*, and *low emissions* attributes. Thus it is a good candidate for inclusion in the vehicle system build list.

However, in this case the designer learns that a *VVT system* has not been previously used in the *2.0L, lean-burn engine* application. Consequently, new component development must be initiated. The model compiler can generate a set of interface and composability requirements that arise when the *VVT system* components are included in the system build list. These requirements can then be used to guide a work task to:

- Determine if a new air management coordination controller (between the *ETC* and *VVT controllers*) is warranted.
- Resolve any interface and composability issues that arise when the new controller and *VVT system* components are added to the build list.
- Generate a complete set of parameter component choices for the new controller component and system application.
- Update the component relationship model upon design validation of the new components and system application.

We believe that the creation and maintenance of the component relationship model will help in the system definition and new component creation phases of development. Design iterations between system and control designers will be significantly enhanced via efficient reuse of compatible components.

4.2 System Validation

In today's practice, it is a year's major accomplishment to assemble a powertrain control application model as described in the introduction. Thus full systems integration models typically do not support the design stage of development. Usually, our downstream engineering customers verify systems integration and performance in the target vehicle application. Of course, this tends to damage inter-departmental relationships, reduce engineering efficiency, and increase time-to-market.

The ability to create large system models in a timely and efficient manner is a critical benefit of the model compiler technology. We believe that systems validation during system and component design (prior to component implementation and subsequent systems integration) will be possible. The assembly models created with the aid of the model compiler provide a complete description of the system dynamics. The functionality of the full system can thus be evaluated using simulation. Here again, Large Scale Model technologies [3] that allow the control designer to switch between model component choices within a given model structure would be useful. Pre-compiled, interpreted, empirical, or first-principles versions of a model component could be made available to manage the simulation speed versus model complexity trade-off.

Full system models are particularly useful for studying the system's:

- Robustness to parameter variation.
- Performance under various environmental conditions and usage scenarios.
- Performance in response to failures and faults.

We believe these studies should be made in the context of validating the system requirements. Requirements test procedures should be embedded in Test Scenario model components that have traceability to the product requirements database.

We intend to provide full system assembly models (complete with Test Scenario components) to our internal and external suppliers so that they can develop a better understanding of our requirements for end-product application. We will also ask the suppliers to contribute component models of their subsystems and devices. Requirements for these requests should be consistent with the compatibility relationships discussed in section 3. Intellectual property protection mechanisms need to be developed to facilitate model sharing across organizations.

4.3 Software Development

The development process and tools that comprise the model compiler can have a significant impact on the development of embedded software implementations of the compiled models. In today's powertrain controller development process, we use the validated control-function component model as a template for software development. Unfortunately, this process is labor intensive and time-consuming due to its inherent reliance on the software developer's ability to analyze the control-function model, write embedded software that is intended to behave as specified in the model, and verify that the embedded software and control-function model have the same behavior.

Model compiler technology, coupled with code generation technology can improve this situation dramatically. Newly available tools now allow us to automatically generate embeddable software directly from the model specification. Consequently, the model compiler should capture data flow interaction among model components in a format that is compatible with today's automatic code generators.

The compatibility relationships previously described in section 3 are important when using the models for software development and automatic code generation. Our internal control-function modeling guidelines address discrete-state behavior, software-tasking architecture, and distinct data flow and control flow partitioning. Thus they provide the determinism exhibited in our implementation language: ANSI C. These characteristics, when embedded in the component object, ensure consistency between the generated code and the control-function model without need for additional configuration.

The consistency of the dataflow interfaces between components will provide the single biggest benefit to the automatic code generation phase. This is because managing the different interfaces of each component or connection is generally time consuming. Certainly, the interfaces on archived components may not align with traditional C boundaries, which are function interfaces. Fortunately, this is not a necessity. Control-function components verified for inclusion in the component library may have some superstructure that naturally maps subcomponents into functions, but the interface at the component level should save the software developer from having to trace every data source and destination for verification.

The specification of each model component as an independent entity will have significant positive impact on the code generation process as well. This process calls for functional validation prior to software development necessitates rigorous up-front requirements capture: a critical quality attribute in the software development process.

The ability to connect multiple components has been acknowledged as beneficial for simulation, but can have many advantages for embedded software development efforts as well. In our current software development process, each element is typically specified and designed in isolation. The interfaces to these elements are defined, and static test inputs are used to evaluate the correctness of the implementation. Using the model compiler technology to replace this approach can have multiple benefits. For example, if multiple components are joined (especially where there is strong data flow connectivity among the components), a single test vector can generate inputs for blocks further down stream. Furthermore, an environment with linked components can be used to test the runtime realities of component interaction more comprehensively. In concert, these two characteristics could dramatically reduce the time it takes to verify software.

In a similar manner, the model compiler's ability to operate on joined components can be used to classify and optimize various software performance metrics. ROM, RAM, and CPU usage are of particular concern when the software is destined for an embedded controller. In the embedded environment, computing resources are constrained, so it becomes important to be able to track the allocation of those resources. Once characterized in a particular execution environment (specific CPU, compiler, test vector, etc.) these data can be assigned to the component. Granted, this method will be more useful for data flow oriented components, which have more deterministic schedules, than for control flow oriented components that are less timing consistent. With information detailing memory usage and performance now available at the component level, intelligent, performance-conscious analysis and optimizations can be made during the model compilation process.

5 Conclusions

We have captured the need for an automated model compiler, described some of its desired features, and discussed its application in automotive control systems development. Our aims are to clarify and express our needs (both for our partners and ourselves) and motivate a dialogue with systems researchers and tool providers. We wish to express our apologies to the research community for not conducting an extensive literature search to support the paper. Perhaps solutions similar to those proposed in this paper already exist. In any event, we look forward to a dialogue on model compiler technology development and application. For example, the maintainability and scalability of the component compatibility notions presented in section 3.2 are un-quantified and must be verified for practicality.

Acknowledgements. We would like to acknowledge support by the Defense Advanced Research Projects Agency under the DARPA MoBIES contract number: F33615-01-C-1841.

References

1. Jennings, M., Tiller, M., Butts, K., "Defining a Flexible Modeling Methodology for Design and Development of Automotive Powertrain Systems," to be published in the Proceedings of the ASME 21st Computers and Information in Engineering Conference, Pittsburgh, PA, September, 2001.
2. Cusumano, M.A.,"Chapter 5 - Toshiba: Linking Productivity and Reusability," Japan's Software Factories, A Challenge to U.S. Management, Oxford University Press, Inc., New York, 1991.
3. Sivashankar, N., Butts, K., "A Modeling Environment for Production Powertrain Controller Development," Proceedings of the 1999 IEEE International Symposium on Computer-Aided Control System Design, Hawaii, August 1999.
4. Sudkamp, T. A., Languages and Machines, An Introduction to the Theory of Computer Science, Second Edition, Addison Wesley, 1998.
5. Lee, E. A., Xiong, Y., System-Level Types for Component-Based Design. Technical Memorandum UCB/ERL M00/8, University of California at Berkeley, February 2000.
6. Lee, E. A., "What's Ahead for Embedded Software?," IEEE Computer, September 2000, pages 18-26.
7. Model-based Integration of Embedded Software, Information Technology Office, Defense Advanced Research Projects Agency, http://www.darpa.mil/ito/research/mobies/.
8. Butts, K., "Analysis needs for Automotive Powertrain Control," Proceedings of the 7th Mechatronics Forum International Conference, Atlanta, Georgia, September 2000.

Embedded Control: From Asynchrony to Synchrony and Back

Paul Caspi

Verimag–CNRS
http://www-verimag.imag.fr/
caspi@imag.fr

Abstract. We propose in this paper a historical perspective of programming issues found in the implementation of control systems, based on the author's observations for more than fifteen years, but especially during the Crisys Esprit project. We show that in contrast with the asynchronous tradition of computer scientists, control engineers were naturally led to a synchronous practice that was later formalised and generalised by computer people. But, we also show that, for the sake of robustness and distribution those practitioners had to incorporate some degree of asynchrony in this synchronous approach and we try to comment the resulting programming style.

1 Introduction

The history of computer implementations of control systems is really an interesting one marked by several unexpected accomplishments, among which we can cite:

- The invention, by control engineers, of their own programming languages in place of those designed for them by computer scientists.
- The extraction by computer scientists of those inventions, for long buried into in-house products, and their promotion to the status of a new programming paradigm called "Synchronous programming".
- The use of simulation tools like Matlab/Simulink as programming languages.
- etc.

Having been involved in this history [20], we were aware of this complicated landscape but we did not still know how far it could go. In the course of an Esprit project (Crisys (97–01)), we had the occasion of observing several achievements in the domain of distributed control systems:

- The Airbus "fly-by-wire" system.
- Schneider's safety control and monitoring systems for nuclear plants.
- Siemens' letter sorting machine control,

and several other distributed safety-critical control systems we had previously studied. We had then the surprise of noticing that in most systems, synchronous

T.A. Henzinger and C.M. Kirsch (Eds.): EMSOFT 2001, LNCS 2211, pp. 80–96, 2001.
© Springer-Verlag Berlin Heidelberg 2001

programming tools are used in an *asynchronous programming style* for quite obvious reasons of robustness, thus returning in some sense to the initial computer science programming styles that had been initially rejected.

This paper intends to trace back the history of this "surprise". It is organised as follows:

- At section 2 we briefly describe the basic needs of the domain and show how the computer science answers to these needs were at least partially unsatisfactory. Then we show on which grounds practitioners developed their own concepts that were later organised within the "synchronous programming" school of thought.
- Then we show at section 3 how this later programming paradigm is also inadequate when dealing with real-time and distribution.
- Finally, we describe at section 4 the "asynchronous-synchronous" programming style that is used in practice.

2 From Asynchrony to Synchrony

2.1 Basic Needs of the Domain

It has been recognised for long that programming control systems needs to address among others, the following problems:

Account for parallelism. There are two basic reasons for this need: on the one hand (we shall see an example of this question at section 4.1), the control program runs in parallel with the environment it aims at controlling, and studying, synthesising, debugging, testing and formally verifying the control program requires running, in one form or another, some kind of a model of this environment. Thus the programming language must provide some notion of parallelism. On the other hand, in most cases, the environment has several degrees of freedom that must be controlled in parallel. For instance, in an aircraft, the pitch and the roll must be controlled *at the same time.* This is another reason why the programming language must provide means of naturally describing these activities in parallel.

Provide guaranteed bounds on memory and execution time. Most control systems exhibit hard real-time requirements and are at the same time safety critical ones. It is thus very important that these requirements be met in some sense *by construction* or, at least that their checking be made as easy as possible.

Allow for distribution. Finally most of these systems are distributed ones, for evident reasons of load, location of sensors and actuators and fault-tolerance (redundancy). The programming environment should then also provide facilities for that purpose.

2.2 The Computer Science Answer: Real-Time Kernels and Languages

At the end of the seventies, computer scientist became aware of these requirements and began looking at ways of fulfilling them. Quite uniformly, their proposals were based on the experience they had of programming parallel and distributed activities and most of this experience came from time sharing operating systems. In these systems the main problem was to regulate the concurrent access of several users to computing resources. From this need came the structuring concepts of:

- Synchronisation: semaphores, monitors, sequential processes...
- Communication: shared memory, messages, mail-boxes...
- Synchronisation + communication: queues, rendez-vous

and these structuring concepts were also the ones which structured the proposals in the field of real-time programming, for instance, the programming languages CSP [13], OCCAM, the tasking part of ADA and the many proposals of real-time operating systems.

2.3 The Evolution of Practices

By the same time, practitioners were thinking of moving from their analog controllers to computerised ones. Figure 1 shows a quite general scheme they used for it: it consists of providing a single real-time periodic clock which triggers both analog to digital converters at the input, digital to analog ones at the output and *the computing activity of the computer*. This activity can result from various programming styles, at the source level, but it appears, at the object level, quite uniformly, as a *single program* looking like the one displayed at table 1, and corresponds to what we would call now a periodic synchronous program.

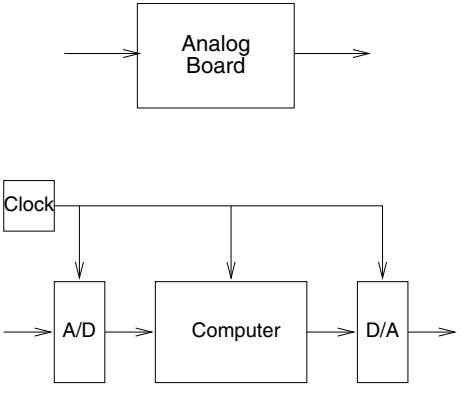


Fig. 1. From analog boards to computers

Table 1. A periodic synchronous program

```
initialize state;

loop each clock tick

   read other inputs;
   compute outputs and state;
   emit outputs

 end loop
```

This kind of implementation has a lot of practical interests:

- First, it perfectly matches the needs for solving, in real-time, the differential equations corresponding to the previous analog boards. It is known that there are many ways of solving differential equations but, if we want to do it in real time, the most practical solution consists of using forward fixed step methods which correspond to the program of table 1.
  It also matches quite closely *the mathematical theory of sampled control systems* [14], a very popular method of control.
- It is also a very simple, safe ant efficient implementation method:
  There is a single interrupt (the real-time clock ticks) and this interrupt should occur when the computer has finished the loop iteration and is idle. Thus there is no need for context saving. This means that there is very little need for an operating system and that this kind of program can run on a bare machine. This is quite safe if we think that validating an OS is a difficult task and that most standards for safety critical programming like DO178B recommend a limited use of interrupts.
  Furthermore, this programming structure eases the checks for guaranteed bounds on memory and execution time. Bounded memory is obtained by construction if the programming language does not allow for dynamic memory nor recursion. Bounded execution time amounts to checking the worst execution time of an acyclic program.

It is thus an appealing method the more so as safety critical systems are concerned and certification authorities have to be convinced. This is why it has been quite widely applied [6,16,3].

2.4 Generalisation: Synchronous Languages

This kind of practice gave birth to the so-called *synchronous programming school*. The activities of this school consisted essentially of:

- Generalising the concept of clock to any kind of event, (the multiple time scale paradigm) yielding the object code structure of table 2.

- Finding several styles of source code: data-flow [10,2], imperative [4], graphic [12,17,1]. The main shared feature of these different styles is the presence of a parallel construct which allows to address one of the requirements stated at section 2.1. It must be noticed here that, in order to yield an object code like the one depicted at table 2, this parallel construct has to be *compiled* instead of being *interpreted* like in the concurrent approaches of section 2.2.
- Equipping the approach with efficient compilers, debugging, simulation and formal verification tools [11,5].

Table 2. Synchronous programming

```
initialize state;

loop each input event

    read other inputs;
    compute outputs and state;
    emit outputs

end loop
```

Yet, it should also be noticed that, in practice, most applications of synchronous programming in the control domain are actually periodic ones.

2.5 Milner's SCCS Theory

As for the theoretical foundations of the synchronous approach, we think that it had been *a priori* provided by Robin Milner's SCCS [19]. It was based on the synchronous product of automata and figure 2 shows how this product contrasts with the asynchronous one of CCS. As a the nice feature of this theory, Milner shows that SCCS can simulate CCS and thus that CCS is a *sub-theory* of SCCS. This, in our opinion provides some theoretical support to control practices: this means that synchronous primitives are stronger than and encompass asynchronous ones. If it is so, why should those practitioners have adopted the computer science asynchronous proposals and have limited themselves to weaker primitives?

2.6 Further Justifications

To this landscape Gérard Berry and the Esterel team added several interesting remarks [4]:

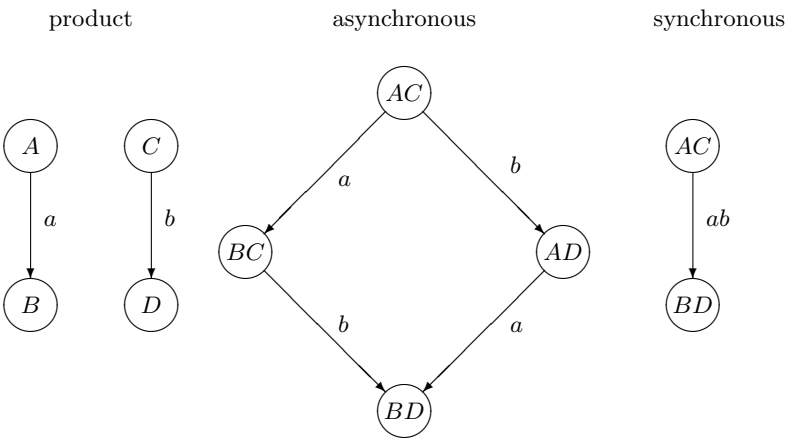


Fig. 2. Asynchronous and synchronous products of automata

- First they remarked that the synchronous product is deterministic and yields less states than the asynchronous one. This has, in their opinion two valuable consequences: programs are more deterministic and thus yield easier debugging and test and have less state explosion and thus yield easier formal verification.
- They also remarked that a step in the synchronous product corresponds exactly to one step of each of the component automata. This provides a "natural" notion of *logical time* shared by all components which allows an easier and clearer reasoning about time.

We thus see that synchronous programming is endowed with both a rich theory and many pragmatic interests and this seems to justify its being largely used in practice, even by people who don't know about it. But ...

3 Some Drawbacks of Synchronous Programming

But the synchronous model is not without problems when applied to control. We discuss here two of them, the first one which is due to hybridity, *i.e.*, the fact that the environment of a control program which provides it with inputs and receives its outputs does not evolve according to logical time but to real time, and the second one which is due to distribution and amounts to the fact that, in many implementations, control programs do not only sample their environment *but also the other control programs* which it cooperates with. In this sense, the latter is just a generalisation of the former.

3.1 Real-Time Is Not Logical Time

The phenomenon can be seen at the two boundaries of the control system, *i.e.*, inputs and outputs:

Sampling inputs. Figure 3 shows two possible periodic samplings of the same couple of inputs. In the first one, a and b are seen to raise at the same logical instant while, in the second one, this is not the case. Clearly, a control program should be in some sense insensitive to this sampling phenomenon and, clearly also, synchronous programming does not provide specific means to ensure it.

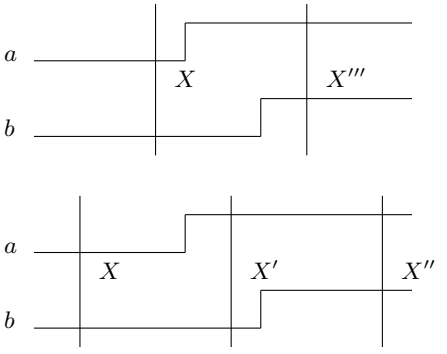


Fig. 3. Sampling is non deterministic

Outputs. Let us consider the following requirement for a control system:

y and z should never be true at the same time

and assume a designer provides you with a solution yielding the chronogram of figure 4(a). Would you be satisfied with this solution, even if he formally proves that it is correct? It is likely that you wouldn't, because, even if the solution is perfectly correct in logical time, it doesn't exclude the possibility of z and y being simultaneously true for some small real-time intervals.

This is why a smart designer is likely to provide you with a solution yielding, for instance, the chronogram of figure 4(b) which ensures you that your desired property holds both in logical and real time.

3.2 Distribution

We have seen at section 2.3 how control engineers moved from analog controllers to computers. But, in general, large control systems like a commercial aircraft flight control or a chemical plant control are made of more than one board and figure 5 shows how, in most cases, they did when dealing with networks of analog boards.

The idea here was to reuse repeatedly the replacement scheme of figure 1, but for the case when two adjacent boards were replaced by computers. Then

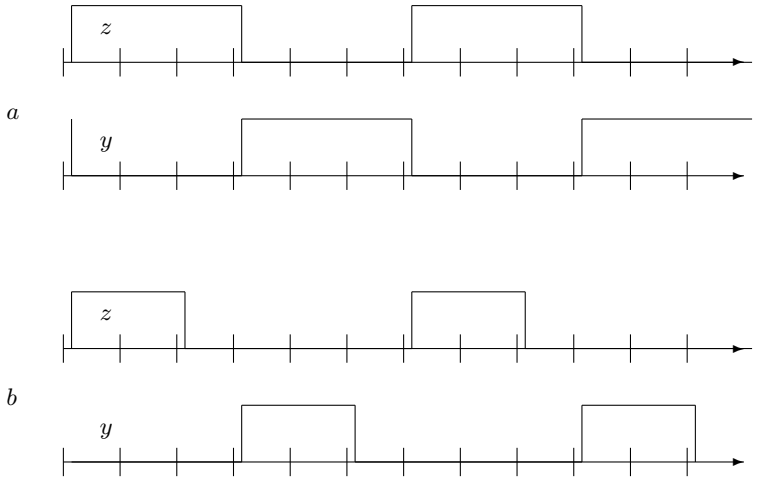


Fig. 4. Non robust (a) and robust (b) mutual exclusion

the previous analog communication was replaced by serial lines and in further evolutions, by so-called field busses [9].

This method has many advantages:

- It is modular, *i.e.*, it allows to progressively replace elements, according to the needs, and even to let several technologies cooperate smoothly.
- It can be seen as robust:
  Each computer is a complete and autonomous one, including its own real-time clock and even, possibly, its own power supply.
  Communications between computers are based on periodic readings and writings. In some sense, they are similar to the communications between computers and environments, that is, based on periodic sampling and, as it, are non blocking. Thus liveness, at least at low level, is not a problem.
- Finally, it is cost effective, in that it does not need too much specialised hardware. This allows to follow the technological advances at lower cost.

However, it also has its drawbacks. These are summarised at figure 6 which shows a possible behaviour of the real-time clocks of two computers: even if these clocks have been initially set to the same period and phase, this exact situation cannot be maintained in time because no resynchronisation is provided here, in contrast with the so-called *Time Triggered Architecture* [15]. Even if the the periods are assumed to show very limited variations, [1] some problems cannot be avoided:

[1] In most of the systems considered here, clock periods should be strictly monitored because otherwise even relative time would be lost yielding unrecoverable consequences on system safety [16].

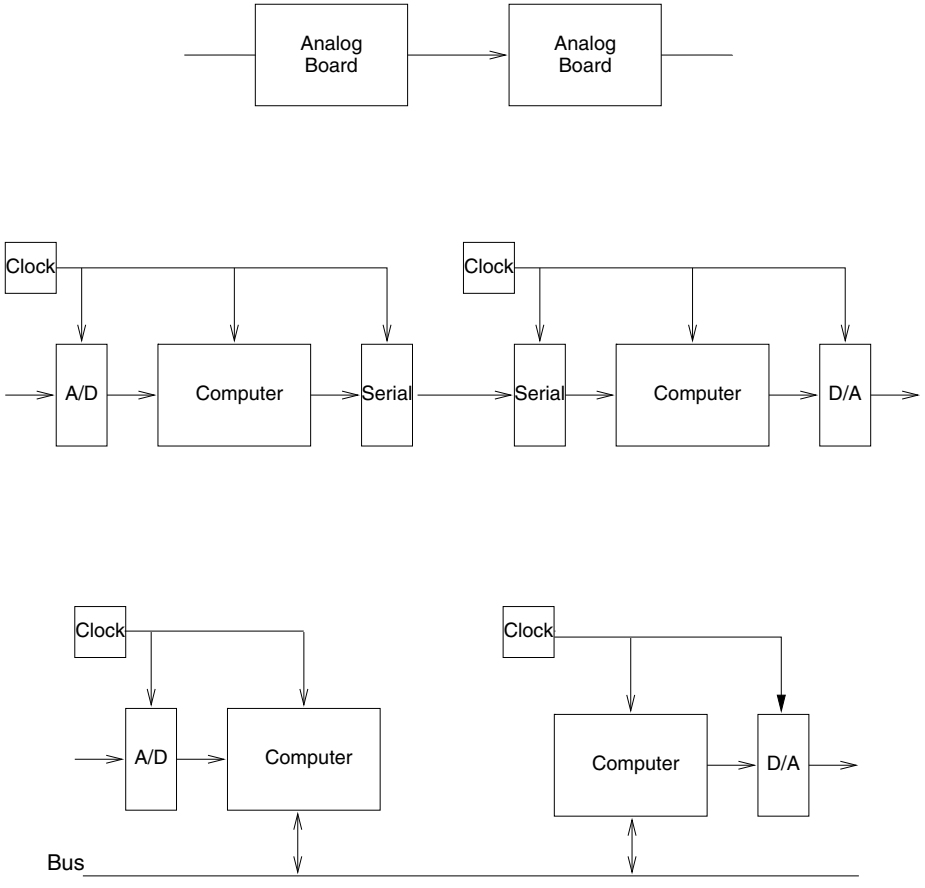


Fig. 5. From networks of analog boards to local area computing networks

- Communication errors occur: it may be that two write clocks take place strictly between two read clocks. Then the first write is overwritten and lost. Conversely, data duplication can take place. Moreover, there are non deterministic communication delays. However these delays are bounded: the worst delay occurs when a read clock occurs just before a write clock. Then the written value has to wait another read period before being read.
- Absolute time is lost.
- Finally, this situation amounts to some kind of bounded fairness:

  It is not the case that a component process executes more than twice between two successive executions of another process.

We thus see here that some care has to be taken when using the synchronous paradigm in programming control systems, because some degree of asynchrony

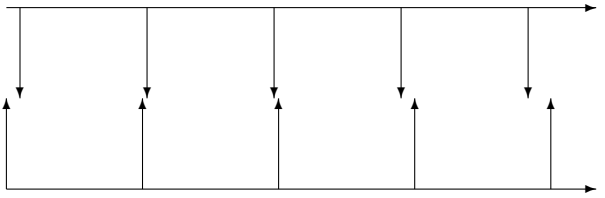


Fig. 6. Two periodic clocks with nearly the same period

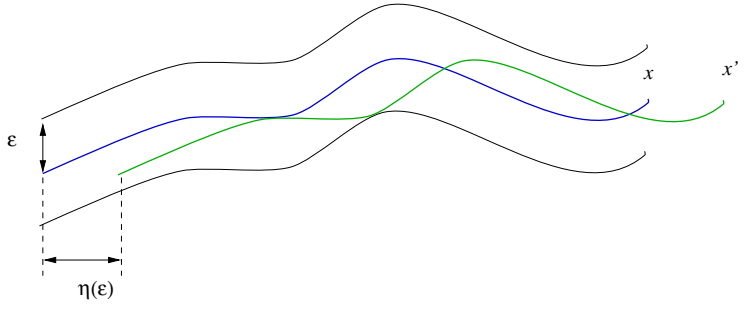
has to be incorporated in it for real-time and distribution reasons. In the following section, we shall see how people use to handle it in practice.

4 Asynchronous–Synchronous Programming

Control systems and programs are usually concerned with two types of computations, continuous ones and discrete ones and also with the interaction of both. These will be examined in sequence:

4.1 Continuous Signals and Systems

Figure 7 shows the basic uniform continuity signal property used in this framework. This property says that given an arbitrary maximum error, we can choose a maximum delay such that, staying within this delay ensures staying within this error.

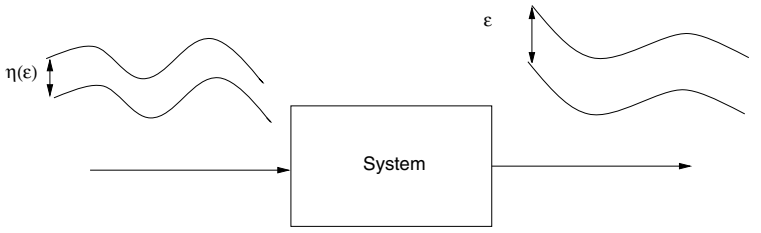


$$\forall \epsilon > 0, \exists \eta > 0, \forall t, t', |t - t'| \leq \eta_x \Rightarrow |x(t) - x(t')| \leq \epsilon$$

Fig. 7. An uniformly continuous signal

Now, this property can be used in conjunction with the corresponding property of systems, depicted at figure 8, which says that a system is uniformly

continuous if, given an arbitrary maximum output error, there exists a maximum input error such that, if the input stays within the latter, the output will stay within the former.



$$\forall \epsilon > 0, \exists \eta > 0, \forall x, x', ||x - x'||_\infty \leq \eta \Rightarrow ||f(x) - f(x')||_\infty \leq \epsilon$$

Fig. 8. An uniformly continuous system

Both properties then combine nicely thanks to the following theorem:

Theorem 1. *An uniformly continuous and time-invariant system, fed with uniformly continuous signals, outputs uniformly continuous signals.*

Thus the property also propagates through acyclic networks of such systems connected by bounded delays in such a way that one can find bounds on delays and on input errors such that output errors remain within given bounds.

This seems to give us a satisfactory theory of robust computations over continuous signals. However, let us note here that the landscape is a bit more complex because even very simple controllers like for instance very popular PIDs do not enjoy the uniform continuity property.[2] This situation is recovered within the framework of closed-loop system: unstable controllers are used in order to stabilise the environment in such a way that the closed-loop behaviour of the system appears stable and hence uniformly continuous (Figure 9).

4.2 Uniform Bounded Variability Signals and Combinational Systems

When we move from continuous to discrete signals, we face the problem that errors cannot be given arbitrary small values. The idea is then to only reason about delays.

Figure 10 illustrates the concept of uniform bounded variability which appears the analogue of uniform continuity for discrete signals. A signal has this property if there exists a least *stable time* between two successive discontinuity points.

[2] This is due to the integral part which accumulate errors without ever forgetting them.

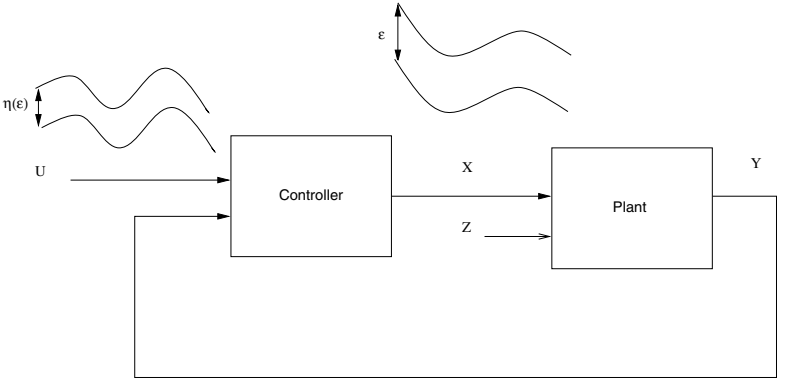


Fig. 9. The closed-loop system computes uniformly continuous signals

Fig. 10. Uniform bounded variability

In this context, one could expect that combinational functions play the part of uniformly continuous systems in the continuous framework. Unfortunately, this is not the case because delays do not combine nicely as errors do. In particular, as soon as we deal with functions of several variables, we face the problem that independent delays on tuples do not yield delayed tuples. This situation is illustrated at figure 11 where we can see that the value $x' = 0, y' = 1$ does not correspond to any value in the original tuple.

A solution to this problem can be found in the confirmation functions [8] shown at table 3 which are used in order to transform incoherent delays into coherent ones. The idea of this function is that, if we know bounds on the delays for each component of a tuple and if these components do not vary too fast (thanks to uniform bounded variability), then, each time a component of the tuple changes, we wait at least for some delay before outputting the change: if the other component of the tuple has not changed meanwhile, we can output the tuple value and be sure that this tuple is a delayed value of the original tuple. This can be summarised as:

Theorem 2 (Confirmation). *Given x', y' two bounded delay images of two uniform bounded variability signals x and y, one can find bounds on the delays, on the clock period and on nmax such that $Confirm_{nmax}(x', y')$ is a bounded delay image of the tuple (x, y).*

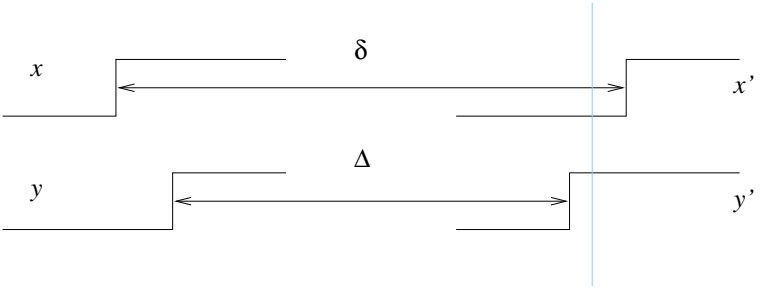


Fig. 11. Delays on tuples do not yield delayed tuples

Here also, we can prove that the uniform bounded variability property propagates through acyclic networks of combinational functions connected by bounded delays and conveniently guarded by confirmation functions in such a way that one can find bounds on the delays, confirmation function parameters, and stable times of the inputs such that the outputs have given stable times and delays (with respect to the ideal undelayed computations). In some sense, theorem 2 appears as an analogue of theorem 1.

Table 3. Confirmation function

```
xp := x0; xpp := x0; n := 0;

loop each clock tick

    read x ;
    if x = xp
    then  if n >= nmax
          then xpp := x; n := 0
          else n := n+1
          end if
    else  xp := x; n := 0
    end if ;
    emit xpp

end loop
```

4.3 Robust Sequential Systems

Unfortunately, this approach does not work as soon as we consider sequential systems. Figure 12 illustrates the critical race phenomenon [7] which forbids it: here we see two state variables computed in distinct locations. In a synchronous framework, both variables are computed at the same step. On the contrary, when each location has its own clock, if there is a dependency between these variables, the resulting state may depend on the order into which these variables are computed.

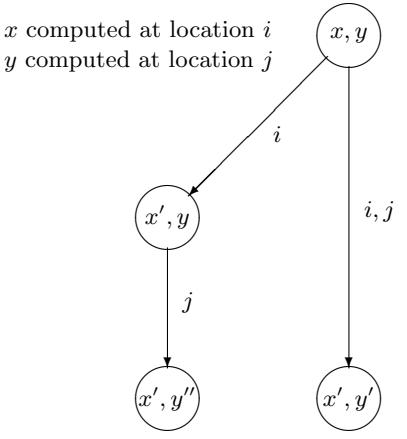


Fig. 12. A critical race

In this sense, sequential systems look very much like continuous unstable systems and cannot be directly implemented in a robust way. There are several ways for checking and enhancing the robustness of sequential systems:

- First, if state variables computed on distinct locations are independent, the critical race phenomenon does not take place and we are left to the combinational situation.
- Another interesting situation appears when dependent state variables cannot vary at the same step. Here also, there is no critical race. Now, there can be two reasons why state variables cannot vary at the same time:

  Timing reasons. If it is not the case, designers can add delays in order to robustify their programs.

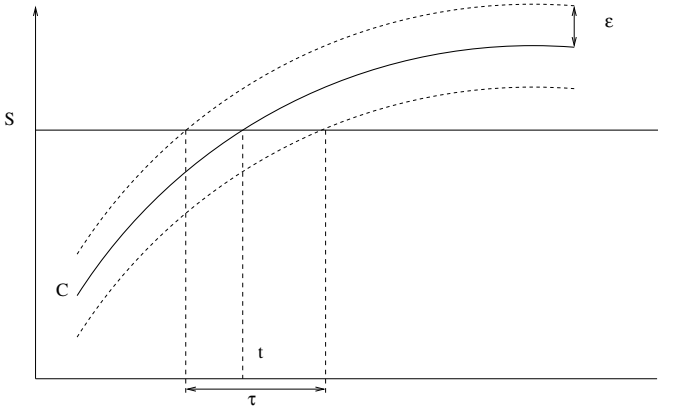
  Causality reasons. For instance, in the mutual exclusion example of figure 4, we can transform the non robust program into the robust one by deciding that:

  y cannot raise but when z has gone down and conversely

When this is used in conjunction with the mutex requirement stating that y and z cannot raise at the same time, we clearly forbid y and z to change at the same time, thus forbidding any race.
Inserting causality chains disallowing races is thus a way of robustifying programs. This kind of programming is reminiscent of typical asynchronous programming methods like so-called *Message Sequence Charts* [18].

4.4 Mixed Systems

Now, more and more systems are mixed ones, and there is not clear way of dealing with them. Figure 13 illustrates the situation where a boolean signal is generated which changes when some continuous signal crosses a threshold. In our framework, where errors are associated with continuous signals, we can see that some error on the continuous signal induces some delay on the corresponding boolean. This fits quite nicely in our *error/delay* framework but for the fact that the relation between errors and delays is non-linear: if the derivative at the crossing point vanishes, the delay can get unbounded!



$$\tau = \frac{2\epsilon}{|\frac{dC}{dt}(t)|}$$

Fig. 13. Threshold crossing

5 Conclusion

In this paper we attempted to illustrate the evolution of ideas in control system programming. Starting from the early real-time languages and operating system propositions, we showed that control engineers elaborated their own practices which where generalised by computer scientists and gave birth to the synchronous paradigm.

But we also showed that for the sake of robustness and in particular distribution, some asynchrony had to be incorporated in the synchronous approach and

we briefly reviewed the methods and theories used in practice for that purpose. What we can see here is that these methods and theories do not constitute, at present, a very mature and well established framework, and many efforts are still required in order to strengthen it and equip it with CAD tools.

Among the pending questions, we can cite:

- How could we merge together synchronous and asynchronous paradigms in order to obtain *robust by construction* distributed systems, while keeping the advantages of synchronous programming ?
- How can we encompass timing and causality within a coherent theory of robust discrete systems.
- How can we unify the continuous and discrete theories in order to obtain a satisfactory theory of mixed (hybrid) systems ?

Acknowledgments. The author is indebted to his colleagues of the Crisys project, mainly R. Salem from Verimag, M. Yeddes and R. David from Grenoble's Automatic Control Laboratory (LAG), C. Bodennec, C. Mazuet and N. Raynaud from Schneider Electric, R. Budde and A. Poigné from GMD, R. Mercadié from EADS-Airbus, for their contributions to the project and to the ideas proposed here. H. Kopetz, as a reviewer of the project, played a very important part in it by his always stimulating questions and many passionate discussions with O. Maler, E. Asarine, S. Tripakis from Verimag, J. Pulou from FranceTelecom and, last but not least, A. Benveniste from INRIA/IRISA, also contributed to the paper.

References

1. C. André. Representation and analysis of reactive behaviors: a synchronous approach. In *Proc. CESA'96*, Lille, July 1996.
2. A. Benveniste, P. LeGuernic, and Ch. Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Science of Computer Programming*, 16:103–149, 1991.
3. J.L. Bergerand and E. Pilaud. SAGA; a software development environment for dependability in automatic control. In *SAFECOMP'88*. Pergamon Press, 1988.
4. G. Berry and G. Gonthier. The ESTEREL synchronous programming language, design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.
5. G. Boudol, V. Roy, R. de Simone, and D. Vergamini. Process algebras and systems of communicating processes. In *Automatic Verification For Finite States Systems*, volume 407 of *Lecture Notes in Computer Science*. Springer Verlag, 1990.
6. D. Brière, D. Ribot, D. Pilaud, and J.L. Camus. Methods and specification tools for Airbus on-board systems. In *Avionics Conference and Exhibition*, London, December 1994. ERA Technology.
7. J. A. Brzozowski and C-J. H. Seger. *Asynchronous Circuits*. Springer-Verlag, 1995.
8. P. Caspi and R. Salem. Threshold and bounded-delay voting in critical control systems. In Mathai Joseph, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 1926 of *Lecture Notes in Computer Science*, pages 68–81, September 2000.

9. A. Chatha. Fieldbus: The foundation for field control systems. *Control Engineering*, pages 47–50, May 1994.
10. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.
11. N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE. *IEEE Transactions on Software Engineering*, 18(9):785–793, september 1992.
12. D. Harel. Statecharts: a visual approach to complex systems. *Science of Computer Programming*, 8(3), 1987.
13. C.A.R. Hoare. Communicating sequential processes. *Communication of the ACM*, 21(8):666–676, 1978.
14. K.J.Åström and B.Wittenmark. *Computer Controlled Systems*. Prentice-Hall, 1984.
15. H. Kopetz, A. Damm, Ch. Koza, M. Mulazzani, W. Schwabl, Ch. Senft, and R. Zainlinger. Distributed fault-tolerant real-time systems: the MARS approach. *IEEE Micro*, 9(1):25–40, 1989.
16. G. LeGoff. Using synchronous languages for interlocking. In *First International Conference on Computer Application in Transportation Systems*, 1996.
17. F. Maraninchi. Operational and compositional semantics of synchronous automaton compositions. In *Proc. of CONCUR'92*, volume 630 of *Lecture Notes in Computer Science*. Springer Verlag, August 1992.
18. S. Mauw. The formalization of message sequence charts. *Computer Networks and ISDN Systems*, 28:1643–1657, 1996.
19. R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25:267–310, 1983.
20. P.Caspi. What can we learn from synchronous data-flow languages. In O.Maler, editor, *Hybrid and Real-Time Systems*, volume 1201 of *Lecture Notes in Computer Science*, pages 255–258. Springer, 1997. invited conference.

Verification of Embedded Software: Problems and Perspectives*

Patrick Cousot[1] and Radhia Cousot[2]

[1] École normale supérieure
Département d'informatique
45 rue d'Ulm
75230 Paris cedex 05, France
Patrick.Cousot@ens.fr
http://www.di.ens.fr/~cousot/

[2] Laboratoire d'informatique
CNRS & École polytechnique
91128 Palaiseau cedex, France
rcousot@lix.polytechnique.fr
http://lix.polytechnique.fr/~rcousot/

Abstract. Computer aided formal methods have been very successful for the verification or at least enhanced debugging of hardware. The cost of correction of a hardware bug is huge enough to justify high investments in alternatives to testing such as correctness verification. This is not the case for software for which bugs are a quite common situation which can be easily handled through online updates. However in the area of embedded software, errors are hardly tolerable. Such embedded software is often safety-critical, so that a software failure might create a safety hazard in the equipment and put human life in danger. Thus embedded software verification is a research area of growing importance. Present day software verification technology can certainly be useful but is yet too limited to cope with the formidable challenge of complete software verification. We highlight some of the problems to be solved and envision possible abstract interpretation based static analysis solutions.

1 Introduction

Since the origin of computer science, software in general, whence embedded software in particular, expands continuously to consume available processor cycles and memory. The exponential complexity growth in VLSI with decreasing or constant costs is therefore accompanied, maybe with a delay of few months or years, by a corresponding proportional growth in software. So an operating system running a large number of applications which crashes on present day computers every 24 hours will crash every 30 minutes within a decade, probably

* This work was supported in part by the RTD project IST-1999-20527 DAEDALUS of the european IST FP5 programme.

T.A. Henzinger and C.M. Kirsch (Eds.): EMSOFT 2001, LNCS 2211, pp. 97–113, 2001.
© Springer-Verlag Berlin Heidelberg 2001

because of software and not hardware faults. If the present software bug rate is preserved or slightly decreased only, but the size of software is multiplied by a factor of ten, then the computer system might even be expected to crash every three minutes. Embedded software is presently simpler than operating systems but complexity is also growing rapidly in this area. Similar failure rates leads to software crashes every few hours, which is hardly acceptable for safety critical systems, even fault tolerant ones [1]. It follows that verification techniques whether formal or informal must scale up in similar proportions, indeed at a much higher rate since the software verification cost is well-known not to be linear in the software size. We highlight some of the problems to be solved and envision possible abstract interpretation based static analysis solutions.

2 Formal Methods

Formal methods such as theorem proving based deductive methods [76], model checking [19] and program static analysis by abstract interpretation [26] have all had success stories.

The embedded software for the driverless METEOR line 14 metro in Paris was formally designed with the B-method [3]. The 115 000 lines specification written in B compiles into a 87 000 lines ADA program. The correctness proof, using interactive theorem proving, required to handle manually 27 800 proof obligations. For that purpose, 1400 rules had to be added to the prover and proved correct, 900 of which automatically. Since the metro is running, no error was ever claimed to be found in the embedded software nor in its B specification. Indeed all errors, if any, could only be found at the interfaces, the specification of which might not have been satisfied by the central control software (not developped in B and itself potentially subject to errors). One may wonder why, after such a successful experience, theorem proving based formal methods are not standard for the design of safety critical embedded software. If the circulating figures of 600 person/years are not exagerated this might be because of the human cost of the software development process.

The ARIANE 5 flight 501 failure was due to the inertial reference system sending incorrect data following a software exception. This overflow exception was caused by an unprotected data conversion from a too large 64-bit floating point to a 16-bit signed integer value. Not all such conversions were protected because a maximum workload target of 80% had been set for the inertial reference system computer. Ironically, the exception was lifted in a part of the software which serves no purpose after the ARIANE 5 launcher lifts off (but was previously required for ARIANE 4). An erroneous reasoning based upon physical limitations and large margins of safety lead to the decision to leave variables unprotected. Unfortunately, the overflow was caused by an unexpected high value because the early part of the trajectory of ARIANE 5 differs from that of ARIANE 4 and results in considerably higher horizontal velocity values. The exception caused the inertial reference system processor to shut down which finally proved fatal [68]. The origin of the error was caught (afterwards) by an abstract interpretation

based [29,30,32] static analysis of the program [65]. Unfortunately, automatic static analysis relies on approximation, so not all software errors will ever be caught statically in this way. There always exists an approximation to prove a given specification of a given computer system semantics/model but discovering this abstraction is logically equivalent to a formal correctness proof [27]. So one either has to manually design the abstraction (often in the hidden form of a model) or to consider general-purpose reusable abstractions which will always be too abstract to proof some peculiar functional specification.

The most industrialized of the formal methods is certainly model checking [16,74]. After the famous FDIV design fault in the Pentium processor, most hardware design companies now have model checkers [6,13]. Present-day hardware model checkers can verify circuit designs of a few hundreds of registers (with abstraction of their surrounding environment). Model checking proceeds by exhaustive enumeration of the state space and is therefore subject to state space explosion: although the checking algorithm may be linear in the size of the specification formula and that of the state space, the state space size often grows exponentially with the size of the description of the model (usually given in the form of a program in some computer language). Despite various symbolic representation techniques using BDDs [12] and their numerous variants, symmetry reduction [17], modular decomposition [62], breadth-first checking with the SAT procedure [9] etc. model checking still has to scale up for hardware, not speaking of software. Difficulties also come out of the temporal logic used for the specification which is often beyond human understanding capabilities [64, 69]. Most of the success of model checking is not so much in the formal verification of refined functional specifications (always subjects to errors in the design of the model and/or specification) but in the finding of bugs not found by other informal methods (such as testing or simulation). Such partial model checking techniques only explore part of the state space (testing or simulation do follow exactly the same principle) thus avoiding the exploration (see e.g. the random pruning of the search space in [56]). Debugging is done at the price of soundness, which is considered abusive by some, practical by others and sometimes is misunderstood.

Despite all these successes, debugging, simulation and run-time tests (using redundant computations to detect faulty numerical computations or to check at run-time that the path traversed is legal) are still the essential computer aided methods in embedded software validation and verification. So the present success of formal methods (mainly in hardware design) is still problematic to scale up for software.

3 Challenges in Embedded Software Verification

3.1 Software Models

Programming Language Semantics. Standard models in software are called semantics [2]. They formalize program execution in abstract mathematical terms. Obviously a programming language semantics can serve as a basis for the analysis

and verification of software written in this language. In practice, there are nevertheless many difficulties. Even if (informal) standards do exist (see e.g. ANSI C [59]), most compilers do not strictly implement these specifications. Moreover the standards are continuously revised [58]. An example of change in [60] is "An array subscript is out of range, even if an object is apparently accessible with the given subscript (as in the lvalue expression `a[1][7]` given the declaration `int a[4][5]`) (6.3.6)." Obviously the (probably erroneous) behavior of programs may be completely modified by such an update of their semantics! Programming environments also include many large libraries which semantics is often only very informally specified. Consequently the semantics of a programming language is often that specified by a compiler on a given machine for specific libraries, which is hardly understandable. In the best case, the consequence of this situation is that program verification tools try to conform to standards and therefore do not fully conform to practice. Nevertheless formal tools based upon programming (or specification) language semantics (or an abstract interpretation of this semantics) have the obvious advantage of providing automatically a model of the program to be verified.

Problem Driven Abstractions for Model Checking. In model-checking, the model is assumed to be given and the verification is relative to that model [20]. The model should preserve only selected characteristics of a real-world artifact, while suppressing others so as to abstract away from the too complex real-world system or program. This abstraction is done informally (or uses abstract interpretation of an already existing more refined model). The requirement to design a model to enable program verification leads to three different descriptions of the real-world system or program: – 1 – in a programming language for the implementation; – 2 – in a verification language for the model and – 3 – in a logic language for the specification of the properties of the model which have to be checked [55]. So the specification is valid for the implementation only if the model is faithful, which is seldom checked (but could be using abstract interpretation to prove the model to be an abstraction of the implementation semantics). Abstraction is sometimes considered in model checking [21], but this is often between a concrete model and a more abstract model thus requiring at least a fourth level in the abstract description of the implementation. Often the concrete model is already assumed to be finite (although too large to be automatically checked) so that the abstraction and concretization functions are now computable. In this context refinement is computable [18], which is not the case in general for the semantics of usual programming languages [49].

Abstraction. More generally the concrete model is not finite, at least if the most concrete model is considered to be the semantics of the implementation and this implementation is described by programs written in a realistic programming language. The question is then whether the abstract model should be finite or infinite. For the verification of a given infinite concrete model, a finite model will always be adequate [27]. This leads to the idea of automatizing the

design of the abstract model from the concrete one, using deductive methods to prove its soundness (e.g. [78]) The difficulty is then that the discovering of the abstraction is logically equivalent to the discovery of an inductive argument (e.g. an invariant) and that the proof that the abstraction is sound is logically equivalent to an inductive proof (e.g. through invariance verification conditions) [27]. Otherwise stated the correctness of the concrete model can always be established by checking a finite abstract model, but the discovery and proof of soundness of the required abstraction is logically equivalent to a direct correctness proof of this concrete model [27]. It is hoped that this will globally simplify the proof (because abstractions like partitioning will decompose the global proof into many local ones [24]). Unfortunately, the soundness proof of the global/local abstractions (which is undecidable) is much more difficult than checking the abstract model (which is finite). The whole difficulty is now in the choice (and soundness proof, if any) of the abstraction so that the benefit is not always clear. Moreover, the whole abstraction/checking process has to be redone after each modification of the program. This is certainly a difficulty for embedded software which often evolves slowly over a long period of times (sometimes up to 20 years). It is therefore necessary to anticipate how the model will be maintained and modified along with the program.

Standard Abstractions for Program Analysis. For model checkers, the initial abstraction out of the embedded software is provided in the form of an often finite model for a given program. In static program analysis, the model of the program to be verified and its abstraction are provided by the analyzer and proved correct for a given programming language. So the user does not have to extract a verification model from his program but only to choose among predefined abstractions. Since analyzers must work for infinitely many programs, it is shown in [36] that no finite abstraction will be as powerful as infinite abstract domains with widening/narrowing [29,30] for Turing equivalent programming languages. A broader class of general-purpose abstractions, implemented in the form of libraries, is needed. The elimination of false alarms through the automatic choice of the appropriate abstract domain is still opened.

Widening/Narrowing and Their Duals. Infinite abstract domains not satisfying chain conditions do require the use of widening/narrowing techniques [29,30] in order to accelerate the convergence of fixpoint computations into approximations from above or to choose among alternatives in absence of a best approximation. Dual notions do exist for approximations from below [23]. Widening/narrowing techniques are also used in model checking although the link with these well-known techniques is not always recognized (e.g. compare the widening for BDDs of [66] to that of [71]).

The widening/narrowing technique is a dynamic approximation technique (during fixpoint computation) whereas abstraction is a static one (before fixpoint computation, at the time the model/abstract semantics is designed) [30]. All abstractions can be expressed as widenings [34] so abstraction is required only

to ensure the existence of an efficient computer-representation of the properties in static analysis and of an initial model in model checking. Otherwise, one can always represent properties as terms although this is not quite adequate in practice since powerful widenings are based on the semantics and the geometry of the fixpoint computation. Widenings based on thresholds (for example the widening to a finite domain [53] in static analysis or the limitation of reachability at a certain depth in model-checking [9]) are equivalent to static abstraction so are not very expressive [34]. Dynamic widenings could be better exploited in model checking to cope with the state space explosion problem, the same model being explored several times at different levels of abstractions determined dynamically by widenings.

3.2 Specifications

The specification language in model checking is typically a temporal logic [16] or a fixpoint calculus [63]. In program analysis, the specification is either provided automatically (e.g. a standard example is absence of run-time errors [32,24]) or provided by the user for abstract testing [11,24,39]. In both cases, the forward/backward and least/greatest fixpoints based static analysis/checking methods are not so different. Since the design of a model for a program is an abstraction in the sense of abstract interpretation, we can establish the following comparison:

		Specification	
		Program-dependent	Language dependent
Abstraction	Program-dependent	Model checking	—
	Language dependent	Abstract testing	Static Analysis

Obviously, one can also think of *Static Checking* where a program-dependent model is designed to check for language dependent properties for which standard abstractions may be a problem (such as threads must eventually enter/exit critical sections, the condition in monitors will eventually be verified for condition variables, etc.).

3.3 Control Structures

The flat modelling of control structures by transition systems initially considered in program analysis [32] and model-checking [16,74] is valid for some programming languages (like Prolog [35]) but this remains an exception (e.g. for functional languages [38]). In this context the finiteness hypothesis on data structures is not enough to ensure the finiteness of the program semantics. An example is the restriction of program variables to booleans in which case it is possible to simulate a Turing machine in Pascal [25] but not in C thus enabling finite model

checking [5]. Control analysis may also require a precise data flow analysis e.g. to trace pointers to functions or handlers (see Sec. 3.5).

Even with simple control structures, control abstractions (which consist in isolating a control-flow skeleton which is void of any knowledge about data [56]), in particular of the veracity of tests and of run-time errors, is very rough and usable only for safety properties. An example of erroneous reasoning based on this abstraction is live variable in dataflow analysis [79] which is a liveness property for the control-flow model but not for the original program so that the analysis determines potentially live variables only (whence dead variables for sure). So deductive methods or model checking methods for proving liveness of the model while ignoring the program control flow (even partially e.g. by ignoring a single test) perform abstractions from above which are valid for safety but not liveness properties. For such upper abstraction models, most of the power of temporal logics over traditional program analysis methods is simply ruled out. Obviously, the dual notion of abstraction from below can also be used [22] (or both can be mixed [57]) but such lower approximation models are hardly usable to prove more than one property at a time so that different models are needed for proving different liveness properties of the same given program (alternative approaches are discussed in Sec. 3.8).

Although this might be still unfrequent, embedded software will certainly evolve towards multithreaded programming which requires both a high level of expertise together with precise analysis tools to cope with the usual accompanying control flow problems such as untrapped exceptions, race conditions, deadlocks, priority inversion, nonreentrant software, etc.

3.4 Numerical Properties

Integer Properties. The first abstractions into non-relational infinite domains [29,30] where designed to handle properties of integers. Non-relational numerical abstraction were rapidly followed by relational ones [41]. Such relational domains do scale up for static analysis provided the number of values which can be related is limited either statically at abstraction time [72] or dynamically using widenings [37].

Relational numerical abstract domains with widening have been extensively used in model checking of infinite state spaces to handle safety properties using exactly classical static analysis techniques [51].

For liveness properties, the techniques used in static analysis (inference of variant functions) and in model checking of finite systems (fixpoint approximation from below) are quite different (see Sec. 3.8 below). This is because in the context of infinite state spaces the only dual widenings which are known are based upon variant functions or on the finite prefix/suffix/intermediate exploration of a finite subset of the execution traces (which is nothing but debugging). More work is needing on that subject to cope with liveness properties of embedded software involving integer computations.

Floating Point Properties. Most present embedded software now involve floating point computations (e.g. to control a trajectory) which used to be performed with fixed precision. A consequence is the uncontrolled loss of precision of the floating-point operations. Transcendental numbers (like π and e) cannot be represented exactly in a computer, since machines only use finite implementations of numbers (floating-point numbers instead of mathematical real numbers); they are truncated to a given number of decimals. Moreover the usual algebraic laws (associativity for instance) are no longer true when manipulating floating-point numbers. This leads to bugs such as run-time errors (here for instance, uncaught numerical exceptions), but also more subtle ones about the relevance of the numerical calculations that are made which in some cases can be completely non-significant. Let us just take an example reported in [50] showing the importance of the loss of precision. On the 25th of February 1991, during the Gulf war, a Patriot anti-missile missed a Scud in Dharan which in turn crashed onto an American barracks, killing 28 soldiers. The official enquiry report (GAO/IMTEC-92-26) attributed this to a fairly simple "numerical bug". An internal clock that delivers a tick every tenth of a second controlled the missile. Internal time was converted in seconds by multiplying the number of ticks by $\frac{1}{10}$ in a 24 bits register. But $\frac{1}{10} = 0.0001100110011001100110011001100\cdots$ in binary format, i.e. is not represented in an exact manner in memory. This produced a truncating error of about 0.000000095 (decimal), which made the internal computed time drift with respect to ground systems. The battery was in operation for about 100 hours which made the drift of about 0.34 seconds. A Scud flies at about 1676m/s, so the clock error corresponded to a localization error of about 500 meters. The proximity sensors supposed to trigger the explosion of the anti-missile could not find the Scud and therefore the Scud fell and hit the ground, exploding onto the barracks.

Sophisticated semantics and history based abstractions are needed to statically analyze the origin (not only the consequences) of this loss of precision in numerical programs [50]. Note that this is completely different from the boolean verification of circuits in floating-point unit by model checking [15].

3.5 Data Structures

In model checking program data structures are most often simply ignored. However the analysis of message-passing transition systems (e.g. for communication protocols) must take message-passing queues and operations into account (often not their content), see e.g. [10]. The abstraction process, which is an abstract interpretation, often remains quite informal or on purely syntactic bases [56], which is not adequate for liveness properties (as noted in previous Sec. 3.3).

Embedded software is often written in C or ADA and uses data structures which cannot be completely ignored when verifying their correctness. An example is the encoding of control into booleans (e.g. when compiling synchronous programs to C) or enumerated types. Type casts may also have to be taken into account. A more complex example is the use of pointers, in the simplest case to pass parameters to procedures (e.g. pointers to buffers, queues, etc.) which

may yield to aliases. Any analysis or correctness proof not taking aliases into account would be incorrect. Such pointer alias analysis attempts to determine when two pointer expressions refer to the same storage location and is useful to detect potential side-effects through assignment and parameter passing (see an overview in [77]). Such memory allocated data structures are used to memorize information which must be traced in some way or another in the correctness analysis or proof. It is then necessary to study the shape of the data structures and the absence of errors in their manipulation (see e.g. [43]). A classical example of error is buffer overflow (which has been often used by attackers of operating systems). Using precise domains, it is possible to check the absence of overflows with a very low rate of false alarms [42]. Standard abstractions for data structures remain to be developped, e.g. for standard libraries.

3.6 Modularity

Modularity has been studied both in model checking and static analysis. Whereas the modules are often designed manually in model checking [62], they often follow the modular structure of the software in program static analysis. Four basic methods for compositional separate modular static analysis of programs by abstract interpretation are known [28]: – 1 – Simplification-based separate analysis (where the equations/constraints to be solved for a module are simply simplified while the fixpoint computation is delayed until the context of use of the module is known); – 2 – Worst-case separate analysis (which consists in considering that absolutely no information is known on the interfaces of the module as in the detection of all potential interactions between the agents of a part of a mobile system interacting with an unknown context [46]) ; – 3 – Separate analysis with (user-provided) interfaces (where the properties of the external objects referenced in the program part are defined by the user so that the analysis of the module can rely on that information while any use of the module must guarantee its veracity); and – 4 – Symbolic relational separate analysis (where the analysis of the module relates symbolically the local information within the module to named external objects through a relational domain as in the pointer analysis of [31, Sec. 4.2.2]). There is also a fifth category which is essentially obtained by iterative composition of the above separate local analyses and global analysis methods [28]. For example, very large software based on a library of elementary functions could be analyzed efficiently by a very precise separate (thus possibly parallel) analysis of the basic functions later reused, maybe at a lower degree of precision, for the whole program analysis [28].

3.7 Timing

Embedded software (in particular when design according to the model of synchronous languages such as Lustre [14] or Signal [8]), must be shown to satisfy timing constraints (typically execution of all simultaneous "instantaneous" actions must take less than a given upper bound, typically of few milliseconds). Modelling such timing constraints is difficult if not impossible when bounds are

tight so that characteristics of modern computers such as pipelines and cache hierarchies must be taken into account. These are numerous extensions of model-checking to handle time (such as timed automata e.g. [4]) but it would be very difficult to manually design appropriate models at the required fine grain level. Indeed in program analysis, the timing semantics can hardly be designed at the programming language source level (for which automatic concrete complexity analysis is certainly useful [47] but insufficient since constants factors do matter [7]!). In practice it is indispensable to consider the program semantics at the assembler level, that is for a given compiler and for a given processor with some hypotheses on the frequency of physical interrupts [81]. The model being automatically generated for the program, one can be confident in its correctness which is established at the assembler language level using a timed model of the considered processor (which is a difficult task). To handle loops [70], one must have an upper-bound on the number of iterations, i.e. handle termination.

3.8 Termination and Unbounded Liveness Properties

Although embedded software must usually be proved not terminate except maybe through an operator imperative interaction (which is an easily checked property through reachability), parts of the software (such as elementary loops in basic functions) must be proved to effectively terminate. This is liveness proof which is often much more difficult than safety proofs.

This is not so much the case for finite models, even with fairness hypotheses [20], since in that case the model itself is a safety property (since no loop can go through infinitely many different states) so that any liveness property of the model can be proved by proving a stronger safety property of the model.

However infinite models (e.g. traces generated by an infinite transition system) are usually not safety properties so that proofs much resort to *variant functions* [48] which are much harder to discover than invariants (since they are abstractions of traces not of sets of states [33]). The models considered in model-checking (such as timed automata [4]) are often too restricted to serve as a basis for a general approach.

The results obtained in program analysis, in particular in the context of partial evaluation [67] or for the termination of imperative [61] or functional [73] or logic/constraint [80] languages seems promising. However fairness and schedulers still have to be considered e.g. for infinite state distributed programs in a local network.

3.9 Distribution and Mobility

The evolution of critical real-time embedded software for avionics, communication, defense, automotive, utilities, space or medical industry is from centralized control to distributed control on a (e.g. Ethernet-based) local area network (LAN). For example, modern automotive, aeronautic and train transportation computer systems certainly contain or will soon contain several dozen of computers communicating on a LAN. This radically changes the programming models

which are presently used, in particular from shared-memory to message-based systems. In this context one must reason on sets of traces (such as UML sequence diagram [54]) and not on sequences of sets of states (as is implicit in most temporal specifications [40]) for which present day set-based abstractions may be inadequate [75]. Experience in this analysis of network protocols [52] is certainly useful but regularity will certainly not be the rule.

Embedded software on LANs will certainly be fully integrated within wide-area networks (such as Internet) before the end of the present decade. For example to meet the constraints resulting from continuous air traffic growth, the future air navigation systems will certainly replace the existing air traffic control systems by effective traffic management that relies on flight path negotiations between the ground and the aircraft to reduce pilot and controller workload which implies network communications e.g. through communication satellites. A characteristic of network software is mobility, at least to replace online components by new ones but also as a communication mean, which implies continuous changes in the communication topology. The proof of non-trivial properties of mobile systems of processes definitely involves unbounded recursive structures which are hard to analyze using uniform models where all instances of processes are merged independently of their instances. Relational abstractions with counting are necessary to obtain precise results [45,44].

3.10 User Interfaces

Tools based on formal methods may require a profound understanding of the methods (e.g. all tools are incomplete so that the user will eventually come with questions that the formal tool cannot fully answer in which case the user will want to understand why no definite positive/negative answer is produced, which even for simple formal systems as simple as type systems is sometime very hard). Interfaces of formal software tools with non-specialists of formal methods, in particular to interpret the output in case of uncertainty and to interact with the tool, is also to be considered.

4 Conclusion

Formal verification is certainly essential to design safety critical embedded systems. Embedded software validation must evolve from debugging to verification, still a long way to go. The increasing complexity of such software systems evolves with the new capacities of hardware and networking capabilities. This continuously increased complexity makes the verification of embedded software a formidable challenge for the next decade.

The success will certainly depends on which models/semantics and specifications of embedded software are considered:

– Hand-made models are a guarantee of success (since finite models allowing for a formal proof of a given program always exist [27]). However hand-made

models are also extremely costly to design (but maybe for relatively small parts of the program) and maintain (in order to follow program evolutions). Methods for proving the soundness of such hand-made models are still to be developped and could strongly rely on abstract interpretation.
- Application specific models constructed by automatic abstraction require a model of reference and a theorem prover based soundness proof. This is logically equivalent to direct formal proofs [27] and so are also likely to be extremely costly (again but for small parts of the program).
- Since abstraction is inevitable and costly, an interesting alternative is to design reusable abstractions for programming languages with respect to predefined specifications (as in static program analysis) or user-defined specifications (as in abstract testing [39]). This is the whole purpose of program semantics abstraction as formalized by abstract interpretation [26] which will certainly be an important formal method leading to the systematic design of useful automatic tools supporting the design of embedded software.

References

[1] J.A. Abraham. The myth of fault tolerance in complex systems, keynote speech. In *The Pacific Rim International Symposium on Dependable Computing, PRDC'99*, Hong Kong, CN. IEEE Comp. Soc. Press, 16–17 Dec. 1999. `http://www.cerc.utexas.edu/~jaa/talks/prdc-1999/`.

[2] S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum, eds. *Semantic Modelling*, volume 4 of *Handbook of Logic in Computer Science*. Clarendon Press, 1995.

[3] J.-R. Abrial. *The B-Book*. Cambridge U. Press, 1996.

[4] R. Alur and D.L. Dill. A theory of timed automata. *Theoret. Comput. Sci.*, 126(2):183–235, 1994.

[5] T. Ball and S.K. Rajamani. Bebop: A symbolic model checker for boolean programs. In K. Havelund, J. Penix, and W. Visser, eds., *Proc. 7th SPIN Workshop*, Stanford, CA, LNCS 1885, pages 113–130. Springer-Verlag, Aug. 30 – Sep. 1, 2000.

[6] I. Beer, S. Ben-David, C. Eisner, D. Geist, L. Gluhovsky, T. Heyman, A. Landver, P. Paanah, Y. Rodeh, G. Ronin, and Y. Wolfsthal. RuleBase: Model checking at IBM. In O. Grumberg, editor, *Proc. 9th Int. Conf. CAV '97*, Haifa, IL, LNCS 1254, pages 480–483. Springer-Verlag, 22–25 Jul. 1997.

[7] A.M. Ben-Amram and N.D. Jones. Computational complexity via programming languages: constant factors do matter. *Acta Informat.*, 37(2):83–120, 2000.

[8] A. Benveniste, P. Le Guernic, and C. Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Sci. Comput. Programming*, 16(2):103–149, 1991.

[9] A. Biere, A. Cimatti, E.M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Proc. 36th Conf. DAC '99*, New Orleans, LA, pages 317–320. ACM Press, 21–25 June 1999.

[10] B. Boigelot and P. Godefroid. Symbolic verification of communication protocols with infinite state spaces using QDDs (extended abstract). In R. Alur and T.A. Henzinger, eds., *Proc. 8th Int. Conf. CAV '96*, New Brunswick, NJ, LNCS 1102, pages 1–12. Springer-Verlag, 31 Jul. –3 Aug. 1996.

[11] F. Bourdoncle. Abstract debugging of higher-order imperative languages. In *Proc. ACM SIGPLAN '93 Conf. PLDI. ACM SIGPLAN Not. 28(6)*, pages 46–55, Albuquerque, NM, 23–25 June 1993. ACM Press.

[12] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Inform. and Comput.*, 98(2):142–170, June 1992.

[13] Cadence®. "formalcheck" model checking verification. `http://www.cadence.com/datasheets/formalcheck.html`.

[14] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. LUSTRE: a declarative language for programming synchronous systems. In *14^{th} POPL*, Munchen, DE, 1987. ACM Press.

[15] Y-A. Chen, E.M. Clarke, P.H. Ho, Y. Hoskote, T. Kam, M. Khaira, J. O'Leary, and X. Zhao. Verification of all circuits in a floating-point unit using word-level model checking. In M.S. Srivas and A.J. Camilleri, eds., *Proc. 1^{st} Int. Conf. on Formal Methods in Computer-Aided Design, FMCAD '96*, number 1166 in LNCS, pages 19–33, Palo Alto, CA, 6–8 Nov. 1996. Springer-Verlag.

[16] E.M. Clarke and E.A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *IBM Workshop on Logics of Programs*, Yorktown Heights, NY, US, LNCS 131. Springer-Verlag, May 1981.

[17] E.M. Clarke, E.A. Emerson, S. Jha, and A.P. Sistla. Symmetry reductions in model checking. In A.J. Hu and M.Y. Vardi, eds., *Proc. 10^{th} Int. Conf. CAV '98*, Vancouver, BC, CA,LNCS 1427, pages 147–158. Springer-Verlag, 28 June – 2 Jul. 1998.

[18] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and . Veith. Counterexample-guided abstraction refinement. In E.A. Emerson and A.P. Sistla, eds., *Proc. TWELFTHInt. Conf. CAV '00*, Chicago, IL, LNCS 1855, pages 154–169. Springer-Verlag, 15–19 Jul. 2000.

[19] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and . Veith. Progress on the state explosion problem in model checking. In R. Wilhelm, editor, « *Informatics — 10 Years Back, 10 Years Ahead* », volume 2000 of *LNCS*, pages 176–194. Springer-Verlag, 2000.

[20] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. MIT Press, 1999.

[21] E.M. Clarke, S. Jha, Y. Lu, and D. Wang. Abstract BDDs: A technique for using abstraction in model checking. In L. Pierre and T. Kropf, eds., *Correct Hardware Design and Verification Methods, Proc. 10^{th} IFIP WG 10.5 Adv. Res. Work. Conf. CHARME '99*, Bad Herrenalp, DE, LNCS 1703, pages 172–186. Springer-Verlag, 27–29 Sep. 1999.

[22] R. Cleaveland, P. Iyer, and D. Yankelevitch. Optimality in abstractions of model checking. In A. Mycroft, editor, *Proc. 2^{nd} Int. Symp. SAS '95*, Glasgow, UK, 25–27 Sep. 1995, LNCS 983, pages 51–63. Springer-Verlag, 1995.

[23] P. Cousot. *Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique de programmes*. Thèse d'État ès sciences mathématiques, Université scientifique et médicale de Grenoble, Grenoble, FR, 21 Mar. 1978.

[24] P. Cousot. Semantic foundations of program analysis. In S.S. Muchnick and N.D. Jones, eds., *Program Flow Analysis: Theory and Applications*, chapter 10, pages 303–342. Prentice-Hall, 1981.

[25] P. Cousot. Methods and logics for proving programs. In J. van Leeuwen, editor, *Formal Models and Semantics*, volume B of *Handbook of Theoretical Computer Science*, chapter 15, pages 843–993. Elsevier, 1990.

[26] P. Cousot. Abstract interpretation based formal methods and future challenges, invited paper. In R. Wilhelm, editor, « *Informatics — 10 Years Back, 10 Years Ahead* », volume 2000 of *LNCS*, pages 138–156. Springer-Verlag, 2000.

[27] P. Cousot. Partial completeness of abstract fixpoint checking, invited paper. In B.Y. Choueiry and T. Walsh, eds., *Proc. 4^{th} Int. Symp. SARA '2000*, Horseshoe Bay, TX, LNAI 1864, pages 1–25. Springer-Verlag, 26–29 Jul. 2000.

[28] P. Cousot. Compositional separate modular static analysis of programs by abstract interpretation. *Proc. SSGRR 2001 – Advances in Infrastructure for Electronic Business, Science, and Education on the Internet*, 6 – 10 Aug. 2001.

[29] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proc. 2^{nd} Int. Symp. on Programming*, pages 106–130. Dunod, 1976.

[30] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4^{th} POPL*, pages 238–252, Los Angeles, CA, 1977. ACM Press.

[31] P. Cousot and R. Cousot. Static determination of dynamic properties of recursive procedures. In E.J. Neuhold, editor, *IFIP Conf. on Formal Description of Programming Concepts, St-Andrews, N.B., CA*, pages 237–277. North-Holland, 1977.

[32] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *6^{th} POPL*, pages 269–282, San Antonio, TX, 1979. ACM Press.

[33] P. Cousot and R. Cousot. 'À la Floyd' induction principles for proving inevitability properties of programs. In M. Nivat and J. Reynolds, eds., *Algebraic Methods in Semantics*, chapter 8, pages 277–312. Cambridge U. Press, 1985.

[34] P. Cousot and R. Cousot. Comparison of the Galois connection and widening/narrowing approaches to abstract interpretation. *Actes JTASPEFL '91, Bordeaux, FR. BIGRE*, 74:107–110, Oct. 1991.

[35] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *J. Logic Programming*, 13(2–3):103–179, 1992. (The editor of J. Logic Programming has mistakenly published the unreadable galley proof. For a correct version of this paper, see `http://www.di.ens.fr/~cousot`.).

[36] P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation, invited paper. In M. Bruynooghe and M. Wirsing, eds., *Proc. 4^{th} Int. Symp. PLILP '92*, Leuven, BE, 26–28 Aug. 1992, LNCS 631, pages 269–295. Springer-Verlag, 1992.

[37] P. Cousot and R. Cousot. Galois connection based abstract interpretations for strictness analysis, invited paper. In D. Bjørner, M. Broy, and I.V. Pottosin, eds., *Proc. FMPA*, Akademgorodok, Novosibirsk, RU, LNCS 735, pages 98–127. Springer-Verlag, 28 June – 2 Jul. 1993.

[38] P. Cousot and R. Cousot. Higher-order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection and PER analysis of functional languages), invited paper. In *Proc. 1994 ICCL*, pages 95–112, Toulouse, FR, 16–19 May 1994. IEEE Comp. Soc. Press.

[39] P. Cousot and R. Cousot. Abstract interpretation based program testing, invited paper. In *Proc. SSGRR 2000 Computer & eBusiness International Conference*, Compact disk paper 248 and electronic proceedings `http://www.ssgrr.it/en/ssgrr2000/proceedings.htm`, L'Aquila, IT, 31 Jul. – 6 Aug. 2000. Scuola Superiore G. Reiss Romoli.

[40] P. Cousot and R. Cousot. Temporal abstract interpretation. In *27th POPL*, pages 12–25, Boston, MA, Jan. 2000. ACM Press.

[41] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *5th POPL*, pages 84–97, Tucson, AZ, 1978. ACM Press.

[42] N. Dor, M. Rodeh, and M. Sagiv. Cleanness checking of string manipulations in c programs via integer analysis. In P. Cousot, editor, *Proc. 8th Int. Symp. SAS '01*, Paris, FR, LNCS 2126, pages 194–212. Springer-Verlag, 16–18 Jul. 2001.

[43] N. Dor, M. Rodeh, and M. Sagiv. Checking cleanness in linked lists. In J. Palsberg, editor, *Proc. 7th Int. Symp. SAS '2000*, Santa Barbara, CA, LNCS 1824, pages 115–134. Springer-Verlag, 29 June – 1 Jul. 2000.

[44] J. Feret. Abstract interpretation-based static analysis of mobile ambients. In P. Cousot, editor, *Proc. 8th Int. Symp. SAS '01*, Paris, FR, LNCS 2126, pages 413–431. Springer-Verlag, 16–18 Jul. 2001.

[45] J. Feret. Occurrence counting analysis for the π-calculus. *ENTCS*, 39, 2001. `http://www.elsevier.nl/locate/entcs/volume39.html`.

[46] J. Feret. Confidentiality analysis of mobile systems. In J. Palsberg, editor, *Proc. 7th Int. Symp. SAS '2000*, Santa Barbara, CA, LNCS 1824, pages 135–154. Springer-Verlag, 29 June – 1 Jul. 2000.

[47] P. Flajolet, B. Salvy, and P. Zimmermann. Automatic average-case analysis of algorithm. *Theoret. Comput. Sci.*, 79(1):37–109, 1991.

[48] R.W. Floyd. Assigning meaning to programs. In J.T. Schwartz, editor, *Proc. Symposium in Applied Mathematics*, volume 19, pages 19–32. AMS, 1967.

[49] R. Giacobazzi and E. Quintarelli. Incompleteness, counterexamples and refinements in abstract model-checking. In P. Cousot, editor, *Proc. 8th Int. Symp. SAS '01*, Paris, FR, LNCS 2126, pages 356–373. Springer-Verlag, 16–18 Jul. 2001.

[50] É. Goubault. Static analyses of the precision of floating-point operations. In P. Cousot, editor, *Proc. 8th Int. Symp. SAS '01*, Paris, FR, LNCS 2126, pages 234–259. Springer-Verlag, 16–18 Jul. 2001.

[51] N. Halbwachs. About synchronous programming and abstract interpretation. In B. Le Charlier, editor, *Proc. 1st Int. Symp. SAS '94*, Namur, BE, 20–22 Sep. 1994, LNCS 864, pages 179–192. Springer-Verlag, 1994.

[52] N. Halbwachs, F. Lagnier, and C. Ratel. An experience in proving regular networks of processes by modular model checking. *Acta Informat.*, 29(6/7):523–543, 1992.

[53] C. Hankin and S. Hunt. Approximate fixed points in abstract interpretation. *Sci. Comput. Programming*, 22(3):283–306, 1994. Erratum: Sci. Comput. Programming 23(1): 103 (1994).

[54] Ø. Haugen. From MSC-2000 to UML 2.0 – the future of sequence diagrams. In R. Reed and J. Reed, eds., *Proc. SDL 2001: Meeting UML, 10th Int. SDL Forum*, Copenhagen, DK, 27–29 June 2001, LNCS 2078, pages 38–51. Springer-Verlag, 2001.

[55] G.J. Holzmann. From code to models. In *Proc. 2nd Int. Conf. ACSD '01*, Newcastle upon Tyne, GB. IEEEpress, 25–29 June 2001.

[56] G.J. Holzmann and M.H. Smith. Software model checking: Extracting verification models from source code. In *Proc. Formal Methods in Software Engineering and Distributed Systems, PSTV/FORTE99*, Beijng china, pages 481–497. Kluwer Acad. Pub., Oct. 1999.

[57] M. Huth, R. Jagadeesan, and D.A. Schmidt. Modal transition systems: A foundation for three-valued program analysis. In D. Sands, editor, *Proc. 10th ESOP '01*, LNCS 2028, pages 155–169, Genova, IT, 2–6 Apr. 2001. Springer-Verlag.

[58] Joint Technical Committee ISO/IEC JTC1, Information Technology. The ISO/IEC 9899:1990 standard for Programming Language C. 1 Dec. 1990.

[59] Joint Technical Committee ISO/IEC JTC1, Information Technology. The ISO/IEC 9899:1999 standard for Programming Language C. 1 Dec. 1999.

[60] Joint Technical Committee ISO/IEC JTC1, Information Technology. The Technical Corrigendum 1 (ISO/IEC 9899 TCOR1) to ISO/IEC 9899:1990 standard for Programming Language C. `http://anubis.dkuug.dk/JTC1/SC22/WG14/www/docs/tc2.htm`, 1995.

[61] N.D. Jones. Program analysis for implicit computational complexity. In 0. Danvy and A. Filinski, eds., *Proc. 2nd Symp. PADO '2001*, Århus, DK, 21–23 May 2001, LNCS 2053, page 1. Springer-Verlag, 2001.

[62] Y. Kesten and A. Pnueli. Modularization and abstraction: The keys to formal verification. In L. Brim, J. Gruska, and J. Zlatuska, eds., *23rd Int. Symp. MFCS '98*, LNCS 1450, pages 54–71. Springer-Verlag, 1998.

[63] D. Kozen. Results on the propositional μ-calculus. *Theoret. Comput. Sci.*, 27:333–354, 1983.

[64] O. Kupferman and M.Y. Vardi. Vacuity detection in temporal model checking. In L. Pierre and T. Kropf, eds., *Correct Hardware Design and Verification Methods, Proc. 10th IFIP WG 10.5 Adv. Res. Work. Conf. CHARME '99*, Bad Herrenalp, DE, LNCS 1703, pages 82–96. Springer-Verlag, 27–29 Sep. 1999.

[65] P. Lacan, J.N. Monfort, L.V.Q. Ribal, A. Deutsch, and G. Gonthier. The software reliability verification process: The Ariane 5 example. In *Proceedings DASIA 98 – DAta Systems In Aerospace*, Athens, GR. ESA Publications, SP-422, 25–28 May 1998.

[66] W. Lee, A. Pardo, J.-Y. Jang, G. Hachtel, and F. Somenzi. Tearing based automatic abstraction for CTL model checking. In *ICCAD 1996*, San Jose, CA, pages 76–81. IEEE Comp. Soc. Press, Nov. 10–14 1996.

[67] M. Leuschel. On the power of homeomorphic embedding for online termination. In G. Levi, editor, *Proc. 5th Int. Symp. SAS '98*, Pisa, IT, 14–16 Sep. 1998, LNCS 1503, pages 200–214. Springer-Verlag, 1998.

[68] J.L. Lions (Chairman of the Board). Ariane 5 flight 501 failure, report by the inquiry board. `http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html`, see also `http://vlsi.colorado.edu/~abel/pubs/anecdote.html\#ariane`.

[69] T. Margaria and W. Yi, eds. *Branching vs. Linear Time: Final Showdown*, Genova, IT, LNCS 2031. Springer-Verlag, 2–6 Apr. 2001.

[70] F. Martin, M. Alt, R. Wilhelm, and C. Ferdinand. Analysis of loops. In K. Koskimies, editor, *Proc. 7th Int. Conf. CC '98*, Lisbon, PT, LNCS 1383, pages 80–94. Springer-Verlag, 28 Mar. – 4 Apr. 1998.

[71] L. Mauborgne. Abstract interpretation using typed decision graphs. *Sci. Comput. Programming*, 31(1):91–112, May 1998.

[72] A. Miné. A new numerical abstract domain based on difference-bound matrices. In 0. Danvy and A. Filinski, eds., *Proc. 2nd Symp. PADO '2001*, Århus, DK, 21–23 May 2001, LNCS 2053, pages 155–172. Springer-Verlag, 2001.

[73] S.E. Panitz and M. Schmidt-Schauß. TEA: Automatically proving termination of programs in a non-strict higher-order functional language. In P. Van Hentenryck, editor, *Proc. 4th Int. Symp. SAS '97*, Paris, FR, 8–10 Sep. 1997, LNCS 1302, pages 345–360. Springer-Verlag, 1997.

[74] J.-P. Queille and J. Sifakis. Verification of concurrent systems in Cesar. In *Proc. Int. Symp. on Programming*, LNCS 137, pages 337–351. Springer-Verlag, 1982.

[75] F. Ranzato. On the completeness of model checking. In D. Sands, editor, *Proc. 10th ESOP '2001*, Genova, IT, 2–6 Apr. 2001, LNCS 2028, pages 137–154. Springer-Verlag, 2001.

[76] J. Rushby. Automated deduction and formal methods. In R. Alur and T.A. Henzinger, eds., *Proc. 8th Int. Conf. CAV '96*, number 1102 in LNCS, pages 169–183, New Brunswick, NJ, Jul. /Aug. 1996. Springer-Verlag.

[77] B.G. Ryder, W. Landi, P.A. Stocks, S. Zhang, and R. Altucher. A schema for interprocedural side effect analysis with pointer aliasing. *TOPLAS*, 2001. To appear.

[78] S. Saïdi. Model checking guided abstraction and analysis. In J. Palsberg, editor, *Proc. 7th Int. Symp. SAS '2000*, Santa Barbara, CA, LNCS 1824, pages 377–396. Springer-Verlag, 29 June – 1 Jul. 2000.

[79] D.A. Schmidt. Data-flow analysis is model checking of abstract interpretations. In *25th POPL*, pages 38–48, San Diego, CA, 19–21Jan. 1998. ACM Press.

[80] C. Speirs, Z. Somogyi, and H. Søndergaard. Termination analysis for Mercury. In P. Van Hentenryck, editor, *Proc. 4th Int. Symp. SAS '97*, Paris, FR, 8–10 Sep. 1997, LNCS 1302, pages 160–171. Springer-Verlag, 1997.

[81] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise WCET prediction by separated cache and path analyses. *Real-Time Syst.*, 18(2–3):157–179, 2000.

A Network-Centric Approach to Embedded Software for Tiny Devices

David E. Culler, Jason Hill, Philip Buonadonna, Robert Szewczyk, and Alec Woo

University of California at Berkeley, Intel Research at Berkeley, Berkeley CA 94720, USA

Abstract. The ability to incorporate low-power, wireless communication into embedded devices gives rise to a new genre of embedded software that is distributed, dynamic, and adaptive. This paper describes the network-centric approach to designing software for highly constrained devices embodied in TinyOS. It develops a tiny Active Message communication model and shows how it is used to build non-blocking applications and higher level networking capabilities, such as multihop ad hoc routing. It shows how the TinyOS event-driven approach is used to tackle challenges in implementing the communication model with very limited storage and the radio channel modulated directly in software in an energy efficient manner. The open, component-based design allows many novel relationships between system and application.[1]

1 Introduction

The emergence of compact, low-power wireless communication, sensors, and actuators in the technology that supports the ongoing miniaturization of processing and storage is giving rise to entirely new kinds of embedded systems and a fundamentally new genre of embedded software. Historically, embedded systems have been highly engineered to a particular task. For example, a disk drive controller sits between a standardized command/response channel and the disk head assembly, with its rotation sensors, positioning actuators, and read/write heads. The controller software is a highly orchestrated command processing loop to parse the request, move the heads, transfer the data, perform signal processing on it, and respond. The system is sized and powered for the particular application. The software is developed incrementally over generations of products and loaded into a device for its lifetime. An engine ignition controller is even more specialized to performing a particular sense/actuate loop autonomously.

The new kinds of embedded systems are distributed, deployed in environments where they may not be designed into a particular control path, and often very dynamic. The fundamental change is communication; collections of devices can communicate to achieve higher level coordinated behavior. For example, wireless sensor nodes may be deposited in offices and corridors throughout

[1] This research was supported by the Defense Advanced Research Projects Agency, the National Science Foundation and Intel Corporation

T.A. Henzinger and C.M. Kirsch (Eds.): EMSOFT 2001, LNCS 2211, pp. 114–130, 2001.
© Springer-Verlag Berlin Heidelberg 2001

a building, providing light, temperature, and activity measurements. Wireless nodes may be attached to circuits or appliances to sense current or to control usage. Together they form a dynamic, multihop routing network that connects each node to more powerful networks and processing resources. Through analysis of radio signal strength and other sensory nodes, nodes determine their location. They tap into local energy sources, perhaps using photovoltaic cells or nearby telephone or AC lines, to restore their energy reserves. Nodes come and go, move around, and are affected by changes in their environment. Collectively, they adapt to these changes, perform analysis of usage patterns and control lighting, temperature, and appliance operation to conform to overall energy usage goals.

The new genre of embedded software is characterized by being agile, self-organizing, critically resource constrained, and communication-centric on numerous small devices operating as a collective, rather than highly engineered to a particular stand-alone task on a device sized to suit. The application space is huge, spanning from ubiquitous computing environments where numerous devices on people and things interact in a context-aware manner, to dense in situ monitoring of life science experiments, to condition-based maintenance, to disaster management in a smart civil infrastructure. A common pattern we find is that the mode of operation is concurrency intensive for bursts of activity and otherwise very passive watching for a significant change or event. In the bursts, data and events are streaming in from sensors and the network, out to the network and to various actuators. A mix of real-time actions and longer-scale processing must be performed. In remaining majority of the time, the device must shutdown to a very low power state, yet monitor sensors and network for important changes while perhaps restoring energy reserves. Net accumulation of energy in the passive mode and efficiency in the active mode determine the overall performance capability of the nodes.

To explore the system design techniques underlying these kinds of applications and the emerging technology of microscopic computing, we have developed a series of small RF wireless sensor devices, a tiny operating system (TinyOS), and a networking infrastructure for low-power, highly constrained devices in dynamic, self-organized, interactive environments. The hardware platform grew out of the 'Macromote' developed in SmartDust project as a demonstration of the current analog of what might be put into a cubic millimeter by 2005 [8]. Our first experimental platform (Figure 1) had a 4MHz Atmel AVR 8535 Microcontroller, 8 KB of program store, 0.5 KB of SRAM, a single-channel low power radio [9], EEPROM secondary store, and a range of sensors on an expansion bus [4]. It operates at about 5 mA when active and 5μA in standby, so a pair of AA batteries provides over a year of lifetime at 1% active duty cycle. The severe resource constraints put this platform far beyond reach of conventional operating systems. TinyOS is a simple, component-based operating system, which primarily is a framework for managing concurrency in a storage and energy limited context. A collection of modular components build up from modulating the radio channel and accessing sensors via ADCs to an event-driven environmental

monitoring application with dynamic network discovery and multihop ad hoc routing. A non-blocking discipline is carried through out the design and most components are essentially reentrant cooperating state machines.

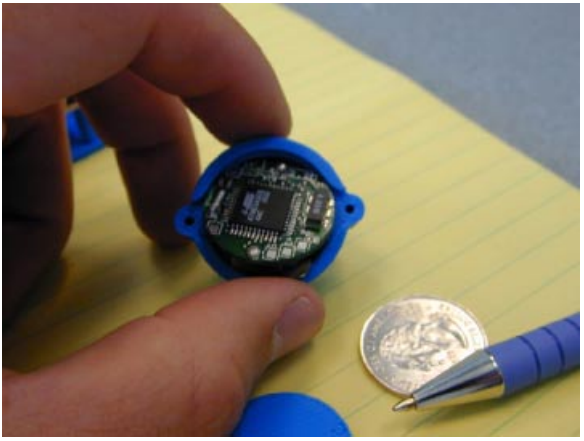


Fig. 1. The DOT mote.

The remainder of this paper describes the communication-centric design issues that have arose in developing TinyOS and its tiny networking stacks. Section 2 provides background and a general introduction to TinyOS. Section 3 introduces the Tiny Active Message communication abstraction and illustrates how it is used to form high level networking capabilities. Section 4 examines a collection of software challenges underlying the communication abstraction. Section 5 provides a brief performance evaluation and Section 6 outlines future directions.

2 TinyOS Concepts

Tiny OS, like conventional operating systems, seeks to reduce the burden of application development by providing convenient abstractions of physical devices and highly tuned implementations of common functions. However, this goal is especially challenging because of the highly constrained resource context, the unusual and application specific character of the devices, and the lack of consensus on what layers of abstraction are most appropriate in this regime. The TinyOS approach is to define a very simple component model and to develop a range of components, subsets of which are composed to support a particular application. With time and experience, the new layers of abstraction are likely to emerge. The TinyOS component model focuses on providing a rich expression of concurrency within limited resources, rather than interface discovery or format adaptation [1,10]. Since the deeply embedded sensor networks that we target must run unattended for long periods, robustness is essential. The component

model with narrow interfaces is a significant aid, but given the processing, storage, and energy constraints, we need very efficient modularity. The programming model provides extensive static information, so that compile-time techniques can remove much of the overhead associated with a modular approach and eventually can provide unusual analyses, such as jitter bounds.

A complete TinyOS application consists of a scheduler and a graph of components. Each component is described by its interface and its internal implementation, in a manner similar to many hardware description languages, such as VHDL and Verilog. An interface comprises synchronous *commands* and asynchronous *events*. We think of the component as having an upper interface, which names the commands it implements and the events it signals, and a lower interface, which names the commands it uses and the events it handles. The implementation is written entirely in terms of the interface name space. A component also has internal storage, structured into a *frame*, and internal concurrency, in the form of very light-weight threads, called *tasks*. The command, event, and task handlers are declared explicitly in the source. The points where an external command is called, event is signaled, or task is posted are also explicit in the static code, as are references to frame storage. A separate application description describes how the interfaces are 'wired together' to form the overall application composition. The wiring need not be 1-1; an event may be delivered to multiple components or multiple components may use the same command. Thus, although the application is extremely modular, the compiler has a great deal of static information to use in optimizing across the whole application, including the operating system. In addition, the underlying run-time execution model and storage model can be optimized for specific platforms. A typical application graph is shown in Figure 2, containing a low-power radio stack, a UART serial port stack, sensor stacks, and higher level network discovery and ad hoc routing to support distributed sensor data acquisition. This entire application occupies about three kilobytes.

The TinyOS concurrency model is essentially a two-level scheduling hierarchy - events preempt tasks, tasks do not preempt other tasks. The philosophy is that the vast majority of operation is in the form of non-blocking state transitions. Inter-component operation in tasks is relatively familiar. Within a task, commands may be called, a command may call subordinate commands, or it may post tasks to continue working logically in parallel with its invocation. By convention, all commands return a status indicating whether the command was accepted, providing a full handshake. Since all components have bounded storage, a component must be able to refuse commands. It is very common for a command to merely initiate an operation, say accessing a sensor or sending a message, leaving the operation to be carried out concurrently with other activities, either using hardware parallelism or tasks.

Events are initiated at the lowest level by hardware interrupts. Events may signal higher level events, call commands, or post tasks. Commands cannot signal events. Thus, an individual event may propagate through multiple levels of components, triggering collateral activity. Whenever the work cannot be ac-

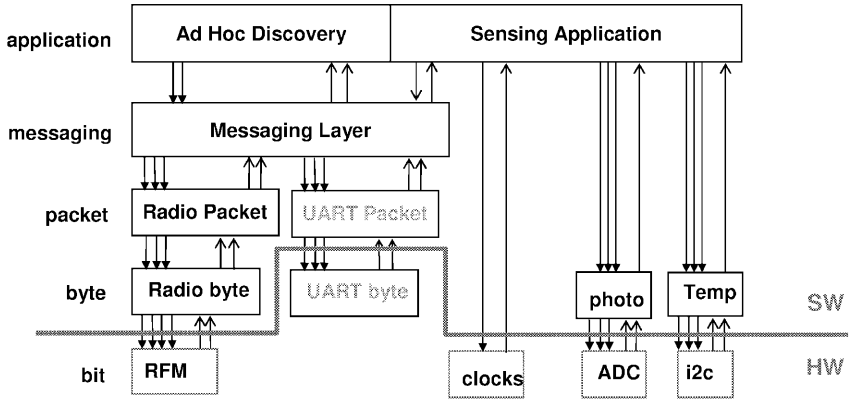


Fig. 2. Typical networking application component graph.

complished in a small, bounded amount of time, the component should record continuation information in its frame and post a task to complete the work. By convention, the lowest level hardware abstraction components perform enough interrupt processing to reenable interrupts before signaling the event. Events (or tasks posted within events) typically complete the split-phase operations initiated by commands, signaling the higher-level component that the operation has completed and perhaps passing it the data.

A non-blocking approach is taken throughout TinyOS. There are no locks and components never spin on a synchronization variable. A lock-free queue data structure is used by the scheduler. Components perform a phase of an operation and terminate, allowing the completion event to resume their execution. Most components are written essentially as reentrant state machines. Currently, TinyOS is written in C with conventional preprocessor macros to highlight the key concepts. Better linguistic support would be natural and desirable as the approach becomes established. In our current implementation, the TinyOS execution model is implemented on a single shared stack with a static frame per component.

To make the discussion concrete, Figure 3 shows a TinyOS C code fragment that periodically obtains a value from a sensor and communicates it to neighboring nodes, which render it on their LEDs. The top level structure declares the component storage frame, a command handler, and four event handlers, one of which handles a message event. Each declaration is decorated with a TOS handler type and each would be reflected in the external interface. Here the frame provides an outgoing message buffer for the component and a state variable. The CHIRP_INIT command illustrates the synchronous module interface; it invokes a subordinate command in the clock module to request periodic events, one per second. A local name is used for the clock initialization command, which is bound to the name used within the particular clock component by the application de-

scription graph. Similarly, a local event handler (CHIRP_CLOCK_EVENT) is wired to the clock output event. Here the clock event initiates acquisition of a sensor data value, unless the previous data has not yet been transmitted. The reference to a frame variable is explicit, using VAR.

```
TOS_FRAME_BEGIN(CHIRP_frame) {
  TOS_Msg msg;              /* Message transmission buffer */
  char send_pending;        /* State of buffer*/
}
TOS_FRAME_END(CHIRP_frame);

char TOS_COMMAND(CHIRP_INIT)(){
     return TOS_CALL_COMMAND(CHIRP_CLOCK_INIT)(tick1ps);
}

void TOS_EVENT(CHIRP_CLOCK_EVENT)(){
  if (VAR(send_pending) == 0) return TOS_CALL_COMMAND(CHIRP_GET_DATA)();
}

char TOS_EVENT(CHIRP_DATA_EVENT)(int data){
  VAR(msg).data[0] = (char)(data >> 2) & 0xff;
  VAR(send_pending) = 1;
  if (TOS_CALL_COMMAND(CHIRP_SEND_MSG)(TOS_BCAST_ADDR,AM_MSG(CHIRP_MSG),
      &VAR(data))
    return 1;
  }else {
    VAR(send_pending) = 0;
    return 0;
  }
}

char TOS_EVENT(CHIRP_MSG_SEND_DONE)(TOS_MsgPtr msg){
  if(&VAR(msg) == msg) VAR(send_pending) = 0;
  return 1;
}

TOS_MsgPtr TOS_MSG_EVENT(CHIRP_MSG)(TOS_MsgPtr msg){
  TOS_CALL_COMMAND(CHIRP_OUTPUT(msg->data[0]);
  return msg;
}
```

Fig. 3. Example Tiny Active Message application.

The sensor data acquisition protocol illustrates a common, split-phase access pattern. The command causes the operation to start. The component will finish a phase of its work and return, typically allowing the task or event to complete. When the operation is complete, a corresponding event is fired. Here,

the ADC runs concurrently and signals the data ready event, which converts the 10-bit sensor value to an 8-bit quantity and requests that it be transmitted in a message. Completion of the transmission operation will signal the CHIRP_MSG_SEND_DONE event on the local node. Arrival of the message at neighboring nodes will signal the CHIRP_MSG event on those nodes, passing it the message data. Details of the messaging portions are described below.

3 Application-Level Communications Challenges

A key test of the communication-centric design approach in TinyOS is its utility in constructing a networking infrastructure for self-organized, deeply embedded collections of devices. We outline several of the key challenges that arise within the networking 'stack' and describe how they are addressed within TinyOS. We begin with the application level messaging model, a variant of Active Messages [11]. Going upward, we describe a simple dynamic network discovery and ad hoc multihop routing layer. Going down, the next section addresses a number of the detailed issues in implementing such a stack in few instructions, little storage, and very little power.

3.1 Tiny Active Messages

Active Messages (AM) is a simple, extensible paradigm for message-based communication widely used in large parallel and distributed computing systems [6, 11]. At its core is the concept of overlapping communication and computation through lightweight remote procedure calls. Each message contains the name of a handler to be invoked on a target node upon arrival and a data payload to pass in as arguments. The handler function serves the dual purpose of extracting the message from the network and either integrating the data into the computation or sending a response. The AM communication model is especially well-suited to the execution framework of TinyOS, as it is event-driven and specifically designed to allow a very lean communication stack to process packets directly off the network, while supporting a wide range of applications.

Initiating an Active Message involves four components, specifying the data arguments, naming the handler, requesting the transmission, and detecting transmission completion. Receiving involves invoking the specified handler on a copy of the transmitted data. This family of issues is illustrated in Figure 3.

The SEND_MSG command identifies intended recipients (here using the broadcast address for the local cell of nodes that pick up the radio transmission), the handler that will process the message on arrival, (here CHIRP_MSG), and the source output message buffer in the local frame. A handler registry is maintained, and TOS_MSG extracts the identifier for the named handler. The status handshake for this command illustrates the general notion of components managing their bounded resources. The messaging component may refuse the send request, for example, if it is busy transmitting or receiving a message and does not have resources with which to queue the request. The reaction to this

occurrence is application specific; in this example we forgo transmitting the particular sensor reading. We might instead adjust data acquisition frequency or phase.

The message arrival event is similar to other events. One key difference is that the Active Message component dispatches the event to the component with the associated message handler. Many components may register one or more message handlers. Additionally, the input to the handler is a reference to a message buffer provided by the Active Message component.

3.2 Managing Packet Buffers

Managing buffer storage is a sticky problem in any communication stack. Traditional operating systems push this complexity into the kernel, providing a simple, unlimited user model at the cost of copying, storage management complexity, and blocking interfaces. High performance models, such as MPI, define a suite of send and receive operations with distinct completion semantics[3]. Three issues must be addressed: encapsulating useful data with transport header and trailer information, determining when output message data storage can be reused, and providing an input buffer for an incoming message before the message has been inspected to determine where it goes. The Tiny Active Message layer provides simple primitives for resolving these issues with no copying and very simple storage management.

The message buffer has a defined type in the frame that provides holes for system specific encapsulation, such as routing information and error detection. These holes are filled in as the packet moves down the stack, rather than following pointers or copying. The application components refer only to the data field or the entire buffer. References to message buffers are the only pointers carried across component boundaries in TinyOS.

Once the send command is called, the transmit buffer is considered 'owned' by the network until the messaging component signals that transmission is complete. The mechanism for tracking ownership is application specific; our example maintains a pending flag. Since a strict ownership exchange is involved, no mutex is required around updates to the flag, although some care in coding must be exercised. If the send command is accepted, the SEND_DONE event will asynchronously clear the pending flag.

Observe that the SEND_DONE event receives a reference to the completed buffer as an argument and must check whether the buffer is its own. This event is delivered to all components that register AM handlers. A component that receives a done signal for another's transmission may use the event to retry a previously refused request.

The message handler receives a reference to a 'system owned' buffer, which is distinct from its frame. The typical behavior is to process information in the message and return the buffer, as in our example. In general, the handler must return a reference to *some* free buffer. It could retain the buffer it was given by the system and return a different buffer that it 'owns'. A common special case of this scenario is a handler that makes a small change to an incoming message

and retransmits it. We would like to avoid copying the remainder of the message. However, we cannot retain ownership of the buffer for transmission and return the same buffer to the system. Such a component should declare a message buffer and a message buffer pointer in its frame. The handler modifies the incoming buffer and exchanges buffer ownership with the system. If its previous transmit buffer is still busy, one of the two operations must be discarded. A component performing reassembly from multiple packets may own multiple such buffers. In any case, runtime buffer storage management is reduced to a simple pointer swap.

3.3 Network Discovery and Ad Hoc Routing

A more sophisticated use of the tiny Active Message model is illustrated by its use in supporting dynamic network discovery and multihop ad hoc routing. Discovery could be initiated from any node, but often it is rooted at gateway nodes that provide connectivity to conventional networks. Each root periodically transmits a message carrying its ID and its distance (zero) to its neighborhood. The message handler checks whether the source is the 'closest' node it has heard from recently (i.e., in the current discovery phase) and, if so, records the source ID as its multihop parent, increments the distance, and retransmits the message with its own ID as the source. The discovery component utilizes the buffer swap described above. Observe, this simple algorithm builds a breadth first spanning tree in a distributed fashion rooted at the original source. Each node records only a fixed amount of information. The specific shape of the tree is determined by the physical propagation characteristics of the network, not any prespecified layout, so the network is self-organizing. With multiple concurrent roots, a spanning forest is formed.

Routing packets up the tree is straightforward. A node transmitting data to be routed specifies a multihop forwarding handler and identifies its parent as the recipient. The handler will fire in each of its neighbors. The parent retransmits the packet to its parent, using the buffer swap. Other neighbors simply discard the packet. The data is thus routed hop-by-hop to the root. Reduction operators can be formed by accumulating data from multiple 'children' before transmitting a packet up the tree.

The discovery algorithm is non-optimal because of redundancy in the outgoing discovery wave front and might be improved by electing cluster leaders or retransmitting the beacon with some probability inversely related to the number of siblings. Alternatively, the discovery phase can be eliminated entirely by piggybacking the distance information on the sensor data messages. When a node hears a packet from a node fewer hops from the base station, it adopts the source as its parent. The root node simply transmits a packet to itself to grow the routing tree. (Nodes must also age their current distance to adapt to changes in network topology due to movement or signal propagation changes.)

These simple examples illustrate the fundamental communication step upon which distributed algorithms for embedded wireless networks are based: receiving a packet, transforming it, and selectively retransmitting it or not. Squelching

retransmission forms an outgoing wave front in discovery and forms a beam on multihop routing. In these algorithms the data structure for determining whether to retransmit is little more than a hop count, more generally it might be a cache of recent packets [5,7].

4 Lower-Level Communication Challenges

This section works down from the messaging component to illustrate the nuts and bolts of realizing the lower layers if a tiny communications stack.

4.1 Crossing Layers without Buffering

One challenge is to move the message data from the application storage buffer to the physical modulation of the channel without making entire copies, and similarly in the reverse direction. A common pattern that has emerged is a cross layer 'data pump'. We find this at each layer of the stack in Figure 2. The upper component has a unit of data partitioned into subunits. It issues a command to request transmission of the first subunit. The lower component acknowledges that it has accepted the subunit and when it is ready for the next one it signals a subunit event. The upper handler provides the next unit, or indicates that no more are forthcoming. Typically this is done by calling the next subunit command within the ready handler. The message layer is effectively a packet pump. The packet layer encodes and frames the packet, pumping it byte-by-byte into the byte layer. On the UART, the byte-by-byte abstraction is implemented directly in hardware, whereas on the radio the byte layer pumps the data bit-by-bit into the radio. Each of these components utilizes the frame, command, and event framework to construct a re-entrant software state machine.

4.2 Listening at Low Power

In traditional remote control or remote monitoring applications, a well-powered stationary device is always receiving and a portable device transmits infrequently. However, in a multi hop data collection network, each node will transmit its own data from time to time and listen the rest of the time for data that it needs to forward toward a sink.

Although active transmission is the most power intensive mode, most radios consume a substantial fraction of the transmit energy when the radio is on and receiving nothing. In ad hoc networks, a device will only be transmitting for short periods of time but must be continually listening in order to forward data for the surrounding nodes. The total energy consumption of a device ends up being dominated by RF reception cost. To address this, we employ two techniques to reduce the power consumption while listening.

A fairly traditional way to accomplish this is through a technique we refer to as *periodic listening*. By creating time periods when it is illegal to transmit, nodes must listen only part time. This approach works well when the time scale

of the invalid periods is quite large relative to the message transmission time. For example, we have an implementation of this mechanism where the transmission window is ten seconds and the sleep window is 90 seconds. This reduces the reception power consumption of the nodes by approximately 90%. However, downside of this simple approach is that it limits the realized bandwidth available by the same factor.

The reduction of network bandwidth into a mobile node is often unacceptable in the context of sensor networks. Any node may be act as a router or data processing point and need to fully utilize the radio bandwidth. To address this issue we have developed second technique that we call *low power listening*. This method keeps the same listener duty cycle concept, but greatly reduces the time scale. Each receiver turns its radio on for 30 μs out of a 300 μs window instead of 10 sec out of 100 sec. This permits the same 90% energy savings as periodic listening yet does not decrease the available channel capacity. The downside of this method is that a transmitter must spend extra energy to make sure that the receiver has its radio on before packet transmission begins. A transmitter must send a packet preamble designed to get the attention of the receiver. Because the sender knows that the receiver will be listening every 300μs, the preamble must be at least the same duration. In our system, the data packet length is 56,100 μs long, so preamble overhead is quite small compared to the transmission cost. A 90% decrease in idle power consumption is gained with less than a 1% increase in transmission cost without changing the channel capacity. In the case of a node that transmits one packet per second and receives one packet per second from other nodes, there is a net power reduction for the radio of approximately 75%. System level power measurements on real hardware have confirmed this power savings.

Table 1. Power savings breakdown for a radio that receives and transmits 1 packet per second with a TX power consumption of 12mA and an RX power consumption of 5mA. Packet transmission and reception takes 50ms.

Operation	Time	Normal	Low Power Mode
Transmit	50 ms/50.5 ms	600uj	606uj
Receive	50 ms	250 uj	250 uj
Listen	900 ms	4500 uj	450 uj
Total	1000 ms	5350 uj	1306 uj
Savings			75%

To further reduce the average power consumption of the network, low power listening can be combined with the periodic listening. Running both schemes simultaneously results in listening at reduced power for only a fraction of the time. The power reductions are multiplicative. These techniques provide a mechanism for trading bandwidth and transmission cost for a reduction in receive power consumption.

4.3 Physical Layer Interface

Traditional I/O subsystems have a controller hierarchy that abstracts the device specific characteristics and timing requirements of the physical layer from the main system controller. In contrast, our hardware directly connects the central microcontroller to the radio. This places all of the real time requirements of the radio onto the microcontroller. It must handle every bit that is transmitted or received in real time. Additionally, it controls the timing of each bit so that any jitter in the control signals that it generates is propagated to the transmitted signal. The TinyOS communication stack has been constructed to handle these constraints while allowing higher level functions to continue in parallel.

At the base of our component stack is a state machine that performs the bit timing. The RFM component transfers a single bit at a time to and from the RF Monolithics radio. For a correct transmission to occur, the transmitted bit must be placed and held on the TX line of the radio for exactly one bit time. In our system that is 100μs. For reception, the RX line of the radio must be sampled at the midpoint of the transmission period. The radio provides no support for determining when bit times have completed.

The interface to our RFM component takes the form of a data pump. It orchestrates a bit-by-bit transfer from a byte level component to the physical hardware. To start the transmission of data, a command is issued to the RFM component to switch into transmit mode. Then a second command is used to transfer a single bit down to the RFM component. This bit is immediately placed onto the transmit line. After 100μs has passed, the RFM component will signal an event to indicate that it is ready for another bit. The byte level components response is to issue another command to the RFM component that contains the next bit. This interaction of signaling and event and receiving the next bit continues until the entire packet is completed. The RFM layer component abstracts the real time deadlines of the transmission process from the higher layer components.

During transmission, complex encoding must be done on each byte while simultaneously meeting the strict real time requirements of the bit layer. The encoding operation for a single byte takes longer than the transmission time of single bit. To ensure that the encoded data is ready in time to meet the bit level transmission deadline, we must start the encoding of the next byte prior to the completion of the transmission of the current byte. We use the TinyOS task mechanism to execute the encoding operation while simultaneously performing the transmission of previous data. By encoding data one byte in advance of transmission, we are using buffering to decouple the bit level timing from the byte encoding process.

Data reception takes the same form as transmission except that the receiver must first detect that a transmission is about to begin and then determine the timing of the transmission. To accomplish this, when there is activity on the radio channel, the RFM layer component is set to sample bits every 50μs, double sampling each byte. These bits are handed up one at a time to the byte level component. The byte level component creates a sliding buffer of these bit values

that contains the last 18 bits. When the value of the last 18 bits received matches the designated start symbol, the start of a packet has been detected. Additionally, the timing of the packet has been determined to within half a bit time. Next, the RFM layer is told to sample a single bit after 75μs. This causes the next sample to fall in the middle of the next bit window, half way between where the double sampling would have occurred if the sample period had remained at 50μs. Finally, the RFM is told to sample every 100μs for the remainder of the packet.

4.4 Media Access and Transmission Rate Control

In wireless embedded systems, the communication path to the devices is not a dedicated link as it is in most traditional embedded system, but instead a shared channel. This channel represents a precious resource that must be shared effectively in the context of resource constrained processing and ad hoc multihop routing. Moreover, many applications require that nodes have roughly equal ability to move data through the network, regardless of position within the network topology. We have extended the low-level TinyOS communication components with an energy-aware media access control (MAC) protocol and developed a simple technique for application specific adaptive rate control [12].

Since the I/O controller hierarchy on our small devices is so primitive, the MAC protocols must be performed on micro-controller concurrently with other operations. The RF transceiver lacks support for collision detection, so we focus on Carrier Sense Multiple Access (CSMA) schemes, where a node listens for the channel and only transmits a packet if the channel is idle. The mechanism for clocking in bits at the physical layer is also used for carrier sensing. Thus, the MAC layer is implemented at both the bit and byte level in the network stack. If consecutive sampling of the channel discovers no signal, the channel is deemed idle and a packet transmission is attempted. However, if the channel is busy, a random back off occurs. The entire process repeats until the channel is idle. A simple 16-bit linear feedback shift register is used as a pseudo random number generator for the back off period. Since energy is a precious resource, the radio is turned off during back off. Many applications collect and transmit data periodically, perhaps after detecting a triggering event, so traffic can be highly correlated. Detection of a busy channel suggests that a neighboring node may indicate that the communication patterns of the nodes are synchronized. The application uses the failure to send as feedback and shifts it sampling phase to potentially desynchronize. This simple scheme has been shown to yield 75% channel utilization for densly populated cell.

Another common application requirement is roughly equal coverage of data sampling over the entire network. In other words, each node in the network should be able to deliver fair allocation of bandwidth to the base station. With our ad hoc routing layer, nodes self organize into a spanning forest, where each node originates and routes traffic to a base station. The competition between originated and route-thru traffic for upstream bandwidth must be balanced in

order to meet the fairness goal. Furthermore, given that the capacity of a multi-hop network is limited [2], nodes must adapt their offered load to the available bandwidth rather than over-commit the channel and waste energy in transmitting packets that can never reach the base station. Our adaptive transmission control scheme is a local algorithm implemented above the Active Message layer and below the application level. The application has a baseline sampling rate that determines in maximum transmission rate and transmits a sample with a dynamically determined probability. On successful transmission the probability is increased linearly, whereas on failure it is decreased multiplicatively. A successful transmission can be indicated by an explicit acknowledgment from the receiver or an implicit acknowledgment when the sender hears its packet being forwarded by its parent. Since implicit acknowledgment is often application specific, the application decides if the transmission was successful and propagates the information down to the transmission control layer. Rejection of application's transmission command at the transmission control level triggers the adaptation.

5 Evaluation

To demonstrate the performance of the Active Messages model, we performed single message round-trip timings (RTT) and measured energy overhead on a sample implementation. The measurements were made on embedded sensors with a 4MHz Atmel AVR 8535 Microocontroller and an RFM radio. The Tiny Active Messages software component consumes 322 bytes of a 2.6KB total binary image. At a 10kbps raw bit rate and using a 4b6 encoding scheme, the wireless link supports 833 bytes/sec of throughput. Figure 4 presents the RTT results for various lengths through a network. A route length of one measures a host computer to base station time (40ms) and reflects the cost of the wired link, device processing, and the host OS overhead. For routes greater than one hop, the RTT includes the latency of the wireless link between two devices. The difference between the two and one hop RTT yields the device-to-device RTT of 78ms. Table 2 presents a cumulative profile of the single hop RTT. The difference between request arrival and reply transmission of .3 ms shows that the Active Message layer only accounts for .75% of the total RTT time over the wired link. This decreases when compared to the longer transmission times of the wireless link.

Exploiting the event based nature of Active Messages enables a high degree of power savings. When no communication or computation is being performed, the device enters a low-power idle state. We measured the power consumption for this idle state, the peak power consumption and the energy required to transmit one bit. The results are presented in Table 3.

6 Conclusion

The TinyOS approach has proven quite effective in supporting general purpose communication among potentially many devices that are highly constrained in

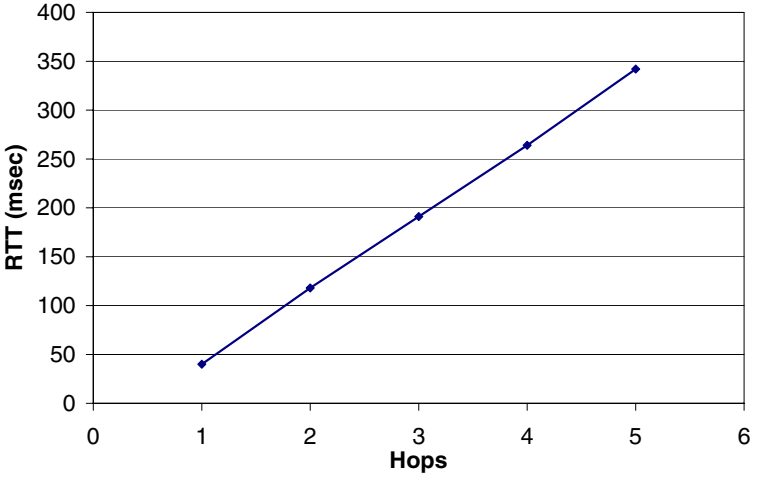


Fig. 4. Round trip times for various route lengths. Note that one hop measures the time for a message between the Host PC and base station device.

Table 2. Cumulative time profile for a single hop RTT test.

Component	Cumulative Time (msec)
First bit of request on device	0
Last bit of request on device	15.6
First bit of reply from device	15.9
Last bit of reply from device	32.8
First bit of next request on device	40.0

Table 3. Power and energy consumption measurements.

Idle State	5 μAmps
Peak	5 mAmps
Energy per bit	1 μJoule

terms of processing, storage, bandwidth, and energy with primitive hardware support for I/O. Its event driven model facilitates interleaving the processor between multiple flows of data and between multiple layers in the stack for each flow while still meeting the severe real-time requirements of servicing the radio. Since storage is very limited, it is common to process messages incrementally at several levels, rather than buffering entire messages and processing them level-by-level. However, events alone are not sufficient; it is essential that an event be able to hand any substantial processing off to a task that will run outside

the real-time window. This provides logical concurrency within the stack and is used at every level except the lowest hardware abstraction layer. By adopting a non-blocking, event-driven approach, we have been able to avoid supporting traditional threads, with the associated multiple stacks and complex synchronization support.

The component approach has yielded not only robust operation despite limited debugging capabilities, it has greatly facilitated experimentation. For example, we had little understanding of what would be the practical error characteristics of the radio channel when the development started, and we ended up building several packet layers that implemented different coding and error detection strategies. The packet components could be swapped with a simple change to the description graph and temporary components could be interposed between existing components, without changing any of the internal implementations. Moreover, the use of components and the TinyOS programming style allows essentially an entire subtree of components to be replaced by hardware and vice versa.

The Tiny Active Message programming model has made it easy to experiment with numerous higher level networking layers and fine-grained distributed algorithms. Although the devices are quite limited, we spend little effort worrying about the low-level machinery while building high-level, often application specific protocols. Several higher level capabilities have recently been developed on this substrate. One example is the ability to reprogram the nodes over the network. A node can obtain code capsules from its neighbors or over multihop routes and assemble a complete execution image in its EEPROM tiny secondary store. The node can then use this to reprogram itself. Other examples include a general purpose data logging and acquisition capability, a facility to query nodes by schema, and to aggregate data from a large number of nodes within the network. We are currently developing mechanisms for operating the radio at five to ten times the bit rate, while keeping all of the higher level structures.

Without the traditional layers of abstraction dictating what kinds of capabilities are available, it is possible to foresee many novel relationships between the application and the underlying system. Our adaptive transmission control scheme is a simple example; rejection of the send request causes the application to adjust its rate of originating data. The application level forwarding of multihop traffic allows the node to keep track of its changing set of neighbors. Moreover, the radio is itself another sensor, since receive signal strength is provided to the ADC. Thus, each packet can be accompanied by signal strength data for use in estimating physical distance or presence of obstructions. The radio is also an actuator, as its signal strength, and therefore cell size, can be controlled. The lowest layer components are synchronizing all receivers to the transmitter to within a fraction of a bit. Thus, very fine grain time synchronization information could be provided with every packet for control applications. What started as a tremendously constraining environment where traditional abstractions were intractable has become a rich and open playground for experimenting with novel software structures for deeply embedded, networked systems.

References

1. Guy Eddon and Henry Eddon. *Inside Distributed COM.* Microsoft Press, Redmond, WA, 1998.
2. Jinyan Li et. al. Capacity of ad hoc wireless networks. In *Proceedings of the Seventh Annual International Conference on Mobile Computing and Networking*, Rome, Italy, July 2001.
3. MPI Forum. Mpi: A message passing interface standard. *International Journal of Supercomputer Applications and High Performance Computing*, 8(3-4):169–416, 1994.
4. Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for networked sensors. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, Cambridge, MA, November 2000.
5. Chalermek Intanagonwiwat, Ramesh Govindan, and Deborah Estrin. Directed: diffusion: a scalable and robust communication paradigm for sensor networks. In *Proceedings of the Sixth Annual International Conference on Mobile Computing and Networking*, August 2000.
6. Alan M. Mainwaring and David E. Culler. Design challenges of virtual networks: Fast, general-purpose communication. In *Proceedings of the 1999 ACM Sigplan Symposium on Principles and Practise of Parallel Programming (PPoPP'99)*, volume 34.8 of *ACM Sigplan Notices*, pages 119–130, A.Y., May 1999.
7. Charles E. Perkins, editor. *Ad Hoc Networking.* Addison-Wesley, New York, NY, 2001.
8. K. S. J. Pister, J. M. Kahn, and B. E. Boser. Smart dust: Wireless networks of millimeter-scale sensor nodes. *1999 Electronics Research Laboratory Research Summary*, 1999.
9. RF Monolithics, Inc. Tr1000 916.50 mhz hybrid transciever. `http://www.rfm.com/products/data/tr1000.pdf`.
10. Sun Microsystems, Inc. Jini network technology. `http://www.sun.com/jini`.
11. T. von Eicken, D. E. Culler, S. C. Goldstein, and K.E. Schauser. Active messages: a mechanism for integrated communication and computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 256–266, Gold Coast, Qld., Australia, May 1992.
12. Alec Woo and David Culler. A transmission control scheme for media acces in sensor networks. In *Proceedings of the Seventh Annual International Conference on Mobile Computing and Networking*, Rome, Italy, July 2001.

Storage Allocation for Real-Time, Embedded Systems*

Steven M. Donahue[1], Matthew P. Hampton[1], Morgan Deters[1], Jonathan M. Nye[1], Ron K. Cytron[1], and Krishna M. Kavi[2]

[1] Washington University
Department of Computer Science
Saint Louis, MO 63130, USA,
cytron@cs.wustl.edu,
http://www.cs.wustl.edu/~doc/
[2] University of North Texas
P.O. Box 311277
Denton, Texas 76203

Abstract. Dynamic storage allocation and automatic garbage collection are among the most popular features that high-level languages can offer. However, time-critical applications cannot be written in such languages unless the time taken to allocate and deallocate storage can be reasonably bounded. In this paper, we present algorithms for automatic storage allocation that are appropriate for real-time and embedded systems. We have implemented these algorithms, and results are presented that validate the predictability and efficiency of our approach.

1 Introduction

Languages featuring dynamic storage allocation and automatic garbage collection continue to grow in popularity—such languages include Java (*Java* is a registered trademark of Sun Microsystems) and ML. Following standard terminology, storage requests are satisfied by *allocating* storage from a *storage heap*. Such storage is *live* or *busy* until such time as the storage is declared *dead*. For languages like Java, an automatic *garbage collection* algorithm can detect dead objects. Other languages offer primitives for dynamically asserting the death of an object. In any case, once the object is declared dead, the storage associated with the object can be *deallocated*, which makes that storage available for subsequent reallocation. Developers of real-time and embedded systems have been slow to embrace automatic storage management for the following reasons.

– *Real-time applications require predictable execution.* Automatic storage management incurs overhead that can be difficult or impossible to predict. In this paper, we examine this issue with respect to storage allocation. For real-time applications, allocation time must be bounded.

* This work is supported by the National Science Foundation under grant 0081214 and by DARPA under contract F33615–00–C–1697

T.A. Henzinger and C.M. Kirsch (Eds.): EMSOFT 2001, LNCS 2211, pp. 131–147, 2001.
© Springer-Verlag Berlin Heidelberg 2001

– *Embedded systems require predictable storage bounds.* Embedded systems are typically deployed without the advantages of a virtual memory backing store. Thus, the heap cannot be extended without physically inserting more RAM into the system.
  As a result, the storage requirements for these kinds of applications must be known in advance. For our purposes, this implies that the size of the run-time heap is fixed and known *a priori*.

Unfortunately, the performance of automatic storage management continues to be problematic for developers of embedded or real-time systems. While there are currently several reasons for this, one concern is that the time taken to perform storage-management functions cannot easily be bounded. Time-critical applications cannot abide such behavior. In our view, the term *time-critical* applies to both of the following.

– An application may be time-critical in the sense that some instruction sequences must execute in a reliable timeframe. Embedded and real-time applications often have this property.
– Hardware support for storage-management functions can be time-critical in the sense that such hardware must be clocked at a predetermined rate for synchronous operation. In this situation, it is better to perform a little work at each storage-management operation than to have some operations execute for-free and others take a very long time.

Ironically, Java was initially designed as a language for embedded-systems applications, but its storage management remains problematic for the following reasons.

– Storage allocation [9] usually involves searching a *free-list* of storage blocks. For garbage-collected programs, the free-list tends to diminish until the point of collection. Following collection, the free-list typically contains a large number of (former) objects. To satisfy a single storage request, searching the free-list could take time proportional to the size of the list—unacceptable for time-critical situations.
– Garbage collection [8] techniques usually require marking (a subset of) the program's live objects; the storage for the unmarked objects can then be returned to the storage manager for reallocation. Collection cycles can happen unexpectedly and can take considerable time; moreover, negative effects on the data and instruction caches are drastic [4,2,1].
– As the program executes, the heap becomes *fragmented*: relatively small holes develop in the heap storage area. To defragment the heap, objects are either copied or compacted during the collection cycle, taking more execution time and toll on the cache.

If the time taken to execute storage-management functions can be reasonably bounded, then languages such as Java could better support time-critical applications. Moreover, it then becomes possible to relegate storage-management activities to hardware where better performance might be obtained.

Our paper's contributions can be summarized as follows.

- Results are presented using Knuth's *buddy system* [6], which bounds the time taken to satisfy a storage allocation request.
- A variation of the buddy system is presented that delays recombination. We present results on the effectiveness and efficiency of this variation.
- An algorithm is presented for defragmenting a buddy system heap. The algorithm is specialized toward satisfying a single request, rather than a mass defragmentation of the heap which can be disruptive to program execution.

Our paper is organized as follows: Section 2 explains our approach and implementation using simple examples. Section 3 presents experiments based on this implementation. Section 4 presents conclusions and ideas for future work in this area.

2 Approach

In this section, we describe our approach for obtaining free blocks of storage in bounded time. Wilson presents an excellent survey of storage allocation [9], and the buddy system upon which we base our work is described well by Knuth [6].

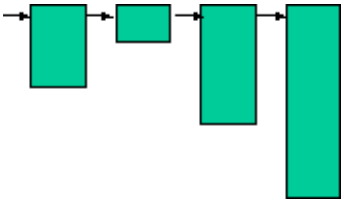


Fig. 1. Unstructured list allocator

2.1 Motivation for Segregated Free-Lists

We first consider a common storage allocation technique, based on maintaining a relatively unstructured list of available blocks, as shown in Figure 1. A storage request is satisfied by searching that free-list for (typically) the first block that is sufficiently large to satisfy the request. While this approach works well in practice, it is possible that the only block that can satisfy a given request is at the far end of the free-list. Because the free-list structure is a function of the allocating program's behavior, it is not easy to bound the time needed to satisfy an allocation request, even if we assume the free-list contains a block of suitable size.

By contrast, consider a storage allocator whose free-lists are *segregated* by size, as shown in Figure 2. For efficiency in obtaining a block of the desired size,

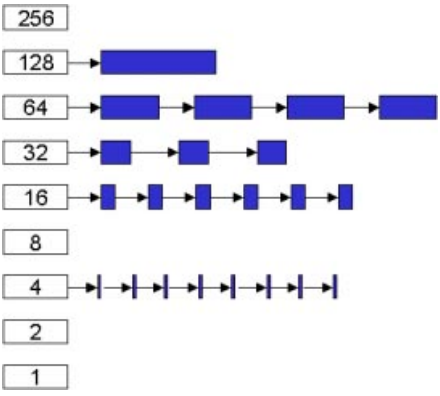


Fig. 2. Data structure for buddy system; objects are segregated by size, and each list of equally-sized blocks is referenced from the *list display*, shown at the left

an indexable slot is reserved to head a linked list for each possible block size. A request for a block of a given size can be satisfied by simply returning a block from the appropriate list. This takes constant time, assuming that a block is available on the appropriate list.

2.2 Buddy System Allocation

Our approach for storage allocation is a segregated-by-size technique, based on Knuth's buddy system [6]. Each storage request is resolved to a block of size 2^k for some positive, integral value of k. Figure 3(a) shows the buddy system's structure in its initial state, assuming the heap is 256 bytes.

The buddy system operates as follows:

1. When the program requests memory, the allocator first calculates the smallest power of 2 that is larger than or equal to the size requested. More specifically, a request of size s is translated into a request of size $2^k, k = \lceil \log_2 s \rceil$.
2. The free-list at index k is consulted for an available block.
3. If a block of size 2^k is not available, then two such blocks can be obtained through bisection of a block of size 2^{k+1}. Figure 3(b) shows the result of subdividing the initial heap into two sub-blocks.
4. Applying this strategy recursively, increasingly larger blocks can be subdivided until a block of size 2^k can be obtained.

For example, the heap in Figure 2 has blocks available of size 16. Thus, a request for a block of size 10 can be satisfied immediately, with the resulting block returned in the time it takes to unlink a block from the size-16 free-list.

Further, consider a request for a block of size 8. Because the list of blocks of size 8 is empty, the buddy system hunts upwards for a larger block that can be subdivided to obtain the desired size. The time necessary for that search is

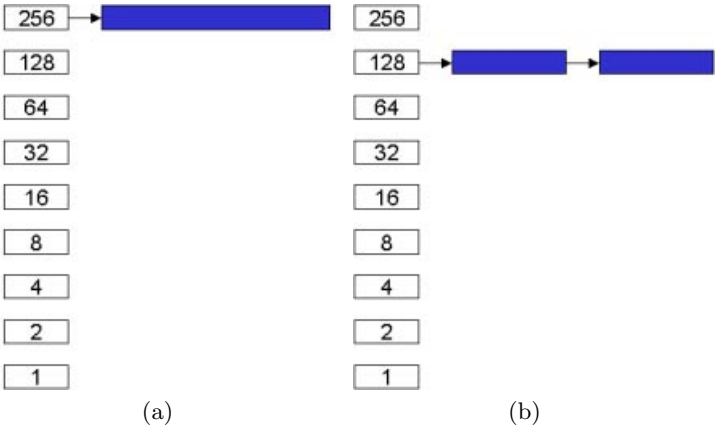


Fig. 3. A block can be divided into two sub-blocks

$O(log(M))$ where M is the size of the storage heap. For embedded and real-time systems, we assume that the heap size is fixed and that $O(log(M))$ time is considered efficient.

2.3 Buddy System Deallocation

When blocks are deallocated, a common problem for most storage-management algorithms is the coalescing of free blocks that happen to lie consecutively into larger blocks. The buddy system greatly simplifies this task. When a block of size 2^{k+1} is bisected into two blocks of size 2^k, the resulting blocks are said to be *buddies* of each other. A buddy of size 2^k can implicitly compute its buddy's address by flipping a predetermined bit of its own address—typically bit k where bit 0 is the rightmost bit.[1]

Figure 4 shows the result of requesting a block of size 16 given the initial condition shown in Figure 2. The initial block is recursively subdivided until two blocks of size 16 are obtained. One of those blocks is returned to satisfy the allocation request, and the other block remains on the free-list for blocks of size 16.

When storage is returned, the buddy system eagerly joins buddies to create ever larger blocks. Thus, if the block allocated in Figure 4 is immediately deallocated, buddies are joined together repeatedly until the heap is returned to the state shown in Figure 2.

2.4 Buddy System Implementation and Behavior

As shown in Figure 2, the buddy system normally keeps an array of linked lists of identically-sized blocks. Each element of the array heads the linked list for blocks

[1] Without loss of generality, we assume the heap's origin is address 0.

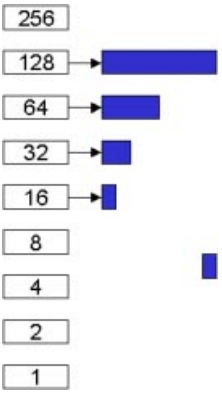


Fig. 4. Allocation using the buddy system

of a given power of two. When a block is deallocated, a check is performed to see if that block's buddy is also free. In support of this test, each block is equipped with one bit to reflect whether it is busy. The bit must be present when the block is actually in use by the allocating program.

When a block is not in use, it is kept on a linked list as shown in Figure 2. The blocks are actually maintained in a *doubly-linked* list, which facilitates quick deletion of a block from its linked list. Space for these links can come from the block itself, since while on the free-list, the space is not otherwise in use. In addition, it is necessary for the block to keep track of its size, which implies that the smallest block that can be managed using the buddy system must be able to accommodate two pointers, along with the size of the block.

Although the buddy system has the advantages stated in this paper, it has not been widely used for storage allocation in programming-language systems for the following reasons.

1. The buddy system tends to fragment storage [6]. This means that the heap may contain sufficient storage to satisfy a request, but the storage is not contiguous within a buddy's boundary. There are two sources of fragmentation:
   Internal fragmentation occurs within a block, when a request for storage is rounded up to a power of two. Consider a request for a block of size s that is translated into a request for size 2^k. For such a request, the number of wasted bytes w must satisfy $0 \leq w < 2^{k-1}$.
   External fragmentation occurs because free storage can be distributed among blocks whose buddies are *busy*—still in use by the allocating program. Such free blocks cannot (at present) be combined.
   From the above, we see that internal fragmentation could waste up to half of a program's data store. We claim that the advantages of the buddy system mitigate such waste. Moreover, we assume that whoever runs a program can establish a reasonable size for the heap, based on program behavior and input data—this must be true for embedded systems which cannot typically

afford a backing store. The bound for the heap's size could be multiplied by 1.5 to obtain a heap where internal fragmentation need not be a concern.
Alternatively, if the size of data types is known *a priori*, then worst-case analysis can identify a (perhaps better) bound on internal fragmentation. This bound can be calculated by finding the data type that maximizes the amount of internal fragmentation. Clearly, the worst-case run of the program would be one in which all allocations are of this type. Thus, the worst-case bound on internal fragmentation would be the level of internal fragmentation created by allocating the worst-case data type.
External fragmentation remains problematic, because the allocating program can cause the heap to reach a state where sufficient storage exists but is unallocatable due to its position in the heap. In this paper we present an algorithm for defragmenting a buddy heap. The algorithm operates on-demand, and defragments just enough storage to satisfy a given request.

2. Performance can be poor for programs that create objects with very short lifetimes. In fact, a typical assumption of Lisp-like programs is that new objects will die soon. In the limiting case, the state of the free-list could oscillate between Figure 2 and Figure 4; if this is repeated many times, the buddy system suffers from overhead as blocks are joined together only to be split again by the next allocation request.
We present a variation of the buddy system in this paper. Dubbed the *estranged buddy system*, we delay block recombination [5] until large storage blocks are needed.

For the purposes of this paper, it is important to understand the extent to which the buddy system operates within bounded time. There are essentially three steps to allocate a block of size 2^k.

1. Starting at index k, search upwards for an available block.
2. Recursively bisect the discovered block until a block of size 2^k is obtained.
3. Return the address of that block.

The first two steps can take time proportional to the size of the list display shown in Figure 2. For a heap of size M, the list display is $\theta(M)$. The final step takes constant time. Most programming language systems insist that any storage returned by an allocator be properly initialized, typically to all-zeros. As reported in Section 3, it is expected that most storage requests are for blocks of relatively small size—16 bytes. For a 16 Mbyte heap, assuming 16 is the smallest request, at most 20 slots could be inspected before a suitable block is found. This bound is quite reasonable when compared with the unknown length of an unstructured free-list.

2.5 Estranged Buddy System

As stated in Section 1, if blocks are allocated and immediately deallocated, then the buddy system could oscillate between subdividing blocks and reuniting buddies. Given our assumptions, such behavior causes no asymptotic difficulties.

As a practical concern, particularly for continuous garbage collection, we investigated the extent to which such wasteful behavior can be eliminated and we report on those results in this paper. The basic idea is to avoid unnecessary block recombination. Although delayed recombination has been previously proposed [5], our goal was to obtain an implementation efficient in terms of its use by embedded and real-time Java applications.

2.6 Motivation for Delayed Recombination

We next examine the conditions under which blocks of storage are deallocated.

Occasional collection is the traditional approach for garbage collection. The collector runs on-demand, when storage becomes scarce or in response to the application program requesting a collection cycle. From the deallocator's point of view, objects are returned in relatively large bursts rather than in a continuous stream.

Continuous collection returns objects in anticipation of future demands. Such a collector could run as a low-priority thread, collecting objects continuously as allowed by available CPU resources. Other techniques include contaminated garbage collection [1], which returns objects upon method return.

For occasional collection, programs appear to alternate between allocating and deallocating states. Objects are not returned in a trickle, but seemingly all at once when the collector runs. For such an approach, it may not be possible to deallocate a given block *immediately* after the block is allocated. The deallocation would have to wait until the next collection cycle.

On the other hand, a continuous collector can exhibit the behavior that calls for delayed recombination. For this paper, our experiments used occasional rather than continuous collection. Future research will investigate the results of the estranged buddy system for such collectors.

2.7 Estranged Buddy Allocation

The estranged buddy system is a variation of Knuth's buddy system that delays recombination of free blocks. When a block is deallocated, it is viewed as *estranged* from its buddy and thus reluctant to rejoin. Although this idea was first proposed by Kaufmann [5], no implementation details were provided. Below, we describe our implementation which is biased toward the behavior we expect from Java programs:

- Programs tend to request many blocks of the same size. In Java, equal type implies equal size, except for arrays. Since programs typically instantiate many objects of the same type, a request for a block of size s implies the likelihood of similar requests in the future.
- Programs tend to allocate relatively small blocks. Good object-oriented design prescribes simple objects with relatively few fields. Thus, most objects in Java are small.

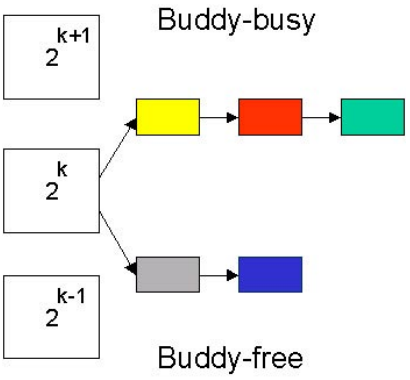


Fig. 5. Estranged buddy list structure

In our implementation of delayed recombination, the estranged buddy system maintains *two* free-lists per size, as shown in Figure 5.

Buddy-busy contains blocks whose buddies are busy. The objects in such blocks are presumably still in use by the allocating program.

Buddy-free contains blocks whose buddies are free. Note that only one of the two free buddies appears on any list. The other buddy's availability is implied by its buddy's presence on the buddy-free list.

The two lists are used so that buddy-busy blocks can be allocated in preference to buddy-free blocks, the latter being saved for recombination should larger blocks be scarce.

Delayed recombination allows increased flexibility in satisfying allocation requests. We implemented the following heuristic to satisfy a request of size 2^k:

1. The buddy-busy list at index k is examined.
2. The buddy-free list at index k is examined.
3. We examine the buddy-free list at index $k-1$, so that two blocks of half the necessary size can be combined. The recombination of such blocks was delayed in the estranged buddy system.
4. We apply the usual buddy algorithm and search above for a large block that can be subdivided.
5. We try to glue from the buddy-free lists of the lowest level up to level k.

Essentially we favor constant-time strategies over searches of the list display. Also, by favoring buddy-busy over buddy-free we tend to preserve opportunities for recombination.

2.8 Defragmenting a Buddy Heap

In this section we examine an algorithm for defragmenting a buddy heap. The algorithm is appropriate for Knuth's buddy system as well as our estranged

buddy system. We assume defragmentation becomes necessary when the heap contains sufficient storage to satisfy a request, but such space is the sum of storage "holes" that are not joinable in the buddy sense. We do not penalize the heap for internal fragmentation—we assume all storage requests are expressed as powers of two, and that wasted space within a block is not allocatable.

With our focus on real-time, embedded systems, a defragmentation algorithm must have the following properties:

- The heap cannot be extended. For an embedded system, the heap size is fixed when the product is delivered. Thus, defragmentation must happen in place.
- The defragmentation must occur in bounded time, since the need for defragmentation cannot be anticipated by the allocating program.

In fact, fragmentation can plague any allocator if blocks of storage can be returned *ad hoc*. Most allocators enter a compaction phase, during which storage is massively reorganized. All live objects are pushed to one end of the heap, and all "holes" pushed toward the other end. The holes are then combined into one large block that is suitable for subsequent allocations. For real-time systems, this approach cannot be reasonably bounded. We therefore developed an approach that liberates sufficient storage to satisfy only the request at hand.

Consider Knuth's buddy system in a situation where an allocation request of size 2^k cannot be satisfied yet space is available. The following must be true:

- There is no block available in any list of size $2^l, l \geq k$. Otherwise, such a block could be bisected until a block of size 2^k is obtained.
- There are no free blocks anywhere that can be combined. This follows from the eager recombination of Knuth's buddy system.

Thus, the only space that is available must be below level k. Defragmentation must then consist of relocating objects so as to free buddies that can be combined to obtain a block of size 2^k.

The key observation is shown in Figure 6. If two blocks are on some free-list at level j, then they are necessarily *not* buddies, as described above. However, each must have a buddy that is currently busy. By exchanging one block's busy buddy with the other free block, two joinable blocks result. This approach can be applied recursively down the size display to obtain a block at level k.

Application of the above defragmentation algorithm to our estranged buddy system is straightforward.

3 Experiments

Based on the implementation described above, we present experiments to investigate the following:

1. How does the performance of our bounded-time allocator compare with a standard, unstructured-list allocator? We are interested in worst-case as well as average performance on standard benchmarks.

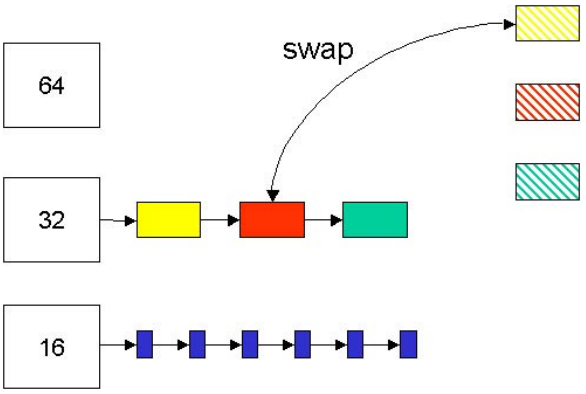


Fig. 6. Defragmentation algorithm

2. How does the performance of our estranged buddy system compare with Knuth's buddy system?
3. What size objects are typically allocated by Java programs?

We implemented our approach in the context of Sun's Java system, JDK 1.1.8. Our changes were confined to those portions of its Java Virtual Machine (JVM) [7] that deal with storage allocation, in particular the `realObjAlloc` method of the `gc` module. Sun's 1.1.8 system offers the following JVM interpreters:

- A reference interpreter is provided, written entirely in C.
- A more efficient interpreter implements the most frequently executed portions in (Sparc) assembly language.

To facilitate our implementation, we based our work on the C version. However, the changes we made are compatible with the architecture of the (speedier) assembly version.

Sun's JVM interpreter manages objects using *handles*. Each handle contains a pointer to the object's current location as well as a reference to an appropriate method table for (virtual) method-lookup. One object can reference another only indirectly through the handles. Thus, if objects are relocated (during garbage collection, for example), only the handle's pointer to the object needs to be updated. To simplify our work, we retained the Sun JVM's use of handles even though our approach avoids relocating objects.

The timings were obtained on a Sparc Ultra 1 running at 167MHz with 128 megabytes of RAM. Our benchmarks consisted of the SPEC benchmarks [3], using their "large" problem size. Figure 7 summarizes the properties of these benchmarks, including the number of objects created and the execution time on the standard JDK 1.1.8 system. The times reported are for the unstructured-list allocator, as shipped with JDK 1.1.8. As shown in Figure 7, the `mpegaudio` and

`compress` benchmarks take significant computational time without allocating many objects. We therefore do not report further results for those benchmarks, but concentrate instead on the others, which do allocate a substantial number of objects.

Name	Description	Lines of source	Objects created	Execution Time (sec)
compress	Modified Lempel-Ziv	6,396	10129	
jess	Expert System	570	7,923,782	1802
raytrace	Ray Tracer	3750	6,346,487	2101
db	Database Manager	1020	3,210,520	3766
javac	Java Compiler	9485	5,902,305	1969
mpegaudio	MPEG-3 decompressor	N/A	7,555	8519
mtrt	Ray Tracer, threaded	3750	6,586,584	2223
jack	PCCTS tool	N/A	6,579,042	2336

Fig. 7. SPEC benchmark properties

3.1 Worst-Case Performance of Buddy over Unstructured-List

A buddy system works well for real-time and embedded systems, not especially due to its average performance, but more due to the bounded nature of its worst-case allocation performance. We compared the JDK 1.1.8 (unstructured-list) allocator against the buddy system on the following contrived example:

1. N objects of constant size k are allocated, filling most of the heap.
2. References to every other object are purged, rendering $N/2$ objects dead.
3. A garbage collection cycle is performed. At the conclusion of the cycle the free-list contains $N/2$ blocks of size k, and a larger "remainder" block at the end.
4. An object of larger size is then requested.

The above scenario causes the unstructured-list allocator to search to the end of its free-list to find a block of suitable size. By contrast, the buddy system would have such blocks segregated by size so they can be found quickly. In Figure 8 we show the results for this example: the buddy system is 72 times faster than the unstructured-list allocator. Because that speedup depends on the length of the unstructured list, it is possible to make the speedup arbitrarily high for a contrived example.

While a contrived example may be unfair, it is important to note that a real program *could* misbehave and that the unstructured-list allocator can provide no *reasonable* bound on allocation time.

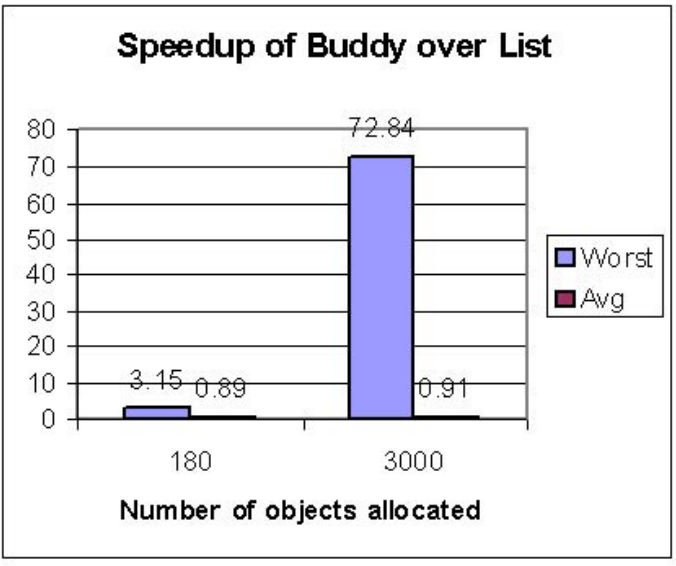


Fig. 8. Speedup of buddy system on a contrived example

Benchmark	Estranged	Knuth
jess	1.04	1.02
raytrace	1.03	0.92
db	1.01	1.01
javac	1.00	1.00
mtrt	1.10	1.02
jack	1.04	1.03

Fig. 9. Speedup of the buddy systems over JDK 1.1.8's unstructured-list allocator

3.2 Average performance of Knuth's Buddy and Estranged Buddy Systems

We next compare the efficiency of the unstructured-list allocator, Knuth's buddy system, and our estranged buddy system on the SPEC benchmarks. We ran the SPEC benchmarks, large size (100) under the following conditions:

- The JDK 1.1.8 system is equipped with a standard, unstructured-list allocator.
- Knuth's buddy system eagerly recombines blocks.
- The estranged buddy system delays recombination as described in Section 2.

Figure 9 shows that Knuth's buddy system operates well on these benchmarks, but sometimes loses performance. On the other hand, the estranged buddy sys-

tem can be up to 10 percent faster than the (JDK 1.1.8) unstructured-list allocator.

Admittedly, neither approach offers tremendous improvement. However, these results show that the advantages of both systems in terms of worst-case performance do not come with a loss in average performance, especially when recombination is delayed.

3.3 Qualitative Analysis of Estranged Buddy

We implemented the estranged buddy system described in Section 2. While the execution times reported above are an overall indication of our allocator's performance, we investigated qualitatively *how* requests are satisfied. The shaded portions of Figure 10 show the percentage of allocation requests that were satisfied immediately, by finding a block of size 2^k available in slot k of the size display. Figure 10 shows that by delaying recombination, significantly more blocks can be found without having to break larger blocks or glue smaller blocks. The estranged buddy system finds a block available in the buddy-free or buddy-busy list almost 90% of the time, while the Knuth implementation finds a block immediately only 50% of the time.

3.4 Object Size

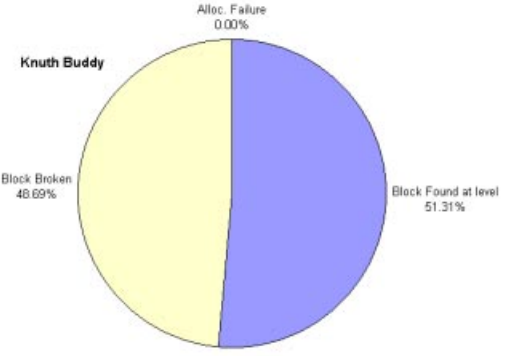
We show in this section that one reason for the estranged buddy system's success is that most allocation requests by Java programs are for blocks of relatively small size. In fact, JDK 1.1.8 cannot allocate fewer than 16 bytes for an object, and a large number of requests are for size-16 blocks. We next examine two of the SPEC benchmarks in detail, showing their distribution of allocation requests.

The `raytrace` application is typical of those SPEC benchmarks that allocate many objects. Figure 11(a) shows the distribution of allocation requests for that `raytrace`. On the other hand, the `compress` benchmark allocates relatively few objects, but Figure 11(b) shows that some of those objects are large, presumably used for holding the data for compression.

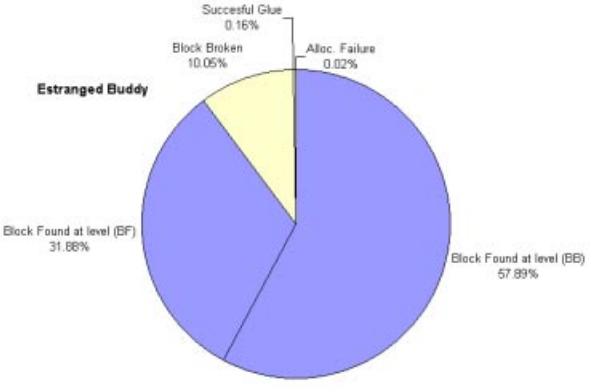
4 Conclusions

From the experiments we have conducted, we have shown the following:

1. An allocator hat segregates the free-list by size offers a reasonable bound on the amount of time required for memory allocation and deallocation.
2. Delayed recombination of memory blocks (as in estranged buddy) increases significantly the chances that memory blocks will be available immediately upon an allocation request.
3. The estranged buddy system offers performance gains over Knuth's buddy system and the standard list allocator.

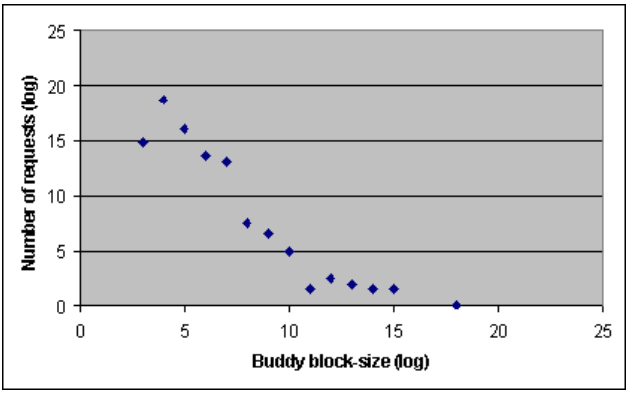


(a)

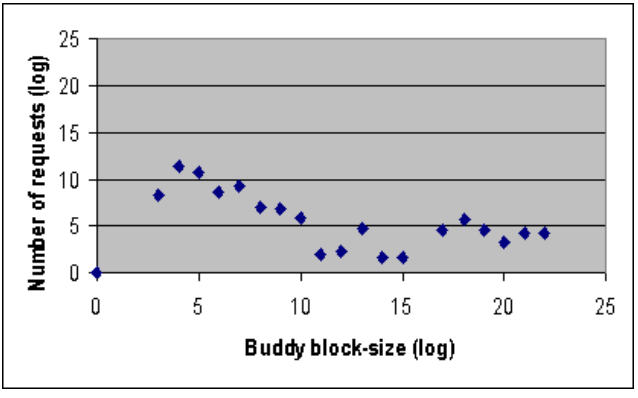


(b)

Fig. 10. Handling of requests by (a) Knuth's buddy system allocator and (b) the estranged buddy system allocator



(a) `raytrace`



(b) `comrpess`

Fig. 11. Allocation requests

Arguably, the structured-list allocator is bounded, but its bound is $O(M)$—in fact, it is difficult to imagine an allocator whose performance is worst than $O(M)$. For real-time applications, cost analysis of an operation must consider worst-case behavior. A worst-case assumption of $O(M)$ for object instantiations causes gross overprovisioning for most allocations but is a necessarily conservative bound. Asymptotically, both Knuth's buddy system and the estranged buddy system operate in $O(\log M)$ time, where M is the size of the heap. For real-time and embedded systems, this bound should be sufficient to obtain reasonable performance without overly provisioning for allocation times. For future work, we will investigate the extent to which the time for responding to an allocation request could be effectively *constant*. To obtain such an improvement, the allocator must go beyond segregation by size.

Acknowledgements. We thank Sun Microsystems for access to their Java interpreter. We thank Conrad E. Warmbold of Washington University for his help in preparing figures for this paper.

References

1. Dante J. Cannarozzi, Michael P. Plezbert, and Ron K. Cytron. Contaminated garbage collection. *Programming Language Design and Implementation*, 2000.
2. Trishul Chilimbi and James Larus. Using generational garbage collection to implement cache-conscious data placement. *Proceedings of the International Symposium on Memory Management*, 1998.
3. SPEC Corporation. Java spec benchmarks. Technical report, SPEC, 1999. Available by purchase from SPEC.
4. Scott Haug. Automatic storage optimization via garbage collection. Master's thesis, Washington University, 1999.
5. Arie Kaufman. Tailored-list and recombination-delaying buddy systems. *ACM Transactions on Programming Languages and Systems*, 6(1):118–125, January 1984.
6. Donald E. Knuth. *Fundamental Algorithms, Volume 1, The Art of Computer Programming, Second Edition*. Addison-Wesley, 1973.
7. Tim Lindholm and Frank Yellin. The *Java Virtual Machine Specification*. Addison-Wesley, 1997.
8. Paul R. Wilson. Uniprocessor garbage collection techniques (Long Version). Submitted to ACM Computing Surveys, 1994.
9. Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In Henry Baker, editor, *Proceedings of International Workshop on Memory Management*, volume 986 of *Lecture Notes in Computer Science*, Kinross, Scotland, September 1995. Springer-Verlag.

Interface Theories for Component-Based Design*

Luca de Alfaro[1] and Thomas A. Henzinger[2]

[1] University of California, Santa Cruz
[2] University of California, Berkeley

Abstract. We classify component-based models of computation into component models and interface models. A component model specifies for each component how the component behaves in an arbitrary environment; an interface model specifies for each component what the component expects from the environment. Component models support compositional abstraction, and therefore component-based verification. Interface models support compositional refinement, and therefore component-based design. Many aspects of interface models, such as compatibility and refinement checking between interfaces, are properly viewed in a game-theoretic setting, where the input and output values of an interface are chosen by different players.

1 Interfaces vs. Components, Informally

A generic way of depicting system structure is the block diagram. A block diagram consists of entities called blocks related by an interconnect, which specifies a topology for communication between the blocks. A block may represent a physical or logical component, such as a piece of hardware or software, but it more often represents either an abstract description of the *component*, or a description of the component *interface*. A component description answers the question *What does it do?*; an interface description answers the question *How can it be used?*. A component description may be very close to the underlying component, or it may specify as little as a single property of the underlying component. Interface descriptions, too, can be more or less detailed, but they must contain enough information for determining how the underlying components can be composed and connected, and like any good abstraction, they should not contain more information. Component designers often make assumptions about the environment in which a component is to be deployed, and such assumptions, while not part of the component itself and therefore not part of any abstract component description, can be part of an interface description.

Components do not constrain the environment; interfaces do. Consider, for example, a division component with inputs x and y and output z. Suppose that our description language is the predicate calculus. The predicate

$$x \in \mathbb{R} \wedge y \in \mathbb{R} \backslash \{0\} \Rightarrow z = x/y \tag{1}$$

* This research was supported in part by the AFOSR MURI grant F49620-00-1-0327, the DARPA ITO grant F33615-00-C-1693, the MARCO grant 98-DT-660, and the NSF ITR grant CCR-0085949.

T.A. Henzinger and C.M. Kirsch (Eds.): EMSOFT 2001, LNCS 2211, pp. 148–165, 2001.
© Springer-Verlag Berlin Heidelberg 2001

is a description of the division component. It does not constrain the environment, but describes the behavior of the component in an arbitrary environment: "for all inputs x and y, if $y \not= 0$, then the output is $z = x/y$." A more abstract description of the same component might be

$$x \in \mathbb{R} \wedge y \in \mathbb{R}\backslash\{0\} \Rightarrow z \in \mathbb{R}.$$

By contrast, the predicate

$$x \in \mathbb{R} \wedge y \in \mathbb{R}\backslash\{0\} \wedge z \in \mathbb{R}$$

is a description of the interface for the division component. It describes an expectation the component has about its environment: "input x is expected to be a real, and input y is expected to be a real different from 0." Such a constraint on the environment is called an *input assumption*. Since the environment consists of other components, a useful interface description not only constrains the environment, but also offers symmetric information about the underlying component, which can then be compared against the input assumptions of interfaces for the environment components. The reciprocal information is called *output guarantee*; in our example, it is "output z is guaranteed to be a real." A more detailed interface description for the division component might be

$$x \in \mathbb{R} \wedge y \in \mathbb{R}\backslash\{0\} \wedge z = x/y. \tag{2}$$

While the component description (1) asserts that "*if* the environment provides proper inputs, *then* the component produces the desired result," the interface description (2) asserts that "the environment provides proper inputs *and* the component produces the desired result."

The informal litmus test *Does it constrain the environment?* applies naturally to many open systems (i.e., systems with free inputs) and open-system descriptions. For instance, a physical circuit is a component, because it behaves (or misbehaves) somehow in all environments; the pin assignment of a hardware chip is an interface, because it puts an expectation on the way in which the chip is deployed. The body of a Pascal procedure is a component; its parameter declaration is an interface. An I/O automaton [13] is a component; an interface automaton [5], which has the same syntax as an I/O automaton, is an interface.

Components and interfaces have different well-formedness criteria. The well-formedness criterion for components is *input-universal*:

$$(\forall x, y)(\exists z)(x \in \mathbb{R} \wedge y \in \mathbb{R}\backslash\{0\} \Rightarrow z = x/y).$$

In other words, a component predicate is well-formed if it is true in *all* environments. This criterion expresses the fact that the component does not constrain the environment and produces an output in an arbitrary environment. The well-formedness criterion for interfaces is *input-existential*:

$$(\exists x, y)(\exists z)(x \in \mathbb{R} \wedge y \in \mathbb{R}\backslash\{0\} \wedge z = x/y).$$

In other words, an interface predicate is well-formed if it is true in *some* environment. An environment that makes the interface predicate true, and thus enables the production of an output, is called a "helpful" environment. The well-formedness criterion expresses the input assumption that the environment is helpful. Input-universal vs. input-existential interpretations of open-system descriptions have also been called *pessimistic* (the environment is allowed to be arbitrary) vs. *optimistic* (the environment is expected to be helpful) approaches to the modeling of open systems [5].

Composition. Composition is a partial function on components, because the result of putting together two well-formed components may not be well-formed. However, as neither component constrains its environment, in many component models there are simple *compatibility* criteria, which ensure that any two compatible components can be composed. For synchronous component models, composition can be more involved because of the possibility of circular I/O dependencies [6,7] (for example, an inverter component $y = \neg x$ cannot be composed with an identity component $x = y$, despite the fact that both are well-formed). The composition of *interfaces* must resolve both input assumptions and output guarantees: two well-formed interfaces can be composed if (1) they mutually satisfy their respective input assumptions, and (2) the composition (which may still have free inputs) is again a well-formed interface. Interface compatibility can be viewed as a game between the two interfaces and their environment: the environment attempts to be helpful and meet the input assumptions of both interfaces; the interfaces attempt to prevent this. If the environment has a winning strategy in this game, then the the two interfaces can be composed, because then the composition has again a helpful environment.

In the formal treatment below, we split composition into two operations, one for collecting sets of blocks (i.e., components or interfaces), and the other for relating them by an interconnect. This separation, which is inspired by block-diagram languages [12], orthogonalizes concerns, and thus guides and simplifies the presentation.

Hierarchy. Hierarchical block diagrams support abstraction and refinement. Abstraction allows a block diagram to be compressed into a single block; refinement allows a block to be expanded into a block diagram. The hierarchical relationship between blocks is *contravariant* on inputs and outputs: a more refined description of an open system may make weaker input assumptions and stronger output guarantees than a more abstract description of the same system. For components, which make no input assumptions, abstraction is therefore weakening, such as implication or trace containment or simulation, and refinement is strengthening. For interfaces, which do make input assumptions, hierarchy maintains its contravariant character, and can be defined as alternating trace containment or alternating simulation [3].

Components typically support *compositional abstraction*: for two compatible components f' and g', if f' can be abstracted to f, and g' can be abstracted

to g, then f and g are again compatible. This is because for components, abstraction is weakening (say, implication) and compatibility (say, nonemptiness of conjunction) is made more likely by weakening. Compositional abstraction is the basis for component-based *verification* methods, which proceed bottom-up from a system description and require the following: in order to prove that the given system $f' \| g'$ refines the specification $f \| g$, it suffices to *independently* prove that the component f' refines the partial specification f, and that g' refines g. Assume-guarantee rules for compositional verification are more elaborate, but have the same "direction" [1,9]: given the composite system, if we establish properties of the components, then we can conclude properties of the composite system.

Interfaces, by contrast, can be made to support *compositional refinement*, which is the "opposite direction" of abstraction: for two compatible interfaces F and G, if F can be refined to F', and G can be refined to G', then F' and G' are again compatible. This is possible because for interfaces, refinement weakens input assumptions and thus can make compatibility more likely. Compositional refinement is the basis for component-based *design*, which proceeds top-down from an interface description and requires the following: in order to refine the given interface $F \| G$ towards an implementation, it suffices to *independently* refine F and G, say, to F' and G', respectively; then the refinements F' and G' are compatible and their composition refines the interface $F \| G$. We present several interface formalisms that permit component-based design in this sense.

Formalism. Formalism enables tool support. Formal component models can support compositional verification with algorithmic tools for component-abstraction checking (e.g., "model checking" [4]). Formal interface models can support compositional design with algorithmic tools for interface-compatibility checking and interface-refinement checking. Interface models are typically less ambitious and smaller than component models, which often attempt to capture behavioral aspects of the underlying components for verification. Formal interface models in general, and automatic compatibility checking in particular, therefore offer an opportunity for formal methods to succeed and have practical impact on the design process.

While formal component models (e.g., [1,10,13]) and informal interface models (e.g., [14]) abound, formal interface models are less common. Notable examples are *trace theory* [8] and *lazy composition* [15], both of which combine, according to our terminology, component and interface aspects. Most formalisms, however, limit the input assumptions to assumptions about the types of input values. In practice, on the other hand, designers make much richer input assumptions. For example, an object-oriented software designer of a class with an initialization method and other methods may require that the initialization method is called before any of the other methods is called. Such temporal-ordering assumptions can be captured by *interface automata*, a formal interface model for asynchronous component interaction [5]. An embedded software designer of a control task may assume that a sensor input is updated with a certain frequency

(an assumption on the plant), or that the task has a certain worst-case execution time (an assumption on the hardware), or that the task finishes execution within a certain time bound (an assumption on hardware and scheduler). Such real-time assumptions require a rich interface model with timing assumptions and timing guarantees, which has to wait for later work. In this paper, we lay the foundation for interface formalisms by defining the framework and presenting a few formal interface models, called *interface theories*, for simple kinds of synchronous component interaction.

2 Interfaces vs. Components, Formally

We capture hierarchical block diagrams formally by a mathematical object called block algebra. A *block algebra* consists of:

- A set of *blocks*.
- For each block F, a set P_F of *ports*. A *port* is a typed variable. We assume that all types are nonempty and write $\mathbb{T}_x$ for the type of port x.
- A partial binary function, called *composition*, mapping two blocks F and G to a block $F\|G$. We require that if the composition $F\|G$ is defined, then $P_{F\|G} = P_F \cup P_G$. We also require that composition is commutative and associative: (1) if $F\|G$ is defined, then $G\|F$ is defined and equal to $F\|G$; (2) if $(F\|G)\|H$ is defined, then $F\|(G\|H)$ is defined and equal to $(F\|G)\|H$. In other words, composition is a partial function that maps a set of blocks to a single block.
- A partial binary function, called *connection*, mapping a block F and an interconnect θ to a block $F\theta$. An *interconnect* is a set of channels, and a *channel* (x, y) is a pair consisting of a port x called *source*, and a port y called *target*, such that $\mathbb{T}_x = \mathbb{T}_y$. Given an interconnect θ, we write $I_\theta = \{x \mid (\exists y)(x, y) \in \theta\}$ for the set of sources, $O_\theta = \{y \mid (\exists x)(x, y) \in \theta\}$ for the set of targets, and ρ_θ for the predicate $\bigwedge_{(x,y)\in\theta}(x = y)$. We require that if the connection $F\theta$ is defined, then $P_{F\theta} = P_F \cup I_\theta \cup O_\theta$. We also require that if $\theta = \emptyset$, then $F\theta$ is defined and equal to F.
- A binary relation $\preceq$, called *hierarchy*, between blocks. If $F' \preceq F$, then the block F' is said to *refine* the block F, and F is said to *abstract* F'. We require that $\preceq$ is reflexive and transitive.

We distinguish between block algebras whose blocks represent interfaces, and block algebras whose blocks represent components: interface algebras support top-down design; component algebras support bottom-up verification. Top-down design iteratively refines a block F into a block F' such that $F' \preceq F$; bottom-up verification iteratively abstracts a block f' into a block f such that $f' \preceq f$. A block algebra is an *interface algebra*, and the blocks are called *interfaces*, if both of the following:

1. For all interfaces F, G, and F', if $F \| G$ is defined and $F' \preceq F$, then $F' \| G$ is defined and $F' \| G \preceq F \| G$.
2. For all interfaces F and F', and all interconnects θ, if $F\theta$ is defined and $F' \preceq F$, then $F'\theta$ is defined and $F'\theta \preceq F\theta$.

A block algebra is a *component algebra*, and the blocks are called *components*, if both of the following:

1. For all components f', g', and f, if $f' \| g'$ is defined and $f' \preceq f$, then $f \| g'$ is defined and $f' \| g' \preceq f \| g'$.
2. For all components f' and f, and all interconnects θ, if $f'\theta$ is defined and $f' \preceq f$, then $f\theta$ is defined and $f'\theta \preceq f\theta$.

Interfaces and components are related by implementations. Given an interface algebra $\mathcal{A}$ and a component algebra $\mathcal{B}$, an *implementation of $\mathcal{A}$ by $\mathcal{B}$* is a binary relation $\lhd$ between the components of $\mathcal{B}$ and the interfaces of $\mathcal{A}$ such that for every interface F, there exists a component f with $f \lhd F$; that is, every interface can be implemented. If $f \lhd F$, then the component f is said to *implement* the interface F, and F is called an *interface for* f. Component-based design is supported by compositional implementations. The implementation $\lhd$ is *compositional* if all of the following:

1. For all components f and g of $\mathcal{B}$, and all interfaces F and G of $\mathcal{A}$, if $f \lhd F$ and $g \lhd G$ and $F \| G$ is defined, then $f \| g$ is defined and $f \| g \lhd F \| G$.
2. For all components f of $\mathcal{B}$, all interfaces F of $\mathcal{A}$, and all interconnects θ, if $f \lhd F$ and $F\theta$ is defined, then $f\theta$ is defined and $f\theta \lhd F\theta$.
3. For all components f of $\mathcal{B}$, and all interfaces F' and F of $\mathcal{A}$, if $f \lhd F'$ and $F' \preceq F$, then $f \lhd F$.

If $\mathcal{A}$ is an interface algebra, $\mathcal{B}$ a component algebra, and $\lhd$ a compositional implementation of $\mathcal{A}$ by $\mathcal{B}$, then the pair $(\mathcal{A}, \lhd)$ is called an *interface theory* for $\mathcal{B}$. There may be several interface theories for a component algebra. Given two interface theories $(\mathcal{A}_1, \lhd_1)$ and $(\mathcal{A}_2, \lhd_2)$ for the component algebra $\mathcal{B}$, the theory $(\mathcal{A}_2, \lhd_2)$ is *as expressive for $\mathcal{B}$ as* the theory $(\mathcal{A}_1, \lhd_1)$ if there is a function α from the interfaces of $\mathcal{A}_1$ to the interfaces of $\mathcal{A}_2$ such that all of the following:

1. For all interfaces F and G of $\mathcal{A}_1$, if $F \| G$ is defined, then $\alpha(F) \| \alpha(G)$ is defined and equal to $\alpha(F \| G)$.
2. For all interfaces F of $\mathcal{A}_1$, and all interconnects θ, if $F\theta$ is defined, then $\alpha(F)\theta$ is defined and equal to $\alpha(F\theta)$.
3. For all interfaces F' and F of $\mathcal{A}_1$, if $F' \preceq F$, then $\alpha(F') \preceq \alpha(F)$.
4. For all interfaces F of $\mathcal{A}_1$, and all components f of $\mathcal{B}$, if $f \lhd_1 F$, then $f \lhd_2 \alpha(F)$.

Interface theories support compositional design. Suppose that we want to design a component that implements a given interface F. An interface theory allows us to split the design task into a number of subtasks handled by independent designers. We can refine the interface F into an interface of the form

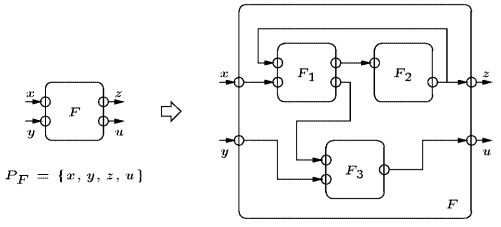


Fig. 1. Component-based design refinement using an interface theory

$(F_1 \| \cdots \| F_k)\theta$; that is, the block F is refined into k blocks that are connected by a set θ of channels (cf. Figure 1). The new interfaces F_1 to F_k can be handed off to k different designers. Suppose that the first designer builds a component f_1 that implements the interface F_1, the second designer buys off the shelf a component f_2 that implements F_2, etc. Then the interface theory guarantees that (1) the k components can be composed and connected to form the system $(f_1 \| \cdots \| f_k)\theta$, and (2) this system implements the given interface F. Mathematically, if $(F_1 \| \cdots \| F_k)\theta \preceq F$, and $f_j \lhd F_j$ for all $1 \leq j \leq k$, then $(f_1 \| \cdots \| f_k)\theta \lhd F$. More generally, such a compositional design process can proceed through multiple levels of refinement. Given two interface theories $(\mathcal{A}_1, \lhd_1)$ and $(\mathcal{A}_2, \lhd_2)$ for a component algebra $\mathcal{B}$, if $(\mathcal{A}_2, \lhd_2)$ is as expressive for $\mathcal{B}$ as $(\mathcal{A}_1, \lhd_1)$, than every compositional design process that is carried out in the interface theory $(\mathcal{A}_1, \lhd_1)$ can also be carried out in the interface theory $(\mathcal{A}_2, \lhd_2)$.

An interface theory also supports the component-wise evolution of a design. Suppose that the system $(f_1 \| \cdots \| f_k)\theta$ implements the interface F. If we want to replace a component f_j by another component f'_j, and we are given the corresponding interface F_j, then we need to ensure only $f'_j \lhd F_j$ in order to put together a revised system $(f_1 \| \cdots \| f'_j \| \cdots \| f_k)\theta$ that implements F.

Component algebras support compositional verification. Suppose that we want to verify that a given system $(f'_1 \| \cdots \| f'_k)\theta$ satisfies a specification f, which may be a more abstract description of the system, or a property of the system. A component algebra allows us to split the verification task into a number of subtasks of smaller complexity. We can establish independently the k proof obligations that each component f'_j satisfies a corresponding specification f_j. Then the component algebra guarantees that the k partial specifications can be composed and connected to form a single specification $(f_1 \| \cdots \| f_k)\theta$, which, it remains to be shown, implies the given specification f. Mathematically, if $f'_j \preceq f_j$ for all $1 \leq j \leq k$, and $(f_1 \| \cdots \| f_k)\theta \preceq f$, then $(f'_1 \| \cdots \| f'_k)\theta \preceq f$. More generally, such a compositional verification process can proceed through multiple levels of abstraction.

3 Some Stateless Interface Algebras

We begin by considering some examples of interfaces without state, and defer interfaces with state to Section 5. Stateless interfaces may of course be implemented by stateful components. For example, while the parameter declaration of a Pascal procedure is a stateless interface similar to the stateless I/O interfaces defined below, the body of a Pascal procedure is a component that typically has state (e.g., local variables). We discuss three classes of stateless interfaces:

1. An *input/output* (I/O) *interface* constrains the environment of a component by specifying the names and types of input ports. Symmetrically, it provides the same information about output ports.
2. An *assume/guarantee* (A/G) *interface* is an I/O interface that constrains, in addition, the ranges of values expected at input ports. Symmetrically, it provides range information about output ports. The example

$$x \in \mathbb{R} \wedge y \in \mathbb{R} \backslash \{0\} \wedge z \in \mathbb{R}$$

   from the introduction is a stateless A/G interface.
3. A *port-dependency* (PD) *interface* is an I/O interface that constrains which output ports may influence the values at which input ports. Symmetrically, it provides dependency information between input and output ports.

3.1 The Stateless Input/Output Interfaces

A *stateless I/O interface* F consists of a set I_F of *input ports*, a set O_F^+ of *available ports* disjoint from I_F, and a set $O_F \subseteq O_F^+$ of *output ports*. The available ports are reserved names for choosing output ports when refining the interface; they are used to ensure that whenever two interfaces are implemented independently by two components, then the components have different output ports. Let $P_F = I_F \cup O_F$ and $P_F^+ = I_F \cup O_F^+$.

`Composition` $F \| G$ is defined iff $P_F^+ \cap P_G^+ = \emptyset$. Then, $I_{F\|G} = I_F \cup I_G$ and $O_{F\|G}^+ = O_F^+ \cup O_G^+$ and $O_{F\|G} = O_F \cup O_G$.

`Connection` $F\theta$ is defined iff (1) $I_\theta \subseteq O_F$, (2) $O_\theta \cap O_F^+ = \emptyset$, and (3) for all channels $(x, y), (x', y') \in \theta$, if $x \not= x'$, then $y \not= y'$; that is, two channels cannot have the same target. Then, $I_{F\theta} = I_F \backslash O_\theta$ and $O_{F\theta}^+ = O_F^+ \cup O_\theta$ and $O_{F\theta} = O_F \cup O_\theta$.

`Hierarchy` $F' \preceq F$ iff $I_{F'} \subseteq I_F$ and $O_{F'}^+ \subseteq O_F^+$ and $O_{F'} \supseteq O_F$. Note the contravariance between inputs and outputs: a component that implements a refinement of the interface F may not use all input ports in I_F, but it must provide values at all output ports in O_F, because those may be expected by the environment.

Proposition 1. *The stateless I/O interfaces are an interface algebra.*

3.2 The Stateless Assume/Guarantee Interfaces

A *stateless A/G interface* F consists of a stateless I/O interface $\Pi_F = (I_F, O_F^+, O_F)$, a satisfiable predicate ϕ_F on I_F called *input assumption*, and a satisfiable predicate ψ_F on O_F called *output guarantee*. The input assumption specifies the value combinations at the input ports which a component that implements the interface must accept, and the output guarantee specifies the value combinations at the output ports which such a component may produce.

`Composition` $F \| G$ is defined iff $\Pi_F \| \Pi_G$ is defined. Then, $\phi_{F\|G} = \phi_F \wedge \phi_G$ and $\psi_{F\|G} = \psi_F \wedge \psi_G$.

`Connection` $F\theta$ is defined iff (1) $\Pi_F\theta$ is defined and (2) the input assumption $\phi_{F\theta}$, as defined next, is satisfiable. Let $\phi_{F\theta} = (\forall O_{F\theta})(\psi_F \wedge \rho_\theta \Rightarrow \phi_F)$ and $\psi_{F\theta} = \psi_F \wedge \rho_\theta$. The input assumption $\phi_{F\theta}$ states that a component that implements the interface $F\theta$ expects inputs so that the input assumption of F is satisfied, provided the outputs of F, some of which may be connected to inputs by θ, do not violate the output guarantee of F.

`Hierarchy` $F' \preceq F$ iff (1) $\Pi_{F'} \preceq \Pi_F$, (2) the input assumption $\phi_{F'}$ is implied by ϕ_F, and (3) the output guarantee $\psi_{F'}$ implies ψ_F. Note the contravariance between input assumptions and output guarantees: a component that implements a refinement of the interface F must be prepared to accept all inputs that satisfy the input assumption ϕ_F, and it may produce only outputs that satisfy the output guarantee ψ_F.

Proposition 2. *The stateless A/G interfaces are an interface algebra.*

Example 1. Consider the stateless A/G interface F with two input ports x and y, and an output port z, all integer-valued. The input assumption is $\phi_F = (x = 0 \Rightarrow y = 0)$; that is, the environment is expected to provide input values so that either the x value is different from 0, or both input values are 0. The output guarantee is $\psi_F = \text{TRUE}$; that is, a component that implements F may produce any integer output. The connection $F\theta$ with $\theta = \{(z, x)\}$ is legal (i.e., defined): as the environment of $F\theta$ does not know whether or not the value at x is 0, to be on the safe side, it must provide the value 0 at the remaining input port y; that is, $\phi_{F\theta} = (y = 0)$. Since now both z and x are output ports, we have $\psi_{F\theta} = (z = x)$. The connection $F\theta'$ with $\theta' = \{(z, y)\}$ is also legal: to be on the safe side, the environment must provide a value different from 0 at the remaining input port x; that is, $\phi_{F\theta'} = (x \neq 0)$ and $\psi_{F\theta'} = (z = y)$.

The stateless A/G interface F' is just like F, except that it has the stronger output guarantee $\psi_{F'} = (z \neq 0)$; that is, a component that implements F' is guaranteed to produce an output value different from 0. The interface F' has fewer implementations than F; indeed, F' refines F. Consequently, the connection $F'\theta$ with $\theta = \{(z, x)\}$ is still legal: in fact, as x is guaranteed to be different from 0, the environment is free to provide any input value at y; that is, $\phi_{F'\theta} = \text{TRUE}$ and $\psi_{F'\theta} = (z \neq 0 \wedge z = x)$. Also the connection $F'\theta'$ with $\theta' = \{(z, y)\}$ is still legal: $\phi_{F'\theta'} = (x \neq 0)$ and $\psi_{F'\theta'} = (z \neq 0 \wedge z = y)$. Note that both $F'\theta \preceq F\theta$ and $F'\theta' \preceq F\theta'$, as predicted. □

Example 2. Consider the stateless A/G interface G with two input ports x and y, and two output ports z and u, all boolean-valued. The input assumption is $\phi_G = (x = y)$ and the output guarantee is $\psi_G = (z = u)$; that is, the environment is expected to provide equal input values at x and y, and the component guarantees the output values at z and u to be equal. The connection $G\theta$ with $\theta = \{(z, x), (u, y)\}$ is legal: $\phi_{G\theta} = \text{TRUE}$ (there are no more inputs) and $\psi_{G\theta} = (z = u \wedge z = x \wedge u = y)$. However, the connection $G\theta_1$ with $\theta_1 = \{(z, x)\}$ is illegal (i.e., undefined), because there is no condition on the remaining input y that would guarantee that $x = y$. The connection $G\theta_2$ with $\theta_2 = \{(u, y)\}$ is similarly illegal. This shows the need for considering *sets* of channels as interconnects, rather than considering individual channels one at a time. □

3.3 The Stateless Port-Dependency Interfaces

A *stateless PD interface* F consists of a stateless I/O interface $\Pi_F = (I_F, O_F^+, O_F)$ and an *I/O-dependency relation* $\kappa_F \subseteq I_F \times O_F$. Intuitively, $(x, y) \in \kappa_F$ means that the value at input port x can influence the value at output port y.

`Composition` $F \| G$ is defined iff $\Pi_F \| \Pi_G$ is defined. Then, $\kappa_{F\|G} = \kappa_F \cup \kappa_G$.

`Connection` $F\theta$ is defined iff (1) $\Pi_F\theta$ is defined and (2) for all channels $(x, y) \in \kappa_F$, we have $(y, x) \not\in \theta$. In other words, the port dependencies introduced by the interconnect θ must not close a dependency cycle. Let $\kappa^*_{F\theta} \subseteq P_{F\theta} \times O_{F\theta}$ be the smallest transitive relation such that $\kappa_F \subseteq \kappa^*_{F\theta}$ and $\theta \subseteq \kappa^*_{F\theta}$. Then, $\kappa_{F\theta} = \kappa^*_{F\theta} \cap (I_{F\theta} \times O_{F\theta})$.

`Hierarchy` $F' \preceq F$ iff $\Pi_{F'} \preceq \Pi_F$ and $\kappa_{F'} \cap (I_F \times O_F) \subseteq \kappa_F$. In other words, a component that implements a refinement of the interface F must not have more I/O dependencies than permitted by the I/O dependency relation κ_F.[1]

Proposition 3. *The stateless PD interfaces are an interface algebra.*

4 Some Component Algebras

For each interface algebra $\mathcal{A}$ from the previous section, we give an example of a component algebra $\mathcal{B}$ such that there is a compositional implementation $\lhd$ of $\mathcal{A}$ by $\mathcal{B}$. All examples of component algebras presented here are *netlists*, i.e., sets of atomic blocks called processes which are connected by channels. In the most general case, each process specifies a nonempty relation between input and output ports. Such a relational net is well-formed if there exist port values that satisfy the I/O relations of all processes, and all identities enforced by channels. The stateless I/O interfaces, the stateless A/G interfaces, and the stateless PD interfaces each capture sufficient conditions on the well-formedness of a subclass of relational nets, and thus provide an interface theory for that subclass.

[1] Exactly the opposite condition is required in *component* algebras with dependency relations: a more abstract component may have fewer I/O dependencies (cf. [1]).

Notation. A *valuation* p on a set P of ports is a function that maps each port $x \in P$ to a value $p(x) \in \mathbb{T}_x$. We write $[P]$ for the set of valuations on P. Given a valuation $p \in [P]$ and a predicate ρ on P, by $\rho@p$ we denote the truth value of ρ evaluated at p. Consider two sets P_1 and P_2 of ports, and two valuations $p_1 \in [P_1]$ and $p_2 \in [P_2]$. We write $p_1 \cong p_2$ if $p_1(x) = p_2(x)$ for all ports in $P_1 \cap P_2$. If $p_1 \cong p_2$, then $p_1 \uplus p_2$ denotes the valuation in $[P_1 \cup P_2]$ such that $(p_1 \uplus p_2)(x) = p_1(x)$ for all $x \in P_1$, and $(p_1 \uplus p_2)(x) = p_2(x)$ for all $x \in P_2$.

4.1 The Relational Nets

A *process* a consists of a set I_a of *input ports*, a set O_a of *output ports* disjoint from I_a, and a satisfiable predicate ρ_a on $I_a \cup O_a$ called *I/O relation*. For a process a, let $P_a = I_a \cup O_a$. A *relational net* f consists of a set A_f of processes and a set C_f of channels, such that all of the following:

1. For all processes $a, b \in A_f$, if $a \not= b$, then $P_a \cap P_b = \emptyset$.
2. For all processes $a \in A_f$ and all channels $(x, y) \in C_f$, we have $y \not\in O_a$.
3. For all channels $(x, y), (x', y') \in C_f$, if $x \not= x'$, then $y \not= y'$.
4. Let $P_f = \bigcup_{a \in A_f} P_a \cup \bigcup_{(x,y) \in C_f} \{x, y\}$. There is an I/O valuation $p \in [P_f]$ such that (a) $\rho_a@p$ is true for all processes $a \in A_f$, and (b) $p(x) = p(y)$ for all channels $(x, y) \in C_f$.

If (a) and (b), then the I/O valuation p is called *consistent* with the relational net f. A port $x \in P_f$ of a relational net f is a *primary input port* if (1) there is no process $a \in A_f$ with $x \in O_a$, and (2) there is no port $y \in P_f$ with $(y, x) \in C_f$. We write I_f for the primary input ports of the relational net f, and $O_f = P_f \backslash I_f$ for the other ports.

`Composition` $f \| g$ is defined iff $P_f \cap P_g = \emptyset$. Then, $A_{f\|g} = A_f \cup A_g$ and $C_{f\|g} = C_f \cup C_g$.

`Connection` $f\theta$ is defined iff the result $(A_{f\theta}, C_{f\theta})$, as defined next, is a relational net. Let $A_{f\theta} = A_f$ and $C_{f\theta} = C_f \cup \theta$.

`Hierarchy` $f' \preceq f$ iff (1) $P_{f'} \supseteq P_f$, (2) $O_{f'} \supseteq O_f$, and (3) for every I/O valuation $p \in [P_{f'}]$, if p is consistent with f', then p is consistent with f. Note the covariance between inputs and outputs: a more abstract relational net f may have fewer input and output ports than f', and it may leave some output ports of f' unconstrained by treating them as free inputs.

Proposition 4. *The relational nets are a component algebra.*

The relational nets are very general, in that they admit processes with (1) partial I/O relations, which do not accept all input valuations, and (2) arbitrary dependencies between input and output valuations. An example of relational processes are the input-constraining Mealy machines whose combinational I/O dependencies do not change in time. Instead of a formal account of this statement, we only give an intuitive explanation. First, a state machine with inputs and outputs, such as a Moore or Mealy machine, is *input-enabling* if in every

state, the machine accepts all possible inputs. We refer to a class of state machines as *input-constraining* if its members are not necessarily input-enabling. Second, stateless interfaces consider only the *combinational* I/O dependencies of a state machine, which assert how an output value depends on the *concurrent* input values. In particular, a Moore machine has *no* combinational I/O dependencies, because an output value may depend only on *previous* input values. Thus, to view a state machine M as a relational process a, we construct the I/O relation ρ_a from the combinational I/O dependencies of M, and abstract away all other detail, such as sequential I/O dependencies.

We have no interface theory for the relational nets. Instead, we consider three restricted classes of relational nets for which simple interface theories exist:

1. In the first restricted class, the *rectangular nets*, the processes can still restrict the accepted input valuations, but there are no I/O dependencies. An example of such processes are the input-constraining Moore machines whose input assumptions do not change in time. An appropriate interface theory are the stateless A/G interfaces. (To capture input assumptions that that may change in time, *stateful* A/G interfaces are needed; see Section 5.)
2. In the second restricted class, the *total nets*, the processes do not restrict the accepted input valuations, but there can be I/O dependencies. An example of such processes are the input-enabling Mealy machines whose combinational I/O dependencies do not change in time. An appropriate interface theory are the stateless PD interfaces, which rule out cycles of combinational I/O dependencies—a common restriction in hardware design. (To capture I/O dependencies that that may change in time, *stateful* PD interfaces are needed; cf. [6].)
3. In the third restricted class, the *total-and-rectangular nets*, the processes do not restrict the accepted input valuations, and there are no I/O dependencies. An example of such processes are the input-enabling Moore machines. An appropriate interface theory are the stateless I/O interfaces.

4.2 The Rectangular Nets

A process a is *rectangular* if there exist a predicate ϕ_a on I_a and a predicate ψ_a on O_a such that ρ_a is equivalent to $\phi_a \wedge \psi_a$. In other words, the I/O relation of a rectangular process may constrain input and output values, but it cannot relate them. It follows that there are no dependencies between input and output values. A relational net f is a *rectangular net* if all atoms in A_f are rectangular. The rectangular nets are closed under composition and connection.

For a relational net f, let $\rho_f^A = \bigwedge_{a \in A_f} \rho_a$ and $\rho_f^C = \bigwedge_{(x,y) \in C_f} (x = y)$ and $\rho_f = \rho_f^A \wedge \rho_f^C$. The predicate ρ_f is true precisely at the I/O valuations in $[P_f]$ that are consistent with f. Given a relational net f and a stateless I/O interface F, define $f \lhd_{I/O} F$ iff $I_f \subseteq I_F$ and $O_f \subseteq O_F^+$ and $O_f \supseteq O_F$. Given a relational net f and a stateless A/G interface F, define $f \lhd_{A/G} F$ iff (1) $f \lhd_{I/O} \Pi_F$, (2) $(\exists O_f)\rho_f$ is implied by the input assumption ϕ_F, and (3) $(\exists I_f)\rho_f$ implies the output guarantee ψ_F.

Proposition 5. *The stateless A/G interfaces with $\lhd_{A/G}$ are an interface theory for the rectangular nets, but not for the relational nets.*

To see the second part of the proposition, consider the inverter process $a_{\neq}$ with boolean input port x and boolean output port y and I/O relation $x \not= y$. This process is not rectangular. The illegal connection $a_{\neq}\theta$ for $\theta = \{(y, x)\}$ is permitted by the stateless A/G interfaces with $\lhd_{A/G}$.

4.3 The Total Nets

A process a is *total* if $(\forall I_a)(\exists O_a)\rho_a$. In other words, a total process accepts all possible input values. A relational net f is a *total net* if all processes in A_f are total. The total nets are closed under composition and connection.

For a relational net f, the *transitive-dependency relation* κ_f^* is a binary relation on the ports P_f, namely, the smallest transitive relation such that (1) for all processes $a \in A_f$, if $x \in I_f$ and $y \in O_f$, then $(x, y) \in \kappa_f^*$, and (2) $C_f \subseteq \kappa_f^*$. Given a relational net f and a stateless PD interface F, define $f \lhd_{PD} F$ iff $f \lhd_{I/O} \Pi_F$ and $\kappa_f^* \cap (I_F \times O_F) \subseteq \kappa_F$.

Proposition 6. *The stateless PD interfaces with $\lhd_{PD}$ are an interface theory for the total nets, but not for the relational nets.*

To see the second part of the proposition, consider the relational net f with two processes a and b, and no channels. Suppose that a has a boolean output port x and the I/O relation $x = 0$, and b has a boolean input port y and the I/O relation $y = 1$. The process b is not total. The illegal connection $f\theta$ for $\theta = \{(x, y)\}$ is permitted by the stateless PD interfaces with $\lhd_{PD}$.

4.4 The Total-and-Rectangular Nets

A relational net f is a *total-and-rectangular net* if f is both a total net and a rectangular net. Note that a process a is both total and rectangular iff the I/O relation ρ_a contains no input ports from I_a. In other words, the I/O relation of a total rectangular process constrains only the output values. The total-and-rectangular nets are closed under composition and connection.

Proposition 7.

1. *The stateless I/O interfaces with $\lhd_{I/O}$ are an interface theory for the total-and-rectangular nets, equally expressive as the stateless A/G interfaces with $\lhd_{A/G}$, and more expressive than the stateless PD interfaces with $\lhd_{PD}$.*
2. *The stateless I/O interfaces with $\lhd_{I/O}$ are not an interface theory for the total nets, nor for the rectangular nets.*

To see that the stateless PD interfaces with $\lhd_{PD}$ are not as expressive for the total-and-rectangular nets as the stateless I/O interfaces with $\lhd_{I/O}$, consider the constant process a_0 with boolean input port x and boolean output port y and I/O relation $y = 0$. This process is both total and rectangular. The connection $a_0\theta$ with $\theta = \{(y, x)\}$ is permitted by the stateless I/O interfaces

with $\lhd_{I/O}$, but is not permitted by the stateless PD interfaces with $\lhd_{PD}$. To see the second part of the proposition, first recall the inverter process $a_{\neq}$ from the proof of Proposition 5. The process $a_{\neq}$ is total. The illegal connection $a_{\neq}\theta$ for $\theta = \{(y, x)\}$ is permitted by the stateless I/O interfaces with $\lhd_{I/O}$. Second, recall the two-process net f from the proof of Proposition 6. Both processes $a, b \in A_f$ are rectangular. The illegal connection $f\theta$ for $\theta = \{(x, y)\}$ is permitted by the stateless I/O interfaces with $\lhd_{I/O}$.

5 Stateful Interfaces Are Games

We present two interface algebras with state. A stateful interface F proceeds in steps through a state space, and with every step, the valuation on the set P_F of ports may change. The stateful interfaces we consider are *deterministic*, in that for every state the port valuation, which is observable, uniquely determines the successor state. Deterministic interfaces may of course be implemented by nondeterministic components. Nondeterminism in interfaces, however, seems unnecessary and is expensive: if an interface is not at all times aware of the state of the other interfaces within a component-based system, then compatibility checking for interfaces becomes difficult [5] (an exponential subset construction is needed to track the possible states of interfaces).

Our first example of stateful interfaces are the deterministic A/G interfaces. They have input assumptions and output guarantees that depend on the state of the interface. A stateful interface is naturally viewed as a two-player concurrent game [2]. The two players are *component*, representing a component that implements the interface, and *environment*, representing the environment. In the case of a stateful A/G interface, with every step, the component chooses an output valuation that satisfies the current output guarantee, and *independently* the environment chooses an input valuation that satisfies the current input assumption. The resulting I/O valuation determines the successor state, and the game repeats. The objective of the environment is to be helpful by always providing inputs that are accepted by the component.

There are two ways for the environment to win the game: either the game continues ad infinitum, or it enters a state in which the component cannot produce an output, because the output guarantee is unsatisfiable. Such a state is called a *termination state*, because it represents the fact that the component terminates. Conversely, the environment loses the game if the game enters a state in which the environment cannot provide an acceptable input, because the input assumption is unsatisfiable, whereas the component is not ready to terminate. Such a state is called an *immediate error state*. A helpful environment tries to prevent an immediate error state from being entered. The states from which the environment cannot prevent this are called *derived error states*. In other words, the derived error states are those states from which the environment has no strategy to avoid an immediate error state. As the objective is to avoid a set of states, this is a *safety* game [2]. The interface that gives rise to the game is well-formed iff the initial state is not an (immediate or derived) error state: if

this is the case, then there is a helpful environment that can provide acceptable inputs either ad infinitum, or until the component terminates.

Given an interface F, a connection $F\theta$ is defined iff the resulting interface is well-formed. In particular, checking if $F\theta$ is defined requires computing the derived error states, i.e., solving a concurrent safety game. This can be done by backward-search from the immediate error states in time linear in the number of states. The game-theoretic view also motivates the definition of hierarchy: if the interface F' refines the interface F, then the well-formedness of $F\theta$ must imply the well-formedness of $F'\theta$; that is, if the environment has a strategy to win the $F\theta$ game, then it must also have a strategy to win the $F'\theta$ game. This preservation of strategies is captured by *alternating* refinement relations [3], such as alternating simulation (an alternative choice is refinement as alternating trace containment, but this would be more expensive to check).

The stateful A/G interfaces can be viewed as *synchronous interface automata*, thus complementing the asynchronous variety defined in [5]. In a similar fashion, state can be added also to the stateless I/O interfaces and to the stateless PD interfaces. Instead, we present as second example a more generic stateful interface algebra, the deterministic game interfaces, which make the game-theoretic view explicit. In a deterministic game interface, at every state, certain moves are available to each player. These moves can be very general, such as functions from input valuations to output valuations for the component, and functions from output valuations to input valuations for the environment. With every step, each player independently chooses an available move, and the combination of both moves determines the current port valuation, which in turn determines the successor state.

5.1 The Deterministic Assume/Guarantee Interfaces

A *deterministic A/G interface* F consists of the following:

- A stateless I/O interface Π_F.[2]
- A finite set Q_F of *states*, including an *initial state* $\hat{q}_F \in Q_F$.
- For every state $q \in Q_F$, a predicate $\phi_F(q)$ on I_F called *input assumption*, and a predicate $\psi_F(q)$ on O_F called *output guarantee*. A state $q \in Q_F$ is a *termination state* if the output guarantee $\psi_F(q)$ is unsatisfiable. A state $q \in Q_F$ is an *error state* if the input assumption $\phi_F(q)$ is unsatisfiable and the output guarantee $\psi_F(q)$ is satisfiable. We require that the initial state $\hat{q}_F$ is not an error state.
- For every pair $q, r \in Q_F$ of states, a predicate $\rho_F(q, r)$ on P_F called *transition guard*. We require that for every state $q \in Q_F$, (1) the disjunction $\bigvee_{r \in Q_F} \rho_F(q, r)$ is valid, and (2) for all states $r, r' \in Q_F$, if $r \not= r'$, then the

[2] In a more general interface algebra, the input and output ports may depend on the state of the interface. This is necessary for modeling bidirectional ports (cf. [7]).

conjunction $\rho_F(q,r) \wedge \rho_F(q,r')$ is unsatisfiable. Condition (2) ensures determinism. In other words, for every state $q \in Q_F$ and every I/O valuation $p \in [P_F]$, there is a unique state $r \in Q_F$ with $\rho_F(q,r)@p$. If $\rho_F(q,r)@p$, then r is called the *p-successor* of q and denoted $\delta_F(q,p)$.

For the deterministic A/G interfaces, composition, connection, and hierarchy are defined as follows:

`Composition` $F \| G$ of two deterministic A/G interfaces F and G is defined iff $\Pi_F \| \Pi_G$ is defined. Then, $Q_{F\|G} = Q_F \times Q_G$ and $\hat{q}_{F\|G} = (\hat{q}_F, \hat{q}_G)$ and for all states $q_F, r_F \in Q_F$ and $q_G, r_G \in Q_G$, let $\phi_{F\|G}(q_F, q_G) = \phi_F(q_F) \wedge \phi_G(q_G)$ and $\psi_{F\|G}(q_F, q_G) = \psi_F(q_F) \wedge \psi_G(q_G)$ and $\rho_{F\|G}((q_F, q_G), (r_F, r_G)) = \rho_F(q_F, r_F) \wedge \rho_G(q_G, r_G)$.

`Connection` $F\theta$ for a deterministic A/G interface F is defined iff (1) $\Pi_F\theta$ is defined and (2) the initial state of $F\theta$, as defined next, is not an error state. Let $Q_{F\theta} = Q_F$ and $\hat{q}_{F\theta} = \hat{q}_F$ and for all states $q, r \in Q_{F\theta}$, let $\psi_{F\theta}(q) = \psi_F(q) \wedge \rho_\theta$ and $\rho_{F\theta}(q,r) = \rho_F(q,r)$. The input assumptions $\phi_{F\theta}$ are computed by the following algorithm, which finds all derived error states:

> `[Step 1]` For all states $q \in Q_{F\theta}$, initialize $\phi_{F\theta}(q)$ to the predicate $(\forall O_{F\theta})(\psi_{F\theta}(q) \Rightarrow \phi_F(q))$.
>
> `[Step 2]` For all states $q, r \in Q_F$, if $\phi_{F\theta}(r)$ is unsatisfiable and $\psi_{F\theta}(r)$ is satisfiable, then replace $\phi_{F\theta}(q)$ by the conjunction of $\phi_{F\theta}(q)$ and $(\forall O_{F\theta})(\psi_{F\theta}(q) \Rightarrow \neg\rho_{F\theta}(q,r))$. In other words, if r is an error state, then a helpful environment chooses, in every state q, an input valuation that prevents r from being entered.
>
> Repeat `[Step 2]` until all input assumptions are replaced by equivalent predicates.

The second step must be iterated, because the strengthening of an input assumption $\phi_{F\theta}(q)$ in `[Step 2]` may cause it to change from satisfiable to unsatisfiable, possibly exposing q as an error state and thus triggering additional changes.

`Hierarchy` $F' \preceq F$ iff (1) $\Pi_{F'} \preceq \Pi_F$ and (2) there is an alternating A/G simulation $\sqsubseteq$ of F' by F with $\hat{q}_{F'} \sqsubseteq \hat{q}_F$. An *alternating A/G simulation* of F' by F is a binary relation $\sqsubseteq$ between the states $Q_{F'}$ and the states Q_F such that $q' \sqsubseteq q$ implies that (1) the input assumption $\phi_{F'}(q')$ is implied by $\phi_F(q)$, (2) the output assumption $\psi_{F'}(q')$ implies $\psi_F(q)$, and (3) for all input valuations $i \in [I_F]$ and all output valuations $o \in [O_{F'}]$, if $\phi_F(q)@i$ and $\psi_{F'}(q')@o$, then $\delta_{F'}(q', i \uplus o) \sqsubseteq \delta_F(q, i \uplus o)$.

Proposition 8. *The deterministic A/G interfaces are an interface algebra.*

5.2 The Deterministic Game Interfaces

A *deterministic game interface* F consists of the following:

- A set P_F of *ports*.
- A finite set Q_F of *states*, including an *initial state* $\hat{q}_F \in Q_F$.
- For every state $q \in Q_F$, a finite set $\mathcal{I}_F(q)$ of *input moves*, a finite set $\mathcal{O}_F(q)$ of *output moves*, a function $\gamma_F(q)$ from $\mathcal{I}_F(q) \times \mathcal{O}_F(q)$ to $[P_F]$ called *outcome function*, and a function $\delta_F(q)$ from $[P_F]$ to Q_F called *successor function*. Determinism means that the successor state depends only on the port valuation, and not on the choice of moves. A state $q \in Q_F$ is a *termination state* if the set $\mathcal{O}_F(q)$ of output moves is empty. A state $q \in Q_F$ is an *error state* if the set $\mathcal{I}_F(q)$ of input moves is empty and the set $\mathcal{O}_F(q)$ of output moves is nonempty. We require that the initial state $\hat{q}_F$ is not an error state.

For the deterministic game interfaces, composition, connection, and hierarchy are defined as follows:

`Composition` $F \| G$ of two deterministic game interfaces F and G is defined iff $P_F \cap P_G = \emptyset$. Then, $P_{F\|G} = P_F \cup P_G$ and $Q_{F\|G} = Q_F \times Q_G$ and $\hat{q}_{F\|G} = (\hat{q}_F, \hat{q}_G)$ and for all states $q_F, r_F \in Q_F$ and $q_G, r_G \in Q_G$, let $\mathcal{I}_{F\|G}(q_F, q_G) = \mathcal{I}_F(q_F) \times \mathcal{I}_G(q_G)$ and $\mathcal{O}_{F\|G}(q_F, q_G) = \mathcal{O}_F(q_F) \times \mathcal{O}_G(q_G)$ and for all moves $m_F \in \mathcal{I}_F$, $m_G \in \mathcal{I}_G$, $n_F \in \mathcal{O}_F$, and $n_G \in \mathcal{O}_G$, let $\gamma_{F\|G}(q_F, q_G)((m_F, m_G), (n_F, n_G)) = \gamma_F(q_F)(m_F, n_F) \uplus \gamma_G(q_G)(m_G, n_G)$, and for all valuations $p_F \in [P_F]$ and $p_G \in [P_G]$, let $\delta_{F\|G}(q_F, q_G)(p_F \uplus p_G) = (\delta_F(q_F)(p_F), \delta_G(q_G)(p_G))$.

`Connection` $F\theta$ for a deterministic game interface F is defined iff the initial state of $F\theta$, as defined next, is not an error state. Let $Q_{F\theta} = Q_F$ and $\hat{q}_{F\theta} = \hat{q}_F$ and for all states $q \in Q_{F\theta}$, let $\mathcal{O}_{F\theta}(q) = \mathcal{O}_F(q)$ and for all valuations $p \in [P_{F\theta}]$, let $\delta_{F\theta}(q)(p) = \delta_F(q)(p)$. The outcome functions $\gamma_{F\theta}$ are the uniquely determined functions such that for all states $q \in Q_{F\theta}$, all input moves $m \in \mathcal{I}_F(q)$, and all output moves $n \in \mathcal{O}_{F\theta}(q)$, we have (1) $\gamma_{F\theta}(q)(m, n) \cong \gamma_F(q)(m, n)$ and (2) for all channels $(x, y) \in \theta$, we have $\gamma_{F\theta}(q)(m, n)(x) = \gamma_F(q)(m, n)(y)$. The input moves $\mathcal{I}_{F\theta}$ are computed by the following algorithm, which finds all derived error states:

> `[Step 1]` For all states $q \in Q_{F\theta}$, initialize $\mathcal{I}_{F\theta}(q)$ to the set of input moves $m \in \mathcal{I}_F(q)$ such that for all output moves $n \in \mathcal{O}_{F\theta}(q)$ and all channels $(x, y) \in \theta$, we have $\gamma_{F\theta}(q)(m, n)(x) = \gamma_{F\theta}(q)(m, n)(y)$.
>
> `[Step 2]` For all states $q, r \in Q_F$, if $\mathcal{I}_{F\theta}(r)$ is empty and $\mathcal{O}_{F\theta}(r)$ is nonempty and there is an output move $n \in \mathcal{O}_{F\theta}(q)$ and a valuation $p \in [P_{F\theta}]$ such that $\gamma_{F\theta}(q)(m, n) = p$ and $\delta_{F\theta}(q)(p) = r$, then remove m from the set $\mathcal{I}_{F\theta}(q)$ of input moves. In other words, if r is an error state, then a helpful environment chooses, in every state q, an input move that prevents r from being entered.
>
> Repeat `[Step 2]` until no input moves can be removed.

The second step must be iterated, because the removal of input moves in `[Step 2]` may cause a set $\mathcal{I}_{F\theta}(q)$ of input moves to change from nonempty to empty, possibly exposing q as an error state and thus triggering additional changes.

`Hierarchy` $F' \preceq F$ iff there is an alternating game simulation $\sqsubseteq$ of F' by F with $\hat{q}_{F'} \sqsubseteq \hat{q}_F$. An *alternating game simulation* of F' by F is a binary relation $\sqsubseteq$ between the states $Q_{F'}$ and the states Q_F such that $q' \sqsubseteq q$ implies that for all input moves $m \in \mathcal{I}_F(q)$, there is an input move $m' \in \mathcal{I}_{F'}(q')$ such that for all output moves $n' \in \mathcal{O}_{F'}(q')$, there is an output move $n \in \mathcal{O}_F(q)$ such that for all valuations $p' \in [P_{F'}]$ and $p \in [P_F]$, if $p' = \gamma_{F'}(q')(m', n')$ and $p = \gamma_F(q)(m, n)$, then $p' \cong p$ and $\delta_{F'}(q')(p') \sqsubseteq \delta_F(q)(p)$.

Proposition 9. *The deterministic game interfaces are an interface algebra.*

Acknowledgment. We thank Edward Lee for inspiring much of this research with his ideas on type theories for component interaction [11], and his project for developing an abstract semantics for PTOLEMY II [12].

References

[1] R. Alur and T.A. Henzinger. Reactive modules. *Formal Methods in System Design*, 15:7–48, 1999.

[2] R. Alur, T.A. Henzinger, and O. Kupferman. Alternating-time temporal logic. In *38th Symp. Foundations of Computer Science*, pp. 100–109. IEEE Computer Society Press, 1997.

[3] R. Alur, T.A. Henzinger, O. Kupferman, and M.Y. Vardi. Alternating refinement relations. In *Concurrency Theory*, LNCS 1466, pp. 163–178. Springer-Verlag, 1998.

[4] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. MIT Press, 1999.

[5] L. de Alfaro and T.A. Henzinger. Interface automata. In *9th Symp. Foundations of Software Engineering*. ACM Press, 2001.

[6] L. de Alfaro, T.A. Henzinger, and F.Y.C. Mang. The control of synchronous systems. In *Concurrency Theory*, LNCS 1877, pp. 458–473. Springer-Verlag, 2000.

[7] L. de Alfaro, T.A. Henzinger, and F.Y.C. Mang. The control of synchronous systems, part II. In *Concurrency Theory*, LNCS, Springer-Verlag, 2001.

[8] D.L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-independent Circuits*. MIT Press, 1989.

[9] T.A. Henzinger, S. Qadeer, and S.K. Rajamani. Decomposing refinement proofs using assume-guarantee reasoning. In *Int. Conf. Computer-aided Design*, pp. 245–252. IEEE Computer Society Press, 2000.

[10] L. Lamport. The temporal logic of actions. *ACM Trans. Programming Languages and Systems*, 16:872–923, 1994.

[11] E.A. Lee. What's ahead for embedded software? *IEEE Computer Magazine*, 33:18–26, 2000.

[12] E.A. Lee. *Overview of the Ptolemy Project.* Technical Memorandum UCB/ERL-M01/11, University of California, Berkeley, 2001.

[13] N.A. Lynch. *Distributed Algorithms*. Morgan-Kaufmann, 1996.

[14] J. Rumbaugh, G. Booch, and I. Jacobson. *The UML Reference Guide*. Addison-Wesley, 1999.

[15] N. Shankar. Lazy compositional verification. In *Compositionality: The Significant Difference*, LNCS 1536, pp. 541–564. Springer-Verlag, 1999.

Giotto: A Time-Triggered Language for Embedded Programming*

Thomas A. Henzinger, Benjamin Horowitz, and Christoph Meyer Kirsch

University of California, Berkeley
{tah,bhorowit,cm}@eecs.berkeley.edu

Abstract. Giotto provides an abstract programmer's model for the implementation of embedded control systems with hard real-time constraints. A typical control application consists of periodic software tasks together with a mode switching logic for enabling and disabling tasks. Giotto specifies time-triggered sensor readings, task invocations, and mode switches independent of any implementation platform. Giotto can be annotated with platform constraints such as task-to-host mappings, and task and communication schedules. The annotations are directives for the Giotto compiler, but they do not alter the functionality and timing of a Giotto program. By separating the platform-independent from the platform-dependent concerns, Giotto enables a great deal of flexibility in choosing control platforms as well as a great deal of automation in the validation and synthesis of control software. The time-triggered nature of Giotto achieves timing predictability, which makes Giotto particularly suitable for safety-critical applications.

1 Introduction

Giotto provides a programming abstraction for hard real-time applications which exhibit time-periodic and multi-modal behavior, as in automotive, aerospace, and manufacturing control. Traditional control design happens at a mathematical level of abstraction, with the control engineer manipulating differential equations and mode switching logic using tools such as Matlab or MatrixX. Typical activities of the control engineer include modeling of the plant behavior and disturbances, deriving and optimizing control laws, and validating functionality and performance of the model through analysis and simulation. If the validated design is to be implemented in software, it is then handed off to a software engineer who writes code for a particular platform (we use the word "platform" to stand for a hardware configuration together with a real-time operating system). Typical activities of the software engineer include decomposing the necessary computational activities into periodic tasks, assigning tasks to CPUs and setting task priorities to meet the desired hard real-time constraints under the given scheduling mechanism and hardware performance, and achieving a degree of fault tolerance through replication and error correction.

* This research was supported in part by the DARPA SEC grant F33615-C-98-3614 and by the MARCO GSRC grant 98-DT-660.

T.A. Henzinger and C.M. Kirsch (Eds.): EMSOFT 2001, LNCS 2211, pp. 166–184, 2001.
© Springer-Verlag Berlin Heidelberg 2001

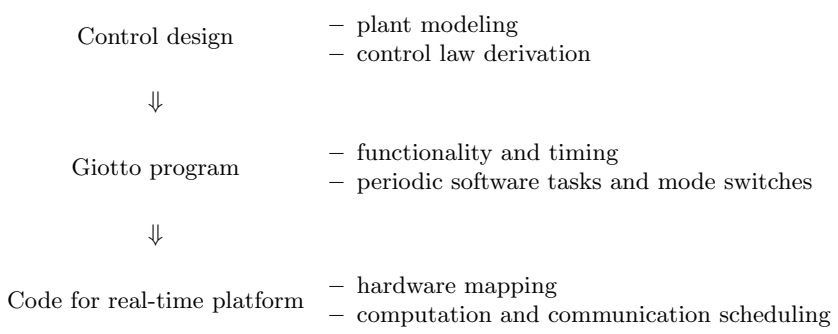


Fig. 1. Real-time control system design with Giotto

Giotto provides an intermediate level of abstraction, which permits the software engineer to communicate more effectively with the control engineer. Specifically, Giotto defines a software architecture of the implementation which specifies its functionality and timing. Functionality and timing are sufficient and necessary for ensuring that the implementation is consistent with the mathematical model of the design. On the other hand, Giotto abstracts away from the realization of the software architecture on a specific platform, and frees the software engineer from worrying about issues such as hardware performance and scheduling mechanism while communicating with the control engineer. After writing a Giotto program, the second task of the software engineer remains of course to implement the program on the given platform. However, in Giotto, this second task, which requires no interaction with the control engineer, is effectively decoupled from the first, and can in large parts be automated by increasingly powerful compilers. The Giotto design flow is shown in Figure 1. The separation of logical correctness concerns (functionality and timing) from physical realization concerns (mapping and scheduling) has the added benefit that a Giotto program is entirely platform independent and can be compiled on different, even heterogeneous, platforms.

Motivating example. Giotto is designed specifically for embedded control applications. Consider a typical fly-by-wire flight control system [LRR92,Col99], which consists of three types of interconnected components (see Figure 2): sensors, CPUs for computing control laws, and actuators. The sensors include an inertial navigation unit (INU), for measuring linear and angular acceleration; a global positioning system (GPS), for measuring position; an air data measurement system, for measuring such quantities as air pressure; and the pilot's controls, such as the pilot's stick. Each sensor has its own timing properties: the INU, for example, outputs its measurement 1,000 times per second, whereas the pilot's stick outputs its measurement only 500 times per second. Three separate control laws —for pitch, lateral, and throttle control— need to be computed.

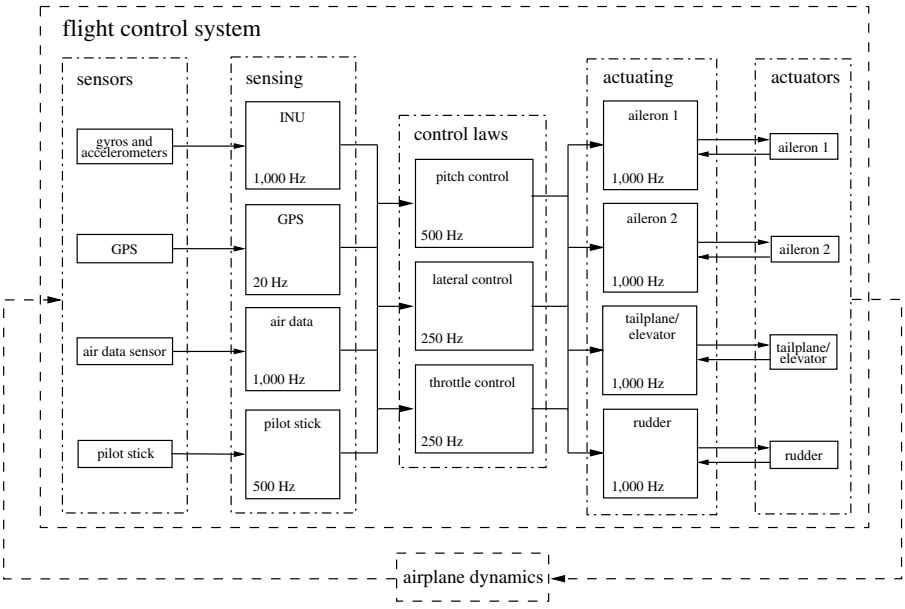


Fig. 2. A fly-by-wire flight control system

The system has four actuators: two for the ailerons, one for the tailplane, and one for the rudder. The timing requirements on the control laws and actuator tasks are also shown in Figure 2. The reader may wonder why the actuator tasks need to run more frequently than the control laws. The reason is that the actuator tasks are responsible for the stabilization of quickly moving mechanical hardware, and thus need to be an order of magnitude more responsive than the control laws.

We have just described one operational mode of the fly-by-wire flight control system, namely the cruise mode. There are four additional modes: the takeoff, landing, autopilot, and degraded modes. In each of these modes, additional sensing tasks, control laws, and actuating tasks need to be executed, as well as some of the cruise tasks removed. For example, in the takeoff mode, the landing gear must be retracted. In the autopilot mode, the control system takes inputs from a supervisory flight planner, instead of from the pilot's stick. In the degraded mode, some of the sensors or actuators have suffered damage; the control system compensates by not allowing maneuvers which are as aggressive as those permitted in the cruise mode.

The Giotto abstraction. Giotto provides a programmer's abstraction for specifying control systems that are structured like the previous fly-by-wire example. The basic functional unit in Giotto is the *task*, which is a periodically executed piece of, say, C code. Several concurrent tasks make up a *mode*. Tasks can be added or removed by switching from one mode to another. Tasks communicate

with each other, as well as with sensors and actuators, by so-called *drivers*, which is code that transports and converts values between *ports*. While a task represents scheduled computation on the application level and consumes logical time, a driver is synchronous, bounded code, which is executed logically instantaneously on the system level (since drivers cannot depend on each other, no issues of fixed-point semantics arise). The periodic invocation of tasks, the reading of sensor values, the writing of actuator values, and the mode switching are all triggered by real time. For example, one task t_1 may be invoked every 2 ms and read a sensor value upon each invocation, another task t_2 may be invoked every 3 ms and write an actuator value upon each completion, and a mode switch may be contemplated every 6 ms. This time-triggered semantics enables efficient reasoning about the timing behavior of a Giotto program, in particular, whether it conforms to the timing requirements of the mathematical (e.g., Matlab) model of the control design.

A Giotto program does not specify where, how, and when tasks are scheduled. The Giotto program with tasks t_1 and t_2 can be compiled on platforms that have a single CPU (by time sharing the two tasks) as well as on platforms with two CPUs (by parallelism); it can be compiled on platforms with preemptive priority scheduling (such as most RTOSs) as well as on truly time-triggered platforms (such as TTA [Kop97]). All the Giotto compiler needs to ensure is that the logical semantics of Giotto —functionality and timing— is preserved. A Giotto program can be annotated with *platform constraints*, which may be understood as directives to the compiler in order to make its job easier. A constraint may map a particular task to a particular CPU, it may schedule a particular task in a particular time interval, or it may schedule a particular communication event between tasks in a particular time slot. These annotations, however, in no way modify the functionality and timing of a Giotto program; they simply aid the compiler in realizing the logical semantics of the program.

Outline of the paper. We first give an informal introduction to Giotto in Section 2, followed by a formal definition of the language in Section 3. In Section 4, we briefly describe annotated Giotto, a refinement of Giotto for guiding distributed code generation. In Section 5, we relate Giotto to the literature and mention ongoing application work.

2 Informal Description of Giotto

Ports. In Giotto all data is communicated through ports. A port represents a typed variable with a unique location in a globally shared name space. We use the global name space for ports as a virtual concept to simplify the definition of Giotto. An implementation of Giotto is not required to be a shared memory system. Every port is persistent in the sense that the port keeps its value over time, until it is updated. There are mutually disjoint sets of sensor ports, actuator ports, and task ports in a Giotto program. The sensor ports are updated by the environment; all other ports are updated by the Giotto program. The task ports are used to communicate data between concurrent tasks and from one mode to

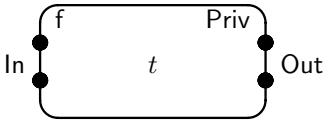


Fig. 3. A task t

the next. In any given mode, a task port may or may not be used; the used ports are called mode ports. Every mode port is explicitly assigned a value every time the mode is entered.

Tasks. A typical Giotto task t is shown in Figure 3. The task t has a set In of two input ports and a set Out of two output ports, all of which are depicted by bullets. The input ports of t are distinct from all other ports in the Giotto program. The output ports of t may be shared with other tasks as long as they are not invoked in the same mode. In general, a task may have an arbitrary number of input and output ports. A task may also maintain a state, which can be viewed as a set of private ports whose values are inaccessible outside the task. The state of t is denoted by Priv. Finally, the task has a function f from its input ports and its current state to its output ports and its next state. The task function f is implemented by a sequential program, and can be written in an arbitrary programming language. It is important to note that the execution of f has no internal synchronization points and cannot be terminated prematurely; in Giotto all synchronization is specified explicitly outside of tasks. For a given platform, the Giotto compiler will need to know the worst-case execution time of f on each CPU.

Tasks invocations. Giotto tasks are periodic tasks: they are invoked at regularly spaced points in time. An invocation of a task t is shown in Figure 4. If the task t is invoked in the mode m, then the output ports of t are included in the mode ports of m, along with the output ports of some other tasks. The task invocation has a frequency ω_{task} given by a non-zero natural number; the real-time frequency will be determined later by dividing the real-time period of the current mode by ω_{task}. The task invocation specifies a driver d which provides values for the input ports In. The first input port is loaded with the value of some other port p, and the second input port is loaded with the constant value κ. In general, a driver is a function that converts the values of sensor ports and mode ports of the current mode to values for the input ports, or loads the input ports with constants. Drivers can be guarded: the guard of a driver is a predicate on sensor and mode ports. The invoked task is executed only if the driver guard evaluates to *true*; otherwise, the task execution is skipped.

The time line for an invocation of the task t is shown in Figure 5. The invocation starts at some time τ_{start} with a communication phase in which the driver guard is evaluated and the input port values are loaded. The Giotto semantics prescribes that the communication phase —i.e., the execution of the

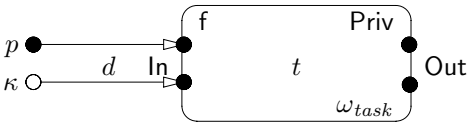


Fig. 4. An invocation of task t

driver d— takes zero time. The synchronous communication phase is followed by a scheduled computation phase. The Giotto semantics prescribes that at time τ_{stop} the state and output ports of t are updated to the (deterministic) result of f applied to the state and input ports of t at time τ_{start}. The length of the interval between τ_{start} and τ_{stop} is determined by the frequency ω_{task}. The Giotto logical abstraction does not specify when, where, and how the computation of f is physically performed between τ_{start} and τ_{stop}. However, the time at which the task output ports are updated is determined, and therefore, for any given real-time trace of sensor values, all values that are communicated between tasks are determined. Instantaneous communication and time-deterministic as well as value-deterministic computation are the three essential ingredients of the Giotto logical abstraction. A compiler must be faithful to this abstraction; for example, task inputs may be loaded after time τ_{start}, and the execution of f may be preempted by other tasks, as long as at time τ_{stop} the values of the task output ports are those specified by the Giotto semantics.

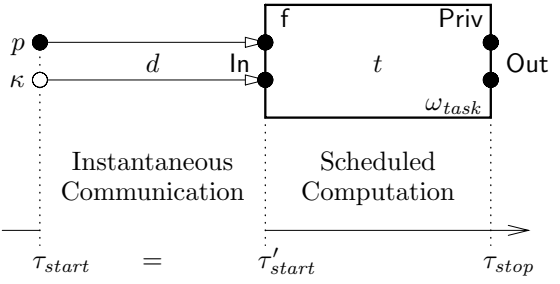


Fig. 5. The time line for an invocation of task t

Modes. A Giotto program consists of a set of modes, each of which repeats the invocation of a fixed set of tasks. The Giotto program is in one mode at a time. A mode may contain mode switches, which specify transitions from the mode to other modes. A mode switch can remove some tasks, and add others. Formally, a mode consists of a period, a set of mode ports, a set of task invocations, a set of actuator updates, and a set of mode switches. Figure 6 shows a mode m that contains invocations of two tasks, t_1 and t_2. The period π of m is 10 ms; that

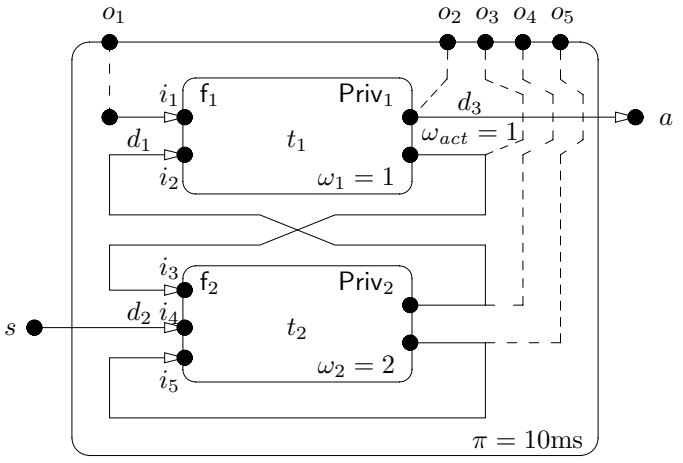


Fig. 6. A mode m

is, while the program is in mode m, its execution repeats the same pattern of task invocations every 10 ms. The task t_1 has two input ports, i_1 and i_2, two output ports, o_2 and o_3, a state Priv_1, and a function f_1. The task t_2 is defined in a similar way. Moreover, there is one sensor port, s, one actuator port, a, and a mode port, o_1, which is not updated by any task in mode m. The value of o_1 stays constant while the program is in mode m; it can be used to transfer a value from a previous mode to mode m. The invocation of t_1 in mode m has the frequency $\omega_1 = 1$, which means that t_1 is invoked once every 10 ms while the program is in mode m. The invocation of t_1 in mode m has the driver d_1, which copies the value of the mode port o_1 into i_1 and the value of the output port o_4 of t_2 into i_2. The invocation of t_2 has the frequency $\omega_2 = 2$, which means that t_2 is invoked once every 5 ms, as long as the program is in mode m. The invocation of t_2 has the driver d_2, which connects the output port o_3 of t_1 to i_3, the sensor port s to i_4, and o_5 to i_5. Note that the mode ports of m, which include all task output ports used in m, are visible outside the scope of m as indicated by the dashed lines. A mode switch may copy the values at these ports to mode ports of a successor mode. The mode m has one actuator update, which is a driver d_3 that copies the value of the task output port o_2 to the actuator port a with the actuator frequency $\omega_{act} = 1$; that is, once every 10 ms.

Figure 7 shows the exact timing of a single round of mode m, which takes 10 ms. As long as the program is in mode m, one such round follows another. The round begins at the time instant τ_0 with an instantaneous communication phase for the invocations of tasks t_1 and t_2, during which the two drivers d_1 and d_2 are executed. The Giotto semantics does not specify how the computations of the task functions f_1 and f_2 are physically scheduled; they could be scheduled in any order on a single CPU, or in parallel on two CPUs. Logically,

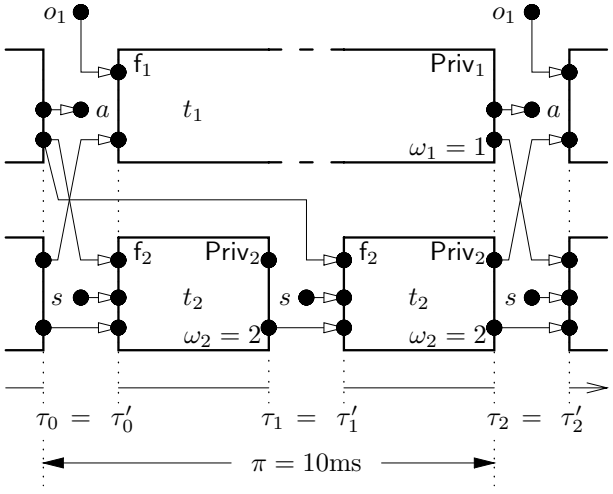


Fig. 7. The time line for a round of mode m

after 5 ms, at time instant τ_1, the results of the scheduled computation of f_2 are written to the output ports of t_2. The second invocation of t_2 begins with another execution of driver d_2, still at time τ_1, which samples the most recent value from the sensor port s. However, the two invocations of t_2 start with the same value at input port i_3, because the value stored in o_3 is not updated until time instant $\tau_2 = 10$ ms, no matter if physically f_1 finishes its computation before τ_1 or not. Logically, the output values of the invocation of t_1 must not be available before τ_2. Any physical realization that schedules the invocation of t_1 before the first invocation of t_2 must therefore keep available two sets of values for the output ports of t_1. The round is finished after writing the output values of the invocation of t_1 and of the second invocation of t_2 to their output ports at time τ_2, and after updating the actuator port a at the same time. The beginning of the next round shows that the input port i_3 is loaded with the new value produced by t_1.

Mode switches. In order to give an example of mode switching we introduce a second mode m', shown in Figure 8. The main difference between m and m' is that m' replaces the task t_2 by a new task t_3, which has a frequency ω_3 of 4 in m'. Note that t_3 has a new output port, o_6, but also uses the same output port o_4 as t_2. Moreover, t_3 has a new driver d_4, which connects the output port o_3 of t_1 to the input port i_6, the sensor port s to i_7, and o_6 to i_8. The task t_1 in mode m' has the same frequency and uses the same driver as in mode m. The period of m', which determines the length of each round, is again 10 ms. This means that in mode m', the task t_1 is invoked once per round, every 10 ms; the task t_3 is invoked 4 times per round, every 2.5 ms; and the actuator a is updated once per round, every 10 ms.

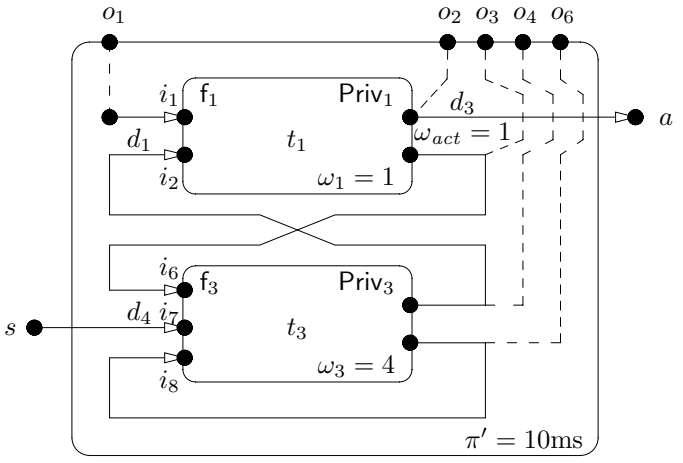


Fig. 8. A mode m'

A mode switch describes the transition from one mode to another mode. For this purpose, a mode switch specifies a switch frequency, a target mode, and a driver. Figure 9 shows a mode switch η from mode m to target mode m' with the switch frequency $\omega_{switch} = 2$ and the driver d_5. The guard of the driver is called *exit condition*, as it determines whether or not the switch occurs. The exit condition is evaluated periodically, as specified by the switch frequency. As usual, the switch frequency of 2 means that the exit condition of d_5 is evaluated every 5 ms, in the middle and at the end of each round of mode m. The exit condition is a boolean-valued condition on sensor ports and the mode ports of m. If the exit condition evaluates to true, then a switch to the target mode m' is performed. The mode switch happens by executing the driver d_5, which provides values for all mode ports of m'; specifically, d_5 loads the constant κ into o_1, the value of o_5 into o_6, and ensures that o_2, o_3, and o_4 keep their values (this is omitted from Figure 9 to avoid clutter). The explicit mention of the persistence of o_2, o_3, and o_4 is helpful, because like tasks, with a mode switch these ports may physically migrate from one CPU to another CPU, and thus may need to be copied. Like all drivers, mode switches are logically performed in zero time.

Figure 10 shows the time line for the mode switch η performed at time τ_1. The program is in mode m until τ_1 and then enters mode m'. Note that until time τ_1 the time line corresponds to the time line shown in Figure 7. At time τ_1, first the invocation of task t_2 is completed, then the mode driver d_5 is executed. This finishes the mode switch. All subsequent actions follow the semantics of the target mode m' independently of whether the program entered m' just now through a mode switch, at 5 ms into a round, or whether it started the current round already in mode m'. Specifically, the driver for the invocation of task t_3 is executed, still at time τ_1. Note that the output port o_6 of t_3 has just received the

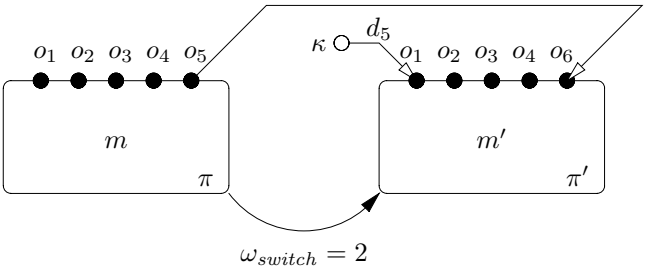


Fig. 9. A mode switch η from mode m to mode m'

value of the output port o_5 from task t_2 by the mode driver d_5. At time τ_2, task t_3 is invoked a second time, and at time τ_3, the round is finished, because this is the earliest time after the mode switch at which a complete new round of mode m' can begin. Now the input port i_1 of task t_1 is loaded with the constant κ from the mode port o_1. In this way, task t_1 can detect that a mode switch occurred.

For a mode switch to be legal, the target mode is constrained so that all task invocations that may be logically interrupted by a mode switch can be logically continued in the target mode. In our example, the mode switch η can occur at 5 ms into a round of mode m, while the task t_1 is logically running. Hence the target mode m' must also invoke t_1. Moreover, since the period of m' is 10 ms, as for mode m, the frequency of t_1 in m' must be identical to the frequency of t_1 in m, namely, 1. If, alternatively, the period of m' were 20 ms, then the frequency of t_1 in m' would have to be 2.

3 Formal Definition of Giotto

3.1 Syntax

Rather than specifying a concrete syntax for Giotto, we formally define the components of a Giotto program in a more abstract way. However, Giotto programs can also be written in a C like concrete syntax [HHK01]. A *Giotto program* consists of the following components:

1. A set of *port declarations*. A port declaration $(p, \mathsf{Type}, \mathsf{init})$ consists of a port name p, a type Type, and an initial value $\mathsf{init} \in \mathsf{Type}$. We require that all port names are uniquely declared; that is, if $(p, \cdot, \cdot)$ and $(p', \cdot, \cdot)$ are distinct port declarations, then $p \not= p'$. The set Ports of declared port names is partitioned into a set $\mathsf{SensePorts}$ of *sensor ports*, a set $\mathsf{ActPorts}$ of *actuator ports*, a set $\mathsf{InPorts}$ of *task input ports*, a set $\mathsf{OutPorts}$ of *task output ports*, and a set $\mathsf{PrivPorts}$ of *task private ports*. Given a port $p \in \mathsf{Ports}$, we use notation such as $\mathsf{Type}[p]$ for the type of p, and $\mathsf{init}[p]$ for the initial value of p. A *valuation* for a set $\mathsf{P} \subseteq \mathsf{Ports}$ of ports is a function that maps each port $p \in \mathsf{P}$ to a value in $\mathsf{Type}[p]$. We write $\mathsf{Vals}[\mathsf{P}]$ for the set of valuations for P.

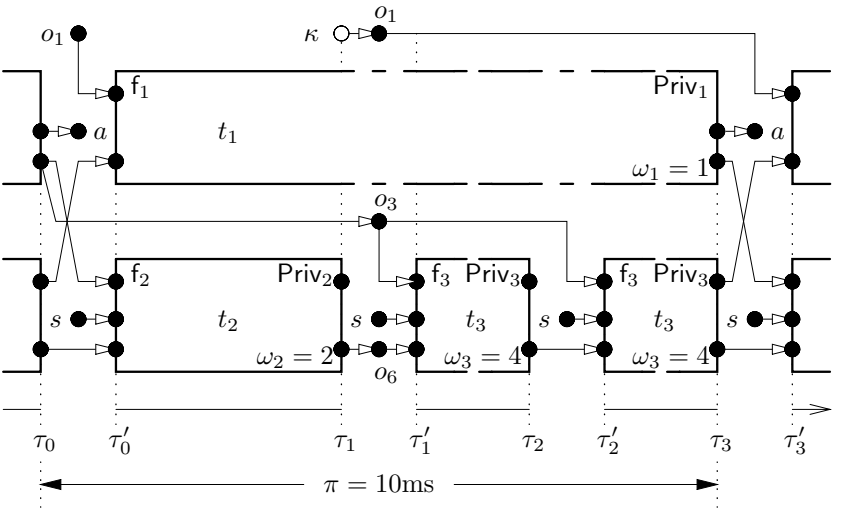


Fig. 10. The time line for the mode switch η at time τ_1

2. A set of *task declarations*. A task declaration $(t, \mathsf{In}, \mathsf{Out}, \mathsf{Priv}, \mathsf{f})$ consists of a task name t, a set $\mathsf{In} \subseteq \mathsf{InPorts}$ of *input ports*, a set $\mathsf{Out} \subseteq \mathsf{OutPorts}$ of *output ports*, a set $\mathsf{Priv} \subseteq \mathsf{PrivPorts}$ of *private ports*, and a *task function* f: $\mathsf{Vals}[\mathsf{In} \cup \mathsf{Priv}] \to \mathsf{Vals}[\mathsf{Out} \cup \mathsf{Priv}]$. If $(t, \mathsf{In}, \mathsf{Out}, \mathsf{Priv}, \cdot)$ and $(t', \mathsf{In}', \mathsf{Out}', \mathsf{Priv}', \cdot)$ are distinct task declarations, then we require that $t \neq t'$ and $\mathsf{In} \cap \mathsf{In}' = \mathsf{Priv} \cap \mathsf{Priv}' = \emptyset$. Tasks may share output ports as long as the tasks are not invoked in the same mode; see below. We write Tasks for the set of declared task names.
3. A set of *driver declarations*. A driver declaration $(d, \mathsf{Src}, \mathsf{g}, \mathsf{Dst}, \mathsf{h})$ consists of a driver name d, a set $\mathsf{Src} \subseteq \mathsf{Ports}$ of *source ports*, a *driver guard* g: $\mathsf{Vals}[\mathsf{Src}] \to \mathbb{B}$, a set $\mathsf{Dst} \subseteq \mathsf{Ports}$ of *destination ports*, and a *driver function* h: $\mathsf{Vals}[\mathsf{Src}] \to \mathsf{Vals}[\mathsf{Dst}]$. When the driver d is called, the guard g is evaluated, and if the result is *true*, then the function h is executed. We require that all driver names are uniquely declared, and we write $\mathsf{Drivers}$ for the set of declared driver names.
4. A set of *mode declarations*. A mode declaration (m, π, $\mathsf{ModePorts}$, $\mathsf{Invokes}$, $\mathsf{Updates}$, $\mathsf{Switches}$) consists of a mode name m, a *mode period* $\pi \in \mathbb{Q}$, a set $\mathsf{ModePorts} \subseteq \mathsf{OutPorts}$ of *mode ports*, a set $\mathsf{Invokes}$ of *task invocations*, a set $\mathsf{Updates}$ of *actuator updates*, and a set $\mathsf{Switches}$ of *mode switches*. We require that all mode names are uniquely declared, and we write Modes for the set of declared mode names.
   a) Each task invocation $(\omega_{task}, t, d) \in \mathsf{Invokes}[m]$ consists of a *task frequency* $\omega_{task} \in \mathbb{N}$, a task $t \in \mathsf{Tasks}$ such that $\mathsf{Out}[t] \subseteq \mathsf{ModePorts}[m]$, and a *task driver* $d \in \mathsf{Drivers}$ such that $\mathsf{Src}[d] \subseteq \mathsf{ModePorts}[m] \cup \mathsf{SensePorts}$ and $\mathsf{Dst}[d] = \mathsf{In}[t]$. The invoked task t only updates mode ports; the task

driver d reads only mode and sensor ports, and updates the input ports of t. If $(\cdot, t, \cdot)$ and $(\cdot, t', \cdot)$ are distinct task invocations in $\mathsf{Invokes}[m]$, then we require that $t \not= t'$ and $\mathsf{Out}[t] \cap \mathsf{Out}[t'] = \emptyset$; that is, tasks sharing output ports must not be invoked in the same mode.

b) Each actuator update $(\omega_{act}, d) \in \mathsf{Updates}[m]$ consists of an *actuator frequency* $\omega_{act} \in \mathbb{N}$, and an *actuator driver* $d \in \mathsf{Drivers}$ such that $\mathsf{Src}[d] \subseteq \mathsf{ModePorts}[m]$ and $\mathsf{Dst}[d] \subseteq \mathsf{ActPorts}$. The actuator driver d reads only mode ports, no sensor ports, and updates only actuator ports. If $(\cdot, d)$ and $(\cdot, d')$ are distinct actuator updates in $\mathsf{Updates}[m]$, then we require that $\mathsf{Dst}[d] \cap \mathsf{Dst}[d'] = \emptyset$; that is, in each mode, an actuator can be updated by at most one driver.

c) Each mode switch $(\omega_{switch}, m', d) \in \mathsf{Switches}[m]$ consists of a *mode switch frequency* $\omega_{switch} \in \mathbb{N}$, a *target mode* $m' \in \mathsf{Modes}$, and a *mode driver* $d \in \mathsf{Drivers}$ such that $\mathsf{Src}[d] \subseteq \mathsf{ModePorts}[m] \cup \mathsf{SensePorts}$ and $\mathsf{Dst}[d] = \mathsf{ModePorts}[m']$. The mode driver d reads only mode and sensor ports, and updates the mode ports of the target mode m'. If $(\cdot, \cdot, d)$ and $(\cdot, \cdot, d')$ are distinct mode switches in $\mathsf{Switches}[m]$, then we require that for all valuations $v \in \mathsf{Vals}[\mathsf{Ports}]$ either $\mathsf{g}[d](v) = \mathit{false}$ or $\mathsf{g}[d'](v) = \mathit{false}$. It follows that all mode switches are deterministic.

5. A *start mode* $\mathsf{start} \in \mathsf{Modes}$.

The program is *well-timed* if for all modes $m \in \mathsf{Modes}$, all task invocations $(\omega_{task}, t, \cdot) \in \mathsf{Invokes}[m]$, and all mode switches $(\omega_{switch}, m', \cdot) \in \mathsf{Switches}[m]$, if $\omega_{task}/\omega_{switch} \not\in \mathbb{N}$, then there exists a task invocation $(\omega'_{task}, t, \cdot) \in \mathsf{Invokes}[m']$ with $\pi[m]/\omega_{task} = \pi[m']/\omega'_{task}$. The well-timedness condition ensures that mode switches do not terminate tasks: if a mode switch occurs when a task may not be completed, then the same task must be present also in the target mode.

3.2 Operational Semantics

The *mode frequencies* of a mode $m \in \mathsf{Modes}$ include (i) the task frequencies ω_{task} for all task invocations $(\omega_{task}, \cdot, \cdot) \in \mathsf{Invokes}[m]$, (ii) the actuator frequencies ω_{act} for all actuator updates $(\omega_{act}, \cdot) \in \mathsf{Updates}[m]$, and (iii) the mode switch frequencies ω_{switch} for all mode switches $(\omega_{switch}, \cdot, \cdot) \in \mathsf{Switches}[m]$. The least common multiple of the mode frequencies of m is called the number of *units* of the mode m, and is denoted $\omega_{max}[m]$. A *program configuration* $C = (\tau, m, u, v, \sigma_{active})$ consists of a *time stamp* $\tau \in \mathbb{Q}$, a mode $m \in \mathsf{Modes}$, an integer $u \in \{0, \ldots, \omega_{max}[m] - 1\}$ called the *unit counter*, a valuation $v \in \mathsf{Vals}[\mathsf{Ports}]$ for all ports, and a set $\sigma_{active} \subseteq \mathsf{Tasks}$ of *active tasks*. The set $\sigma_{active} \subseteq \mathsf{Tasks}$ contains all tasks that are logically running, whether or not they are physically running by expending CPU time.

A program configuration is updated essentially as follows: first, some tasks are completed (i.e., removed from the active set); second, some actuators are updated; third, a mode switch may occur; fourth, some new tasks are activated. We therefore need the following definitions:

- A task invocation $(\omega_{task}, t, \cdot) \in \mathsf{Invokes}[m]$ is *completed* at configuration C if $t \in \sigma_{active}$ and $u \cdot \omega_{task}/\omega_{max}[m] \in \mathbb{N}$.
- An actuator update $(\omega_{act}, d) \in \mathsf{Updates}[m]$ is *enabled* at configuration C if $u \cdot \omega_{act}/\omega_{max}[m] \in \mathbb{N}$ and $\mathsf{g}[d](v) = true$.
- A mode switch $(\omega_{switch}, \cdot, d) \in \mathsf{Switches}[m]$ is *enabled* at configuration C if $u \cdot \omega_{switch}/\omega_{max}[m] \in \mathbb{N}$ and $\mathsf{g}[d](v) = true$.
- A task invocation $(\omega_{task}, \cdot, d) \in \mathsf{Invokes}[m]$ is *enabled* at configuration C if $u \cdot \omega_{task}/\omega_{max}[m] \in \mathbb{N}$ and $\mathsf{g}[d](v) = true$.

For a program configuration C and a set $\mathsf{P} \subseteq \mathsf{Ports}$, we write $C[\mathsf{P}]$ for the valuation in $\mathsf{Vals}[\mathsf{P}]$ that agrees with C on the values of all ports in P. The program configuration C_{succ} is a *successor configuration* of $C = (\tau, m, u, v, \sigma_{active})$ if C_{succ} results from C by the following nine steps. These are the steps a Giotto program performs whenever it is invoked, initially at time $\tau = 0$ with $u = 0$ and $\sigma_{active} = \emptyset$:

1. [**Task output and private ports.**] Let $\sigma_{completed}$ be the set of tasks t such that a task invocation of the form $(\cdot, t, \cdot) \in \mathsf{Invokes}[m]$ is completed at configuration C. Consider a port $p \in \mathsf{OutPorts} \cup \mathsf{PrivPorts}$. If $p \in \mathsf{Out}[t] \cup \mathsf{Priv}[t]$ for some task $t \in \sigma_{completed}$, then define $v_{task}(p) = \mathsf{f}[t](C[\mathsf{In}[t] \cup \mathsf{Priv}[t]])(p)$; otherwise, define $v_{task}(p) = v(p)$. This gives the new values of all task output and private ports. Note that ports are persistent in the sense that they keep their values unless they are modified. Let C_{task} be the configuration that agrees with v_{task} on the values of $\mathsf{OutPorts} \cup \mathsf{PrivPorts}$, and otherwise agrees with C.
2. [**Actuator ports.**] Consider a port $p \in \mathsf{ActPorts}$. If $p \in \mathsf{Dst}[d]$ for some actuator update $(\cdot, d) \in \mathsf{Updates}[m]$ that is enabled at configuration C_{task}, then define $v_{act}(p) = \mathsf{h}[d](C_{task}[\mathsf{Src}[d]])(p)$; otherwise, define $v_{act}(p) = v(p)$. This gives the new values of all actuator ports. Let C_{act} be the configuration that agrees with v_{act} on the values of $\mathsf{ActPorts}$, and otherwise agrees with C_{task}.
3. [**Sensor ports.**] Consider a port $p \in \mathsf{SensePorts}$. Let $v_{sense}(p)$ be any value in $\mathsf{Type}[p]$; that is, sensor ports change nondeterministically. This is not done by the Giotto program, but by the environment. All other parts of a configuration are updated deterministically, by the Giotto program. Let C_{sense} be the configuration that agrees with v_{sense} on the values of $\mathsf{SensePorts}$, and otherwise agrees with C_{act}.
4. [**Target mode.**] If a mode switch $(\cdot, m_{target}, \cdot) \in \mathsf{Switches}[m]$ is enabled at configuration C_{sense}, then define $m' = m_{target}$; otherwise, define $m' = m$. This determines if there is a mode switch. Recall that at most one mode switch can be enabled at any configuration. Let C_{target} be the configuration with mode m' that otherwise agrees with C_{sense}.
5. [**Mode ports.**] Consider a port $p \in \mathsf{OutPorts}$. If $p \in \mathsf{Dst}[d]$ for some mode switch $(\cdot, \cdot, d) \in \mathsf{Switches}[m]$ that is enabled at configuration C_{sense}, then define $v_{mode}(p) = \mathsf{h}[d](C_{target}[\mathsf{Src}[d]])(p)$; otherwise, we define $v_{mode}(p) = C_{target}[\mathsf{OutPorts}](p)$. This gives the new values of all mode ports of the target mode. Note that mode switching updates also the output ports of all tasks t that are logically running. This does not affect the execution of t. When

t completes, its output ports are again updated, by t. Let C_{mode} be the configuration that agrees with v_{mode} on the values of $\mathsf{OutPorts}$, and otherwise agrees with C_{target}.

6. [**Unit counter.**] If no mode switch in $\mathsf{Switches}[m]$ is enabled at configuration C_{sense}, then define $u' = (u+1) \bmod \omega_{max}[m]$. Otherwise, suppose that a mode switch is enabled at configuration C_{sense} to the target mode m'. Let $\sigma_{running} = \sigma_{active} \setminus \sigma_{completed}$. If $\sigma_{running} = \emptyset$, then define $u' = 1$. Otherwise, let $u_{complete}$ be the least common multiple of the set $\{\omega_{max}[m]/\omega_{task} \mid (\omega_{task}, t, \cdot) \in \mathsf{Invokes}[m] \text{ for some } t \in \sigma_{running}\}$; this is the least number of units of m at which all running tasks complete simultaneously. Let u_{actual} be the least multiple of $u_{complete}$ such that $u_{actual} \geq u$; this is the earliest unit number after u at which all running tasks complete simultaneously. Let $\delta = (\pi[m]/\omega_{max}[m]) \cdot (u_{actual} - u)$; this is the duration until the next simultaneous completion point. Let $u_{togo} = (\omega_{max}[m']/\pi[m']) \cdot \delta$; this is the number of units of the target mode m' until the next simultaneous completion point. Finally, define $u' = (1 - u_{togo}) \bmod \omega_{max}[m']$; this is the unit number in mode m' with $u_{togo} - 1$ units to go until the last simultaneous completion point in a round of mode m'. Thus a mode switch always jumps as close as possible to the end of a round of the target mode. Let C_{unit} be the configuration with the unit counter u' that otherwise agrees with C_{mode}.
7. [**Task input ports.**] Consider a port $p \in \mathsf{InPorts}$. If $p \in \mathsf{Dst}[d]$ for some task invocation $(\cdot, \cdot, d) \in \mathsf{Invokes}[m']$ that is enabled at configuration C_{unit}, then define $v_{input}(p) = \mathsf{h}[d](C_{unit}[\mathsf{Src}[d]])(p)$; otherwise, define $v_{input}(p) = v(p)$. This gives the new values of all task input ports. Let C_{input} be the configuration that agrees with v_{input} on the values of $\mathsf{InPorts}$, and otherwise agrees with C_{unit}.
8. [**Active tasks.**] Let $\sigma_{enabled}$ be the set of tasks t such that a task invocation of the form $(\cdot, t, \cdot) \in \mathsf{Invokes}[m']$ is enabled at configuration C_{input}. The new set of active tasks is $\sigma'_{active} = (\sigma_{active} \setminus \sigma_{completed}) \cup \sigma_{enabled}$. Let C_{active} be the configuration with the set σ'_{active} of active tasks that otherwise agrees with C_{input}.
9. [**Time stamp.**] The next time instant at which the Giotto program is invoked is $\tau' = \tau + \pi[m']/\omega_{max}[m']$. An implementation may use a timer interrupt set to τ'. Let C_{succ} be the configuration with the time stamp τ' that otherwise agrees with C_{active}.

An *execution* of a Giotto program is an infinite sequence $C_0, C_1, C_2, \ldots$ of program configurations C_i such that (i) $C_0 = (0, \mathsf{start}, 0, v, \emptyset)$ with $v(p) = \mathsf{init}[p]$ for all ports $p \in \mathsf{Ports}$, and (ii) C_{i+1} is a successor configuration of C_i for all $i \geq 0$. Note that there can be a mode switch at time 0, but there can never be two mode switches in a row without any time passing.

4 Annotated Giotto

A Giotto program can in principle be run on a single sufficiently fast CPU, independent of the number of modes and tasks. However, taking into account

performance constraints, the timing requirements of a program may or may not be achievable on a single CPU. Additionally, a particular application may require that tasks be located in specific places, e.g., close to the physical processes that the tasks control, or on processors particularly suited for the operations of the tasks. Lastly, in order to achieve fault tolerance, redundant, isolated CPUs may be desirable. For these reasons, it may be necessary to distribute the work of a Giotto program between multiple CPUs. In order to aid the compilation on distributed, possibly heterogeneous, platforms, we allow the annotation of Giotto programs with platform constraints. While pure Giotto is platform-independent, annotated Giotto contains directives for mapping and scheduling a program on a particular platform. An *annotated Giotto program* is a formal refinement of a pure Giotto program in the sense that the logical semantics of the pure Giotto program, as defined in Section 3.2, is preserved.

Annotated Giotto consists of multiple annotation levels. Conceptually, annotations at the higher levels occur prior to annotations at the lower levels. This structured approach has several advantages. First, it permits the incremental refinement of a pure Giotto program into an executable image. Specifically, it allows a modular architecture for the Giotto compiler, with separate modules for mapping and scheduling. Second, it enables the generation of formal models at all annotation levels. These models can be checked for consistency with the annotations at the higher levels [VB93], in particular, for consistency with the pure Giotto semantics.

Formally, a *hardware configuration* consists of a set of *hosts* and a set of *networks*. A host is a CPU that can execute Giotto tasks. A network connects two or more hosts and can transport values. The passing of a value from one port to another (e.g., from a sensor port or a task output port to a task input port) is called a *connection*. Annotated Giotto consists of the following three levels of annotations:

Giotto-H (H for "hardware") specifies a set of hosts, a set of networks, and worst-case execution time information. The WCET information includes the time needed to execute tasks on hosts, and the time needed to transfer connections on networks.

Giotto-HM (M for "map") specifies, in addition, an assignment of task invocations to hosts, and an assignment of connections to networks. The same task, when invoked in different modes, may be assigned to different hosts. The mapping of a task invocation also determines the physical location of the task output ports.

Giotto-HMS (S for "schedule") specifies, in addition, scheduling information for each host and network. For example, every task invocation may be assigned a priority, and every connection may be assigned a time slot.

An annotation is *complete* if it fully determines all assignments at its annotation level, and is *partial* otherwise. In particular, a complete HM annotation maps every task invocation to a host, and maps every connection to a network. The information that a complete Giotto-HMS program needs to specify may vary depending on the scheduling strategy of the RTOS on the hosts, and on the

communication protocols on the networks. For instance, a Giotto-HMS program may specify priorities for task invocations, relative deadlines, or time slots, depending on whether the underlying RTOS uses a priority-driven, deadline-driven, or time-triggered scheduling strategy.

An annotated Giotto program may be *overconstrained*, in that it does not permit any execution that is consistent with the annotations. An annotated Giotto program is *valid* if (i) it is not overconstrained, and (ii) it is consistent with the semantics of the underlying pure Giotto program. A Giotto compiler takes a partially annotated program and can have one of three outcomes: either it determines that the input program is not valid, or it produces a completely annotated, valid HMS refinement (which can then be turned into executable code), or it gives up and asks for more annotations from the programmer. For answering the validity question, a Giotto compiler can generate a formal model on each annotation level. For example, the constraints imposed by a Giotto-HM program can be expressed as a graph of conditional process graphs [EKP+98], one for each mode, which can be checked for validity. A completely annotated Giotto-HMS program, provided it is not overconstrained, specifies a unique behavior of all hosts and networks for every given real-time trace of sensor valuations. These behaviors can be checked for conformance against the higher-level graph model to guarantee Giotto semantics. Given a partially annotated Giotto program, a compiler can generate the missing HMS-annotations based on holistic schedulability analysis for distributed real-time systems that use time-triggered communication protocols [TC94]. Such a compiler can be evaluated along several dimensions: (i) how many annotations it requires to generate valid code, and (ii) what the cost is of the generated code. For instance, a compiler can use a cost function that minimizes jitter of the actuator updates.

5 Discussion

While many of the individual elements of Giotto are derived from the literature, we believe that the study of strictly time-triggered task invocation together with strictly time-triggered mode switching as a possible organizing principle for *abstract, platform-independent real-time programming* is an important, novel step towards separating *reactivity*, i.e., functionality and timing requirements, from *schedulability*, i.e., scheduling guarantees on computation and communication. Giotto decomposes the development process of embedded control software into high-level real-time programming of reactivity and low-level real-time scheduling of computation and communication. Programming in Giotto is real-time programming in terms of the requirements of control designs, i.e., their reactivity, not their schedulability.

The strict separation of reactivity from schedulability is achieved in Giotto through time- and value-determinism: given a real-time trace of sensor valuations, the corresponding real-time trace of actuator valuations produced by a Giotto program is uniquely determined. The separation of reactivity from schedulability has at least two important ramifications. First, reactive (i.e., func-

tional and timing) properties of a Giotto program may be subject to formal verification against a mathematical model of the control design [Hen00]. Second, Giotto is compatible with any scheduling algorithm, which therefore becomes a parameter of the Giotto compiler. There are essentially two reasons why even the best Giotto compiler may fail to generate an executable: (i) not enough platform utilization, or (ii) not enough platform performance. Then, independently of the program's reactivity, utilization can be improved by a better scheduling module, while performance can be improved by faster hardware or leaner software that implements the actual functionality (i.e., the individual tasks) more efficiently.

5.1 Related Work

Giotto is inspired by the time-triggered architecture (TTA) [Kop97], which first realized the time-triggered paradigm for meeting hard real-time constraints in safety-critical distributed settings. However, while the TTA encompasses a hardware architecture and communication protocols, Giotto provides a hardware-independent and protocol-independent abstract programmer's model for time-triggered applications. Giotto can be implemented on any platform that provides sufficiently accurate clock primitives or supports a clock synchronization scheme. The TTA is thus a natural platform for Giotto programs.

Giotto is similar to architecture description languages (ADLs) [Cle96]. Like Giotto, ADLs shift the programmer's perspective from small-grained features such as lines of code to large-grained features such as tasks, modes, and inter-component communication, and they allow the compilation of scheduling code to connect tasks written in conventional programming languages. The design methodology [KZF+91] for the MARS system, a predecessor of the TTA, distinguishes in a similar way *programming-in-the-large* and *programming-in-the-small*. The inter-task communication semantics of Giotto is particularly similar to the MetaH language [Ves97], which is designed for real-time, distributed avionics applications. MetaH supports periodic real-time tasks, multi-modal control, and distributed implementations. Giotto can be viewed as capturing the time-triggered fragment of MetaH in an abstract and formal way. In particular, unlike MetaH, Giotto specifies not only inter-task communication but also mode switches in a time-triggered fashion, and it does not constrain the implementation to a particular scheduling scheme.

The goal of Giotto —to provide a platform-independent programming abstraction for real-time systems— is shared also by the synchronous reactive programming languages [Hal93], such as Esterel [Ber00], Lustre [HCRP91], or Signal [BGJ91]. While the synchronous reactive languages are designed around zero-delay value propagation, Giotto is based on the formally weaker notion of unit-delay value propagation, because in Giotto, scheduled computation (i.e., the execution of tasks) takes time, and synchronous computation (i.e., the execution of drivers) consists only of independent, non-interacting processes. This decision shifts the focus and the level of abstraction in essential ways. In particular, for analysis and compilation, the burden for the well-definedness of values is shifted from logical fixed-point considerations to physical constraints about

platform resources and performance (in Giotto all values are, logically, always well-defined). Thus, Giotto can be seen as identifying a class of synchronous reactive programs that support (i) typical real-time control applications as well as (ii) efficient schedule synthesis and code generation.

5.2 Giotto Implementations

We briefly review the existing Giotto implementations. The first implementation of Giotto was a simplified Giotto run-time system on a distributed platform of Lego Mindstorm robots. The robots use infrared transceivers for communication. Then we implemented a full Giotto run-time system on a distributed platform of Intel x86 robots running the real-time operating system VxWorks. The robots use wireless Ethernet for communication. We also implemented a Giotto program running on five robots, three Lego Mindstorms and two x86-based robots, to demonstrate Giotto's applicability for heterogeneous platforms. The communication between the Mindstorms and the x86 robots is done by an infrared-Ethernet bridge implemented on a PC. For an informal discussion of this implementation, and embedded control systems development with Giotto in general, we refer to the earlier report [HHK01].

In collaboration with Marco Sanvido and Walter Schaufelberger at ETH Zürich, we have been working on a high-performance implementation of a Giotto run-time system on a single-processor platform that controls an autonomously flying model helicopter [San99]. The implementation language is a subset of Oberon for embedded real-time systems [Wir99]. The existing helicopter control software has been reimplemented as a combination of a Giotto program and Oberon code that implements the controller tasks. We have implemented a Giotto compiler that generates, from such a Giotto program, the Giotto executable as Oberon code. The executable uses the Giotto run-time system on the helicopter to control the hard real-time scheduling of the navigation and controller software. We have also been working on an implementation of a virtual hard real-time scheduling machine [Kir01], as an alternative to the Giotto run-time system on the helicopter. The Giotto compiler can generate machine code of the virtual machine instead of Giotto executables in Oberon. This approach has two advantages: (i) code generation is more flexible, because the virtual machine semantics is finer-grained than the API of the Giotto run-time system, and (ii) increased portability of the generated code.

Acknowledgments. We thank Rupak Majumdar for implementing a prototype Giotto compiler for Lego Mindstorms robots. We thank Dmitry Derevyanko and Winthrop Williams for building the Intel x86 robots. We thank Edward Lee and Xiaojun Liu for help with implementing Giotto as a "model of computation" in Ptolemy II [DGH+99]. We thank Marco Sanvido for his suggestions on the design of the Giotto drivers.

References

[Ber00] G. Berry. The foundations of Esterel. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*, pp. 425–454. MIT Press, 2000.

[BGJ91] A. Benveniste, P. Le Guernic, and C. Jacquemot. Synchronous programming with events and relations: The Signal language and its semantics. *Science of Computer Programming*, 16:103–149, 1991.

[Cle96] P. Clements. A survey of architecture description languages. In *Proc. 8th International Workshop on Software Specification and Design*, pp. 16–25. IEEE Computer Society Press, 1996.

[Col99] R.P.G. Collinson. Fly-by-wire flight control. *Computing & Control Engineering*, 10:141–152, 1999.

[DGH+99] J. Davis, M. Goel, C. Hylands, B. Kienhuis, E.A. Lee, J. Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Reekie, N. Smyth, J. Tsay, and Y. Xiong. *Ptolemy II: Heterogeneous Concurrent Modeling and Design in Java.* Technical Report UCB/ERL-M99/44, University of California, Berkeley, 1999.

[EKP+98] P. Eles, K. Kuchcinski, Z. Peng, A. Doboli, and P. Pop. Process scheduling for performance estimation and synthesis of hardware/software systems. In *Proc. 24th EUROMICRO Conference*, pp. 168–175, 1998.

[Hal93] N. Halbwachs. *Synchronous Programming of Reactive Systems.* Kluwer, 1993.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proc. IEEE*, 79:1305–1320, 1991.

[Hen00] T.A. Henzinger. Masaccio: A formal model for embedded components. In *Proc. First IFIP International Conference on Theoretical Computer Science*, LNCS 1872, pp. 549–563. Springer-Verlag, 2000.

[HHK01] T.A. Henzinger, B. Horowitz, and C.M. Kirsch. Embedded control systems development with Giotto. In *Proc. SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, ACM Press, 2001.

[Kir01] C.M. Kirsch. *The Embedded Machine.* Technical Report UCB/CSD-01-1137, University of California, Berkeley, 2001.

[Kop97] H. Kopetz. *Real-time Systems: Design Principles for Distributed Embedded Applications.* Kluwer, 1997.

[KZF+91] H. Kopetz, R. Zainlinger, G. Fohler, H. Kantz, P. Puschner, and W. Schütz. The design of real-time systems: From specification to implementation and verification. *IEE/BCS Software Engineering Journal*, 6:72–82, 1991.

[LRR92] D. Langer, J. Rauch, and M. Rößler. *Real-time Systems: Engineering and Applications*, chapter 14, pp. 369–395. Kluwer, 1992.

[San99] M. Sanvido. *A Computer System for Model Helicopter Flight Control; Technical Memo 3: The Software Core.* Technical Report 317, Institute of Computer Systems, ETH Zürich, 1999.

[TC94] K. Tindell and J. Clark. Holistic schedulability for distributed hard real-time systems. *Microprocessing and Microprogramming*, 40:117–134, 1994.

[VB93] S. Vestal and P. Binns. Scheduling and communication in MetaH. In *Proc. 14th Annual Real-Time Systems Symposium.* IEEE Computer Society Press, 1993.

[Ves97] S. Vestal. MetaH support for real-time multi-processor avionics. In *Proc. Fifth International Workshop on Parallel and Distributed Real-Time Systems*, pp. 11–21. IEEE Computer Society Press, 1997.

[Wir99] N. Wirth. *A Computer System for Model Helicopter Flight Control; Technical Memo 2: The Programming Language Oberon SA*, second edition. Technical Report 285, Institute of Computer Systems, ETH Zürich, 1999.

Directions in Functional Programming for Real(-Time) Applications*

Walid Taha**, Paul Hudak, and Zhanyong Wan

Department of Computer Science,
Yale University, New Haven, CT, USA.
{taha,hudak,zwan}@cs.yale.edu

Abstract. We review the basics of functional programming, and give a brief introduction to emerging techniques and approaches relevant to building real-time software. In doing so we attempt to explain the relevance of functional programming concepts to the real-time applications domain. In particular, we address the use of *types* to classify properties of real-time computations.

"If thought corrupts language, language can also corrupt thought."
George Orwell, *Politics and the English Language*

1 Introduction

An important challenge facing functional programming is the successful application of its principles to the domain of real-time software. Examples of real-time software include controllers in audio systems, video cameras, video games, imaging and data acquisition systems, and telecommunications hardware. The significance of this challenge is two fold: First, real-time applications are constantly growing in complexity, and society is growing more dependent on their correctness. Second, real-time applications possess characteristics that have been traditionally considered outside the scope of functional programming languages. For example, real-time systems are often required to be:

- *Responsive, reactive, or live*: a response must be made to every input.
- *Resource-bounded*: responses must happen in a limited amount of time, using limited hardware.
- *Concurrent*: components may run in parallel and manipulate shared data.
- *Networked or distributed*: communication between components of the system may involve time delays or loss of information.

The purpose of this paper is to illustrate how many fundamental aspects of functional programming are highly relevant not only to programming but also

* Funded by DARPA F33615-99-C-3013 and NSF CCR-9900957.
** Funded by NSF ITR-0113569 and subcontract #8911-48186 from Johns Hopkins University under NSF agreement Grant # EIA-9996430.

T.A. Henzinger and C.M. Kirsch (Eds.): EMSOFT 2001, LNCS 2211, pp. 185–203, 2001.
© Springer-Verlag Berlin Heidelberg 2001

to *understanding* real-time systems. Following on a long tradition of mathematical modeling using functions [3,22,38,89,96], we summarize the key benefits of functional programming. In doing so, we put into context many on-going efforts both in and outside the area of functional programming to address issues that are specific to the real-time domain.

1.1 Audience and Organization of This Paper

The classic paper by Hughes presents excellent motivation for general purpose functional programming [48]. The present paper is aimed at a reader interested in using functional programming for real-time applications. We assume the reader has a background in discrete mathematics, a few years experience with programming in a mainstream language, and an informal familiarity with basic programming language concepts. We focus on:

- Programming by writing and composing *functions* (Section 2),
- Understanding of the notion of *types* and *type systems* (Section 3),
- Using *higher-order* functions (Section 4), and
- Appreciating the presence of a wealth of formal *semantics treatments* for various aspects of functional languages (Section 5).

2 The Big Deal about Functions

What sets functions apart from arbitrary relations is two simple properties: for two given sets A and B, a relation $f \subseteq A \times B$ is a *function* (written $f : A \to B$) if and only if it satisfies:

- *Totality*: for every $a \in A$, there exists an element $b \in B$ such that $(a, b) \in f$, and
- *Uniqueness*: for any $a \in A$ and $b, b' \in B$, if $(a, b) \in f$ and $(a, b') \in f$, then $b = b'$. We write $f(a)$ to denote this unique element b.

A relation that satisfies only uniqueness is called a *partial function*.

A framework that allows us to model programs as functions (from input to output) has a natural appeal in the context of real-time systems, where each of these properties is desirable: Totality means that the program will always respond to its input, and uniqueness means that the program will return the same result whenever it is subjected to the same input. While it may be hard to see how this simple intuition scales to more complex settings, a key quality of functions is that they compose naturally and without surprising interactions.

In this section we begin with a brief review of the state of the art both in functional programming languages and programming languages for real-time applications. We then proceed to illustrate how a "language of functions" can be an expressive tool for the specifying real-time computational models.

2.1 State of the Art

A wide variety of functional programming languages are available today, including Scheme [86], SML [66], Haskell [78], and OCaml [60]. But none of these provides full support for programming with functions in the sense described above, for two reasons. First, all of them permit the definition of partial functions, thus leading to potentially non-terminating computations. Second, except for Haskell, none of these languages are "purely" functional: that is, they allow the expression of imperative programs that violate the uniqueness property. For example, while Scheme is a highly expressive and versatile programming language, historically its main goal was list and symbolic processing, and not so much programming with functions. Scheme is not statically-typed and can express "impure functions" that depend implicitly on interactions with both machine state and the real world. ML and OCaml are typed functional languages, but still permit partial and imperative computations.

Of these popular languages, Haskell comes closest to being "purely" functional, in that both local state and interactions with the real world are modeled without side-effects. Such interactions are also typed explicitly (see the discussion of *monads* below), ensuring that the type still reflects these interactive properties. At the same time, Haskell provides special syntax for "imperative" features. Although this paper presents most concepts in generic mathematical notation, it is fairly straightforward to encode our equations in Haskell.

There are frameworks that fully support programming with functions. Important examples of such systems are Elementary Strong Functional Programming (ESFP) [94], Nuprl [2], Martin-Löf's Type Theory [74], the Calculus of Constructions [21,28,73], Charity [19,20], LEGO [61], and Twelf [80]. These languages are still not in the mainstream, but we predict that they will grow in popularity in the coming years.

Functional Programming for Real-time Applications. While many languages developed for the real-time domain are "imperative" (such as ESTEREL [9] and STATECHARTS [37]), a number of these languages are expressly functional, including LUSTRE [15], synchronous Kahn networks [16], SAFL [70] and FRP [30,99]. But essentially all the "imperative"' ones are deterministic (thus satisfying uniqueness) and terminating (thus satisfying totality), and so it is reasonable to classify them as functional.[1] This statement is less surprising when we note that Haskell also supports and encourages the imperative style, as long as side-effects are properly encapsulated.

To guarantee termination, most real-time languages disallow general recursion. Synchronous Kahn networks were developed as an extension to LUSTRE that provides recursion and higher-order programming, but termination is sacrificed for this expressivity. Implementations of LUSTRE also have a macro-like

[1] ESTEREL first translates programs into circuits (or state machines), then these circuits are analysed to ensure determinism.

facility that supports recursion, but runs the risk of causing the compiler to diverge. FRP is embedded in Haskell [46,31,47] and so inherits recursion and nontermination. RT-FRP [100] is a subset of FRP that guarantees resource-bounded program execution: every interactive RT-FRP computation terminates. Using a special type system, RT-FRP still allows two forms of recursion similar in spirit to the idea of tail-recursion [86]. Because RT-FRP is a closed language [56], it is possible to *guarantee* that no partiality is accidentally introduced.

2.2 Functions as a Tool for Analysis and Modeling of Computation

If we temporarily abstract away from issues of performance, practically all interesting features of program behavior can be specified precisely using functions. This means that we can easily *simulate* or *prototype* most interesting computational phenomena in a functional language. We believe that this has tremendous conceptual benefits, and that, in the long term, will have the same impact on software engineering processes. In what follows, we give a brief description of such computational phenomena, and review how they can be described in a purely functional manner.

Input/Output, Uniqueness, and Interaction. A function cannot implicitly interact with an "outside world" during the course of its computation. If it did, then it is very possible that each time it is applied to a value it would produce a different output. In other words, this computation would not have the uniqueness property.

A simple approach to modeling such computations is to view them as a chain of functions that determine successive interactions with the outside world. Given two types input and output, a recursive type equation can be written to describe such a computation as follows:

$$\mathsf{computation} \rightleftharpoons \mathsf{input} \to (\mathsf{output} \times \mathsf{computation}).$$

This equation says that a computation is a function which takes an input and returns a pair. The pair consists of both the output of the first computation and the computation that should be carried out on the next input. As such the type $\mathsf{computation}$ models a strictly infinite sequence of interactions with the outside world (as long as the outside world is providing an input). This model of computations is well-suited for reactive and interactive systems.

A number of reactive languages, such as Lava [18], Hawk [62], and FRP [46], have used streams to implement the idea presented above. *Streams* are infinite datastructures which can be easily defined in a lazy language. However, most of these systems have been developed as languages embedded into Haskell. RT-FRP implements the same model, but as a closed language, which makes it more suitable for direct use in a real-time setting.

Runtime Errors and Partiality. A common challenge that one encounters when trying to model various features of computation as functions is dealing with partiality. For example, consider the following definition:

$$f(x) \triangleq 1/x.$$

When $x = 0$, $f(x)$ is not defined. Definitions similar to the one above are allowed in most programming languages, even statically-typed ones that "guarantee that runtime errors do not occur." This is not a flaw with these languages, because what they do guarantee is the preclusion of a *specific* class of runtime errors, not all [13,14]. Whenever we start with primitives such as division that can themselves generate runtime errors, those errors are generally ignored by type systems. However, in the design of real-time systems where raising errors is simply not acceptable, such primitives become a concern.

It is possible to prevent errors by replacing the faulty set of primitives by a modified set. For example, we can define a new division function:

$$x /\!\!/ y \triangleq \text{ if } y = 0 \text{ then } 0 \text{ else } x/y.$$

A more elegant way to dealing with partiality is to introduce a distinguished value $\perp$ (read "bottom") to be returned in the case of an error. To introduce such a value in a type-sound manner, we use a *lifting* type constructor $_\perp$ defined as follows:

$$\alpha_\perp \ ::= \ \perp \mid \alpha_\perp.$$

Intuitively, this specification can be read as a parameterized BNF definition. The variable α on the left-hand side is a type variable that can be instantiated to a specific type, and on the right-hand side denotes a term of that type. The first variant in this definition tells us that $\perp$ has type $\alpha_\perp$ for any type α. The second variant tells us that a term $e_\perp$ has type $\alpha_\perp$ whenever e is a term of type α. The type constructor $\alpha_\perp$ is a standard concept that allows us to say that we may or may not get a value as a result, and hence is useful for modeling partiality.[2]

Now we can redefine our division function as:

$$x /\!\!/ y \triangleq \text{ if } y = 0 \text{ then } \perp \text{ else } (x/y)_\perp$$

All we need to do is to check that the result of this operation is not $\perp$ before we continue on to other computations.

Infinite Loops. In any Turing-complete language, it is well known that one can write programs that run forever without returning a result. This means that we cannot directly treat arbitrary programs as functions. But we can still provide functional models of such programs. In fact, we can do this using a combination of the two techniques just introduced. We use a *sum* type constructor $_ + _$ that takes two type parameters, and is defined as follows:

$$\alpha + \beta \ ::= \ \mathsf{Alpha}\ \alpha \mid \mathsf{Beta}\ \beta.$$

[2] In implementations of typed functional programming languages $\alpha_\perp$ is known as either a "maybe" or an "option" type.

This new constructor allows us to package up values of two different types α and β in one common type $\alpha + \beta$, and still retain the ability to recover the original values.[3] For any term e of some type α, the term $\mathsf{Alpha}\ e$ has type $\alpha + \beta$, for any type β. The Beta variant lets us do things the other way around. Now, we can define a new kind of computation as:

$$\begin{aligned} \mathsf{computation} &\rightleftharpoons \mathsf{input} \to \mathsf{initialized\text{-}computation} \\ \mathsf{initialized\text{-}computation} &\rightleftharpoons \mathsf{output} + \mathsf{initialized\text{-}computation} \end{aligned}$$

Such a computation takes an input and returns an initialized computation. An initialized computation is either an output, in which case we are done, or another initialized computation. In the second case, we have "try again." Intuitively, this model is similar to observing a Turing machine as it performs a step of a program. If a program is finished, it generates an output. If not, we get back a machine that has been advanced by one step, and then we can try again.

Note that this model of computation is, at least intuitively, a special case of the model for interaction presented above. This idea has been used to provide a form of recursion suitable for real-time applications in RT-FRP.

Concurrency, Randomness, Non-determinism, and Sets. Possibly the biggest conceptual challenge to programming with functions is *non-determinism*. Applications that try to take advantage of either concurrency or randomness often end up having to deal with non-determinism in one form or another. A basic approach to modeling non-determinism in the functional setting is to use of *sets* [82,88]. But a naive account of sets can itself get in the way of programming functionally. For example, the order of elements in a set is generally considered irrelevant. If our language provides any mechanism for turning an arbitrary set into an ordered data structure, such as a list, then the language is forcing us to make one of two choices: either the order of elements has to be fixed using some mechanism (which would weaken our ability to model non-determinism accurately), or the mechanism for turning sets into lists would itself be non-deterministic (in which case we would lose the uniqueness property of the language).

A simple approach to soundly modeling a set is by its characteristic function:

$$\mathsf{set}(\alpha) \ \rightleftharpoons\ \alpha \to \mathsf{bool}$$

For example, a specific set of integers can be represented by a characteristic function of type $\mathsf{int} \to \mathsf{bool}$. With this representation, it is easy to define basic set-theoretic functions, such as union, intersection, complement, and membership. Slightly richer models allow us to define other operations on sets, such as size. A large body of techniques exist for the specification of functional data structures that have very reasonable performance characteristics [76].

To model concurrent/distributed computation, we must consider the effect of running two or more interactive computations simultaneously. The model

[3] As such, sums provide a kind of overloading.

would have to explicitly construct the set of all possible interactions between these computations. Although reasoning about such a model is not easier than about any non-deterministic computation, using function can help us identify the sources of complexity.

Stateful Computation. Pure functions do not have "memory." So, no matter what a function has been applied to before, it will always return the same result for the same input. How can we write a program that reads a sequence of numbers, and continually prints the sum of the numbers up to this point? The essential idea is to explicitly pass around the relevant *state*. In fact, the interactive model that we presented earlier is also suitable for realizing this idea. That is, a computation of type:

$$\mathsf{computation} \rightleftharpoons \mathsf{int} \to (\mathsf{int} \times \mathsf{computation})$$

can be a function that takes an input and returns the current sum as output together with a computation that continues this process indefinitely. This idea is exploited in RT-FRP, for example, to realize operations like integration.

3 Types and Type Systems

In addition to their "traditional role" in programming, types have an important role in the classification of models and data structures. We begin this section by first presenting a possible application of types as a tool for classifying models of computation for real-time applications, and then move on to discuss their newly emerging role as a tool for making assertions about performance.

3.1 Types as a Tool for Classifying Models of Computation

Model theory was developed partly as a tool for the classification of various mathematical structures [42]. Types can do the same for models of computation.

In general, we can think of a reactive system as a map H that takes an input X from the environment and returns an output Y, that is,

$$Y = H(X).$$

The accuracy, tractability, and over all appropriateness of a particular model depends on the specifics of each of these variables. A particularly important aspect of these models is the treatment of time (or "the clock"), which can be treated very concretely as a real number ($\mathbb{R}$), more abstractly as a natural number ($\mathbb{N}$), or even more abstractly as a partial order [57]. In what follows we briefly demonstrate how types can capture many of the key characteristics of some common models for reactive systems [59].

Continuous/Differential Model. The finest scale which we can use to model the physical world (in the Newtonian sense) is achieved by using a time-line of reals, and measuring features of the world as real numbers. We can also assume that we can respond to these measurements in any manner, as long as it is causal. Mathematically, our model would be:

$$Y(t) \;=\; H(X|_{[0,t]}, t) \quad \text{and} \quad \begin{array}{l} X : \mathbb{R} \to \mathbb{R}^i, \\ Y : \mathbb{R} \to \mathbb{R}^o, \end{array}$$

where i and o are natural numbers corresponding to the number of inputs and outputs to the system, and R^j is the Cartesian product of R repeated j times. The restriction $|_{[0,t]}$ on the domain of X captures the standard constraint that H is causal. Finding solutions to such equations where H is unconstrained, however, would be extremely hard. More often we restrict our model to be one where H is a tractable transformation on reals. For example, we may restrict ourselves to ordinary differential equations. In this case, H can be a polynomial of integro-differential operators on the inputs X. While this model is still extremely accurate for many applications, it ignores the digital aspect of the machinery that is typically used to implement such computations.

Discrete/Difference Model. A more tractable model is the discrete/difference-equation model. In a formal sense, this model is an approximation of the one presented above. It can be described as follows:

$$Y(n) \;=\; H(X|_{0..n}, n) \quad \text{and} \quad \begin{array}{l} X : \mathbb{N} \to \mathbb{R}^i, \\ Y : \mathbb{N} \to \mathbb{R}^o. \end{array}$$

In this case, H is typically a polynomial of values of X at times $0..n$.

Synchronous/Reactive Model. Several real-time programming languages are built upon the notion of *synchronous* computation. In the literature, the word "synchrony" has two meanings, corresponding to two important properties of synchronous languages [7]: first, it implies the ability to share a common time scale (that is, the logic time does not advance until all computations in the current "step" have been completed); second, the elements of the streams are to be consumed as soon as produced, making it possible to allocate only one cell for one stream. The first part of this definition is already present in the discrete model. The second is a performance issue, which is determined by the primitives that we are allowed to use in combining signals. A key change in the model, in our view, is that it becomes possible to use conditionals and partial functions on signals. From this point of view, it is only a minor variation on the model above:

$$\begin{array}{l} Y(0) \quad\;\; = Y_0 \\ Y(n+1) = H(X|_{n+1}, Y|_n) \end{array} \quad \text{and} \quad \begin{array}{l} X : \mathbb{N} \to (\mathbb{R}_\perp)^i, \\ Y : \mathbb{N} \to (\mathbb{R}_\perp)^o. \end{array}$$

where Y_0 is some initial state. An important change here is that the memory of the system is restricted to a fixed size (width of Y). With the constant developments of exact-arithmetic methods [29], this model can be realized quite effectively.

Finite State Machine Model. Reals can in general require an arbitrary amount of space to represent. This makes analyzing space requirements non-trivial, and makes another highly tractable model very attractive: finite state machines. This model can be described as follows:

$$\begin{array}{l} Y(0) \quad\;\; = Y_0 \\ Y(n+1) = H(X|_{n+1}, Y|_n) \end{array} \quad \text{and} \quad \begin{array}{l} X : \mathbb{N} \to \mathsf{one\text{-}bit\text{-}input}^i, \\ Y : \mathbb{N} \to \mathsf{one\text{-}bit\text{-}state}^o, \end{array}$$

The main difference between this model and the previous one is that the set of inputs and states is restricted to be finite. This yields a model of computation that is especially tractable and easy to analyze, at least for reasonably small state sizes.

Abstract State Machine Model. This model is a generalization of finite state machines, where a state consists of a term rather than just a fixed number of bits. At any n, the size of each term is finite, but there is no static limit on this size.

$$\begin{array}{l} Y(0) \quad\;\; = Y_0 \\ Y(n+1) = H(X|_{n+1}, Y|_n) \end{array} \quad \text{and} \quad \begin{array}{l} X : \mathbb{N} \to \mathsf{term\text{-}input}^i, \\ Y : \mathbb{N} \to \mathsf{term\text{-}state}^o. \end{array}$$

Discrete Event (or Event-Driven) Model. An increasingly popular model is the discrete event model. It can be viewed as a special case of the finite state machine model, where updating a special set of inputs (the "event") is what drives the global clock. This model can be described as follows:

$$\begin{array}{l} Y(0) \quad\;\; = Y_0 \\ Y(n+1) = H(I|_{n+1}, X|_{n+1}, Y|_n) \end{array} \quad \text{and} \quad \begin{array}{l} I \;\; : \mathbb{N} \to \mathsf{event}, \\ X : \mathbb{N} \to \mathsf{input}^i, \\ Y \;\; : \mathbb{N} \to \mathsf{output}^o, \end{array}$$

where $I(n)$ is the nth event that occurs during the system's life. So-called cycle-driven models are special cases of discrete event models, where some events are planned to occur at fixed intervals.

3.2 Types for Performance

Most functional programming languages are based on the lambda-calculus [17,4]. The untyped lambda-calculus is deterministic but not terminating. In fact, in the absence of recursion and primitives that may introduce partiality, all programs in the simply-typed lambda-calculus are terminating (see Hindley [39] for an excellent introduction). The same is also true for more sophisticated typed lambda-calculi, such as the Calculus of Constructions [5]. As such, typing is essential for using the lambda-calculus as a language for programming with functions. But while these type systems guarantee termination, it might require super-exponential time [101]. This means that a traditional typed lambda-calculus is not suitable for the real-time domain. It is only with the advent of more modern concepts that type systems are proving to be useful for characterizing resource consumption at a finer level. We give a brief introduction to some of these developments in this section.

Abstract Functors and Di-Functors. A *functor* is essentially a parameteric data type, such as a list. For any type α, the list functor allows us to form the type $\mathsf{List}(\alpha)$, which is a list of values of type α. Adding functors to a language can extend its expressivity in many ways.

As with data types, functors can be concrete or abstract. List is usually a concrete data type. Modern programming languages allow users to introduce new functors using either data type declarations or class instances. Abstract functors, on the other hand, allow us to go beyond what can be defined in our language, and can provide powerful encapsulation mechanisms. For example, consider a situation where we want to handle values of type $\mathsf{CouldDiverge}(\alpha)$, which are computations that could either yield α when evaluated or diverge. Can we introduce such a data type into a functional setting without losing uniqueness or totality? This and many other interesting problems can be addressed by a variety of functors:

Monads: The very question raised above, for example, turns out to be fundamental. It can be expanded to include many *computational effects* other than non-termination, such as state, concurrency, and exceptions. A particular flavor of abstract functors called monads [68,97,79] was shown to be suitable for precisely that. Using a monad, a wide variety of effectful computations can be passed around, combined, and extended within a purely functional setting.

Linear types: Another kind of abstract functor is one based on linear logic [6]. Linear logic provides a means to expressing a notion of a "resource" in the lambda-calculus. Recent work in this area has shown how linear types can be used to build expressive programming languages with dynamic data structures that can be compiled into malloc-free C [45]. Almost exactly the same system has also been used to build a language where all programs run in constant space and can take at most polynomial time to execute [1,43]. Programs in both languages are terminating.

Reactivity: FRP is built around two functors called Behavior and Event, corresponding to continuous-time behaviors and discrete event occurrences. Semantically, the same type constructors are present in RT-FRP, but they are made implicit in the types. This is achieved by having a specialized type system that only addresses the reactive part of FRP.

Staging: Multi-stage languages [12,90,92] are based on the key ideas of multi-level [25,26,33] and two-level languages [52,71], and provide mechanisms for building programs that execute in multiple distinct stages. This is achieved by providing constructs for building, combining, and executing code at runtime. The presence of these constructs provides both hygienic program generation and reflection mechanisms, all in one, statically-typed framework. Multi-stage languages also provide a basis for heterogeneous languages, which combine more than one "traditional" language in the same framework. All of these languages provide a

functor for "code". It is an abstract notion of code because it is usually not possible to inspect its text. So, a more accurate way of describing this kind of code is as a "future stage" computation. A variant of the code functor has been used in RT-FRP to model "exportability annotations:" only those variables whose type is marked as "exportable" can participate in computations in the base language. This allows a certain kind of cyclic definitions to be detected statically, and to ensure totality.

We are not aware of many results on multi-stage programming in a resource-bounded setting. McKay and Singh use partial evaluation for the dynamic specialization of FPGAs [63]. We see this as an important direction of future research.

Arrows: More recently, it has been noted that it is useful to distinguish between two kinds of types that a functor can be parameterized by: input types and output types.[4] This gives rise to a particular brand of di-functors called *arrows*. Arrows can be viewed as a generalization of monads, and have been shown to model a variety of interesting kinds of computations. Efforts are under way to explore the utility of the notion of arrows in FRP.

Type-and-Effect Systems and Indexed-Types. Another approach to increasing the power of type systems is to enrich types with annotations that capture additional information about a value. An early example of this approach is the type-and-effect system used by Talpin and Jouvelot [93] to introduce side-effects in a functional manner. In this system, the type of every "impure function" is enriched with information about which variables are read or written when it is executed. This approach is as an alternative to monads. In some instances the two are indeed equivalent [54,98]. Effects were later used to develop region-based memory management [11,95]. In this approach, each type carries the name of a region where it is allocated, in addition to the standard effect information. This enables a safe and high-level form of explicit allocation/deallocation of memory. Explicit memory management can make the execution of programs more predictable than if it is left to a garbage collector. Effects have also been used to build a type system for a multi-threaded language where freedom of deadlocks can be statically tested [32].

Sized types are types enriched to capture the amount of space needed to store a value. For example, a list of length n carrying values of type α would have the type $\mathsf{List}_n(\alpha)$. This idea has been used to build languages for reactive systems, where properties such as termination and resource-boundedness can be verified statically [50]. An especially interesting aspect of this work is that one of the rules explicitly encodes an induction principle, which allows the programmer to use recursion, as long as the type system can check that it is well-founded. This idea is explored in the context of an execution model where reactive systems

[4] For the reader familiar with subtyping, the distinction alluded to here is related to co- and contra-variance of type constructors.

are viewed as stream-processors where the rates of the various streams can be different, and the key constraint is to ensure the "liveness" of the output.

It has also been shown that sized types and region effects can be combined naturally in a first-order system [49].

Using ideas from dependent typing, Crary and Weirich [24] develop a type system that provides an explicit upper bound on the number of steps needed to complete the computation. Space is conservatively bounded by the same bound as time. The language does not have recursion, rather, provides special iterators that are always guaranteed to terminate. The language supports higher-order functions.

LUSTRE and Synchronous Kahn networks use the notion of a *clock calculus*, which is essentially a type system characterizing the clocks underlying each expression. Programs that are well-typed in this system, and that also pass a cyclic dependency test, are guaranteed to be well-behaved.

4 Higher-Order Functions

Intuitively, higher-order functions are programming patterns. Formally, higher-order functions are ones that can take functions as arguments, or return functions as results. We have already seen the usefulness of returning functions: it was used in various models of computations that we discussed earlier. In particular, it provides the ability to build new functions that are "evolved" versions of older ones. Passing functions as arguments is equally useful, as it provides a mechanism for parameterizing programs by functions.

Consider, for example, the pattern of performing point-wise operations on elements of a container type, such as $\mathsf{List}(\alpha)$. One way to do this is to write a recursive (or iterative) program each time we want to carry out this pattern. But with higher-order functions, we can capture this pattern once and for all with one function (call it map) that has the following type:

$$\mathsf{map} : (\alpha \to \beta) \times (\mathsf{List}(\alpha)) \to (\mathsf{List}(\beta)).$$

The presence of higher-order functions can drastically enhance the expressivity of a programming language [75].

Higher-order programming is generally not supported in programming languages intended for the real-time domain. There are, however, a few notable exceptions, such as FRP and synchronous Kahn networks. The main reason seems to be that it is harder to guarantee resource boundedness in the presence of higher-order functions. In RT-FRP, for example, higher-order functions are not supported directly.

5 Mathematical Semantics

The semantics of a programming language is a mathematical specification of what the programming language is supposed to do. While a semantics can be

abstract, it is not necessarily so. There is a colorful spectrum of ways to define the semantics of a programming language. A number of balanced textbook treatments of the various approaches already exist [36,67,72], as do more advanced studies [23]. There are two flavors of semantics that are often viewed as being in strong contrast: *denotational* semantics and *operational* semantics.

Denotational semantics is generally presented by a translation from syntax to a more abstract mathematical domain, often called "the meaning" or "denotation" of the syntax. It is concerned with traditional program equivalence, which is a subtle and technical subject. A denotational semantics is frequently chosen as the reference when it is the simplest semantics. This simplicity is often what makes it abstract. While abstractness is perfect if we are only interested in reasoning about program equivalence, it can have two disadvantages: first, it can be alienating for the practitioner who does not have the mathematical background to understand it. Second, it can be too abstract. For example, a denotational semantics does not traditionally describe the cost of a computation, which is a crucial concern in real-time applications.

Operational semantics is generally presented by a set of rules for "building" a computation, and thus can provide a basis for discussing performance. Originally promoted by Plotkin [83,85,10], in recent years this approach has gained substantial popularity [35,69,81,87]. When combined with typing, operational techniques can also provide powerful proof techniques [34].

Essentially all of the languages mentioned above have been given some kind of formal semantics. A large effort has been made to develop the semantics of ESTEREL, including a denotational semantics in the point view of Scott's ternary logic, an operational semantics based on an interpretation scheme expressed by term rewriting rules defining microstep sequences, and a circuit semantics based on a translation of programs into circuits. These semantics are shown to correspond in a certain way, constrained only by a notion of stability [8].

A distinguishing characteristic of FRP is that it has a continuous-time based denotational semantics. It has also been shown that a stream-based implementation of FRP converges to this semantics at the limit as sampling intervals drop to zero, modulo some uniform continuity conditions [99].

RT-FRP has a deterministic operational semantics. This semantics allows a notion of cost to be defined in terms of derivation size, which is necessary for proving that RT-FRP is resource bounded.

6 How Do I Learn More?

Jones [51] presents a language-based account of complexity and computability. Hofmann [44] presents a detailed overview of results in the area of programming languages that capture complexity classes.

While there are few implementations of purely functional programming languages, there is a number of good introductions to programming in Haskell which

would still serve as an excellent introduction to the subject nevertheless [47,46]. Di Cosmo's monograph on isomorphisms of types [27] and Hindley's treatment of the simply-typed lambda-calculus [39] are excellent introductions to types in the sense that we have used them here.

Lee [58] gives an overview of how many ongoing efforts fit into the greater picture of transferring software engineering techniques to the area of embedded and real-time systems.

Finally, we have not discussed traditional real-time techniques like priority-based and rate-monotonic scheduling [55]. We expect that these approaches can fit within the traditional frameworks for concurrency [64,84,40,41,65], and that the same approaches discussed above for the encapsulation of concurrency apply.

Acknowledgments. We would like to thank Françoise Bellegarde, Adriana Compagnoni, Bill Harrison, Gordon Pace and John Peterson for reading and giving us feedback on an earlier draft of the paper.

References

1. Klaus Aehlig and Helmut Schwichtenberg. A syntactical analysis of non-size-increasing polynomial time computation. In *the Symposium on Logic in Computer Science (LICS' 00)*, pages 84–94. IEEE, June 2000.
2. Stuart Allen, Robert Constable, Richard Eaton, Christoph Kreitz, and Lori Lorigo. The nuprl open logical environment. In D. McAllester, editor, *the International Conference on Automated Deduction*, volume 1831 of *Lecture Notes in Artificial Intelligence*, pages 170–176. Springer-Verlag, 2000.
3. John Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, 1978.
4. Henk P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, revised edition, 1984.
5. Henk P. Barendregt. Lambda calculi with types. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*. Oxford University Press, Oxford, 1991.
6. Nick Benton and Philip Wadler. Linear logic, monads and the lambda calculus. In *the Symposium on Logic in Computer Science (LICS '96)*, New Brunswick, 1996. IEEE Computer Society Press.
7. Gerard Berry. Real time programming: Special purpose or general purpose languages. In *IFIP World Computer Congress*, San Francisco, 1989.
8. Gerard Berry. The constructive semantics of pure Esterel (draft version 3). Draft Version 3, Ecole des Mines de Paris and INRIA, July 1999.
9. Gerard Berry and the Esterel Team. *The Esterel v5.21 System Manual*. Centre de Mathématiques Appliquées, Ecole des Mines de Paris and INRIA, March 1999. Available at http://www.inria.fr/meije/esterel.
10. Manfred Broy. A fixed point approach to applicative multi-programming. In *Lecture Notes*. International Summer School on Theoretical Foundations of Programming Methodology, 1981.

11. Cristiano Calcagno. Stratified operational semantics for safety and correctness of the region calculus. In *the Symposium on Principles of Programming Languages (POPL'01)*, 2001.
12. Cristiano Calcagno, Eugenio Moggi, and Walid Taha. Closed types as a simple approach to safe imperative multi-stage programming. In *the International Colloquium on Automata, Languages, and Programming (ICALP '00)*, volume 1853 of *Lecture Notes in Computer Science*, pages 25–36, Geneva, 2000. Springer-Verlag.
13. Luca Cardelli. Typeful programming. In E. J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts*, IFIP State-of-the-Art Reports, pages 431–507. Springer-Verlag, New York, 1991.
14. Luca Cardelli. Type systems. In Allen B. Jr Tucker, editor, *The Computer Science and Engineering Handbook*. CRC Press, 1997.
15. Paul Caspi, Halbwachs Halbwachs, Nicolas Pilaud, and John A. Plaice. Lustre: A declarative language for programming synchronous systems. In *the Symposium on Principles of Programming Languages (POPL '87)*, January 1987.
16. Paul Caspi and Marc Pouzet. Synchronous Kahn networks. In *the International Conference on Functional Programming (ICFP'96)*, pages 226–238, Philadelphia, Pennsylvania, 24–26 May 1996.
17. Alonzo Church. *The Calculi of Lambda Conversion*. Princeton University Press, Princeton, 1941.
18. Koen Claessen. *Embedded Languages for Describing and Verifying Hardware*. PhD thesis, Chalmers, 2001.
19. J. Robin B. Cockett and Dwight Spencer. Strong categorical datatypes I. In R. A. G. Seely, editor, *Proceedings International Summer Category Theory Meeting, Montréal, Québec, 23–30 June 1991*, volume 13 of *Canadian Mathematical Society Conf. Proceedings*, pages 141–169. American Mathematical Society, Providence, RI, 1992.
20. J. Robin B. Cockett and Dwight Spencer. Strong categorical datatypes II: A term logic for categorical programming. *Theoretical Computer Science*, 139(1–2):69–113, 1995.
21. Thierry Coquand and Gérard Huet. A theory of constructions. Presented at the International Symposium on Semantics of Data Types, Sophia-Antipolis, 1984.
22. Antony Courtney and Conal Elliott. Genuinely functional user interfaces. In *Proceedings of the Haskell Workshop*, September 2001.
23. Patrik Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. In *Mathematical Foundations of Programming Semantics*, 1997.
24. Karl Crary and Stephanie Weirich. Resource bound certification. In *the Symposium on Principles of Programming Languages (POPL '00)*, pages 184–198, N.Y., January 19–21 2000. ACM Press.
25. Rowan Davies. A temporal-logic approach to binding-time analysis. In *the Symposium on Logic in Computer Science (LICS '96)*, pages 184–195, New Brunswick, 1996. IEEE Computer Society Press.
26. Rowan Davies and Frank Pfenning. A modal analysis of staged computation. In *the Symposium on Principles of Programming Languages (POPL '96)*, pages 258–270, St. Petersburg Beach, 1996.
27. Roberto Di Cosmo. *Isomorphisms of Types: from λ-calculus to information retrieval and language design*. Progress in Theoretical Computer Science. Birkhäuser, 1995.

28. Gilles Dowek, Amy Felty, Hugo Herbelin, Gérard Huet, Chet Murthy, Catherine Parent, Christine Paulin-Mohring, and Benjamin Werner. The Coq proof assistant user's guide. Rapport Techniques 154, INRIA, Rocquencourt, France, 1993. Version 5.8.
29. Abbas Edalat and Peter John Potts. A new representation for exact real numbers. *Electronical Notes in Theoretical Computer Science*, 6:14 pp., 1997. Mathematical foundations of programming semantics (Pittsburgh, PA, 1997).
30. Conal Elliott and Paul Hudak. Functional reactive animation. In *International Conference on Functional Programming*, pages 163–173, June 1997.
31. John Peterson et. al. Haskell 1.4: A non-strict, purely functional language. Technical Report YALEU/DCS/RR-1106, Department of Computer Science, Yale University, Mar 1997. World Wide Web version at http://haskell.cs.yale.edu/haskell-report.
32. Cormac Flanagan and Martín Abadi. Types for safe locking. In *European Symposium on Programming (ESOP)*, volume 1576 of *Lecture Notes in Computer Science*, pages 91–108. Springer-Verlag, 1999.
33. Robert Glück and Jesper Jørgensen. Fast binding-time analysis for multi-level specialization. In Dines Bjørner, Manfred Broy, and Igor V. Pottosin, editors, *Perspectives of System Informatics*, volume 1181 of *Lecture Notes in Computer Science*, pages 261–272. Springer-Verlag, 1996.
34. Healfdene Goguen. *A Typed Operational Semantics for Type Theory.* PhD thesis, University of Edinburgh, 1994.
35. Andrew D. Gordon. *Functional Programming and Input/Output.* Distinguished Dissertations in Computer Science. Cambridge University Press, September 1994.
36. Carl A. Gunter. *Semantics of Programming Languages.* MIT Press, 1992.
37. David Harel. STATECHARTS: a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.
38. Peter Henderson. Functional programming, formal specification and rapid prototyping. *IEEE Transactions on Software Engineering*, 12(2):241–250, 1986.
39. J. Roger Hindley. *Basic Simple Type Theory*, volume 42 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, Cambridge, 1997.
40. C. A. R. Hoare. Communicating sequential processes. *Comm. ACM*, 21(8):666–677, 1978.
41. C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.
42. Wilfred Hodges. *A Shorter Model Theory.* Cambridge University Press, 1997.
43. Martin Hofmann. Linear types and non-size-increasing polynomial time computation. In *the Symposium on Logic in Computer Science (LICS '99)*, pages 464–473. IEEE, July 1999.
44. Martin Hofmann. Programming languages capturing complexity classes. *SIGACTN: SIGACT News (ACM Special Interest Group on Automata and Computability Theory)*, 31, 2000.
45. Martin Hofmann. A type system for bounded space and functional in-place update. *Nordic Journal of Computing*, 7(4), Winter 2000.
46. Paul Hudak. *The Haskell School of Expression – Learning Functional Programming through Multimedia.* Cambridge University Press, New York, 2000.
47. Paul Hudak and Joe Fasel. A gentle introduction to Haskell. *ACM SIGPLAN Notices*, 27(5), May 1992.
48. R.J.M. Hughes. Why functional programming matters. Technical Report 16, Programming Methodology Group, Chalmers University of Technology, November 1984.

49. R.J.M. Hughes and Lars Pareto. Recursion and dynamic data-structures in bounded space: Towards embedded ML programming. In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming (ICFP-99)*, volume 34.9 of *ACM Sigplan Notices*, pages 70–81, N.Y., September 27–29 1999. ACM Press.
50. R.J.M. Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In Guy L. Steele Jr, editor, *In proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, volume 23, St Petersburg, Florida, 1996. ACM Press.
51. Neil D. Jones. *Computability and Complexity From a Programming Perspective*. Foundations of Computing. The MIT Press, Cambridge, MA, USA, 1997.
52. Neil D Jones and C. K. Gomard. A partial evaluator for the untyped lambda calculus. DIKU report, University of Copenhagen, Copenhagen, Denmark, 1990. Extended version of [53].
53. Neil D. Jones, C. K. Gomard, A. Bondorf, O. Danvy, and T. Mogensen. A self-applicable partial evaluator for the lambda calculus. In *IEEE International Conference on Computer Languages*, pages 49–58, 1990.
54. Richard Kieburtz. Taming effects with monadic typing. In *the International Conference on Functional Programming (ICFP '98)*, volume 34(1) of *ACM SIGPLAN Notices*, pages 51–62. ACM, June 1999.
55. Richard Kieburtz. Real-time reactive programming for embedded controllers. Available from author's home page, March 2001.
56. Richard B. Kieburtz. Implementing closed domain-specific languages. In *[91]*, pages 1–2, 2000.
57. Leslie Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7), 1978.
58. Edward A. Lee. What's ahead for embedded software? In *IEEE Computer*, September 2000.
59. Edward A. Lee. Computing for embedded systems. In *IEEE Instrumentation and Measurement Technology Conference*, Budapest, Hungary, 2001.
60. Xavier Leroy. Objective Caml, 2000. Available from http://caml.inria.fr/ocaml/.
61. Zhaohui Luo and Robert Pollack. The LEGO proof development system: A user's manual. Technical Report ECS-LFCS-92-211, University of Edinburgh, May 1992.
62. John Matthews, Byron Cook, and John Launchbury. Microprocessor specification in Hawk. In *Proceedings of the 1998 International Conference on Computer Languages*, pages 90–101. IEEE Computer Society Press, 1998.
63. Nicholas McKay and Satnam Singh. Dynamic specialization of XC6200 FPGAs by partial evaluation. In Reiner W. Hartenstein and Andres Keevallik, editors, *International Workshop on Field-Programmable Logic and Applications*, volume 1482 of *Lecture Notes in Computer Science*, pages 298–307. Springer-Verlag, 1998.
64. Robin Milner. *A Calculus of Communicating Systems*, volume 81 of *Lecture Notes in Computer Science*. Springer-Verlag, 1981.
65. Robin Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989.
66. Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.
67. John C. Mitchell. *Foundations for Programming Languages*. MIT Press, Cambridge, 1996.
68. Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1), 1991.

69. Andrew Moran and David Sands. Improvement in a lazy context: An operational theory for call-by-need. In *the Symposium on Principles of Programming Languages (POPL '99)*, pages 43–56, San Antonio, Texas, January 1999. ACM.
70. Alan Mycroft and Richard Sharp. A statically allocated parallel functional language. In *Automata, Languages and Programming*, pages 37–48, 2000.
71. Flemming Nielson and Hanne Riis Nielson. Two-level semantics and code generation. *Theoretical Computer Science*, 56(1):59–133, 1988.
72. Hanne Rijs Nielson and Flenning Nielson. *Semantics with Applications : A Formal Introduction*. John Wiley & Sons, Chichester, 1992. Available online from http://www.daimi.au.dk/~bra8130/Wiley_book/wiley.html.
73. Bengt Nordström. The ALF proof editor. In *Proceedings of the Workshop on Types for Proofs and Programs*, pages 253–266, Nijmegen, 1993.
74. Bengt Nordström, Kent Peterson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory: An Introduction*, volume 7. Oxford University Press, New York, NY, 1990.
75. Chris Okasaki. Even higher-order functions for parsing or why would anyone ever want to use a sixth-order function? *Journal of Functional Programming*, 8(2):195–199, March 1998.
76. Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, Cambridge, UK, 1998.
77. Oregon Graduate Institute Technical Reports. P.O. Box 91000, Portland, OR 97291-1000,USA. Available online from ftp://cse.ogi.edu/pub/tech-reports/README.html. Last viewed August 1999.
78. Paul Hudak Simon Peyton Jones, Philip Wadler, Brian Boutel, John Fairbairn, Joseph Fasel, Maria M. Guzman, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Will Partain, and John Peterson. Report on the programming language Haskell. *SIGPLAN Notices*, 27(5):Section R, 1992.
79. Simon Peyton Jones and Philip Wadler. Imperative functional programming. In *the Symposium on Principles of Programming Languages (POPL '93)*. January 1993. 71–84.
80. Frank Pfenning and Carsten Schürmann. System description: Twelf — A metalogical framework for deductive systems. In H. Ganzinger, editor, *the International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag LNAI 1632.
81. Andrew M. Pitts. Operationally-based theories of program equivalence. In P. Dybjer and Andrew M. Pitts, editors, *Semantics and Logics of Computation*. Cambridge University Press, 1995. Based on lectures given at the CLICS-II Summer School on Semantics and Logics of Computation, Isaac Newton Institute for Mathematical Sciences, Cambridge UK, September 1995.
82. Gordon D. Plotkin. A powerdomain construction. *SIAM Journal of Computing*, 5(3):452–487, September 1976.
83. Gordon D. Plotkin. A structural approach to operational semantics. Technical report, Computer Science Department, Aarhus University, 1981.
84. Gordon D. Plotkin. An operational semantics for CSP. Technical report, University of Edinburgh, Department of Computer Science, 1982.
85. Jean-Claude Raoult and Jean Vuillemin. Operational and semantic equivalence between recursive programs. *JACM*, 27(4):772–796, October 1980.

86. Jonathan Rees, William Clinger, H. Abelson, N. I. Adams IV, D. Bartley, G. Brooks, R. K. Dybvig, D. P. Friedman, R. Halstead, C. Hanson, C. T. Haynes, E. Kohlbecker, D. Oxley, K. M. Pitman, G. J. Rozas, G. J. Sussman, and M. Wand. Revised[4] report on the algorithmic language Scheme. Technical Report AI Memo 848b, MIT Press, 1992.
87. David Sands. A naïve time analysis and its theory of cost equivalence. *Journal of Logic and Computation*, 5(4), 1995.
88. Michael B. Smyth. Powerdomains. In *the Mathematical Foundations of Computer Science Symposium*, volume 45 of *Lecture Notes in Computer Science*, pages 537–543. Springer-Verlag, 1976.
89. Joseph E. Stoy. Some mathematical aspects of functional programming. In John Darlington, Peter Henderson, and David A. Turner, editors, *Functional Programming and its Applications*, pages 217–252. Cambridge University Press, 1982.
90. Walid Taha. *Multi-Stage Programming: Its Theory and Applications.* PhD thesis, Oregon Graduate Institute of Science and Technology, 1999. Available from [77].
91. Walid Taha, editor. *Semantics, Applications, and Implementation of Program Generation*, volume 1924 of *Lecture Notes in Computer Science*, Montréal, 2000. Springer-Verlag.
92. Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *Proceedings of the Symposium on Partial Evaluation and Semantic-Based Program Manipulation (PEPM)*, pages 203–217, Amsterdam, 1997. ACM Press.
93. Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. In A. Scedrov, editor, *Proceedings of the 1992 Logics in Computer Science Conference*, pages 162–173. IEEE, 1992.
94. Alastair Telford and David Turner. Ensuring Streams Flow. In Michael Johnson, editor, *Algebraic Methodology and Software Technology, 6th International Conference, AMAST '97, Sydney Australia, December 1997*, volume 1349 of *Lecture Notes in Computer Science*, pages 509–523. AMAST, Springer-Verlag, December 1997.
95. Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1 February 1997.
96. David A. Turner. Functional programs as executable specifications. *Philosophical Transactions of the Royal Society of London*, A312:363–388, 1984.
97. Philip Wadler. The essence of functional programming. In *the Symposium on Principles of Programming Languages (POPL '92)*, pages 1–14. ACM, January 1992.
98. Philip Wadler. The marriage of effects and monads. In *the International Conference on Functional Programming (ICFP '98)*, volume 34(1) of *ACM SIGPLAN Notices*, pages 63–74. ACM, June 1999.
99. Zhanyong Wan and Paul Hudak. Functional reactive programming from first principles. In *the Symposium on Programming Language Design and Implementation (PLDI '00)*. ACM, 2000.
100. Zhanyong Wan, Walid Taha, and Paul Hudak. Real-time FRP. In *the International Conference on Functional Programming (ICFP '01)*, Florence, Italy, September 2001.
101. Hongwei Xi. Upper bounds for standardizations and an application. *Journal of Symbolic Logic*, 64(1):291–303, March 1999.

Rate-Based Resource Allocation Models for Embedded Systems*

Kevin Jeffay[1] and Steve Goddard[2]

[1]Department of Computer Science
University of North Carolina at Chapel Hill
Chapel Hill, NC 27599-3175 USA
jeffay@cs.unc.edu

[2]Computer Science & Engineering
University of Nebraska - Lincoln
Lincoln, NE 68588-0115 USA
goddard@cse.unl.edu

Abstract. Run-time executives and operating system kernels for embedded systems have long relied exclusively on static priority scheduling of tasks to ensure timing constraints and other correctness conditions are met. Static priority scheduling is easy to understand and support but it suffers from a number of significant shortcomings such as the complexity of simultaneously mapping timing and importance constraints onto priority values. Rate-based resource allocation schemes offer an attractive alternative to traditional static priority scheduling as they offer flexibility in specifying and managing timing and criticality constraints. This paper presents a taxonomy of rate-based resource allocation and summarizes the results of some recent experiments evaluating the real-time performance of three allocation schemes for a suite of intra-kernel and application-level scheduling problems encountered is supporting a multimedia workload on FreeBSD UNIX.

1. Introduction

Run-time executives and operating system kernels for embedded systems have long relied on static priority scheduling of tasks to ensure timing constraints and other correctness conditions are met. In static priority scheduling tasks are assigned an integer priority value that remains fixed for the lifetime of the task. Whenever a task is made ready to run (*e.g.*, when the arrival of an interrupt releases a waiting task), the active task with the highest priority commences or resumes execution, preempting the currently executing task if need be. There is rich literature that analyzes static priority scheduling and demonstrates how timing and synchronization constraints can be met using static priority scheduling (see for [14] for a good summary discussion). For these and other reasons virtually all commercial real-time operating systems, including VxWorks [27], VRTX [17], QNX [22], pSOSystem (pSOS) [21], and LynxOS [16], support static priority scheduling.

* This work supported in parts by grants from the National Science Foundation (grants CDA-9624662, ITR-0082870, and ITR-0082866), and the IBM and Intel corporations.

T.A. Henzinger and C.M. Kirsch (Eds.): EMSOFT 2001, LNCS 2211, pp. 204–222, 2001.
© Springer-Verlag Berlin Heidelberg 2001

However, despite the popularity and simplicity of static priority scheduling, the method has significant shortcomings. As discussed in greater detail in Section 2, static priority scheduling suffers from a number of problems including:

- An inability to directly map timing or importance constraints into priority values,
- The problem of dealing with tasks whose execution time is either unknown or may vary over time,
- The problem of dealing with tasks whose execution time or execution rate deviates significantly at run-time from the behavior expected at design-time,
- The problem of degrading system performance gracefully in times of overload, and
- The problem of ensuring full utilization of the processor or other system resources in tightly resource constrained systems.

As a solution to these and other problems, we are investigating the use of rate-based resource allocation methods for real-time and embedded systems. In a rate-based system a task is guaranteed to make progress according to a well-defined rate specification such as "process *x* samples per second," or "process *x* messages per second where each message consists of 3-5 consecutive network packets."

Recently a number of rate-based resource allocation paradigms have been developed and reported in the real-time systems and multimedia computing literature. These include:

- The "constant-bandwidth" abstraction for a server algorithm for executing aperiodic workloads [2, 23, 24],
- The Lottery [28], *SMART* [19], *SFQ* [6], *EEVDF* [25], and *BERT* [3] variants of proportional share real-time resource allocation in UNIX, and
- A series of rate-based extension to the Liu and Layland theory of real-time scheduling [5, 7, 9, 29].

This paper summarizes recent developments in rate-based resource allocation and informally demonstrates how rate-based methods can provide a framework for the natural specification and realization of timing constraints in embedded and real-time systems. To structure the discussion, three dimensions of the basic resource allocation problem are considered:

- *The type of resource allocation problem.* To fully explore the utility of rate-based resource allocation three scheduling problems are considered: application-level scheduling (*i.e.*, scheduling of user programs or tasks/threads), scheduling the execution of system calls made by applications ("top-half" operating system-level scheduling), and scheduling asynchronous events generated by devices ("bottom-half" operating system-level scheduling). This treatment is motivated by the logical structure of traditional, monolithic real-time (and general purpose) operating systems and kernels with hardware enforced protection boundaries. Moreover, this simple taxonomy of scheduling problems emphasizes that in real systems one must consider issues of intra-kernel resource allocation and scheduling as well as application task scheduling.
- *The type or class of rate-based resource allocation method*. We consider three broad classes of rate-based resource allocation paradigms: allocation based on a

fluid-flow paradigm, allocation based on a polling or periodic server paradigm, and allocation based on a generalized Liu and Layland paradigm. For an instance of the fluid-flow paradigm the proportional share scheduling algorithm *earliest eligible virtual deadline first* (*EEVDF*) [25] is considered. For an instance of the polling server paradigm, scheduling based on the *constant bandwidth server* (*CBS*) server concept [2] is considered. Finally, for the generalized Liu and Layland paradigm a rate-based extension to the original Liu and Layland task model called *rate-based execution* (*RBE*) [9] is considered.

- *The characteristics of the workload generated.* For each rate-based allocation scheme above, the likely expected real-time performance of an application is considered under a set of execution environments where the applications execute at various rates. Specifically, cases are considered wherein the applications execute at "well-behaved," constant rates, at bursty rates, and at uncontrolled "misbehaved" rates. For well-behaved workloads, the three rate-based schemes considered (and indeed virtually any real-time scheduling scheme) can execute a given workload in real-time. However, the rate-based schemes perform quite differently when applications need to be scheduled at bursty or uncontrolled rates.

The goal is demonstrate how rate-based methods naturally solve a variety of resource allocation problems that traditional static priority scheduling methods are inherently poorly suited to. However, it will also be argued that "one size does not fit all." One rate-based resource allocation scheme does not suffice for all scheduling problems that arise within the layers of a real system. While one can construct an execution environment wherein all of the rate-based schemes considered here perform well, for more realistic environments that are likely to be encountered in practice, the best results are likely to be achieved by employing different rate-based allocation schemes at different levels in the operating system.

To be sure, rate-based resource allocation methods are not a panacea. There are other scheduling architectures based on extensions to (or indirect uses of) static priority scheduling that can also provide effective solutions for many of the problems considered herein (see [14]), however, because of space constraints, in this paper only rate-based methods are considered.

The following section describes the shortcomings of static priority scheduling in more detail. Section 3 presents a taxonomy of rate-based resource allocation methods. Three classes of rate-based resource allocation are described and the strengths and weaknesses of each approach are discussed. The results of some recent experiments performed to evaluate specific instances of rate-based schedulers are reviewed in Section 4. Section 5 proposes and evaluates a hybrid scheme that combines different forms of rate-based resource allocation within different layers of the operating system and application. Section 6 summaries the results and concludes with a discussion of directions for further study in rate-based resource allocation.

2. Traditional Static Priority Scheduling

Traditional models of real-time resource allocation are based on the concept of a discrete but recurring event, such as a periodic timer interrupt, that causes the release of task. The task must be scheduled such that it completes execution before a well-

defined deadline. For example, most real-time models of execution are based on the Liu and Layland periodic task model [15] or Mok's sporadic task model [18]. In these models each execution of a task must complete before the next instance of the same task is released. The challenge is to assign priority values to tasks such that all releases of all tasks are guaranteed to complete execution before their respective deadlines when scheduled by a preemptive priority scheduler. Common priority assignment schemes include the *rate-monotonic* scheme [15] wherein tasks that recur at high rates have priority over tasks that recur at low rates (*i.e.*, a task's priority is equal to its recurrence period), and the *deadline monotonic* scheme [13] wherein a task's priority is related to its response time requirement (it's deadline).[1] In either case, static assignment of priority values to tasks leads to a number of problems.

2.1. Mapping Performance Requirements to Priorities

Simple timing constraints for tasks that are released in response a single event, such as a response time requirement for a specific interrupt, can be easily specified in a static priority system. More complex constraints are frequently quite difficult to map into priority values. For example, consider a signal processing system operating on video frames that arrive over a LAN from a remote acquisition device. The performance requirement may be to process 30 frames/second and hence it would be natural to model the processing as a periodic task and trivial to schedule using a static priority scheduler. However, it is easily the case that each video frame arrives at the processing node in a series of network packets that must be reassembled into a video frame and it is not at all obvious how the network packet and protocol processing should be scheduled. That is, while there is a natural constraint for the application-level processing, there is no natural constraint (*e.g.*, no unique response time requirement) for the processing of the individual events that will occur during the process of realizing the higher-level constraint. Nonetheless in a static priority scheduler the processing of these events must be assigned a single priority value.

The problem here is that the system designer implicitly creates a response time requirement when assigning a priority value to the network processing. Since there is no natural unique value for this requirement a conservative (*e.g.*, short response time) value is typically chosen. This conservative assignment of priority has the effect of reserving more processing resources for the task than will actually ever be consumed and ultimately limits the either the number or the complexity of tasks that can be executed on the processor.

2.2. Managing "Misbehaved" Tasks

In order to analyze any resource allocation problems, assumptions must be made about the environment in which the task executes. In particular, in virtually all real-time problems the amount of resources required for the execution of the task (*e.g.*, the amount of processor time required to execute the task) is assumed to be known in advance. A second problem with static priority resource allocation occurs when assumptions such as these are violated and a task "misbehaves" by consuming more resources than expected. The problem is to ensure that a misbehaving task does not compromise the execution of other "well-behaved" tasks in the system.

---

[1] We assume throughout that low priority values indicate high scheduling priority.

An often-touted advantage of static priority systems is that if a task violates its execution assumptions, higher priority tasks are not affected. While it is true that higher priority tasks are not affected by misbehaving lower priority tasks (unless higher priority tasks share software resources with the lower priority tasks), all other tasks have no protection from the misbehaving task. For example, a task that is released at a higher rate than expected can potentially block *all* lower priority tasks indefinitely.

The issue is that static priority scheduling fundamentally provides no mechanism for isolating tasks from the ill-effects of other tasks other than the same mechanism that is used to ensure response time properties. Given that isolation concerns typically are driven by the relative importance of tasks (*e.g.*, a non-critical task should never effect or interfere with the execution of a mission-critical task), and importance and response time are often independent concepts, attempting to manage both concerns with a single mechanism is inappropriate.

2.3. Providing Graceful/Uniform Degradation

Related to the task isolation problem is that of providing graceful performance degradation under transient (or persistent) overload conditions. The problem of graceful degradation can be considered a generalization of the isolation problem; a set of tasks (or the environment that generates work for the tasks) misbehaves and the processing requirements for the system as a whole increase to the point where tasks miss deadlines. In these overload situations it is again useful to control which tasks' performance degrades and by how much.

Static priority scheduling again has only a single mechanism for managing importance and response time. If the mission-critical tasks also have the smallest response-time requirements then they will have the highest priority and will continue to function. However, if this is not the case then there is no separate mechanism to control the execution of important tasks. Worse, even if the priority structure matches the importance structure, in overload conditions under static priority scheduling only one form of degraded performance is possible: high priority tasks execute normally while the lowest priority task executes at a diminished rate if at all. That is, since a task with priority p will *always* execute to completion before any pending task with priority less than p commences or resumes execution, it is impossible to control how tasks degrade their execution. (Note however, as frequently claimed by advocates of static priority scheduling, the performance of a system under overload conditions is predictable.)

2.4. Achieving Full Resource Utilization

The rate-monotonic priority assignment scheme is well known to be an optimal static priority assignment. (Optimal in the sense that if a static priority assignment exists which guarantees periodic tasks have a response time no greater than their period, then the rate-monotonic priority assignment will also provide the same guarantees.) One drawback to static priority scheduling, however, is that the achievable processor utilization is restricted. In their seminal paper, Liu and Layland showed that a set of n periodic tasks will be schedulable under a rate-monotonic priority assignment if the processor utilization of a task set does not exceed $n(2^{1/n} - 1)$ [15]. If the utilization of a task set exceeds this bound then the tasks may or may not be schedulable. (That is, this condition is a sufficient but not necessary condition for schedulability). Moreover, Liu and Layland showed that in the limit the utilization bound approached $ln\ 2$ or

approximately 0.69. Thus 69% is the least upper bound on the processor utilization that guarantees feasibility. A least upper bound here means that this is the minimum utilization of all task sets that fully utilize the processor. (A task set fully utilizes the processor if increasing the cost of any task in the set causes a task to miss a deadline.) If the utilization of the processor by a task set is less than or equal to 69% then the tasks are guaranteed to be schedulable.

Lehoczky, Sha, and Ding formulated an exact text for schedulabilty under a rate-monotonic priority assignment and showed that ultimately, schedulability is not a function of processor utilization [12]. However, nonetheless, in practice the utilization test remains the dominant test for scheduability as it is both a simple and an intuitive test. Given this, a resource allocation scheme wherein scheduability was more closely tied to processor utilization (or a similarly intuitive metric) would be highly desirable.

3. A Taxonomy of Rate-Based Resource Allocation Models

The genesis of rate-based resource allocation schemes can be traced to the problem of supporting multimedia computing and other soft-real-time problems. In this arena it was observed that while one could support the needs of these applications with traditional real-time scheduling models, these models were not the most natural ones to apply [6, 8, 11, 28]. Whereas Liu and Layland models typically dealt with response time guarantees for the processing of periodic/sporadic events, the requirements of multimedia applications were better modeled as aggregate, but bounded, processing rates.

From our perspective three classes of rate-based resource allocation models have evolved: *fluid-flow allocation*, *server-based allocation*, and *generalized Liu and Layland style allocation*. Fluid-flow allocation derives largely from the work on fair-share bandwidth allocation in the networking community. Algorithms such as *generalized processor sharing* (GPS) [20], *packet-by-packet generalized processor sharing* (PGPS) [20] (better known as *weighted fair queuing* (WFQ) [4]), were concerned with allocating network bandwidth to connections ("flows") such that for a particular definition of fairness, all connections continuously receive their fair share of the bandwidth. Since connections were assumed to be continually generating packets, fairness was expressed in terms of a guaranteed transmission rate (*i.e.*, some number of bits per second). These allocation policies were labeled as "fluid flow" allocation because since transmission capacity was continuously available to be allocated, analogies were drawn between conceptually allowing multiple connections to transmit packets on a link and allowing multiple "streams of fluid" to flow through a "pipe."

These algorithms stimulated tremendous activity in both real-time CPU and network link scheduling. In the CPU scheduling realm numerous algorithms were developed, differing largely in the definition and realization of "fair allocation" [19, 25, 28]. Although fair/fluid allocation is in principle a distinct concept from real-time allocation, it is a powerful building block for realizing real-time services [26].

Server-based allocation derives from the problem of scheduling aperiodic tasks in a real-time system. The salient abstraction is that a "server process" is invoked periodically to service any requests for work that have arrived since the previous invocation of the server. The server typically has a "capacity" for servicing requests (usually expressed in units of CPU execution time) in any given invocation. Once this capacity is exhausted, any in-progress work is suspended until at least the next server

invocation time. Numerous server algorithms have appeared in the literature; differing largely in the manner in which the server is invoked and how its capacity is allocated [2, 23, 24]. Server algorithms are considered to be rate-based forms of allocation as the execution of a server is not (in general) directly coupled with the arrival of a task. Moreover, server-based allocation has the effect of ensuring aperiodic processing progresses at a well defined, uniform rate.

Finally, rate-based generalizations of the original Liu and Layland periodic task model have been developed to allow more flexibility in how a scheduler responds to events that arrive at a uniform average rate but unconstrained instantaneous rate. Representative examples here include the (m, k) allocation models that requires only m out of every k events be processed in real-time [5], the *window-based allocation* (DWYQ) method that ensures a minimum number of events are processed in real-time within sliding time windows [29], and the *rate-based execution* (RBE) algorithm that "reshapes" the deadlines of events that arrive at a higher than expected rate to be those that the events would have had had they arrived at a uniform rate [9].

To illustrate the utility of rate-based resource allocation, one algorithm from the literature from each class of rate-based allocation methods will be discussed in more detail. The choice is motivated by the prior work of the authors, specifically our experience implementing and using these algorithms in production systems [7, 10]. For an instance of the fluid-flow paradigm the proportional share scheduling algorithm *earliest eligible virtual deadline first* (*EEVDF*) [25] will be discussed. For an instance of the polling server paradigm the *constant bandwidth server* (*CBS*) server concept [2] will be discussed. For the generalized Liu and Layland paradigm the *rate-based execution* (*RBE*) model [9] will be discussed. Although specific algorithms are chosen for discussion, ultimately the results presented are believed to hold for each algorithm in the same class as the algorithm discussed.

3.1. Fluid-Flow Resource Allocation: Proportional Share Scheduling

In a proportional share (*PS*) system each shared resource r is allocated in discrete quanta of size at most q_r. At the beginning of each time quantum a task is selected to use the resource. Once the task acquires the resource, it may use the resource for the entire time quantum, or it may release the resource before the time quantum expires. For a given resource, a *weight* is associated with each task that determines the relative *share* of the resource that the task should receive. Let w_i denote the weight of task i, and let $A(t)$ be the set of all tasks active at time t. Define the (instantaneous) share $f_i(t)$ of task i at time t as

$$f_i(t) = \frac{w_i}{\sum_{j \in A(t)} w_j} \tag{1}$$

A share represents a fraction of the resource's capacity that is "reserved" for a task. If the resource can be allocated in arbitrarily small sized quanta, and if the task's share remains constant during any time interval $[t_1, t_2]$, then the task is entitled to use the resource for $(t_2 - t_1)f_i(t)$ time units in the interval. As tasks are created/destroyed or blocked/released, the membership of $A(t)$ changes and hence the denominator in (1) changes. Thus in practice, a task's share of a given resource will change over time. As the total weight of tasks in the system increases, each task's share of the resource decreases. As the total weight of tasks in the system decreases, each task's share of the

resource increases. When a task's share varies over time, the service time S that task i should receive in any interval $[t_1, t_2]$, is

$$S_i(t_1, t_2) = \int_{t_1}^{t_2} f_i(t)\, dt \tag{2}$$

time units.

Equations (1) and (2) correspond to an ideal "fluid-flow" system in which the resource can be allocated for arbitrarily small units of time. In such a system tasks make progress at a uniform rate as if they were executing on a dedicated processor with a capacity that is $f_i(t)$ that of the actual processor. In practice one can implement only a discrete approximation to the fluid system. When the resource is allocated in discrete time quanta it is not possible for a task to always receive exactly the service time it is entitled to in all time intervals. The difference between the service time that a task should receive at a time t, and the time it actually receives is called the service time lag (or simply lag). Let t_0^i be the time at which task i becomes active, and let $s(t_0^i, t)$ be the service time task i receives in the interval $[t_0^i, t]$. Then if task i is active in the interval $[t_0^i, t]$, its lag at time t is defined as

$$lag_i(t) = S_i(t_0^i, t) - s_i(t_0^i, t). \tag{3}$$

Since the lag quantifies the allocation accuracy, it is used as the primary metric for evaluating the performance of *PS* scheduling algorithms. Previously we have shown that one can schedule a set of tasks in a *PS* system using a "virtual time" *earliest deadline first* rule such that the lag is bounded by a constant over all time intervals [25]. By using this algorithm, called *earliest eligible virtual deadline first* (*EEVDF*), a *PS* system's deviation from a system with perfectly uniform allocation is bounded and thus, as explained below, real-time execution is possible.

Scheduling to Minimize Lag

The goal in proportional share scheduling is to minimize the maximum possible lag. This is done by conceptually tracking the lag of tasks and at the end of each quantum, considering only tasks whose lag is positive [25]. If a task's lag is positive then it is "behind schedule" compared to the perfect fluid system — it should have accumulated more time on the CPU than it has up to the current time. If a task's lag is positive it is considered eligible to execute. If its lag is negative, then the task has received more processor time than it should have up to the current time and it is considered ineligible to execute

When multiple tasks are eligible, in *EEVDF* they are scheduled earliest deadline first, where a task's deadline is equal to its estimated execution time cost divided by its share of the CPU $f_i(t)$. This deadline represents a point in the future when the task should complete execution if it receives exactly its share of the CPU. For example, if a task's weight is such that its share of the CPU at the current time is 10% and it requires 2 *ms* of CPU time to complete execution, then its deadline will be 20 *ms* in the future. If the task actually receives 10% of the CPU, over the next 20 *ms* it will execute for 2 *ms*.

Proportional share allocation is realized through a form of weighted round-robin scheduling wherein in each round the task with the earliest deadline is selected. In [25] it was shown that the *EEVDF* algorithm provides optimal (*i.e.*, minimum possible) lag bounds.

Realizing Real-Time Execution

In principle, there is nothing "real-time" about proportional share resource allocation. Proportional share resource allocation is concerned solely with *uniform* allocation (often referred to in the literature as "fluid" or "fair" allocation). A *PS* scheduler achieves uniform allocation if it can guarantee that tasks' lags are always bounded.

Real-time computing is achieved in a *PS* system by (*i*) ensuring that a task's share of the CPU (and other required resources) remains constant over time, and by (*ii*) scheduling tasks such that each task's lag is always bounded by a constant. If these two conditions hold over an interval of length t for a task i, then task i is guaranteed to receive $(f_i \times t) \pm \varepsilon$ units of the resource's capacity, where f_i is the fraction of the resource reserved for task i, and ε is the allocation error, $0 \bullet \varepsilon \bullet \delta$, for some constant δ (for *EEVDF* δ = the quantum size q) [25]. Thus, although real-time allocation is possible, it is not possible to provide hard and fast guarantees of adherence to application-defined timing constraints. Said another way, all guarantees have an implicit, and fundamental, "$\pm \varepsilon$" term. In FreeBSD-based implementations of *EEVDF*, ε has been a tunable parameter and was most commonly set to 1 *ms* [7].

The deadline-based *EEVDF* proportional share scheduling algorithm ensures that each task's lag is bounded by a constant [25] (condition (*i*)). To ensure a task's share remains constant over time (condition (*ii*)), whenever the total weight in the system changes, a "real-time" task's weight must be adjusted so that its initial share (as given by equation (1)) does not change. For example, if the total weight in the system increases (*e.g.*, because new tasks are created), then a real-time task's weight must increase by a proportional amount. Adjusting the weight to maintain a constant share is simply a matter of solving equation (1) for w_i when $f_i(t)$ is a constant function. (Note that w_i appears in both the numerator and denominator of the right-hand side of (1).) If the sum of the processor shares of the real-time tasks is less than 1.0 then all tasks will execute in real-time (*i.e.*, under *EEVDF* real-time scheduablity is a simple function of processor utilization).

3.2. Liu and Layland Extensions: Rate-Based Execution (*RBE*)

The traditional Liu and Layland model of periodic real-time execution has been extended in a number of directions to be more flexible in way in which real-time requirements were modeled and realized. For example, all of the traditional theory assumes a minimum separation in time between releases of instances of the same task. This requirement does not map well in actual systems that, for example, receive inputs over a network. For example, in a video processing application, video frames may be transmitted across an internetwork at precise intervals but arrive at a receiver with arbitrary spacing in time because of the store-and-forward nature of most network switches. Although there is explicit structure in this problem (frames are generated at a precise rate), there is no way to capture this structure in a Liu and Layland model.

The *RBE* paradigm is one extension to the Liu and Layland model to address this problem. In *RBE*, each task is associated with three parameters (x, y, d) which define a rate specification. In an *RBE* system, each task is guaranteed to process at least x events every y time units, and each event j will be processed before a relative deadline d. The actual deadline for processing of the j^{th} event for task i is given by the following recurrence. If t_{ij} is the time of the arrival of the j^{th} event, then the instance of task i servicing this event will complete execution before time:

$$D_i(j) = \begin{cases} t_{ij} + d_i & \text{if } 1 \le j \le x_i \\ \max(t_{ij} + d_i, D_i(j - x_i) + y_i) & \text{if } j > x_i \end{cases} \quad (4)$$

The deadline for the processing of an event is the larger of the release time of the job plus its relative deadline, or the deadline of the x^{th} previous job plus the y parameter (the averaging interval) of the task. This deadline assignment function confers two important properties on *RBE* tasks. First, up to x consecutive jobs of a task may contend for the processor with the same deadline and second, the deadlines for processing events j and $j+x$ for task i are separated by at least y time units. Without the latter restriction, if a set of events for a task arrive simultaneously it would be possible to saturate the processor. However, with the restriction, the time at which a task must complete its execution is not wholly dependent on its release time. This is done to bound processor demand. Under this deadline assignment function, requests for tasks that arrive at a faster rate than x arrivals every y time units have their deadlines postponed until the time they would have been assigned had they actually arrived at the rate of exactly x arrivals every y time units [9].

The *RBE* task model derives from the *linear bounded arrival process* (*LBAP*) model as defined and used in the DASH system [1]. In the *LBAP* model, tasks specify a desired execution rate as the number of messages to be processed per second, and the size of a buffer pool used to store bursts of messages that arrive for the task. The *RBE* model generalizes the *LBAP* model to include a more generic specification of rate and adds an independent response time (relative deadline) parameter to enable more precise real-time control of task executions.

RBE tasks can be scheduled by a simple *earliest-deadline-first* rule so long as the combined processor utilization of all tasks does not saturate the processor. (Hence there are no undue limits on the achievable processor utilization.) Although nothing in the statement of the *RBE* model precludes static priority scheduling of tasks, it turns out that the *RBE* model points out a fundamental distinction between deadline-based scheduling methods and static priority based methods. Analysis of the *RBE* model has shown that under deadline-based scheduling, feasibility is solely a function of the distribution of task deadlines in time and is independent of the rate at which tasks are invoked. In contrast, the opposite is true of static priority schedulers. For *any* static priority scheduler, feasibility is a function of the rate at which tasks are invoked and is independent of the deadlines of the tasks [9]. Said more simply, the feasibility of static priority schedulers is solely a function of the periodicity of tasks, while the feasibility of deadline schedulers is solely a function of the periodicity of the occurrence of a task's deadlines. Given that it is often the operating system or application that assigns deadlines to tasks, this means that the feasibility of a static priority scheduler is a function of the behavior of the external environment (*i.e.* arrival processes), while the feasibility of a deadline driven scheduler is a function of the implementation of the operating system/application. This is a significant observation as one typically has more control over the implementation of the operating system than they do over the processes external to the system that generate work for the system. For this reason deadline based scheduling methods have a significant and fundamental advantage over static priority based methods when there is uncertainty in the rates at which work is generated for a real-time system, such as is the case in virtually all distributed real-time systems.

3.3. Server-Based Allocation: The Constant Bandwidth Server (*CBS*)

The final class of rate-based resource allocation algorithms are server algorithms. At a high-level, the *CBS* algorithm combines aspects of both *EEVDF* and *RBE* scheduling (although it was developed independently of both works). Like *RBE* it is an event based scheduler, however, like *EEVDF* it has a notion of a quantum. Like both, it achieves rate-based allocation by a form of deadline scheduling

In *CBS*, and its related cousin the *total bandwidth server* (*TBS*) [23, 24], a portion of the processor's capacity, denoted U_S, is reserved for processing aperiodic requests of a task. When an aperiodic request arrives it is assigned a deadline and scheduled according to the *earliest-deadline- first* algorithm. However, while the server executes, its capacity linearly decreases. If the server's capacity for executing a single request is exhausted before the request finishes, the request is suspended until the next time the server is invoked.

A server is parameterized by two additional parameters C_S and T_S, where C_S is the execution time available for processing requests in any single server invocation and T_S is the inter-invocation period of the server ($U_S = C_S/T_S$). Effectively, if the k^{th} aperiodic request arrives at time t_k, it will execute as a task with a deadline

$$d_k = \max(t_k, d_{k-1}) + c_k/U_S \tag{5}$$

where c_k is the worst case execution time of the k^{th} aperiodic request, d_{k-1} is the deadline of the previous request from this task, and U_S is the processor capacity allocated to the server for this task.

CBS resource allocation is considered a rate-based scheme because deadlines are assigned to aperiodic requests based on the rate at which the server can serve them and not (for example) on the rate at which they are expected to arrive. Note that like *EEVDF* and *RBE*, scheduability in *CBS* is solely a function of processor utilization. Any real-time task set that does not saturate the processor is guaranteed to execute in real-time under any of these three algorithms.

4. Using Rate-Based Scheduling

To see how rate-based resource allocation methods can be used to realize real-time constraints and overcome the shortcoming of static priority scheduling, the three algorithms above were used to solve various resource allocation problems that arose in FreeBSD UNIX when executing a set of interactive multimedia applications. The details of this study are reported in [10]. The results are summarized here.

4.1. A Sample Real-Time Workload

To compare the rate-based schedulers, three simple multimedia applications were considered. These applications were chosen because they illustrate many of the essential resource allocation problems found in distributed real-time and embedded applications. The applications are:

- An Internet telephone application that handles incoming 100 byte audio messages at a rate of 50/second and computes for 1 millisecond on each message (requiring 5% of the CPU on average),

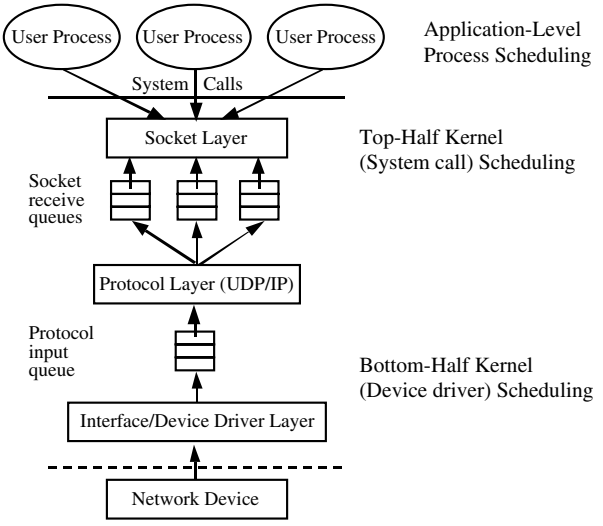


Fig. 1. Architectural diagram of UDP/IP protocol processing in FreeBSD.

- A motion-JPEG video player that handles incoming 1,470 byte messages at a rate of 90/second and computes for 5 milliseconds on each message (requiring 45% of the CPU on average), and
- A file transfer program that handles incoming 1,470 byte messages at a rate of 200/second and computes for 1 millisecond on each message (requiring 20% of the CPU on average).

The performance of different rate-based resource allocation schemes was considered under varying workload conditions. The goal was to evaluate how each rate-based allocation scheme performed when the rates of tasks to be scheduled varied from constant (uniform), to "bursty" (erratic instantaneous rate but constant average rate), to "mis-behaved" (long-term deviation from average expected processing rate).

4.2. Rate-Based Scheduling of Operating System Layers

Our experiments focused on the problem of processing inbound network packets and scheduling user applications to consume these packets. Figure 1 illustrates the high-level architecture of the FreeBSD kernel. Briefly, in FreeBSD, packet processing occurs as follows. (For a more complete description of these functions see [30].) When packets arrive from the network, interrupts from the network interface card are serviced by a device-specific interrupt handler. The device driver copies data from buffers on the adapter card into a chain of fixed-size kernel memory buffers sufficient to hold the entire packet. This chain of buffers is passed on a procedure call to a general interface input routine for a class of input devices. This procedure determines which network protocol should receive the packet and enqueues the packet on that protocol's input queue. It then posts a software interrupt that will cause the protocol layer to be executed when no higher priority hardware or software activities are pending.

Processing by the protocol layer occurs asynchronously with respect to the device driver processing. When the software interrupt posted by the device driver is

serviced, a processing loop commences wherein on each iteration the buffer chain at the head of the input queue is removed and processed by the appropriate routines for the transport protocol. This results in the buffer chain enqueued on the receive queue for the destination socket. If any process is blocked in a kernel system call awaiting input on the socket, it is unblocked and rescheduled. Normally, software interrupt processing returns when no more buffers remain on the protocol input queue.

The kernel socket layer code executes when an application task invokes some form of receive system call on a socket descriptor. When data exists on the appropriate socket queue, the data is copied into the receiving task's local buffers from the buffer chain(s) at the head of that socket's receive queue. When there is sufficient data on the socket receive queue to satisfy the current request, the kernel completes the system call and returns to the application task.

The problem of processing inbound network packets was chosen for study because it involves a range of resource allocation problems at different layers in the operating system. Specifically, there are three distinct scheduling problems: scheduling of device drivers and network protocol processing within the operating system kernel, scheduling system calls made by applications to read and write data to and from the network, and finally the scheduling of user applications. These are distinct problems because the schedulable work is invoked in different ways in different layers. Asynchronous events cause device drivers and user applications to be scheduled but synchronous events cause system calls to be scheduled. Systems calls are, in essence, extensions of the application's thread of control into the operating system. Moreover, these problems are of interest because of the varying amount of information that is available to make real-time scheduling decisions at each level of the operating system. At the application and system call-level it is known exactly which real-time entity should be "charged" for use of system resources while at the device driver-level one cannot know which entity to charge. For example, in the case of inbound packet processing, it cannot be determined which application to charge for the processing of a packet until the packet is actually processed and the destination application is discovered. On the other hand, the cost of device processing can be known exactly as device drivers typically perform simple, bounded-time functions (such as placing a string of buffers representing a packet on a queue). This is in contrast to the application-level where often one can only estimate the time required for an application to complete.

The challenge is to allocate resources throughout the operating system so that end-to-end system performance measures (*i.e.*, network interface to application performance) can be ensured.

4.3. Workload Performance under Proportional Share Allocation

A version of FreeBSD was constructed that used *EEVDF* proportional share scheduling (with a 1 *ms* quantum) at the device, protocol processing, and the application layers. In this system each real-time task is assigned a share of the CPU equal to its expected utilization of the processor (*e.g.*, the Internet phone application requires 5% of the CPU and hence is assigned a weight of 0.05). Non-real-time tasks are assigned a weight equal to the unreserved capacity of the CPU. Network protocol processing is treated as a real-time (kernel-level) task that processes a fixed number of packets when it is scheduled for execution.

In the well-behaved senders case all packets are moved from the network interface to the socket layer to the application and processed in real-time. When the file transfer sender misbehaves and sends more packets than the *ftp* receiver can process given its CPU reservation, *EEVDF* does a good job of isolating the other well-behaved processes from the ill-effects of *ftp*. Specifically, the excess *ftp* packets are dropped at the lowest level of the kernel (at the IP layer) before any significant processing is performed on these packets. These packets are dropped because the kernel task associated with their processing is simply not scheduled often enough (by design) to move the packets up to the socket layer. In a static priority system (such as the unmodified FreeBSD system), network interface processing is the second highest priority task. In this system, valuable time would be "wasted" processing packets up through the kernel only to have them later discarded (because of buffer overflow) because the application wasn't scheduled often enough to consume them.

The cost of this isolation comes in the form of increased packet processing overhead as now the device layer must spend time demultiplexing packets (typically a higher-layer function) to determine if they should be dropped. Performance is also poorer when data arrives for all applications in a bursty manner. This is an artifact of the quantum-based allocation nature of *EEVDF*. Over short intervals, data arrives faster than it can be serviced at the IP layer and the IP interface queue overflows. With a 1 *ms* quantum, it is possible that IP processing can be delayed for upwards of 8-10 *ms* and this is sufficient time for the queue to overflow in a bursty environment. This problem could be ameliorated to some extent by increasing the length of the IP queue, however, this would also have the effect of increasing the response time for packet processing.

4.4. Workload Performance under Generalized Liu and Layland Allocation

When *RBE* scheduling was used for processing at the device, protocol, and application layers, each task had a simple rate specification of (1, *p*, *p*) (*i.e.*, one event will be processed every *p* time units with a deadline of *p*) where *p* is the period of the corresponding application or kernel function.

Perfect real-time performance is realized in the well-behaved and bursty arrivals cases but performance is significantly poorer than *EEVDF* in the case of the misbehaved file transfer application. On the one hand, RBE provides good isolation between the file transfer and the other real-time applications, however, this isolation comes at the expense of the performance of non-real-time/background applications. Unlike *EEVDF*, as an algorithm, *RBE* has no mechanism for directly ensuring isolation between tasks as there is no mechanism for limiting the amount of CPU time an *RBE* task consumes. All events in an *RBE* system are assigned deadlines for processing. When the work arrives at a faster rate than is expected, the deadlines for the work are simply pushed further and further out in time. Assuming sufficient buffering, all will eventually be processed.

In the FreeBSD system, packets are enqueued at the socket layer but with deadlines that are so large that processing is delayed such that the socket queue quickly fills and overflows. Because time is spent processing packets up to the socket layer that are never consumed by the application, the performance of non-real-time applications suffers. Because of this, had the real-time workload consumed a larger

cumulative fraction of the CPU, *RBE* would not have isolated the well-behaved and misbehaved real-time applications. (That is, the fact that isolation was observed in these experiment is an artifact of the specific real-time workload.)

Nonetheless, because the *RBE* scheduler assigns deadline to all packets, and because the system under study was not overloaded, *RBE* scheduling results in the smaller response times for real-time events than seen under *EEVDF* (at the cost non-real-time task performance).

4.5. Workload Performance under Server Based Allocation

When *CBS* server tasks were used for processing throughout the operating system and at the application layers, each server task was assigned a capacity equal to its application's/function's CPU utilization. The server's period was made equal to the expected interarrival time of data (packets). Non-real-time tasks were again assigned to a server with capacity equal to the unreserved capacity of the CPU.

As expected, performance is good when work arrives in a well-behaved manner. In the case of the misbehaved file transfer, *CBS* also does a good job of isolating the other well-behaved tasks. The excess *ftp* packets are dropped at the IP layer and thus little overhead is incurred. In the case of bursty senders *CBS* scheduling outperforms *EEVDF* scheduling. This is because like *RBE*, scheduling of *CBS* tasks is largely event driven and hence *CBS* tasks respond quicker to the arrival of packets. Under *EEVDF* the rate at which the IP queue can be serviced is a function of the quantum size and the number of processes currently active (which determines the length of a scheduling "round" [4]). In general these parameters are not directly related to the real-time requirements of applications. Under *CBS* the service rate is a function of the server's period which is a function of the expected arrival rate and thus is a parameter that is directly related to application requirements. For the choices of quantum size for *EEVDF,* and server period for *CBS*, good performance under *CBS* and poor performance under *EEVDF* results. However, we conjecture that is likely the case that these parameters could be tuned to reverse this result.

Although *CBS* outperforms *EEVDF* in terms of throughput, the results are mixed for response times for real-time tasks. Even when senders are well behaved some deadlines are missed under *CBS*. This results in a significant number of packets being processed later than with *EEVDF* or *RBE* scheduling. This is problematic since in these experiments the theory predicts that no deadlines should be missed. The cause of the problem here relates to the problem of accounting for the CPU time consumed when a *CBS* task executes. In the implementation of *CBS*, the capacity of a *CBS* task is updated only when the task sleeps or is awaken by the kernel. Because of this, many other kernel related functions that interrupt servers (*e.g.,* Ethernet driver execution) are inappropriately charged to *CBS* tasks and hence bias scheduling decisions. This accounting problem is fundamental to the server-based approach and cannot be completely solved without significant additional mechanism (and overhead).

5. Hybrid Rate-Based Scheduling

The results of applying a single rate-based resource allocation policy to the problems of device, protocol, and application processing were mixed. When processing occurs at rates that match the underlying scheduling model (*e.g.*, when work arrives at a constant rate), all the policies considered achieved real-time performance. When work

arrives for an application that exceeds the application's rate specification or resource reservation, then only the *CBS* server-based scheme and the *EEVDF* proportional share scheme provide true isolation between well-behaved and misbehaved applications. When work arrives in a bursty manner, the quantum-based nature of *EEVDF* leads to less responsive protocol processing and more (unnecessary) packet loss. *CBS* performs better but suffers from the complexity of the CPU-time accounting problem that must be solved. *RBE* provides the best response times but only at the expense of decreased throughput for the non-real-time activities. The obvious question is whether or not there is utility in applying different rate-based resource allocation schemes in different layers of the kernel to better match the solution to a particular resource allocation problem to the characteristics of the problem.

To test this conjecture two hybrid rate-based FreeBSD systems were constructed. For application and system call level processing *EEVDF* scheduling was used. This choice was made because the quantum nature of *EEVDF*, while bad for intra-kernel resource allocation, is a good fit given the existing round-robin scheduling architecture in FreeBSD (and many other operating systems such as Linux). It is easy to implement and to precisely control and gives good real-time response when schedulable entities execute for long periods relative to the size of a quantum. For device and protocol processing inside the kernel both *CBS* and *RBE* scheduling were considered. Since the lower kernel layers operate more as an event driven system, a paradigm which takes into account the notion of event arrivals is appropriate. Both of these policies are also well-suited for resource allocation within the kernel because, in the case of *CBS*, it is easier to control the levels and degrees of preemption within the kernel and hence it is easier to account for CPU usage within the kernel (and hence easier to realize the results predicted by the *CBS* theory). In the case of *RBE*, processing within the kernel is more deterministic and hence *RBE*'s inherent inability to provide isolation between tasks that require more computation than they reserved is less of a factor.

The forms of hybrid rate-based resource allocation described here remains the topic of active study. Preliminary results show that when work arrives at well-behaved rates both *CBS+EEVDF* and *RBE+EEVDF* systems perform flawlessly. (Thus hybrid allocation performs no worse than the universal application of a single rate-based scheme throughput the system.) In the misbehaved *ftp* application case, both hybrid implementations provide good isolation, comparable to the best single-policy systems. However, in both hybrid approaches, response times and deadline miss ratios are now much improved. In the case of bursty arrivals, all packets are eventually processed and although many deadlines are missed, both hybrid schemes miss fewer deadlines than did the single-policy systems. Overall the *RBE+EEVDF* system produces the lowest overall deadline miss ratios. While we do not necessarily believe this is a fundamental result (*i.e.*, there are numerous implementation details to consider), it is the case that the polling nature of the *CBS* server tasks increases response times over the direct event scheduling method of *RBE*.

6. Summary, Conclusions, and Future Work

Rate-based resource allocation schemes are a good fit for providing real-time services in distributed real-time and embedded systems. Allocation schemes exist that are a good fit for the scheduling architectures used in the various layers of a traditional monolithic UNIX kernel such as FreeBSD. Three such rate-based schemes were

considered: the *earliest eligible virtual deadline first* (*EEVDF*) fluid-flow paradigm, the *constant bandwidth server* (*CBS*) polling server paradigm, and the generalization of Liu and Layland scheduling known as *rate-based execution* (*RBE*). We compared their performance for three scheduling problems found in FreeBSD: application-level scheduling of user programs, scheduling the execution of system calls made by applications in the "top-half" of the operating system, and scheduling asynchronous events generated by devices in the "bottom-half" of the operating system. For each scheduling problem we considered the problem of network packet and protocol processing for a suite of canonical multimedia applications. We tested each implementation under three workloads: a uniform rate packet arrival process, a bursty arrival process, and a misbehaved arrival process that generates work faster than the corresponding application process can consume it.

The results were mixed. When work arrives at rates that match the underlying scheduling model (the constant rate senders case), all the policies we considered achieve real-time performance. When work arrives that exceeds the application's rate specification, only the *CBS* server-based scheme and the *EEVDF* proportional share scheme provide isolation between well-behaved and misbehaved applications. When work arrives in a bursty manner, the quantum-based nature of *EEVDF* gives less responsive protocol processing and more packet loss. *CBS* performs better but suffers from CPU-time accounting problems that result in numerous missed deadlines. *RBE* provides the best response times but only at the expense of decreased throughput for the non-real-time activities.

We next investigated the application of different rate-based resource allocation schemes in different layers of the kernel and considered *EEVDF* proportional share scheduling of applications and system calls combined with either *CBS* servers or *RBE* tasks in the bottom half of the kernel. The quantum nature of *EEVDF* scheduling proves to be well suited to the FreeBSD application scheduling architecture and the coarser-grained nature of resource allocation in the higher-layers of the kernel. The event driven nature of *RBE* scheduling gives the best response times for packet and protocol processing. Moreover, the deterministic nature of lower-level kernel processing avoids the shortcomings observed when *RBE* scheduling is employed at the user-level.

In summary, we conclude that more research is needed on the design of rate-based resource allocation schemes that are tailored to the requirements and constraints of individual layers of an operating system kernel. All of the schemes we tested worked well for application-level scheduling (the problem primarily considered by the developers of each algorithm). However, for intra-kernel resource allocation, these schemes give significantly different results. By combining resource allocation schemes we are able to alleviate specific shortcomings, however, this is likely more accidental than fundamental as none of these policies were specifically designed for scheduling activities within the kernel. By studying these problems in their own right, significant improvements should be possible.

References

1. D. Anderson, Tzou, S., Wahbe, R., Govindan, R., Andrews, M., Support for Live Digital Audio and Video, Proc. 10th Intl. Conf. on Distributed Computing Systems, Paris, France, May 1990, pp. 54-61.

2. L. Abeni, G. Buttazzo, *Integrating Multimedia Applications in Hard Real-Time Systems*, Proc. of the 19th IEEE Real-Time Systems Symposium, Madrid, Spain, December 1998, pp. 4-13.
3. A. Bavier, and L.L. Peterson, *BERT: A Scheduler for Best Effort and Real-time Tasks*, Technical Report, Department of Computer Science, Princeton University, 2001.
4. A. Demers, S. Keshav, and S. Shenkar, *Analysis and Simulation of a Fair Queueing Algorithm*, *Jour. of Internetworking Research & Experience*, October 1990, pp. 3-12.
5. M. Hamdaoui and P. Ramanathan. *A dynamic priority assignment technique for streams with (m,k)-firm deadlines*, IEEE Transactions on Computers, April 1995.
6. P. Goyal, X. Guo, H. Vin, *A Hierarchical CPU Scheduler for Multimedia Operating Systems*, USENIX Symp. on Operating Systems Design & Implementation, Seattle, WA, Oct. 1996, pp. 107-121.
7. K. Jeffay, F. D. Smith, A. Moorthy, J. Anderson, *Proportional Share Scheduling of Operating Systems Services for Real-Time Applications*, Proc. of the 19th IEEE Real-Time Systems Symposium, Madrid, Spain, December 1998, pp. 480-491.
8. K. Jeffay, D. Bennett, *Rate-Based Execution Abstraction for Multimedia Computing*, Proc. of the Fifth Intl. Workshop on Network & Operating System Support for Digital Audio & Video, Durham, NH, April 1995, *Lecture Notes in Computer Science*, Vol. 1018, pp. 64-75, Springer-Verlag, Heidelberg.
9. K. Jeffay, S. Goddard, *A Theory of Rate-Based Execution*, Proc. 20th IEEE Real-Time Systems Symposium, Dec. 1999, pp. 304-314.
10. K. Jeffay, G. Lamastra, A Comparative Study of the Realization of Rate-Based Computing Services in General Purpose Operating Systems, Proceedings of the Seventh IEEE International Conference on Real-Time Computing Systems and Applications, Cheju Island, South Korea, December 2000, pages 81-90.
11. M.B. Jones, D. Rosu, M.-C. Rosu, *CPU Reservations & Time Constraints: Efficient, Predictable Scheduling of Independent Activities*, Proc., Sixteenth ACM Symposium on Operating Systems Principles, Saint-Malo, France, October 1997, pp. 198-211.
12. Lehoczky, J., Sha, L., Ding, Y., The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior, Proc. of the 10th IEEE Real-Time Systems Symp., Santa Monica, CA, December 1989, pp. 166-171.
13. J. Leung, and J. Whitehead, On the complexity of fixed-priority scheduling of periodic, real-time tasks, Performance Evaluation, 2, 1982, pp. 237-50.
14. J.W. S. Liu, *Real-Time Systems*, Prentice Hall, 2000.
15. C. L. Liu and J. W. Layland, Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment, Journal of the ACM, Vol. 20, No. 1, January 1973, pp. 46-61.
16. LynuxWorks, *LynxOS and BlueCat real-time operating systems*, *http://www.lynx.com/index.html.*
17. Mentor Graphics, *VRTX Real-Time Operating System*.
18. A.K.-L., Mok, *Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment*, Ph.D. Thesis, MIT, Dept. of EE and CS, MIT/LCS/TR-297, May 1983.
19. J. Nieh, M.S. Lam, *Integrated Processor Scheduling for Multimedia*, Proc. 5th Intl. Workshop on Network and Operating System Support for Digital Audio & Video, Durham, N.H., April 1995, Lecture Notes in Computer Science, T.D.C. Little & R. Gusella, eds., Vol. 1018, Springer-Verlag, Heidelberg.
20. A. K. Parekh and R. G. Gallager, *A Generalized Processor Sharing Approach To Flow Control in Integrated Services Networks-The Single Node Case*, *ACM/IEEE Transactions on Networking*, Vol. 1, No. 3, 1992, pp. 344-357.
21. *pSOS+TM/68K Real-Time Executive*, User's Manual, Motorola, Inc.
22. *QNX Operating System*, System Architecture and Neutrino System Architecture Guide, QNX Software Systems Ltd, 1999.

23. M. Spuri, G. Buttazzo, *Efficient Aperiodic Service Under the Earliest Deadline Scheduling*, Proc. 15th IEEE Real-Time Systems Symp., Dec. 1994, pp. 2-11.
24. M. Spuri, G. Buttazzo, F. Sensini, *Robust Aperiodic Scheduling Under Dynamic Priority Systems*, Proc. 16th IEEE Real-Time Systems Symp., Dec. 1995, pp. 288-299.
25. I. Stoica, H. Abdel-Wahab, K. Jeffay, S. Baruah, J. Gehrke, C. Plaxton, *A Proportional Share Resource Allocation Algorithm for Real-Time, Time-Shared Systems*, Proc. 17th IEEE Real-Time Systems Symposium, Dec. 1996, pp. 288-299.
26. I. Stoica, H. Abdel-Wahab, K. Jeffay, *On the Duality between Resource Reservation and Proportional Share Resource Allocation*, Multimedia Computing & Networking '97, SPIE Proceedings Series, Vol. 3020, Feb. 1997, pp. 207-214.
27. *VxWorks Programmer's Guide*, WindRiver System, Inc., 1997.
28. C.A. Waldspurger, W.E. Weihl, *Lottery Scheduling: Flexible Proportional-Share Resource Management*, Proc. of the First Symp. on Operating System Design and Implementation, Nov. 1994, pp. 1-12.
29. R. West, K. Schwan, and C. Poellabauer, *Scalable scheduling support for loss and delay constrained media streams*, Proceedings of the 5th IEEE Real-Time Technology and Applications Symposium, Vancouver, Canada, June 1999.
30. G.R. Wright, W.R. Stevens, *TCP/IP Illustrated, Volume 2, The Implementation*, Addison-Wesley, Reading MA, 1995.

The Temporal Specification of Interfaces in Distributed Real-Time Systems

H. Kopetz

Institut für Technische Informatik
Technische Universität Wien, Austria
hk@vmars.tuwien.ac.at

Abstract. The constructive design of dependable distributed real-time systems out of prevalidated components requires precise interface specifications of the components in the temporal domain and in the value domain. This paper focuses on the temporal specification of interfaces in composable distributed real-time systems. It establishes four principles of composability and gives examples of common composability violations in existing systems. It then classifies interfaces from the point of view of composability and presents, in the final section, as an example, the rational for the interface design in the time-triggered architecture. The time-triggered architecture is a distributed architecture for dependable embedded systems that supports the principles of composability.

1 Introduction

One of the most pressing problems in the domain of distributed real-time systems relates to the constructive design of large systems out of independently developed pre-validated components. This problem appears in the literature under different names: *composability*, *reuse of software*, *systematic integration*, *modularity*, *component-based design* and many more. At the core of this problem is the provision of precise interface specifications, both in the value domain and in the temporal domain.

Composability has many facets. In this paper we focus on the temporal properties of interfaces as they relate to composability. Temporal properties of interfaces can only be specified at the architecture level if the architecture provides a global notion of time and supports a two-level design methodology: architecture design and component design must be strictly separable. The precise specification of the data properties and the temporal properties of the component interfaces, including an appropriate model for the interpretation of these data, must be developed during the architecture design phase. During component design, these interface specifications are considered to be the constraints that must be satisfied by the component implementation. Given the precise interface specification of existing components, it is also possible to determine if a given application architecture meets the specification of these components. This is an approach to the solution of the *reuse* problem.

It is the objective of this workshop contribution to investigate interface design and temporal composability issues in distributed real-time computer systems. The paper is organized as follows: Section two presents the system model, discussing the structure, the model of time, and the information types that are exchanged among the nodes of a distributed real-time system. Section three introduces the different interface types that must be supported in distributed real time systems, defines the concept of compos

T.A. Henzinger and C.M. Kirsch (Eds.): EMSOFT 2001, LNCS 2211, pp. 223-236, 2001.
© Springer-Verlag Berlin Heidelberg 2001

ability, establishes four principles of composability, and reviews common temporal composability violations. Section four classifies existing interfaces according to the criteria that are relevant from the point of view of temporal composability. Finally, as a practical example, in Section five the rational for the design of the interfaces in the time-triggered architecture is presented. The paper closes with a conclusion in Section six.

2 System Model

In this section we present the system model and introduce the basic concepts that are used throughout this paper. These basic concepts are further refined in the context of the EU sponsored DSOS project [4].

2.1 Model of Time

Our model of time is based on Newtonian physics. Real time progresses along a dense timeline, consisting of an infinite set of *instants*, from the past to the future. A *duration (or interval)* is a section of the timeline, delimited by two instants. A happening that occurs at an instant (i.e., a cut of the timeline) is called an *event*. An *observation* of the state of the world is thus an event. The *time-stamp* of an event is established by assigning the state of the global time of the observing node to the event immediately after the event occurrence. An internal fault-tolerant synchronization algorithm established this global time. Due to the impossibility of synchronizing clocks perfectly and the denseness property of real time, there is always the possibility of the following sequence of events: clock in node *j* ticks, event *e* occurs, clock in node *k* ticks. In such a situation, the single event *e* is time-stamped by the two clocks *j* and *k* with a difference of one tick. The finite precision of the global time-base and the digitalization of the time makes it impossible in a distributed system to order events consistently on the basis of their global time-stamps. This problem is solved by the introduction of a *sparse time base* [5], p.55. In the sparse-time model the continuum of time in partitioned into an infinite sequence of alternating durations of *activity* and *silence* as shown in Fig. 1. The duration of the activity interval, i.e., a granule of the global time, must be larger than the precision of the clock synchronization.

From the point of view of temporal ordering, all events that occur within a duration of activity are considered to happen *at the same time*. Events that happen in the distributed system at different nodes at the same clock-tick are thus considered *simultaneous*. Events that happen during different durations of activity and are separated by the required interval of silence can be temporally ordered on the basis of their global timestamps. The architecture must make sure that significant events, such as the sending of a message, occur only during an interval of activity. The timestamps of events that are outside the control of the distributed computer system (and therefore happen on a dense timeline) must be assigned to an agreed duration of activity by an *agreement protocol*.

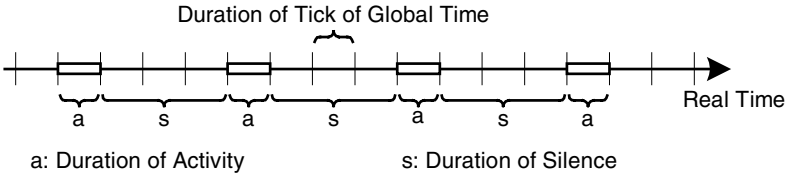


Fig. 1. Sparse time base

2.2 RT Entities and RT Images

A distributed real-time computer system is modeled by a set of nodes that are interconnected by a real-time communication system as shown in Fig. 2. All nodes have access to the global time. A node consists of a *communication controller (CC)* and a *host computer*. The common boundary between the communication controller and the host computer within a node is called the *communication network interface CNI* (thick black line in Fig. 2).

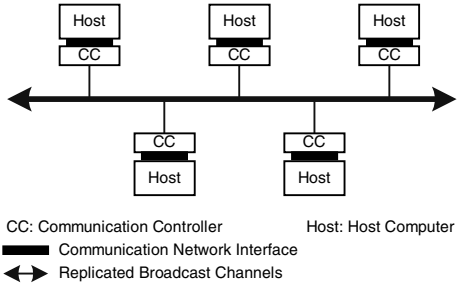


Fig. 2. Distributed real-time system with five nodes

The dynamics of a real-time application is modeled by a set of relevant state variables, the *real-time (RT) entities*, that change their state as time progresses. Examples of RT entities are the flow of a liquid in a pipe, the setpoint of a control loop or the intended position of a control valve. An RT entity has static attributes that do not change during the lifetime of the RT entity, and has dynamic attributes that change with time. Examples of static attributes are the name, the type, the value domain, and the maximum rate of change. The value set at a particular instant is the most important dynamic attribute. Another example of a dynamic attribute is the rate of change at a chosen instant.

The information about the state of an RT entity at a particular instant is captured by the notion of an *observation*. An observation is an *atomic data structure*

$$Observation = <Name, t_{obs}, Value>$$

consisting of the name of the RT entity, the point in real time when the observation was made (t_{obs}), and the observed value of the RT entity. A continuous RT entity can

be observed at any instant while a discrete RT entity can only be observed when the state of this RT is not changing.

A *real-time (RT) image* is a *temporally accurate* picture of an RT entity at instant *t*, if the duration between the time-of-observation and the instant *t* is less than the accuracy interval d_{acc}, which is an application specific parameter associated with the given RT entity. An RT image is thus *valid* at a given instant if it is an accurate representation of the corresponding RT entity, both in the value and the time domains [8]. While an *observation* records a fact that remains valid forever (a statement about an RT entity that has been observed at an instant), the validity of an RT image is *time-dependent* and thus likely to be invalidated by the progression of real-time.

At the communication network interfaces (CNI) within a node, the pictures of the RT entities are periodically updated by the real-time communication system to establish temporally accurate *RT-images* of the RT-entities. The computational tasks within the host of a node take these temporally accurate RT-images as inputs to calculate the required outputs within an *a priori* known worst-case execution time (WCET). The outputs of the host are stored in the CNI and transported by the time-triggered communication system to the CNIs of other nodes at *a priori* determined instants. The *interface nodes* transform the received data to/from the representation required by the controlled object or the human operator and activate control actions in the physical world.

2.3 State-Information versus Event-Information

The information that is exchanged across an interface is either *state information* or *event information*, as explained in the following paragraphs. Any property of *a real-time (RT) entity* (i.e., a *relevant state variable)* that is a observed by a node of the distributed real-time system at a particular instant, e.g, the temperature of a vessel, is called a *state attribute* and the corresponding information *state information.* A *state observation* records the state of a state variable at a particular instant, the *point of observation*. A *state observation* can be expressed by the atomic triple

<Name of the observed state variable, observed value, time of observation>

For example, the following is a state observation: *"The position of control valve A was at 75 degrees at 10:42 a.m."* State information is *idempotent* and requires an *at-least once semantics* when transmitted to a client. At the sender, state information *is not consumed on sending* and at the receiver, state information requires an *update-in-place* and a *non-consumable read.* State information is transmitted in *state messages*.

A sudden change of state of an RT entity that occurs at an instant is an *event*. Information that describes an event is called *event information.* Event information contains the *difference* between the state *before* the event and the state *after* the event.

An *event observation* can be expressed by the atomic triple

<Name of the observed state variable, value difference, time of event>

For example, the following is an event observation: *"The position of control valve A changed by 5 degrees at 10:42 a.m."* Event observations requires *exactly-once semantics* when transmitted to a consumer. At the sender, event information is *consumed on sending* and at the receiver, event information must be *queued* and *consumed on reading*. Event information is transmitted in *event messages*.

Periodic state observations or sporadic event observations are two *alternative* approaches for the observation of a dynamic environment in order to reconstruct *the states and events* of the environment at the observer [12]. Periodic state observations produce a sequence of equidistant "snapshots" of the environment that can be used by the observer to reconstruct those events that occur within a minimum temporal distance that is longer than the duration of the sampling period. Starting from an initial state, a complete sequence of (sporadic) event observations can be used by the observer to reconstruct the complete sequence of states of the RT entity that occurred in the environment. However, if there is no minimum duration between events assumed, the observer and the communication system must be infinitely fast.

3 Temporal Composability

In many engineering disciplines, large systems are built by the constructive integration of well-specified and pre-tested subsystems, called *components*. The components are characterized by their physical parameters and the services they provide across well-specified interfaces. In a composable architecture, this integration should proceed without unintended side effect. In this section we analyze the interface issues that have to addressed if temporal composability of information processing components is to be attained. An information processing component can guarantees the timely specified service (post-conditions) only if the required inputs are provided on time (pre-conditions) and if the necessary computational resources are available.

From the point of view of the analysis of a composable architecture, it is reasonable to distinguish between the following two service classes of an integrated distributed control system:

1. **Prior Services:** Given a component that has been developed independently to provide a specified service, e.g., the control of an engine by an engine-control system. This service has been validated at the component level and is thus available prior to the integration of the component into an integrated control system. We call such a service a *prior service.*
2. **Emerging Services**: The integration of components into a system generates new services that are more than the sum of the prior services. Take the example of a car: The integration of the four wheels, the chassis and the engine, all individual components, give rise to a new level of service, the transport service, that was not present at the component level. We call these additional services that come about by the integration the *emerging services.*

In a distributed real-time computer system, the emerging services are the result of information exchanges among the interacting nodes across the component interfaces. Therefore, the communication system plays a central role in determining the composability of distributed computer architecture.

3.1 Component Interfaces

From the point of view of complexity management and composability, it is useful to distinguish between three different types of interfaces of a component: the real-time-service (RS) interface, the diagnostic and management (DM) interface, and the configuration planning (CP) interface [7]. For the temporal composability, the most important interface is the RS interface.

The Real-Time-Service (RS) Interface: The RS interface provides the timely real-time services to the component environment during the operation of the system. In real-time systems it is a time-sensitive interface that must meet the temporal specification of the architecture in all specified load and fault scenarios. The composability of architecture depends on the proper support of the specified RS interface properties (in the value and in the temporal domain) during the operation. >From the point of a user, the internals of the component are not visible, since they are hidden behind the RS interface.

The Diagnostic and Management (DM) Interface: The DM interface opens a communication channel to the internals of a component. It is used for setting component parameters and for retrieving information about the internals of the component, e.g., for the purpose of internal fault diagnosis. The maintenance engineer that accesses the component internals via the DM interface must have detailed knowledge about the internal structure and behavior of the component. The DM interface is not contributing to the composability. Normally, the DM interface is not time-critical.

The Configuration Planning (CP) Interface: The CP interface is used to connect a component to other components of a system. It is used during the integration phase to generate the "glue" between the nearly autonomous components. The use of the CP interface does not require detailed knowledge about the internal operation of a component. The CP interface is not time critical.

3.2 The Principles of Composability

In a distributed real-time system the components interact via a communication system to provide the emergent real-time services. These emerging services depend on the timely provision of the real-time information at the RS interfaces of the components. For an architecture to be composable, it must adhere to the following four principles with respect to the RS-interfaces:

1. Independent development of components.
2. Stability of prior services.
3. Constructive integration of the components to generate the emerging services.
4. Replica determinism

Independent Development of Components: A composable architecture must distinguish sharply between architecture design and component design. Principle one of a composable architecture is concerned with design at the architecture level. Components can only be designed independently of each other, if the architecture supports the precise specification of all component services at the level of architecture design. In a real-time system the RS-interface specification of a component must comprise the precise CNI specification in the value domain and in the temporal domain and a proper abstract model of the component service, as perceived by the user of the com-

ponent. Only then is the component designer in the position to know exactly what can be expected from the environment and what must be delivered by the component.

Stability of Prior Services: Principle two of a composable architecture is concerned with the design at the component level. A component is a nearly autonomous subsystem that comprises the hardware, the operating system and the application software. The component must provide the intended services across the well-specified component interfaces. The design of the component can take advantage of any established software engineering methodology, such as object based design methods. The stability-of-prior-service principle ensures that the validated service of a component—both in the value domain and in the time domain—is not refuted by the integration of the component into a system. For example, the integration of a self-contained component, e.g., an engine controller, into the integrated vehicle control system may require additional computational resources of the component to service the RS-interface, both in processing time and in memory space. Consider the case where the CNI is based on a queue of messages that must be maintained by the host computer: memory space for the queue must be allocated by the component-local operating system and processing time of the host processor for the management of the queue must be made available. In a time-critical component it may happen that these additional resource requirements that are needed for the timely interface service, are in conflict with the resource requirements of the application software that implements the prior services of the component. In such a situation, failures in the component services may occur sporadically.

Constructive Integration: Principle three of a composable architecture is concerned with the design of the communication system. Normally, the integration of the components into the system follows a step-by-step procedure. The constructive integration principle requires that if n components are already integrated, the integration of the n+1 component may not disturb the correct operation of the n already integrated components. The constructive-integration principle ensures that this integration activity is linear and not circular.

This constructive integration principle has severe implications for the management of the network resources. If network resources are managed dynamically, it must be ascertained that even at the critical instant, i.e., when all components request the network resources at the same point in time, the timeliness of all communication requests can be satisfied. Otherwise sporadic failures will occur with a failure rate that is increasing with the number of components integrated.

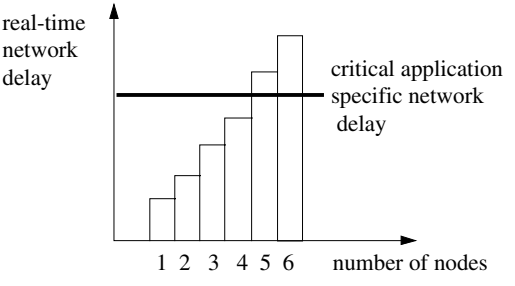


Fig. 3. Maximum network delay at critical instant as a function of the number of nodes.

For example, if a real-time service requires that the maximum latency of the network must always remain below a critical upper limit (because otherwise a local timeout within the component may signal a communication failure) then the dynamic extension of the network latency by adding new components may be a cause of concern. In a dynamic network the message delay at the critical instant (when all components request service at the same instant) increases with the number of components. The system of Fig. 3 will work correctly with up to four components. The addition of the fifth component may lead to sporadic failures.

Other applications, e.g., when a time-sensitive control loop is closed by the network, may require a network of known and constant jitter in order to support this principles of constructive integration.

Replica Determinism: If fault-tolerance is implemented by the replication of components, then the architecture and the components must support replica determinism. A set of replicated components is *replica determinate* [10] if all the members of this set have the same externally visible state, and produce the same output messages at points in time that are at most an interval of *d* time units apart (as seen by the omniscient outside observer). In a fault-tolerant system, the time interval *d* determines the time it takes to replace a missing message or an erroneous message from a node by a correct message from redundant replicas. The implementation of replica determinism is simplified if all components have access to a globally synchronized sparse time base.

3.3 Common Composability Violations

In this section we elaborate on some common scenarios that can lead to sporadic composability violations. To be able to illustrate these scenarios on a concrete example, let us assume a distributed system consisting of set of 10 nodes that are connected by a CAN based [2] communication network that can transport up to 5000 broadcast messages per second (in this system it takes 200 μsec to transport a single message).

Missing Temporal Interface Specification: Many of the present event-triggered architectures do not provide the capability to define the temporal properties of the CNIs with the required rigor. The exact points in time when messages are expected to arrive or are supposed to be sent by a component, and the phase relationships between the messages are not contained in many CNI specifications because there exists no notion of a global time. This lose specification of the CNIs makes it difficult for the component designer to thoroughly validate a component behavior in isolation, i.e., outside the system context. For example, if the peak-load scenario of service requests to a component (rate and phase relationships between the requests) is not precisely specified at the architecture level (this cannot be done in a CAN system), then vague assumptions about the system context of component use may replace the missing interface specifications. Such a component cannot be validated independently and may not operate correctly in another system context. The principle of *Independent Development of Components* is thus violated.

Excessive Sending Delay: In a system where the communication channel is shared dynamically between the nodes, the maximum sending delay for all but the highest priority message depends on the number of nodes and the sending activity of each node. Even if we assume that a node will not send another message before his previous message has been transmitted [11] (an assumption that is not enforced by the

CAN protocol, but is enforced by other protocols, e.g., the ARINC 629 [1]) the worst case sending delay (disregarding the occurrence of transmission failures) at the critical instant will be

$$(n+1)\ 200\ \mu sec$$

If a critical application timeout in the system is around 1000 μsec, the system will violate the *constructive integration principle* if more than 5 nodes are connected. The reproduction of such a failure will be difficult, because the occurrence of the failure is dependent on the temporal phase relationship among the send operation of the sending nodes. If the above assumption on the minimum time between the message transmissions of a single host is not maintained, a host can monopolize all other hosts by sending a sequence of high-priority message (*starvation* of the other hosts).

Sporadic Interruptions of Time-sensitive Host Tasks: Another composability violation can occur due the unplanned interruption of a time-sensitive task by the sporadic arrival of messages. Assume that worst-case execution time of a time-critical task has been carefully analyzed and a correct schedule has been designed in the order that the task meets its deadline. If the execution time of this task is extended by an interrupt service routine that responds to an unplanned interrupt from the communication controller or some other source, then the task will miss its critical deadline. In this scenario the *stability of prior service principle* is violated.

Excessive Load: Every host computer has a finite processing capacity. In a real-time node, a known proportion of this processing capacity is allocated for application scheduling such that all real-time tasks will meet their deadline. Another proportion is reserved for the middleware to service the input/output system and perform the housekeeping functions within the host. If the load generated by the input-output system, i.e., the number of arriving messages that must be handled by the middleware, is excessive, then the host will enter an overload situation and will lose messages or miss deadlines of time-critical tasks. In the above example, the raw communication system is capable to deliver 5000 messages a second to each node. In order to protect the node from such a heavy load, the CAN protocol contains a filter mechanism to discard messages the node is not interested in before they are transferred to the CNI. If these filters are not set correctly and the host will receive more messages than it can process, either the queues will grow beyond their limits and messages will be lost or application tasks will miss their deadlines. The principle of *constructive integration* or *stability of prior service principle* is violated.

Dense Time-base: If the time-base of the global time in a distributed system is *dense* [5], then it is in general not possible to generate a consistent temporal order on the basis of the time-stamps. (A consistent temporal order of events can only be established if the time-base is *sparse*.) An inconsistent order of events can lead to a loss of replica determinism and, in consequence of the planned fault-tolerance [10]. The principle of *replica determinism* is thus violated.

4 Interface Classification

A system architect has many design choices for the layout of the communication network interface (CNI). In this section we elaborate on these design choices. Our main interest is an interface that supports, at the transport level, a unidirectional data flow.

4.1 State Message versus Event Message

The CNI can be designed to handle the incoming and outgoing messages as state messages or as event messages.

State Message: A *state message* is a periodic time-triggered message that copies the sending buffer at the sender without consuming the message and delivers the message at the receiver by overwriting the single receive buffer. A state message interface is intended to transfer state information (see Sect. 2.3). The semantics of state messages is closely related to the semantics of a variable in a programming language. Since only the "freshest" state message is contained in the CNI, no queues are needed at the sender or the receiver. State messages do not require a tight synchronization between sender and receiver. The sender is not required to update the send buffer between send operations. The receiver is not required to read every arriving state message. The receiver can read a given state message more than once. At the *a priori* specified send and receive instants (which are contained in the communication system), the state message is copied by the communication system from the send buffer and is delivered to the receiver by overwriting the receive buffer atomically.

Event Message: An *event message* is a sporadic event-triggered message that consumes the message at the sender from a *send queue* and delivers the message at the receiver by adding the message to the *receive queue*. An event message interface is intended to transfer event information (see Sect. 2.3). The instant of sending the event message is determined by the occurrence of the relevant event (e.g., the execution of a *send* statement) at the sending host. The receiver is required to eventually read every incoming event message, thus implying a one-to-one synchronization between send operations at the sender and the receive operations at the receiver. Since event messages contain event information (i.e., the difference between the old and new state—see Sect. 2.3), an exactly once semantics must be implemented. Since it is not known *a priori* when an event message is supposed to arrive at a receiver, bi-directional error detection protocol must be implemented to detect the loss of an event message, even if the information transfer is only uni-directional [6].

4.2 Information Push versus Information Pull

The CNI can be designed to handle the incoming and outgoing messages according to the *information-push* model or the i*nformation-pull* model [3].

Information Push: The *information push model* presumes that the sender presses the information on the receiver. According to this model, an incoming message will trigger a control (interrupt) signal across an interface as soon as it arrives to alert the receiver.

Information Pull: The *information pull model* assumes that the receiver periodically or sporadically interrogates a data structure to find out whether new information is available.

In a distributed real-time system, the information-push model is ideal from the point of view of the sender and the information-pull model is ideal from the point of view of the receiver, since a time-critical task of the receiver will never be interrupted by an incoming (pushed) message.

4.3 Elementary versus Composite Interface

We call a unidirectional data-flow interface *elementary*, if the there is also a unidirectional control flow [6]. An interface that supports periodic state message with error detection at the receiver is an example of an elementary interface.

We call a data-flow interface *composite*, if even a unidirectional data flow requires a bi-directional control flow. An event-message interface with error detection is an example for a composite interface. Composite interfaces are inherently more complex than elementary interfaces, since the correct operation of the sender *depends on* the control signals from the receiver. This can be a problem in multicast communication since many control messages are generated for every unidirectional data transfer, and each one of the receivers can affect the operation of the sender. Multicast communication is common in distributed real-time systems.

4.4 Error Detection and Error Handling

Interfaces can be classified according to the mechanisms they contain for error detection and error handling. In non-real-time communication systems, where most information exchanges are based on event messages, error detection and error handling are often lumped together into a single protocol. After an erroneous message is detected, the retransmission of this message is requested in order to correct the error (time-redundancy). In real-time systems, where time is a critical resource, the automatic application of time redundancy may be counter-productive. For example, if a time-triggered message containing the most recent sensor reading is corrupted, it makes more sense to wait for the next message then to try to retransmit the original message, which may have lost its temporal accuracy already by the time it is retransmitted (see Sect. 2.4).

Error Detection: Every interface should contain mechanisms to detect errors and to inform its client about a detected errors. In real-time systems, prompt error detection at the receiver is more important than error detection at the sender [5], because in many cases the receiver can initiate application specific error handling actions as soon as the error has been detected.

Error Handling: In distributed real-time systems, the mechanisms for error handling should be separated from the mechanisms for error detection.

5 Interfaces in the TTA

In this section we explain which design decisions have been taken during the interface design of the time triggered architecture (TTA). The TTA is a distributed computer architecture for the implementation of dependable real-time systems [5]. Safety, composability, and flexibility (in this order) have been the objectives during architecture design.

5.1 The Time-Triggered Architecture

Characteristic of the TTA is the availability of a global time of known precision in every node of the distributed computer system. A large TTA system can be decom-

posed into a set of nearly autonomous clusters that are linked by replicated gateway components. Fault tolerance is achieved at the cluster level by replicating the nodes within a cluster and by replicating the communications channels between the nodes.

A TTA cluster consists of a set of TTA nodes connected by means of replicated communications channels. A TTA node comprises a host computer and a communications controller (as shown in Fig. 1) with two bi-directional communication ports. Each one of these two ports is connected to an independent channel of a dual-channel broadcast system. The TTA distinguishes between two types of nodes, core nodes (c-nodes) and non-core (nc-nodes). A c-node participates in the provision of the system services, i.e., clock synchronization and membership. An nc-node takes advantage of the global time established by the c-nodes and is allowed to transmit in it's *a priori* determined transmission slot. Nc-nodes are not participating in the distributed clock synchronization and membership. The decision whether a node is a c-node or an nc-node is made *a priori* and stored in a controller internal data structure before run time. The TTA monitors the behavior of all nodes and ensures the integrity of the temporal and spatial partitions. In the following we consider a TTA cluster that consists of c-nodes only.

A TTA node considers a transmission successful, as judged by the receiver, if one syntactically correct message is received on at least one of the two communications channels. Communication on the broadcast system proceeds according to an *a priori* established time-division multiple access (TDMA) scheme stored in the controller-internal data structure called MEDL (message descriptor list). The TDMA scheme divides time into slots. Every slot is assigned to a TTA node, which is exclusively granted write permission to the broadcast system during the slot. The slots are grouped into rounds: during a (TDMA) round every TTA node is granted write permission exactly once. Furthermore, nodes always send in the same relative slot of a round and, finally, the slots assigned to a particular node have always the same length. A distributed fault-tolerant clock synchronization algorithm, which is executed among the ensemble of c-nodes, establishes the global time, which is necessary for the execution of the TDMA scheme.

A cluster cycle comprises several TDMA rounds and – similar to the TDMA round, which multiplexes the broadcast system between several nodes – multiplexes the slots assigned to a node in succeeding TDMA rounds between different messages produced by the respective node.

Since every node has knowledge of the complete TDMA scheme (and not only of the slots assigned to the respective node itself), nodes know *a priori* when a message must be sent or received. This knowledge is used for a number of purposes since it allows

1. deriving the sender identification from the reception time of a message, (there is thus no need for transmitting the sender identification explicitly)
2. deriving the message identification from the reception time (there is no need for transmitting explicit message identifications)
3. immediate error detection at the receiver if a correct message does not arrive at the *a priori* specified receive time.

Error detection at the receiver it realized by the membership protocol which is an integral part of the TTA. The TTA provides a consistent distributed computing base to all nodes and informs the nodes if the consistency cannot be maintained because of the occurrence of more failures than specified in the fault hypothesis.

5.2 Temporal Firewalls

The TTA provides a layered communication service based on *elementary* interfaces as shown in Fig. 4. The transport layer carries autonomously—driven by its time-triggered schedule—state messages from the sender's CNI to the receiver's CNI. The sender can deposit the information into its CNI according to the information *push* paradigm, while the receiver must *pull* the information out of its CNI. Since no control signals cross the CNI, control-error propagation is eliminated by design. We call an interface that prevents control-error propagation *a temporal firewall* [9].

This basic layer can be used for the provision of the predictable real-time (RS) service interface. On input, the precise interface specifications (in the temporal and value domain) are the *pre-conditions* for the correct operation of the host software. On output, the precise interface specifications are the *post-conditions* that must be satisfied by the host, provided the preconditions have been satisfied by the host environment. Since the bandwidth is allocated statically to the host, no *starvation* of any host can occur due to the high-priority message transmission from other hosts.

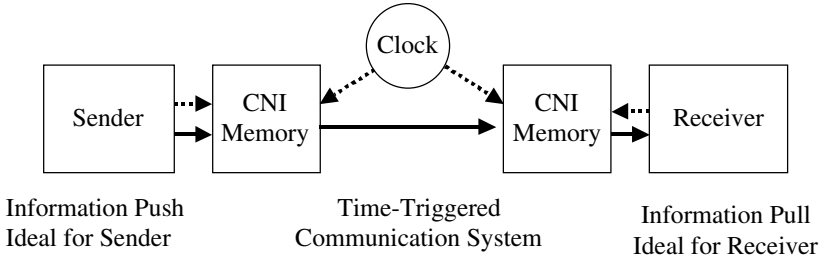


Fig. 4. Data flow (full line) and control flow (dashed line) at an TTA interface

5.3 Event-Triggered Channels

Whereas the time-critical RS service can be satisfied by the time-triggered basic TTA communication layer, the non-time-critical DM and the CP service require an event-triggered communication service. The TTA provides these event triggered communication service by implementing event-triggered channels on top of the basic time-triggered layer. For this purpose, message queues and message filtering are implemented in the communication subsystem at the sender and at the receiver. A portion of the bandwidth of the time-triggered slots of every node is allocated to this event-triggered channel. The clock synchronization and the membership service of the basic TTA are used to implement error detection and retransmission of the event messages in the event-triggered channels. In order to avoid an information-push interface at the receiver, the message queue at the receiver must be configured such that the host processor at the receiver will never be interrupted during the processing of time-critical tasks. From a host-local scheduling point of view, this communication architecture is attractive: Within every cycle, the host can first process the time-critical messages and, in the remaining time before the next cycle starts, empty the receive queue of non-time critical event messages.

6 Conclusions

The solution of the composability problems in large distributed real-time systems requires an appropriate architecture that supports the precise specification of the component interfaces in the temporal domain and in the value domain. At the receiver site, incoming information should be handled according to the information-pull paradigm, which assures that no time-critical task of the receiver is interrupted by the sporadic arrival of messages. Time-critical information should be transmitted in state messages that are periodically updated by an autonomous time-triggered communication system. Non time-critical information can be transported in the form of event messages which are queued inside the communication system until the receiver is in the position to pull the incoming information out of this queue.

Acknowledgments. This work has been supported, in part, by the IST project DSOS and by the IST project FIT.

References

1. ARINC (1991). Multi-Transmitter Data Bus ARINC 629--Part 1: Technical Description, Aeronautical Radio Inc., Annapolis, Maryland 21401.
2. CAN (1990). Controller Area Network CAN, an In-Vehicle Serial Communication Protocol. SAE Handbook 1992, SAE Press. SAE J1583: 20.341-20.355.
3. Deline, R. (1999). Resolving Packaging Mismatch. Computer Science Department. Pittsburgh, Carnegie Mellon University**:** 178.
4. Jones, C., H. Kopetz, et al. (2001). Revised Conceptual Model of DSOS. University of Newcastle upon Tyne, Computer Science Department.
5. Kopetz, H. (1997). Real-Time Systems, Design Principles for Distributed Embedded Applications; ISBN: 0-7923-9894-7, Third printing 1999. Boston, Kluwer Academic Publishers.
6. Kopetz, H. (1999). Elementary versus Composite Interfaces in Distributed Real-Time Systems. Proc. of ISADS 99, IEEE Press, Tokyo, Japan.
7. Kopetz, H. (2000). Software Engineering for Real-Time: A Roadmap. Software Engineering Conference 2000, Limmerick, Ireland, IEEE Press.
8. Kopetz, H. and K. Kim (1990). Temporal Uncertainties in Interactions among Real-Time Objects. Proc. 9th Symposium on Reliable Distributed Systems, Huntsville, AL, USA, IEEE Computer Society Press.
9. Kopetz, H. and R. Nossal (1997). Temporal Firewalls in Large Distributed Real-Time Systems. Proceedings of IEEE Workshop on Future Trends in Distributed Computing, Tunis, Tunesia, IEEE Press.
10. Poledna, S. (1994). Replica Determinism in Fault-Tolerant Real-Time Systems, Technical University of Vienna.
11. Tindell, K. (1995). "Analysis of Hard Real-Time Communications." Real-Time Systems **9**(2): 147-171.
12. Tisato, F. and F. DePaoli (1995). On the Duality between Event-Driven and Time Driven Models. Proc. of 13th. IFAC DCCS 1995, Toulouse France.

System-Level Types for Component-Based Design

Edward A. Lee and Yuhong Xiong
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
Berkeley, CA 94720, USA
{eal, yuhong}@eecs.berkeley.edu

Abstract. We present a framework to extend the concept of type systems in programming languages to capture the dynamic interaction in component-based design, such as the communication protocols between components. In our system, the interaction types and the dynamic behavior of components are defined using interface automata - an automata-based formalism. Type checking, which checks the compatibility of a component with a certain interaction type, is conducted through automata composition. Our type system is polymorphic in that a component may be compatible with more than one interaction type. We show that a subtyping relation exists among various interaction types and this relation can be described using a partial order. This system-level type order can be used to facilitate the design of polymorphic components and simplify type checking. In addition to static type checking, we also propose to extend the use of interface automata to the on-line reflection of component states and to run-time type checking. We illustrate our framework using a component-based design environment, Ptolemy II, and discuss the trade-offs in the design of system-level type systems.

1 Introduction

Type systems are one of the most successful formal methods in software design. Modern polymorphic type systems, with their early error detection capabilities and the support for software reuse, have led to considerable improvements in development productivity and software quality.

For embedded systems, component-based design is established as an important approach to handle complexity. In this area, type systems can be used to greatly improve the quality of design environment. Fundamentally, a type system detects mismatch at component interfaces and ensures component compatibility. Interface mismatch can happen at (at least) two different levels. One is the data type level. For example, if a component expects to receive an integer at its input, but another component sends it a string, then the first component may not be able to function correctly. Many type system techniques in general purpose languages can be applied effectively to ensure compatibility at this level (see [15] and the references therein). The other level of mismatch is the dynamic interaction behavior, such as the communication protocol the components use to exchange data. Since embedded systems often have many concurrent computational activities and mix widely differing operations, components may follow widely different communication protocols. For example, some might use synchronous interaction (rendezvous) while others use asynchronous message passing (see [8] for many more examples). So far, most type system research for component-

T.A. Henzinger and C.M. Kirsch (Eds.): EMSOFT 2001, LNCS 2211, pp. 237–253, 2001.
© Springer-Verlag Berlin Heidelberg 2001

based design, as well as for general purpose languages, concentrates on data types, and leaves the checking of dynamic behavior to other techniques.

In this paper, we extend the concept of type systems to capture the dynamic aspects of component interaction. We call the result *system-level types* [10]. In our approach, different interaction types and the dynamic behavior of components are described by automata, and type checking is conducted through automata composition. In this paper, we choose a particular automata model called *interface automata* [4] to define types. The particular strength of interface automata is their composition semantics.

Traditionally, automata models are used to perform model checking at design time. Here, our emphasis is not on model checking to verify arbitrary user code, but rather on compatibility of the composition of pre-defined types. As such, the scalability of the methods is much less an issue, since the size of the automata in question is fixed. We also propose to extend the use of automata to on-line reflection of component state, and to do run-time type checking. To explore these concepts, we have built an experimental platform based the Ptolemy II component-based design environment [3].

We have found that the design of system-level types shares the same goals and trade-offs with the design of a data-level type system. At the data level, research has been driven to a large degree by the desire to combine the flexibility of dynamically typed languages with the security and early error-detection potential of statically typed languages [13]. As mentioned earlier, modern polymorphic type systems have achieved this goal to a large extend. At the system-level, type systems should also be polymorphic to support component reuse while ensuring component compatibility.

In programming languages, there are several kinds of polymorphism. In [1], Cardelli and Wegner distinguished two broad kinds of polymorphism: *universal* and *ad hoc* polymorphism. Universal polymorphism is further divided into *parametric* and *inclusion* polymorphism. Parametric polymorphism is obtained when a function works uniformly on a range of types. Inclusion polymorphism appears in object oriented languages when a subclass can be used in place of a superclass. Ad hoc polymorphism is also further divided into overloading and coercion. In systems with subtyping and coercion, types naturally form a partial order [2]. For example, in object-oriented languages, the partial order is the inheritance hierarchy, and in languages that support type conversion, the relation in the partial order is the conversion relation, such as int $\leq$ double, which means that an integer can be converted to a double. This latter relation is sometimes considered as subtyping between primitive data types [12]. In the Ptolemy II data type system, the type hierarchy if further constrained to be a lattice, and type constraints are formulated and solved over the lattice [15].

We form a polymorphic type system at the system-level through an approach similar to subtyping. Using the alternating simulation relation of interface automata, we organize all the interaction types in a partial order. Given this hierarchy, if a component is compatible with a certain type A, it is also compatible with all the subtypes of A. This property can be used to facilitate the design of polymorphic components and simplify type checking.

Even with the power of polymorphism, no type system can capture all the properties of programs and allow type checking to be performed efficiently while keeping the language flexible. So the language designer always has to decide what properties to

include in the system and what to leave out. Furthermore, some properties that can be captured by types cannot be easily checked statically before the program runs. This is either because the information available at compile time is not sufficient, or because that checking those properties is too costly. Hence, the designer also needs to decide whether to check those properties statically or at run time. Any type system represents some compromise. For example, array bound checking is vary helpful in detecting program errors, but it is hard to do efficiently by static checks. Some languages, such as C, do not perform this check. Other languages, such as ML and Java, perform the check, but at run time, and at the cost of run time performance. Some researchers propose to perform this check at compile time [14], but the technique requires the programer to insert annotations in the source code, since modern languages do not include array bounds in their type systems.

Type systems at the system-level have similar trade-offs. Among all the properties in a component-based design environment, we choose to check the compatibility of communication protocols as the starting point. This is because communication protocols are the central piece in many models of computation [8] and determine many other properties in the models. Our type system is extensible so other properties, such as deadlock in concurrent models, can be included in type checking. Another reason we choose to check the compatibility of communication protocols is that it can be done efficiently, when a component is inserted in a model. More complicated checking may need to be postponed to run time.

The rest of this paper is organized as follows. Section 2 describes Ptolemy II, with emphasis on the implementation of various communication protocols. Section 3 gives an overview of interface automata. Section 4 presents our system-level type system, including the type definition, the type hierarchy and some type checking examples. Section 5 discusses the trade-offs in the design of system-level types and run-time type checking. The last section concludes the paper and points out our future research directions.

2 Ptolemy II - A Component-Based Design Environment

Ptolemy II [3] is a system-level design environment that supports component-based heterogeneous modeling and design. The focus is on embedded systems. In Ptolemy II, components are called *actors*, and the channel of communication between actors is implemented by an object called a *receiver*, as shown in figure 1. Receivers are contained in *IOPorts* (input/output ports), which are in turn contained in actors.

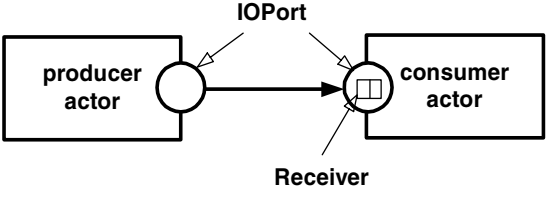


Fig. 1. A simple model in Ptolemy II.

Ptolemy II is implemented in Java. The methods in the receiver are defined in a Java interface Receiver. This interface assumes a producer/consumer model, and communicated data is encapsulated in a class called Token. The put() method is used by the producer to deposit a token into a receiver. The get() method is used by the consumer to extract a token from the receiver. The hasToken() method, which returns a boolean, indicates whether a call to get() will trigger a NoTokenException.

Aside from assuming a producer/consumer model, the Receiver interface makes no further assumptions. It does not, for example, determine whether communication between actors is synchronous or asynchronous. Nor does it determine the capacity of a receiver. These properties of a receiver are determined by concrete classes that implement the Receiver interface. Each one of these concrete classes is part of a Ptolemy II *domain*, which is a collection of classes implementing a particular model of computation. In each domain, the receiver determines the communication protocol, and an object called a *director* controls the execution of actors. From the point of view of an actor, the director and the receiver form its execution environment.

Each actor has a fire() method that the director uses to start the execution of the actor. During the execution, an actor may interact with the receivers to receive or send data. Some of the domains in Ptolemy II are:

- *Communicating Sequential Processes (CSP)*: As the name suggests, this domain implements a rendezvous-style communication (sometimes called synchronous message passing), as in Hoare's communicating sequential processes model [5]. In this domain, the producer and consumer are separate threads executing the fire() method of the actors. Which ever thread calls put() or get() first blocks until the other thread calls get() or put(). Data is exchanged in an atomic action when both the producer and consumer are ready.
- *Process Networks (PN)*: This domain implements the Kahn process networks model of computation [6]. The Ptolemy II implementation is similar to that by Kahn and MacQueen [7]. In that model, just like CSP, the producer and consumer are separate threads executing the fire() method. Unlike CSP, however, the producer can send data and proceed without waiting for the receiver to be ready to receive data. This is implemented by a non-blocking write to a FIFO queue with (conceptually) unbounded capacity. The put() method in a PN receiver always succeeds and always returns immediately. The get method, however, blocks the calling thread if no data is available. To maintain determinacy, it is important that processes not be able to test a receiver for the presence of data. So the hasToken() method always returns *true*. Indeed, this return value is correct, since the get() method will never throw a NoTokenException. Instead, it will block the calling thread until a token is available.
- *Synchronous Data Flow (SDF)*: This domain supports a synchronous dataflow model of computation [9]. This is different from the thread-based domains in that the producer and consumer are implemented as finite computations (firings of a dataflow actor) that are scheduled (typically statically, and typically in the same thread). In this model, a consumer assumes that data is always available when it calls get() because it assumes that it would not have been scheduled otherwise. The capacity of the receiver can be made finite, statically determined, but the

scheduler ensures that when put() is called, there is room for a token. Thus, if scheduling is done correctly, both get() and put() succeed immediately and return.

- *Discrete Event (DE)*: This domain uses timed events to communicate between actors. Similar to SDF, actors in the DE domain implement finite computations encapsulated in the fire() method. However, the execution order among the actors is not statically scheduled, but determined at run time. Also, when a consumer is fired, it cannot assume that data is available. Very often, when an actor with multiple input ports is fired, only one of the ports has data. Therefore, for an actor to work correctly in this domain, it must check the availability of a token using the hasToken() method before attempting to get a token from the receiver.

As can be seen, different domains impose different requirements for actors. Some actors, however, can work in multiple domains. These actors are called *domain-polymorphic* actors. One of the goals of the system-level type system is to facilitate the design of domain-polymorphic actors.

In Ptolemy II, there are more than ten domains implementing various models of computation, including the ones discussed above. One of these domains implements interface automata.

3 Overview of Interface Automata

3.1 An Example

Interface automata [4] are a light-weight formalism for the modeling of components and their environments. As other automata models, interface automata consist of states and transitions[1], and is usually depicted by bubble-and-arc diagrams. There are three different kinds of transitions in interface automata: input, output, and internal transitions. When modeling a software component, input transitions correspond to the invocation of methods on the component, or the returning of method calls from other components. Output transitions correspond to the invocation of methods on other components, or the returning of method calls from the component being modeled. Internal transitions correspond to computations inside the component.

For example, figure 2 shows the interface automata model for an implementation of the consumer actor in figure 1. This figure is a screen shot of Ptolemy II. The convention in interface automata is to label the input transitions with an ending "?", the output transitions with an ending "!", and internal transitions with an ending ";". Figure 2 does not contain any internal transitions. The block arrows on the sides of figure 2 denote the inputs and outputs the automaton. They are:

- *f*: the invocation of the fire() method of the actor.
- *fR*: the return from the fire() method.
- *g*: the invocation of the get() method of the receiver at the input port of the actor.
- *t*: the token returned in the get() call.
- *hT*: the invocation of the hasToken() method of the receiver.
- *hTT*: the value *true* returned from the hasToken() call, meaning that the receiver

[1] Transitions are called actions in [4].

contains one of more tokens.
- *hTF*: the value *false* returned from the hasToken() call, meaning that the receiver does not contain any token.

The initial state is state 0. When the actor is in this state, and the fire() method is called, it calls get() on the receiver to obtain a token. After receiving the token in state 3, it performs some computation, and returns from fire(). This example illustrates an important characteristic of interface automata. That is, they do not require all the states to accept all inputs. In figure 2, the input *f* is only accepted in state 0, but not in any other states. This is opposed to other automata-based formalisms, such as I/O automata [11], where every input must be enabled at every state. By not requiring the model to be input enabled, interface automata models are usually more concise, and do not include states that model error conditions. In fact, interface automata take an optimistic approach for modeling, and they reflect the intended behavior of components under a good environment.

In the SDF domain, an actor assumes that its fire() method will not be called again if it is already inside this method. Also, the scheduler guarantees that data is available when a consumer is fired, so the transition from state 2 to state 3 assumes that the receiver will return a token. An error condition, such as the receiver throws NoTokenException when get() is called, is not explicitly described in the model.

3.2 Composition and Compatibility

Two interface automata can be composed if their transitions are disjoint, except that an input transition of one may coincide with an output transition of the other. These overlapping transitions are called shared transitions. Shared transitions are taken synchronously, and they become internal transitions in the composition. Figure 3 shows two automata that can be composed with the automaton in figure 2. These two automata do not correspond to real Ptolemy II components, we just use them to illustrate automata composition. The *DirectorA* in figure 3(a) calls the fire() method of the SDFActor, then expects the call to return. When composed with the *SDFActor* automaton, *f* and *fR*

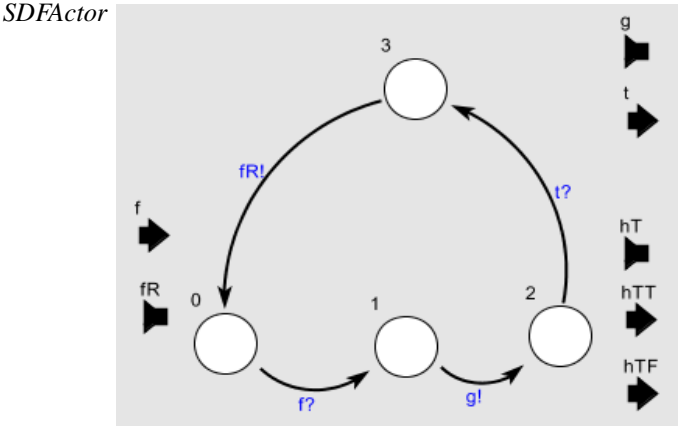


Fig. 2. Interface automaton model for an SDF actor.

are shared transitions, and the composition result is shown in figure 4(a). The *DirectorB* in figure 3(b) may call the fire() method again before the first call returns. When this automaton is composed with *SDFActor*, it may issue an output that the *SDFActor* does not accept. For example, when both automata are in state 1, *DirectorB* may issue *f*, which *SDFActor* does not accept. This means that the pair of states (1, 1) in the composition (*DirectorB*, *SDFActor*) is *illegal*.

In interface automata, illegal states are pruned out in the composition. Furthermore, all states that can reach illegal states through output or internal transitions are also pruned out. This is because the environment cannot prevent the automata from entering illegal states from these states. As a result, the composition of *DirectorB* and *SDFActor* is an empty automaton without any states, as shown in figure 4(b). This is a key property of interface automata. More conventional automaton composition always results in a state space that is the product of the composed state spaces, and hence is significantly larger. Interface automata often compose to form smaller automata.

The above examples illustrate the key notion of *compatibility* in interface automata. Two automata are compatible if their composition is not empty. This notion gives a

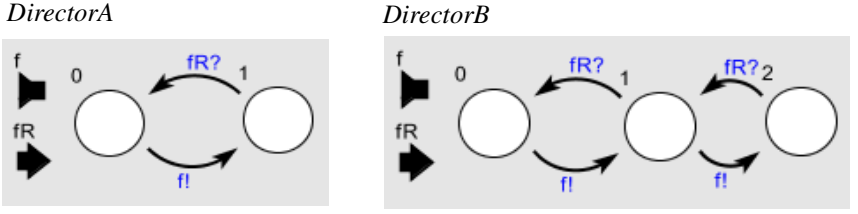


Fig. 3. Two (artificial) director automata.

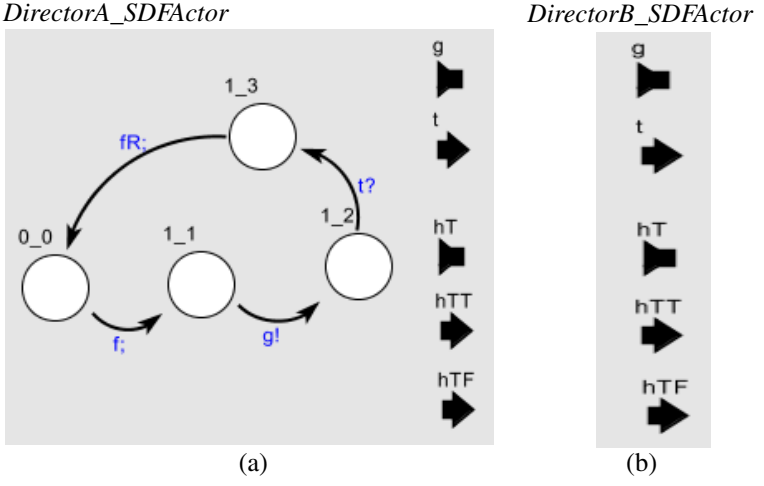


Fig. 4. The composition of the artificial directors (figure) with *SDFActor* (figure 2).

formal definition for the informal statement "two components can work together". The composition automaton defines exactly how they can work together. In system-level types, we use interface automata to describe various communication protocols, or the interaction types for components. To check whether a certain component is compatible with a communication protocol, we can simple compose the automata models of the component and the protocol, and check whether the result is empty. This yields a straightforward algorithm for type checking, which is the main attraction of interface automata to system-level types.

The approach to composition in interface automata is optimistic. If two components are compatible, there is some environment that can make them work together. In the traditional pessimistic approach, two components are compatible if they can work together in all environments. Because of this difference, the composition of interface automata is usually smaller than the composition in other automata models. Before adopting interface automata, we also attempted to describe system-level types using a more traditional finite state machine model [10]. Compatibility checking in that setting proved to be more difficult.

3.3 Alternating Simulation

Interface automata have a notion of *alternating simulation*, which is used in our type system to define subtyping. Informally, for two interface automata P and Q, there is an alternating simulation relation from Q to P if all the input steps of P can be simulated by Q, and all the output steps of Q can be simulated by P. The formal definition is given in [4]. A theorem states that if a third automaton R is compatible with P, and the input transitions of Q that are shared with the output transitions of R is a subset of the input transitions of P that are shared with the output transitions of R, then Q and R are also compatible.

4 System-Level Types

4.1 Type Definition

The automaton *SDFActor* in figure 2 describes a consumer actor designed for the SDF (synchronous dataflow) domain. The automaton shown in figure 5 describes an actor that can operate in wider variety of domains. Since this actor is not designed under the assumption of the SDF domain, it does not assume that data are available when it is fired. Instead, it calls hasToken() on the receiver to check the availability of a token. If hasToken() returns false, it immediately returns from fire(). This is a simple form of domain-polymorphism.

In Ptolemy II, actors interact with the director and the receivers of a domain. In figures 2 and 5, the block arrows on the left side denote the interface with the director, and the ones on the right side denote the interface with the receiver. As discussed in section 2, the implementation of the director and the receiver determines the semantics of component interaction in a domain, including the flow of control and the communication protocol. If we use an interface automaton to model the combined behavior of the director and the receiver, this automaton is then the type signature for the domain. Figure 6 shows such an automaton for the SDF domain. Here, p and pR represent the

call and the return of the put() method of the receiver. This automaton encodes the assumption of the SDF domain that the consumer actor is fired only after a token is put into the receiver[2].

PolyActor

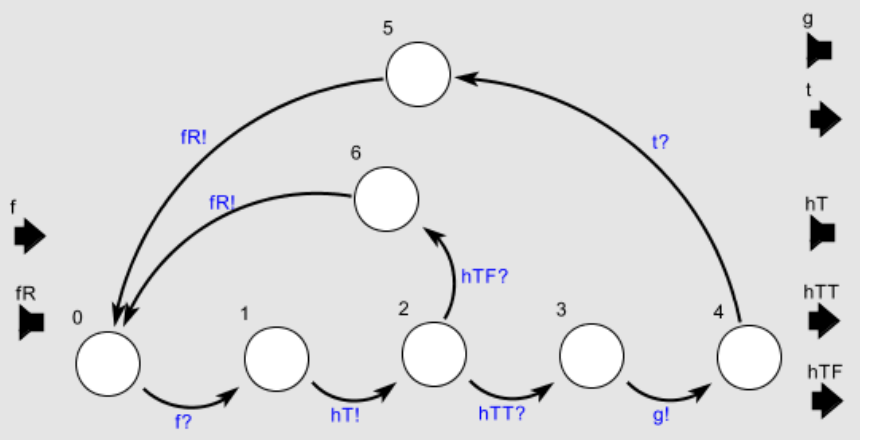


Fig. 5. Interface automaton for a domain-polymorphic actor.

SDFDomain

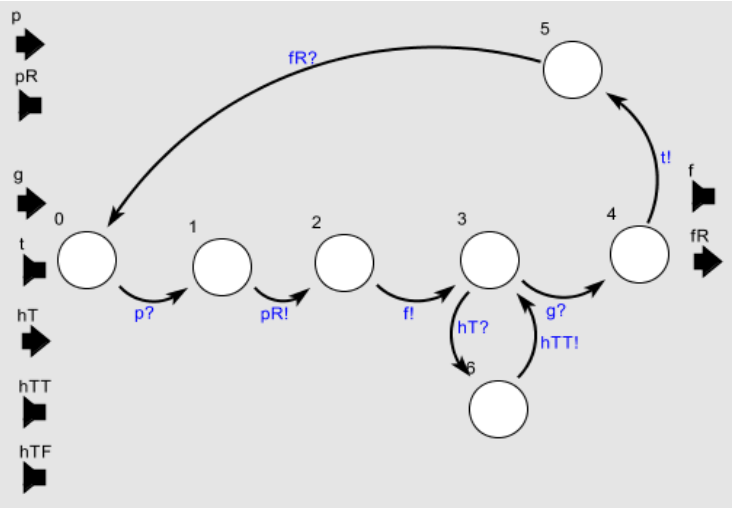


Fig. 6. Type signature of the SDF domain.

[2] This is a simplification of the SDF domain, since an actor may require more than one token to be put in the receiver before it is fired. This simplification makes our exposition clearer, but otherwise makes no material difference.

The type signature of the DE domain is shown in figure 7. In DE, an actor may be fired without a token being put into the receiver at its input. This is indicated by the transition from state 0 to state 7. Figures 6 and 7 also reflect the fact that both of the SDF and the DE domains have a single thread of execution, so the hasToken() query may happen only after the actor is fired but before it calls get(), during which time the actor has the thread of control.

CSP and PN are two domains in Ptolemy II in which each actor runs in its own thread. Figures 8 and 9 give the type signature of these two domains. These automata are simplified from the true implementation in Ptolemy II. In particular, CSPDomain omits conditional rendezvous, which is an important feature in the CSP model of computation. In the CSP and PN domains, an actor is fired repeatedly by its thread, as modeled by the transitions between state 0 and 1.

In CSP, the communication is synchronous; the first thread that calls get() or put() on the receiver will be stalled until the other thread calls put() or get(). The case where get() is called before put() is modeled by the transitions among the states 1, 3, 4, 5, 1. The case where put() is called before get() is modeled by the transitions among the states 1, 6, 8, 9, 1.

In PN, the communication is asynchronous. So the put() call always returns immediately, but the thread calling get() may be stalled until put() is called. The case where get() is called first in PN is modeled by the transitions among the states 1, 3, 4, 5, 1 in figure 9, while the case where put() is called first is modeled by the transitions among the states 1, 6, 8, 10, 1.

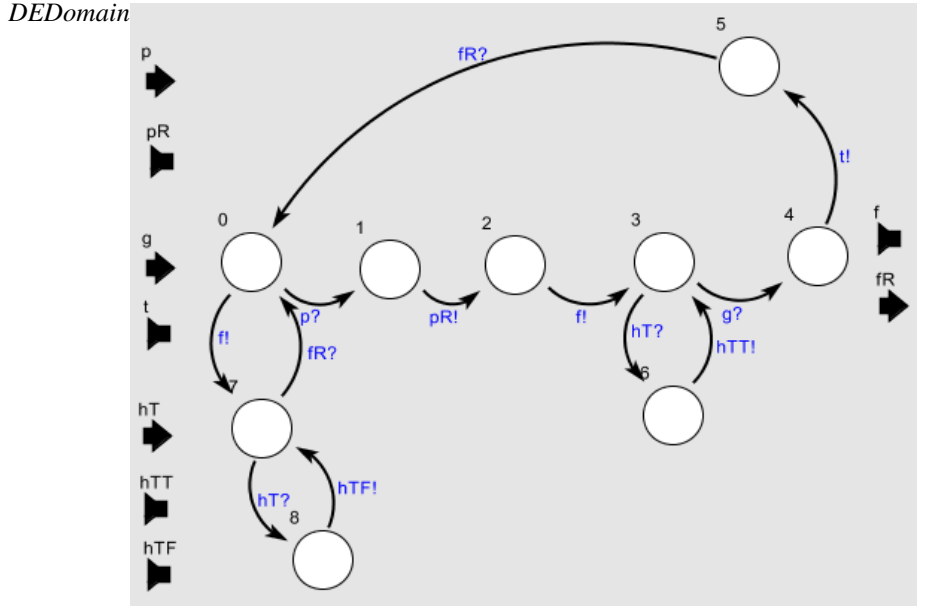


Fig. 7. Type signature of the DE domain.

Given an automaton modeling an actor and the type signature of a domain, we can check the compatibility of the actor with the communication protocol of that domain by composing these two automata. Type checking examples will be shown below in section 4.3.

SPDomain

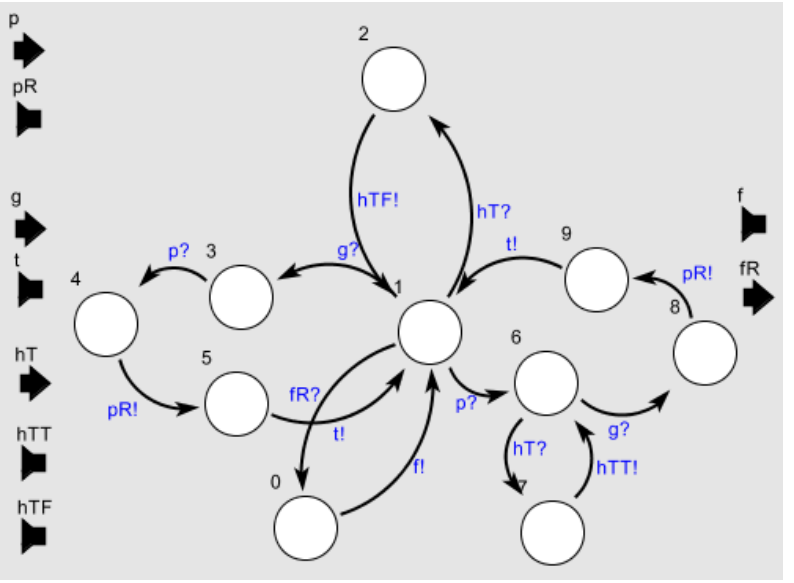


Fig. 8. Type signature of the CSP domain.

PNDomain

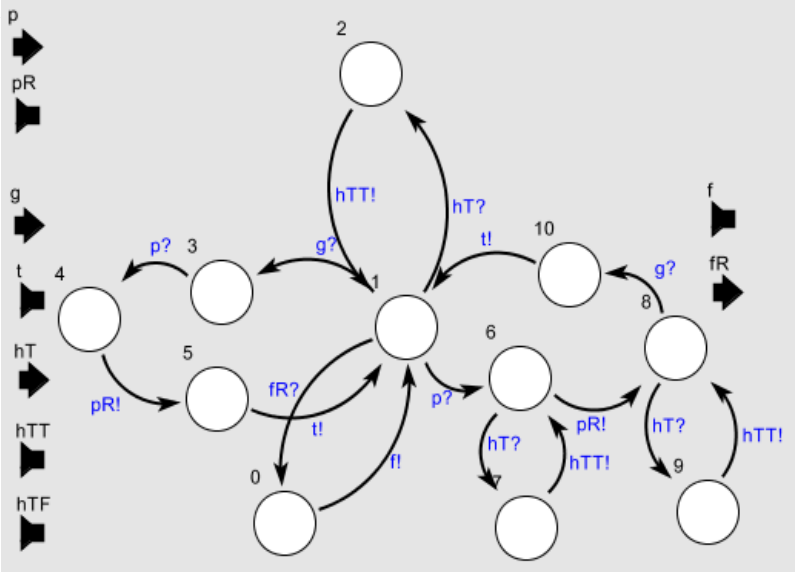


Fig. 9. Type signature of the PN domain.

4.2 System-Level Type Order and Polymorphism

If we compare the SDF and DE domain automata, we can see that they are closely related. This relationship can be captured by the alternating simulation relation of interface automata. In particular, there is an alternating simulation relation from SDF to DE.

In the set of all domain types, the alternating simulation relation induces a partial order, or system-level type order. An example of this partial order is shown in figure 10. From a type system point of view, this order is the subtyping hierarchy for the domain types. If we view the automata as functions with inputs and outputs, then the alternating simulation relation is exactly analogous to the standard function subtyping relation in data type systems. According to the definition of alternating simulation, the automaton lower in the hierarchy can simulate all the input steps of the ones above it, and the automaton higher in the hierarchy can simulate all the output steps below it. In function subtyping, if a function $A \rightarrow B$ is a subtype of another function $A' \rightarrow B'$, then A' is a subtype of A and B is a subtype of B' [1]. Note that in both relations, the order is inverted (contravariant) for the inputs and goes in the same direction (covariant) for the outputs.

In [4], alternating simulation is used to capture the refinement relation from the specification to the implementation of components. Our use of this relation is not for refinement, but for subtyping. In the system-level type order, *SDFDomain* is not a refinement of *DEDomain*, but a subtype of *DEDomain*. In fact, *SDFDomain* has fewer states than *DEDomain*. This subtyping relation can help us design actors that can work in multiple domains. According to the theorem in section 3, if an actor is compatible with a certain domain *D*, and there are other domains below D in the system-level type order, then the actor is also compatible with those lower domains. Therefore, this actor is domain polymorphic.

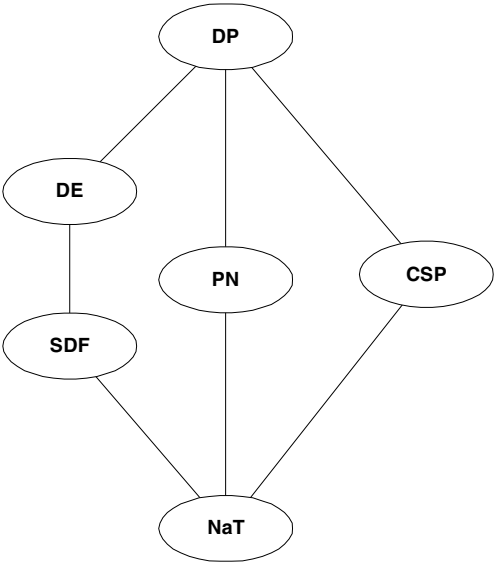


Fig. 10. An example of system-level type order.

In figure 10, we added a bottom and a top element to the type hierarchy. The name of the bottom element *NaT* stands for "not a type", and the name of the top element *DP* stands for "domain polymorphic". The DP automaton has an alternating simulation relation from all the domain-specific automata. So if an actor is compatible with this automaton, it is compatible with all the domains. The exact design of this automaton is part of our future research, and depends on the set of domains to be included.

Note that by adding the top and bottom elements, the system-level type order becomes a lattice. In a lattice, every subset of elements has a least upper bound and a greatest lower bound. It might be possible to explore this fact in the type system, as we have done for data types [15].

4.3 Type Checking Examples

Let's perform a few type checking operations using the actors and domains in the earlier sections. To verify that the *SDFActor* in figure 2 can indeed work in the *SDFDomain*, we compose it with the *SDFDomain* automaton in figure 6. The result is shown in figure 11. As expected, the composition is not empty so *SDFActor* is compatible with *SDFDomain*.

Now let's compose *DEDomain* with *SDFActor*. The result is an empty automaton shown in figure 12. This is because the actor may call get() when there is no token in the receiver, and this call is not accepted by an empty DE receiver. The exact sequence

SDFDomain_SDFActor

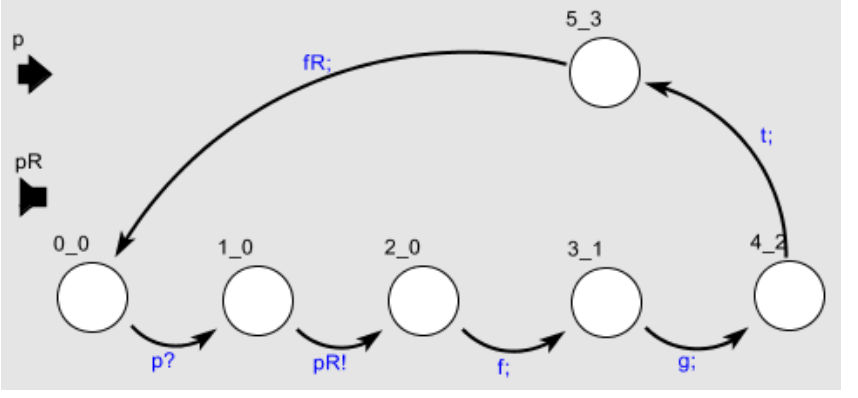


Fig. 11. Composition of *SDFDomain* in figure and *SDFActor* in figure 2.

DEDomain_SDFActor

Fig. 12. Composition of *DEDomain* in figure and *SDFActor* in figure 2.

that leads to this condition is the following: first, both automata take a shared transition *f*. In this transition, *DEDomain* moves from state 0 to state 7, and *SDFActor* moves from state 0 to state 1. At state 1, *SDFActor* issues *g*, but this input is not accepted by *DEDomain* at state 7. So the pair of states (7, 1) in (*DEDomain*, *SDFActor*) is illegal. Since this state can be reached from the initial state (0, 0), the initial state is pruned out from the composition. As a result, the whole composition is empty. This means that the SDF actor cannot be used in DE Domain.

The *PolyActor* in figure 5 checks the availability of a token before attempting to read from the receiver. By composing it with *DEDomain*, we verify that this actor can be used in the DE Domain. This composition is shown in figure 13. Since *SDFDomain* is below *DEDomain* in the system-level type order of figure 10, we have also verified that *PolyActor* can work in the SDF domain. Therefore, *PolyActor* is domain polymorphic. As a sanity check, we have composed *SDFDomain* with *PolyActor*, with the result is shown in figure 14.

We have also composed *PolyActor* and *SDFActor* with *CSPDomain* and *PNDomain*. The result shows that these two actors can be used in both domains. This is not surprising, because both domains have blocking read semantics, so the actors will work regardless of whether they check the availability of a token before calling get(). For the sake of brevity, we do not include these compositions in this paper.

In Ptolemy II, there is a library of about 100 domain-polymorphic actors. The communication behavior for many of these actors can be modeled by the *PolyActor* automaton.

DEDomain_PolyActor

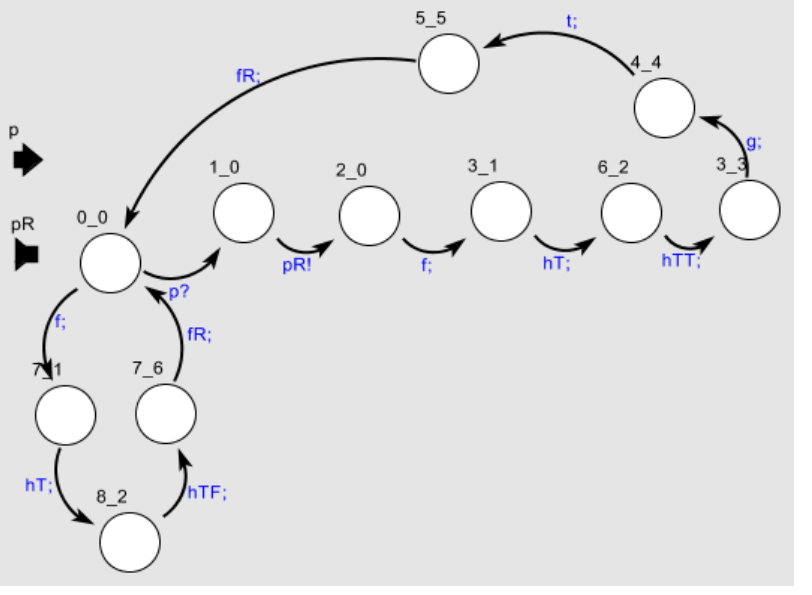


Fig. 13. Composition of *DEDomain* in figure and *PolyActor* in figure 5.

4.4 Reflection

So far, interface automata have been used to describe the operation of Ptolemy II components. These automata can be used to perform compatibility checks between components. Another interesting use is to reflect the component state in a run-time environment. For example, we can execute the automaton *SDFActor* of figure 2 in parallel with the execution of the actor. When the fire() method of the actor is called, the automaton makes a transition from state 0 to state 1. At any time, the state of the actor can be obtained by querying the state of the automaton. Here, the role of the automaton is reflection, as realized for example in Java. In Java, the Class class can be used to obtain the static structure of an object, while our automata reflect the dynamic behavior of a component. We call an automaton used in this role a *reflection automaton*.

5 Trade-Offs in Type System Design

The examples in the previous section focus on the communication protocol between a single actor and its environment. This scope can be broadened by including the automata of more actors and using a more detailed director model in the composition. Also, properties other than the communication protocol, such as deadlock freedom in thread-based domains, can be included in the type system. However, these extensions will increase the cost of type checking. So there is a trade-off between the amount of information carried by the type system and the cost of type checking.

Another dimension of the trade-offs is static versus run-time type checking. The examples in the last section are static type checking examples. If we extend the scope of the type system, static checking can quickly become impractical due to the size of the composition. An alternative is to check some of the properties at run time. One way to perform run-time checking is to execute the reflection automata of the components in parallel with the execution of the components. Along the way, we periodically check the states of the reflection automata, and see if something has gone wrong.

These trade-offs imply that there is a big design space for system-level types. In this space, one extreme point is completely static checking by composing the automata modeling all the system components, and check the composition. This amounts to model checking. To explore the boundary in this direction, we did an experiment by

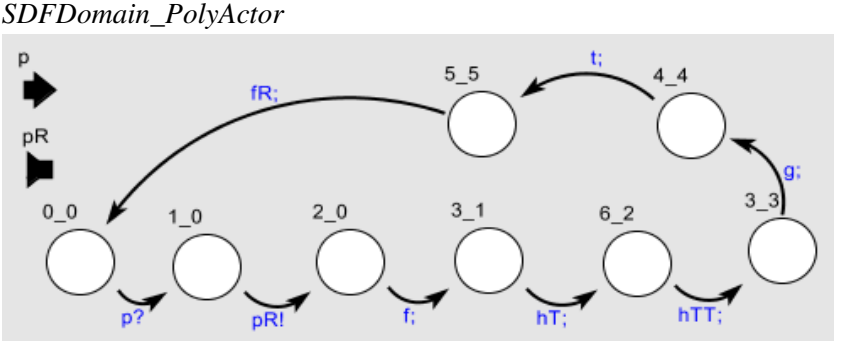


Fig. 14. Composition of *SDFDomain* in figure and *PolyActor* in figure 5.

checking an implementation of the classical dining philosophers model implemented in the CSP domain in Ptolemy II. Each philosopher and each chopstick is modeled by an actor running in its own thread. The chopstick actor uses conditional send to simultaneously check which philosopher (the one on its left or the one on its right) wants to pick it up. We created interface automata for the Ptolemy II components *CSPReceiver*, *Philosopher*, and *Chopstick*, and a simplified automaton to model conditional send. We are able to compute the composition of all the components in a two-philosopher version of the dining philosopher model, and obtain a closed automaton with 2992 states. Since this automaton is not empty, we have verified that the components in the composition are compatible with respect to the synchronous communication protocol in CSP. We also checked for deadlock inherent in the implementation, and are able to identify two deadlock states in the composition, which correspond to the situation where all the philosophers are holding the chopsticks on their left and waiting for the ones on the right, and the symmetrical situation where all philosophers are waiting for the chopsticks on their left.

Our goal here is not to do model checking, but to perform static type checking on a non-trivial models. Obviously, when the model grows, complete static checking will become intractable due to the well-known state explosion problem.

Another extreme point in the design space for system-level types is to rely on run-time type checking completely. For deadlock detection, we can execute the reflection automata in parallel with the Ptolemy II model. When the model deadlocks, the states of the automata will explain the reason for the deadlock. In this case, the type system becomes a debugging tool. The point here is that a good type system is somewhere between these extremes. We believe that a system that checks the compatibility of communication protocols, as illustrated in sections 4, is a good starting point.

6 Conclusion and Future Work

We have described a type system that captures the interaction dynamic in a component-based design environment. The interaction types and component behavior are described by interface automata, and type checking is done through automata composition. Our approach is domain polymorphic in that a component may be compatible with multiple interaction types. The relation among the interaction types is captured by a system-level type order using the alternating simulation relation of interface automata. We have shown that our system can be extended to capture more dynamic properties, and the design of a good type system involves a set of trade-offs. Our experimental platform is the Ptolemy II design environment. All the automata in the paper are built in Ptolemy II and their compositions are computed in software, except that some manual layout is applied for better readability of the diagrams.

We also proposed using automata to do on-line reflection of component states. In addition to run-time type checking, the resulting reflection automata can add value in a number of ways. For example, in a reconfigurable architecture or distributed system, the state of the reflection automata can provide information on when it is safe to perform mutation. Reflection automata can also be valuable debugging tools. This is part of our future work.

In addition to its usual use in type checking, our type system may facilitate the design of new components or Ptolemy II domains. In Ptolemy II, domains can be combined hierarchically in a single model. Using system-level types, it might be possible to show that the composition of a domain director and a group of actors behaves like a polymorphic actor in some other domains. This is also part of our future research.

Acknowledgments. We thank Xiaojun Liu for his help in the implementation of interface automata in Ptolemy II. This work is part of the Ptolemy project, which is supported by the Defense Advanced Research Projects Agency (DARPA), the State of California MICRO program, and the following companies: Agilent, Cadence, Hitachi, and Philips.

References

1. L. Cardelli and P. Wegner, "On Understanding Types, Data Abstraction, and Polymorphism," *ACM Computing Surveys*, Vol.17, No.4, Dec. 1985.
2. B. A. Davey and H. A. Priestly, *Introduction to Lattices and Order*, Cambridge University Press, 1990.
3. J. Davis II, C. Hylands, B. Kienhuis, E.A. Lee, J. Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Tsay, B. Vogel, and Y. Xiong, "Heterogeneous Concurrent Modeling and Design in Java," *Technical Memorandum UCB/ERL M01/12*, EECS, University of California, Berkeley, March 15, 2001. (http://ptolemy.eecs.berkeley.edu/publications/papers/01/HMAD/)
4. L. de Alfaro and T. A. Henzinger, "Interface Automata," to appear in *Proc. of the Joint 8th European Software Engineering Conference and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (ESEC/FSE 01)*, Austria, 2001.
5. C. A. R. Hoare, "Communicating Sequential Processes," *Communications of the ACM*, 28(8), August 1978.
6. G. Kahn, "The Semantics of a Simple Language for Parallel Programming," *Proc. of the IFIP Congress 74*, North-Holland Publishing Co., 1974.
7. G. Kahn and D. B. MacQueen, "Coroutines and Networks of Parallel Processes," *Information Processing 77*, B. Gilchrist, editor, North-Holland Publishing Co., 1977.
8. E. A. Lee, "Computing for Embedded Systems," *IEEE Instrumentation and Measurement Technology Conference*, Budapest, Hungary, May 21-23, 2001.
9. E. A. Lee and D. G. Messerschmitt, "Synchronous Data Flow," *Proc. of the IEEE*, Sept., 1987.
10. E. A. Lee and Yuhong Xiong, "System-Level Types for Component-Based Design," *Technical Memorandum UCB/ERL M00/8*, EECS, University of California, Berkeley, Feb. 29, 2000. (http://ptolemy.eecs.berkeley.edu/publications/papers/00/systemLevel/)
11. N. Lynch and M. Tuttle, "Hierarchical Correctness Proofs for Distributed Algorithms," *Proc. 6th ACM Symp. Principles of Distributed Computing*, pp 137-151, 1981.
12. J. C. Mitchell, "Coercion and Type Inference," *11th Annual ACM Symposium on Principles of Programming Languages*, 175-185, 1984.
13. M. Odersky, "Challenges in Type System Research," *ACM Computing Surveys*, 28(4), 1996.
14. H. Xi and F. Pfenning, "Eliminating Array Bound Checking Through Dependent Types," *Proc. of ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI '98)*, pp. 249-257, Montreal, June, 1998.
15. Y. Xiong and E. A. Lee, "An Extensible Type System for Component-Based Design," *6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Berlin, Germany, March/April 2000. LNCS 1785.

Embedded Software Implementation Tools for Fully Programmable Application Specific Systems

Sharad Malik

Princeton University

Abstract. A variety of diverse pressures are shaping how we will design digital systems in the near future. Shrinking geometries into the deep submicron range raise electrical design challenges that make it impossible to use existing methodologies for application specific system design. In addition, the corresponding exponential increase in the number of devices per chip results in a complexity problem which by itself threatens to cripple existing design methodologies. Finally, increased non-recurring engineering costs for masks and design tools force designs to be limited to higher volume products. All of these point to a gradual reduction of designs done using conventional ASIC (application specific integrated circuits) design methodology.

The above forces are resulting in an increase in systems that contain programmable components that are specialized for a specific application domain, while at the same time providing design flexibility that permits the same device to be used for a range of related products and also generations of a product. These systems need to satisfy one or more of high computation, low power, and real-time constraints. Architecturally, these constraints are met by exploiting concurrency at all levels - at the bit operation level through specialized functional units, at the instruction level through instruction level parallelism and at the coarse grain level through on-chip multiprocessing.

This class of highly specialized embedded systems requires specialized software implementation tools to ensure an efficient mapping of the application onto the architecture. These tools - particularly the compiler and simulator - are needed for two distinct aspects of the design. During architectural evaluation, they are needed to provide feedback about the suitability of the architecture for the application. During system implementation, they are needed to ensure efficient mapping and validation of design constraints. These functions result in the following requirements:

Retargetability: Architectural evaluation requires the ability to compile the application to different architectural choices and evaluate, through simulation or other analysis, the choices for the metrics of interest. This requires the mapping tools (compiler) and the evaluation tools (simulator) to be rapidly retargetable to each architectural choice being explored. This poses several challenges both on the architectural specification front, as well as the tool synthesis front. In

T.A. Henzinger and C.M. Kirsch (Eds.): EMSOFT 2001, LNCS 2211, pp. 254–256, 2001.
© Springer-Verlag Berlin Heidelberg 2001

terms of architectural specification, the ability to specify an interesting range of processors for use in both compilation as well as simulation is a hard problem which has not seen much success despite some previous efforts. A retargetable simulator needs execution semantics while a retargetable compiler needs information on the constrained usage of resources. Providing both in a single consistent description is hard.

The problem is difficult enough for single processor - the fact that the systems of interest are multi-processing systems further exacerbates it. For a single processor, we have some notion of what primitives need to be described in the architectural specification (instruction information, registers, pipelines, resource reservation tables etc.). However, given the wide range of single-chip multiprocessor systems emerging (bus based, packet switch based), it is hard to even determine the primitives for describing such systems.

High Speed Simulation: Architectural evaluation requires the ability to explore several architectural choices for a range of possible metrics of interest. This typically involves simulating the application (or at least several key parts of it) on the design points of interest. A bottleneck in this entire process is the simulation speed. While developing accurate simulators for processors is not new, it is made complicated in our context by two factors. The first is the requirement for retargetability which has already been touched above. The second is the need to handle multiprocessing using possibly complex communication architectures.

Concurrency Mapping: A critical aspect of the implementation is mapping the concurrency present in the application to the concurrency available in the architecture. This involves handling both the explicit as well as the implicit concurrency. The explicit concurrency is specified through threads/processes for coarse grained concurrency, and specialized functions for bit level concurrency. The implicit fine grained concurrency typically needs to be extracted as part of the compilation process.

Efficient Compilation: In addition to handling fine grained concurrency mapping (instruction level parallelism) from retargetable descriptions, the compilation needs to deal with specialized storage (register files and memory) as well as data paths. While some traditional compilation techniques can be used here, a host of new techniques need to be developed that are capable of handling irregular constraints in a retargetable environment.

Validation of Power Constraints: Power/Energy is increasingly becoming a first class citizen in the design process, and in many cases the limiting factor in designs. Given that, it is important that both the synthesis (compiler) and evaluation (simulator) tools can handle this metric. This implies accurate power models for both, as well as optimization algorithms for use in the compiler. As with the other metrics, this needs to be handled in a retargetable environment.

Validation of Real-Time Constraints: Most application domains under consideration have real-time requirements. Soft requirements can be validated through simulation, however hard requirements re-

quire formal analysis that needs to be built as part of the software development environment to enable to compiler to generate code that meets these constraints.

The MESCAL (Modern Embedded Systems: Compilers, Architectures and Languages) project is focussed on developing a complete set of design tools for architectural exploration, architectural evaluation as well as system implementation. This is joint work with David August at Princeton and Kurt Keutzer and Richard Newton at UC Berkeley. This presentation will focus on the software implementation tools and associated methodology used for the latter two aspects, viz. architectural evaluation and system implementation, and describe some solutions to the challenges posed by the above requirements.

Compiler Optimizations for Adaptive EPIC Processors

Krishna V. Palem[1], Surendranath Talla[1⋆], and Weng-Fai Wong[2]

[1] Center for Research on Embedded Systems and Technology,
Georgia Institute of Technology
palem@ece.gatech.edu

[2] Dept. of Computer Science,
National University of Singapore

Abstract. Advances in VLSI technology have lead to a tremendous increase in the density and number of devices that can be manufactured in a single microchip. One of the interesting ways in which this silicon may be used is to leave portions of it uncommitted and re-programmable depending on an applications needs. In an earlier paper, we proposed a machine architecture for achieving this reconfigurability and compilation issues that such an architecture will face. In this paper, we will elaborate on the compiler optimization issues involved. In particular, we will outline a framework for code partitioning, instruction synthesis, configuration selection, resource allocation, and instruction scheduling. Partitioning is the problem of identifying code sections that may benefit by mapping them on to the programmable logic resources. The instruction synthesis phase generates suitable implementations for the candidates partitions and updates the machine description database with the new instructions. Configuration selection is the problem of narrowing down the choices of which synthesized instruction (from the set generated by the instruction synthesis phase) to use for each of the code regions that will be mapped to programmable logic. Unlike traditional optimizing compilers, the adaptive EPIC compiler must deal with the existence of synthesized instructions. Compilation techniques addressing each of these problems will be presented.

1 Introduction

Phenomenal advances in semiconductor technology have made it possible to put an increasing amount of silicon devices into the same surface area. This dramatic increase in device density has brought up a fundamental question: how can the extra silicon be effective employed to improve the execution performance of applications? Many straightforward ideas like increasing the number of functional units or the size of the caches run into the problem of diminishing returns.

⋆ Author's current affiliation: StarCore Technology Center, Atlanta, GA 30328.

T.A. Henzinger and C.M. Kirsch (Eds.): EMSOFT 2001, LNCS 2211, pp. 257–273, 2001.
© Springer-Verlag Berlin Heidelberg 2001

One interesting proposal is to keep a portion of the silicon uncommitted—as re-programmable logic. Processors demonstrating this design, having a core RISC processor and an on-chip, tightly coupled re-programmable logic pool, have been introduced[27]. Depending on the granularity of the re-programmable logic portion, these processors can be used in two ways. The first approach that was adopted by the early generation of such processors is to go for mapping larger granularity computations on the re-programmable logic. The re-programmable logic essentially implements a significantly large grain computation and interfaces it to the application executing in the core is of the nature of a subroutine call. Programmers are expected to know the syntax and semantics of such subroutines and to write the subroutine calls into their application. The alternative that is generally still under-researched is to use the re-programmable logic to implement application-specific instructions. This finer grain approach requires a greater involvement of the compiler. This approach is interesting in that it is an automated approach. On the flip side, we will need a different variation of the generic RISC-core with re-programmable logic processor to support fine grain operations as well as new compiler algorithms.

In this paper, we will describe our proposal for an *Adaptive Explicitly Parallel Instruction Computer* (AEPIC) processor. The bulk of the paper, however, will be devoted to new compiler algorithms needed to generate code for such a machine.

2 Previous Work

The earliest known computing system based on reconfigurable devices wa proposed and implemented by Gerald Estrin at UCLA [11]. It is a hybrid machine consisting of a general purpose processor augmented with high speed logic devices (ALU's, memories) which were interconnected via application specific interconnect. Due to a lack of enabling technology, the reconfiguration was done manually. Mario Schaffner's Circulating Page Structure (CPS) machine [23] implemented a form of hardware paging scheme where the application task was partitioned into pages which circulate through the programmable hardware to compute the task.

The introduction of *field programmable gate array* (FPGA) devices by Xilinx in the mid 80's [32,26] spurred a flurry of research in the development of FPGA based reconfigurable computing engines. PRISM [2] developed at Brown University demonstrates substantial speedup in the case of large binary operations. PAM, a universal reconfigurable hardware co-processor developed by researchers at DEC Paris Research Labs [3,4,29], has been used to demonstrate superior performance/cost ratio compared to every other existing technology of its time on a dozen applications ranging from computer arithmetic, cryptography, image analysis, neural networks, video compression, high-energy physics, biology and astronomy. Another such reconfigurable co-processor board developed by Super Computing Research Center at Maryland called SPLASH-2 [12] has been used to achieve two orders of magnitude speedup on genome sequence matching

compared to supercomputers of that time (Cray2). The cover story of an issue of Scientific American [28] written by researchers at UCLA outlines some novel applications of reconfigurable devices.

Other notable reconfigurable computing projects include the Programmable Reduced Instruction Set Computer (PRISC) [21,22], GARP [16], DISC [31], RAPID [13], the CMU Cached Virtual Hardware (CVH), PipeRench [5], and Chimaera [14,15].

Reconfigurable Architecture Workstation (RAW) proposed by Agarwal and his colleagues at MIT [30,19,1] consists of processors that are sets of interconnected tiles each of which contains instruction and data memories, an arithmetic-logic unit, registers, configurable logic, and a programmable switch that supports both dynamic and static (compiler orchestrated) routing, interconnected by programmable interconnects. Compiling to a RAW machine is complicated by two factors:

- Unlike traditional super-scalars, a RAW processor does not bind specialized logic structures such as register renaming or dynamic instruction issue logic into hardware. Scheduling and resource allocation are the responsibility of the compiler.
- Communication patterns within the code need to be analyzed in order to schedule inter-tile communication.

Preliminary experience with compiling to the RAW machine can be found in [1, 19].

So far, most research efforts have focused on the architectural aspects of reconfigurable systems. Little attention has been paid to compilation issues. Many performance studies have been done and impressive speedups were demonstrated. However, important issues like compilation times, target cost have been neglected. Our research attempts to address these latter issues.

3 Adaptive Explicitly Parallel Instruction Computing (AEPIC)

Processors that take advantage of instruction level parallelism generally fall into two main categories: superscalar (dynamically scheduled) and VLIW (statically scheduled). With the addition of features such as predicated execution, support for software pipelining etc., the latter has been renamed Explicitly Parallel Instruction Computer (EPIC). We refer to Schlansker and Rau [24] for an evaluation of these two ILP approaches. It is to EPIC that we propose adding a reconfigurable component that is amenable to compiler optimizations. We call this new configuration Adaptive EPIC (AEPIC) [25]. Figure 1 shows the abstract execution model for AEPIC. The "hardwired component" in Figure 1 is the EPIC core processor. The adaptive component consists of functional units that have been configured into the datapath by some reconfiguration instructions. Operations performed by the configured functional units are triggered by specific AEPIC instructions invoked on the hardwired functional units.

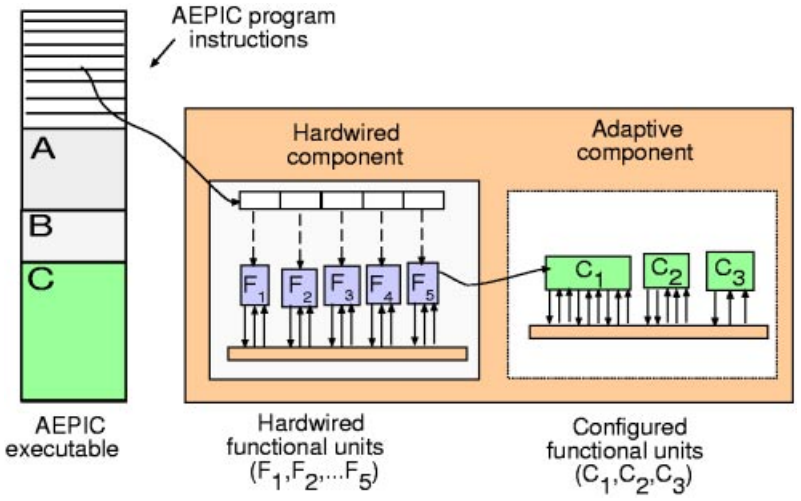


Fig. 1. AEPIC executable and abstract data-path

Figure 2 shows the details of a AEPIC machine. The core component consists of a standard EPIC machine. The adaptive component of the AEPIC processor consists of the *Configuration Cache Hierarchy*, *Multi- context Reconfigurable Logic Array* (MRLA) and *Array Register File* (ARF) connected together via bus interconnect. The MRLA provides the programmable logic resources to host the *Configured Functional Units* (CFUs). The *C-Cache* serves as a temporary cache for configurations before they are instantiated on the MRLA. This is analogous to the way registers serve as storage for program values. The rest of the configuration cache hierarchy consists of the C1 cache connected to external memory. The *Configuration Register File* (CRF) consists of a set of *configuration registers* (CRs). Each CR serves as an alias to either a configured functional unit or a configuration allocated in the C-Cache. Most of the AEPIC instructions take a configuration register as an operand. These are the AEPIC instructions that perform operations such as delete a CFU, etc. The instruction refers to the CFU by its alias—the configuration register. For example, the delete CFU operation in AEPIC is `DELC cr, L, p`. Here, `cr` is the configuration register associated with the desired CFU, `L` is the latency assumed by the compiler for this operation and p the predicate guard for this operation.

We shall now give a more detailed description of the MRLA as it is pertinent to the compiler optimizations discussed in the rest of the paper. The Multi-context Reconfigurable Logic Array (MRLA) is the primary resource used for hosting the configured functional units. Like a typical Field Programmable Gate Array (FPGA), the MRLA is a two dimensional region of the processor die that is composed of programmable logic and interconnect blocks. We shall use the term *Programmable Element* (PE) to refer to both the programmable logic block as well as the programmable interconnect block. Each PE is associated with a

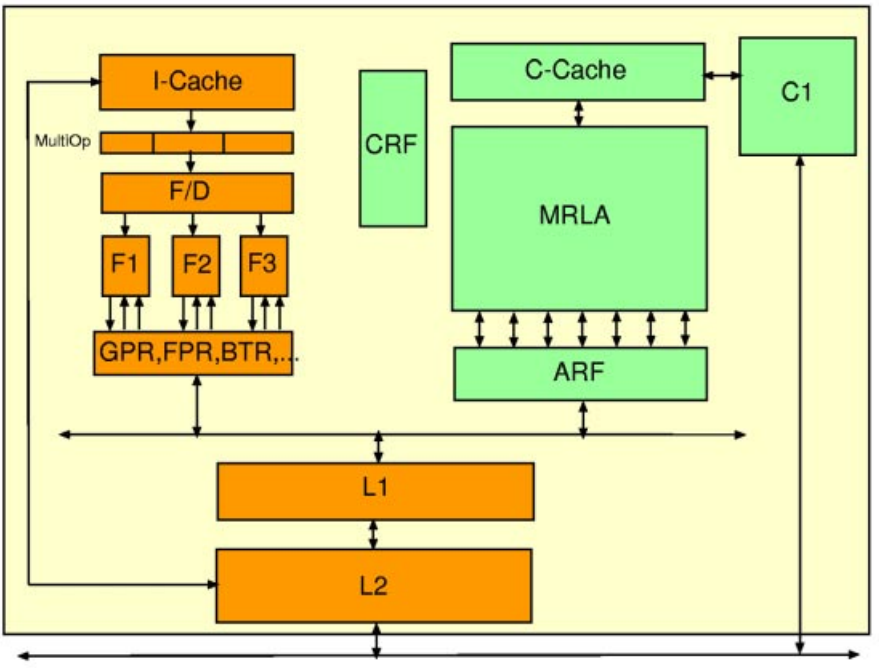


Fig. 2. AEPIC machine model

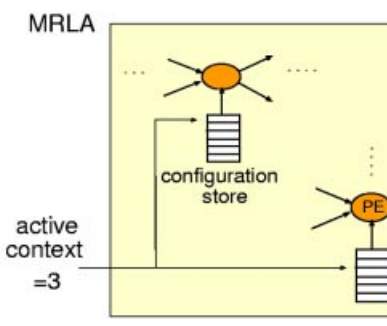


Fig. 3. Structure of MRLA

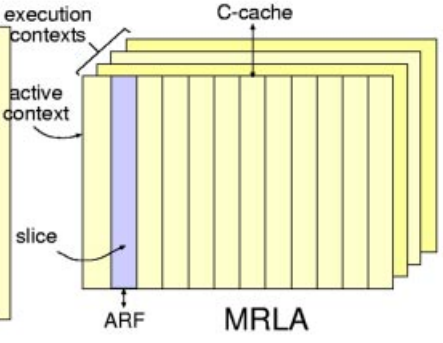


Fig. 4. MRLA multiple contexts

configuration instruction (its "program") which determines the behavior of that programmable element. Any given logic design can be emulated on the MRLA by supplying suitable *configuration instructions* for all the programmable elements of the array. In the context of AEPIC machines, these logic designs are the CFUs.

In a standard FPGA only one configuration instruction is associated with each programmable element. This implies that only one logic design can be resident on the array until the device is reconfigured (i.e., a new set of configuration instructions are associated with the programmable elements of the array). In an MRLA, each programmable element can be associated with multiple configuration instructions. This allows multiple logic designs (CFUs) to be simultaneously resident on the MRLA. The desired logic design can be activated by selecting the appropriate configuration instruction for each of the programmable element.

Configuration instruction slots for each PE (called the *configuration memory*) are stored in an ordered sequence and all PEs have the same number (D) of configuration instruction slots. MRLA takes an input called *context_id* which can take values from 1 to D. A value of k to the *context_id* input selects the k^{th} configuration instruction from the configuration memory as the instruction for each PE. The k^{th} confi guration instruction is referred to as the **active configuration instruction** for that PE.

The set of configuration instructions with identical index in the configuration memory of a PE is referred to as an **execution context**. The execution context that is associated with currently active configuration is called the **active context**. MRLA can be effectively viewed as an array of FPGAs, one array per execution context; and the *context_id* serves as the index into this array. Selection of an execution context, makes all the CFUs of that context available for instruction processing by subsequent instructions.

With this brief introduction to our proposed AEPIC architecture, we shall now deal with the compilation issues of such a machine.

4 Compilation Framework

Inputs to an AEPIC compiler are (a) the application source program written in a high level language such as C/C++, (b) a description of the particular instance of the AEPIC processor described in a machine description language (c) a library containing parameterized configurations for popular computational routines such as FFT, DCT, etc. The input source code may be instrumented with pragmas intended to give partitioning/mapping hints to the compiler. These cues are communicated to different phases of the compilation process through the intermediate code representations.

The first phase is a standard lexical and syntactic analysis phase. The partitioning module will help delineate portions of the program that might benefit from execution on the configurable portion of the target. The partitioning phase is followed by two independent phases that may be performed in parallel: the high-level optimization phase and the operation synthesis phase. In the high-level optimization phase, the code to be executed on the hardwired functional units (which execute instructions from the AEPIC ISA) is optimized as is typically performed in a standard ILP compiler. The instruction synthesis phase generates suitable mappings of the code partitions identified by the partitioning phase. Each of these mappings are packaged as *custom operations*. The ma-

chine description is updated with the synthesized instructions. The instruction synthesis phase may generate multiple implementations for the identified partitions reflecting different performance/device-area tradeoffs. The configuration selection phase tags different regions of the intermediate code with semantically equivalent custom operations. At the end of the configuration selection phase, every region of the code identified for mapping on to the configurable part of the target has a unique configuration associated with it.

Subsequent phases of the compilation are similar in structure to back-end phases of a typical ILP compiler suitably adapted to consider the special characteristics of configurations/CFUs. A typical ILP compiler back-end comprises of (at the minimum) the following phases, in sequence: pre-pass scheduling, register allocation, post-pass scheduling and code generation. In addition to these, AEPIC compilation introduces an extra phase: *configuration allocation*. The configuration allocation phase is aimed at optimizing allocation of resources for configurations—a task analogous to that of register allocation. The resources meant for configuration are the C-cache and the MRLA. These resources are independent of those intended for register allocation. Hence, configuration allocation may be performed in parallel or in any order with respect to register allocation. The main task of the scheduler is to reduce the critical path through the code by masking reconfiguration overheads.

In the above framework, the scheduled and allocated code is transformed into machine code, which is then translated to object code for simulation. Simulation yields correctness as well as performance data for the program on the given input data. The execution profile can be re-instrumented into the IR for profile based optimizations.

4.1 Partitioning

Partitioning is the task of determining the set of code segments (referred to as *candidates*) in the application which may be synthesized as application specific instructions. These application specific instructions are implemented on the MRLA as configured functional units.

The partitioning module takes as input (a) an intermediate code representation of the source program and (b) the machine description of the target AEPIC processor and identifies regions of the intermediate code that can benefit from mapping to the MRLA. Although not necessary, the partitioner can only benefit from an execution profile of the application. The execution profile gives execution frequencies of different regions of the intermediate code. This information can be obtained by compiling to a base EPIC architecture and re-instrumenting the intermediate code with the profile data from the simulator. Since the reconfiguration overhead can be quite large, it may not pay to reconfigure the processor for a certain segment of the code if it is known that this section will rarely be executed. The execution profile can be used by the partitioner to eliminate such code segments from consideration for mapping onto the MRLA.

The code-partitioner is composed of four main steps. These steps are performed in sequence on each intermediate representation that is created during the compilation process.

1. Profile. The profile phase is composed of four steps:

- *Instrument:* The IR nodes are tagged with code for gathering the execution profile for each node. So for example, in a basic block IR, each basic block will be associated with a "execution frequency" variable and a piece of code at the entry of the basic block which updates this variable whenever control enters the basic block.
- *Translate:* During this step the instrumented IR is translated to C language code which is then compiled using the native compiler to yield a semantically equivalent (to the application program) executable with one additional attribute: this executable also gathers the runtime properties of the program as specified by the instrumentation code.
- *Execute:* The execute step runs the instrumented and compiled application using the inputs associated with the application program.
- *Re-instrument:* After the execute step, the application generates the execution profile (written out by the instrumented code) and re-instruments the starting IR with the actual execution profile values (such as the frequency of execution of a basic block, etc.)

2. Analyze. After the **profile** step, the IR is tagged with the execution profile. This information is used to identify the most frequently executing regions of the code. The **analyze** phase traverses the IR and determines various properties of the program that may be useful in determining if a region is a candidate for partitioning. Some of these properties are bit-widths/types of variables, whether certain arrays are constants or are bounded of known dimensions, etc.

3. Identify. The **identify** step performs a bottom-up traversal of the IR and tags each region as a candidate for mapping using certain heuristics based on control structure, operation types and other attributes identified in the **analyze** phase. The identify phase also considers machine resource constraints (from the machine description database) and any user supplied cues transmitted through the IR, in deciding whether a region of the IR is a feasible candidate for mapping.

4. Coalesce. The coalesce phase merges adjacent regions marked by the **identify** phase if the merged regions can still be accommodated on the MRLA assuming the given machine constraints.

4.2 Instruction Synthesis

Instruction synthesis is a systematic technique for defining new instruction set for a given micro-architecture. The process typically involves analyzing the benchmark(s) to infer the most suitable operation repertoire based on its computational characteristics for the intended micro-architecture. In certain cases the micro-architecture itself is synthesized in parallel with the instruction set.

The Instruction Synthesis (IS) module takes a list of candidate partitions that have been identified for mapping onto the programmable logic and synthesizes a set of functional units that can implement all the computations of the code partitions. In addition to the candidate partitions list, the IS module may also take as input, a library of pre-synthesized macros for various basic operators and frequently used kernels such as FFT, DCT, FIR/IIR filters, etc. The purpose of this library is to speedup the process of mapping a given partition to the MRLA. It helps to keep these pre-synthesized macros generic so that they are applicable to a wide range of the target programmable logic parameters, and also accommodate variations in the structure of the input partitions.

4.3 Configuration Selection

After the partitioning and instruction synthesis phase, all the regions of the intermediate code that may be mapped to the MRLA have been identified. Each of the candidate partitions can now be replaced with one of the equivalent configurations synthesized by the IS phase. Configuration Selection (CS) is the problem of determining which semantically equivalent synthesized configuration (custom instruction) to associate with each candidate partition that is mapped to MRLA.

We extend the EPIC code generation path to include configuration selection. The key difference compared to traditional instruction selection is that the selection can happen at multiple levels in the IR. Configuration selection can happen for leaf level operations as well as higher level structures such as groups of instructions, program statements, loops, and functions, etc.

4.4 Configuration Allocation

After configuration selection, some of the nodes of the intermediate code may be tagged with application specific instructions. The configuration selection module *does not consider availability of resources on chip* when it decides on which configuration to assign with each partition that is synthesized into application specific instruction. Any realistic AEPIC processor would have limited programmable logic resources available for CFUs and hence it is quite possible that all the desired configurations cannot be accommodated on chip simultaneously.

Configuration Allocation (CA) is the problem of optimally allocating/deallocating on-chip resources that are intended for holding configurations (on C-cache) or CFUs (on MRLA). Poor allocation of resources for configurations can lead to long periods of processor stalls caused either due to waiting for the configurations to load into the MRLA or due to excessive thrashing in the configuration memory hierarchy. Hence, optimal allocation of configurations to C-cache and MRLA resources is critical for achieving high performance on an AEPIC processor. In addition, we would like the allocation algorithm itself to be time and space efficient.

This problem is analogous to the problem of implementing virtual memory on systems with limited physical memory or to the problem of allocating registers

for program variables performed in most standard compilers. In the remainder of this section, we present a formal definition of the configuration allocation problem and relate it to the well studied problem of register allocation. We show how extensions to the register allocation techniques yield solutions to the configuration allocation problem.

Register Allocation (RA) allocates program variables (also referred to as temporaries) to registers in order to minimize the overall number of accesses to external memory. Fundamentally, both register allocator and configuration allocator perform the same task—allocation of on-chip resources for program values (variables in the case of RA and configurations in the case of CA). Configuration allocation differs from register allocation in the following ways:

1. *Non-uniform allocation units.* Sizes of configurations are typically much larger and *vary widely* compared to the sizes of data values stored in registers which are much smaller and almost always uniform in size.
2. *Heterogeneous resources to be allocated.* There are two types of local memories for configurations: (1) the C-cache and (2) the MRLA. These resources differ in their capacities, access (read/write) costs and sizes of allocation units. There is only one type of resource to be allocated in register allocation - the register set.
3. *Immutable values.* Currently, AEPIC architecture configurations are immutable. So memory write back of configurations is not an issue.
4. *No copies or moves.* AEPIC does not provide any architecturally visible features to create copies or move configurations with in the MRLA or the C-cache.

There are two types of storage classes are available for hosting configurations: (1) C-cache and (2) MRLA. Allocated configurations are present in exactly one of these storage classes. Every allocated configuration whether it is on the MRLA or in the C-cache is associated with a distinct configuration register from the configuration register file (CRF). The unit of allocation in the C-cache is a C-cache *block* and on the MRLA it is a *slice*. All configurations are constrained to consume an integral number of consecutive slices in the MRLA. Configuration data for each MRLA slice requires an integral number of blocks in the C-cache. The basic steps involved in using configurations are:

1. Allocate a configuration register with the configuration to be loaded.
2. Allocate requisite number of blocks in the C-cache for the configuration. If the C-cache is insufficient for the configuration, then the allocation fails and application is aborted. If the configuration is smaller than the C-cache size but the total free space is less than that which the configuration requires, then some of the resident configurations are evicted (since configurations are immutable, they are simply deleted and not written back to memory).
3. Schedule the loading of configuration data into the C-cache into the allocated blocks.
4. Allocate consecutive slices on the MRLA to load the configuration that was loaded into the C-cache (to instantiate the CFU corresponding to the configuration).

5. Schedule transfer of configuration data from the C-cache to MRLA.
6. One or more operations are executed on the CFU.
7. At some point if the MRLA resources are required for another CFU or if this configuration will not be used any more, then it is evicted to the C-cache (if it may be used again) or just deleted from MRLA.
8. Allocated configuration register is freed - it can be allocated to a new configuration.

Note that the same configuration register refers to the configuration data when it was in the C-cache and to the corresponding CFU when loaded onto the MRLA. Once the configuration is completely moved to MRLA from the C-cache, the C-cache resources allocated for the configuration may freed since it is wasteful blocking those resources as long as the configuration data is available on the MRLA. However, in certain cases it might be useful to architecturally expose the deallocation operation. For example, if multiple instances of the CFU corresponding to the configuration are needed on the MRLA, it might be more efficient to copy the configuration data from the C-cache to the MRLA once for each of the CFU instances instead of loading from external memory into the MRLA to make copies of the CFU already available on the MRLA.

4.5 A Simplified AEPIC Allocation Model

For the remainder of this section we consider a simplified version of the configuration allocation problem. In the simplified version, the machine is assumed to contain N configuration registers and each "virtual" configuration in the program intermediate code specifies an integer k which is the number of physical configuration registers it requires. The only difference between this problem and the conventional register allocation is that, in the conventional register allocation problem, each virtual register (also called program temporary) is assigned to a single physical register.

A live range is an isolated and connected group of nodes in the control flow graph that connects the definitions and uses of a given program variable. It is the principle data structure for register allocation. Two live ranges interfere if one of them is live at the definition point of the other. As the reader may have already noted, there is a close resemblance between register live ranges and configuration. The algorithm to construct the interference graph for configurations is very similar to that for register live ranges.

Pruning for Configuration Allocation. If there are program regions where the total resource requirement of configurations that are live at that point exceeds the available resources on the processor, configuration allocation will fail. Analogous to the concept of *register pressure*, we define the total resource requirement at any program point the *resource pressure* at that point. *Pruning* is a technique of preprocessing the live ranges of configurations to ensure that the *resource pressure* never exceeds the total resources available on the machine.

Pruning involves (a) determining the set of live ranges to split and, (b) determining the right split points for the selected live ranges. Once a live range

is split, compensation code needs to be inserted to fetch the values (configurations) for the uses encountered in the second (or later, if multiple splits were performed) part of the live range. One downside of reducing the resource pressure is the additional time taken for executing the compensation code. Hence pruning decisions cannot be made arbitrarily. In addition to the resource pressure, execution frequency should be taken into account before a selecting a live range for splitting since that determines the number of times the compensation code would be executed.

We propose a pruning technique based on the hierarchical technique proposed by Callahan and Koblenz [6]. In their algorithm, gaps between references to a register in a live range are identified. These are possible regions which can be spilled to memory. The maximal length live-range gap is referred to as *wedges* in [9]. Non-overlapping and maximal wedges are identified using the control tree [20]. The choice of wedge s to prune is a function of (a) the runtime cost of compensation code that would be added in the pruned region and, (b) size of the region of the program region that would benefit from the pruning decision. Our algorithm takes into account the special requirements of configurations (their non-uniform sizes - which implies non-uniform spill costs; immutability - which implies stores to memory are not required) and also enhances the scheme with regard to identifying spill candidates and spill locations based on execution profile as well. The algorithm, adapted from the register live range pruning algorithm from [9] and is described in Algorithm 1. The relevant parameters and the data-structures used by the algorithm are listed below.

$$
\begin{aligned}
R &= \text{total number of recourse units available for allocation} \\
C &= \text{configurations (candidates) to be allocated} \\
T &= \text{Control tree of the program} \\
ResUnits[c] &= \text{number of resource units required by the configuration } c \\
Live[n] &= \text{set of configurations live in control node } n \\
Refs[n] &= \text{configurations that are referenced in control node } n \\
Wedges[n] &= \text{list of candidates with wedges that have heads at } n \\
Freq[e] &= \text{number of times control traversed along edge } e \\
Excess[n] &= \begin{cases} \sum_{c \in Live[n]} ResUnits[c] - R & n \in T.Leaves, \\ Max_{c \in T.Children[n]}\{Excess[c]\} & n \notin T.Leaves. \end{cases} \\
LiveUnits[n, C] &= \begin{cases} 1 & n \in T.Leaves \wedge C \in Live[n], \\ 0 & n \in T.Leaves \wedge C \notin Live[n], \\ \sum_{p \in T.Children[n]} LiveUnits[p, C] & n \notin T.Leaves. \end{cases}
\end{aligned}
$$

4.6 Graph Multi-coloring Configuration Allocator (GMCA)

Here we provide a simple configuration allocator called Graph Multi-coloring Configuration Allocator (GMCA), based on Chaitin's graph coloring register

Algorithm 1 Pruning

```
function InitPrune(N) {
    if (N ∈ T.Leaves) {
        LivePart[N] ← Live[N];
        Excess[N] ← (∑_{n∈Live[N]} ResUnits[n]) − R;
        ∀C ∈ Live[N] :   LiveSize[C, N] ← |C|;
    } else {
        ∀M ∈ N.children :   InitPrune(M);
        LivePart[N] ← ∪_{M∈N.children} LivePart[M];
        Refs[N] ← ∪_{M∈N.children} Refs[M];
        Excess[N] ← Max_{M∈N.children} Excess[M];
        ∀C ∈ LivePart[N] :   LiveSize[C, N] ← ∑_{M∈N.children} LiveSize[C, M] ;
        ∀C ∈ Refs[N] ∧ ∀M ∈ N.children:
        if (C ∉ Refs[M] ∧ C ∈ LivePart[M])
            NewWedge(C,M);
    }
}
function UpdatePressure (W,N) {
    if (N ∈ T.leaves) {
        Live[N] ← Live[N] - {W};
        Excess[N] ← Max{(∑_{n∈Live[N]} ResUnits[n]) − R, 0};
    } else {
        Excess[N] ← Max_{m∈N.Children}{UpdatePressure(W, m)};
    }
    return Excess[N];
}
function Prune (N) {
    PrioritizeWedges(N);
    while (Excess[N] > 0 ∧ |Wedges[N]| > 0) {
        W ← Wedges[N].top();
        PruneWedge(W);
        UpdatePressure(W, N);
    }
    ∀M ∈ N.Children ∧ Excess[N] > 0:   Prune(M);
}
```

allocator [7,8] with the spill and split decision modifications suggested by Hansoo and Leung [18] for the simplified AEPIC configuration resource model. All past graph coloring based register allocation schemes are based on the idea of *simplification* [17]. If a graph G contains a node v with fewer than K neighbors and if $G - v$ can be colored with K colors, then G is K colorable. In the case of *configuration allocation*, multiple colors may be allocated to each node and hence it calls for a stronger version of the simplification step. We first state and prove this *simplification lemma* and then show it is used in our GMCA algorithm.

Let $G(V, E, w)$ be an undirected graph where $w : v \rightarrow \mathbf{Z}$ is a weight function defined on the vertices. Let $C : v \rightarrow S$ be a function on the vertex set such that $S \subset \{1, \ldots, K\}$. Then C is a valid *K-multi-coloring* of the graph if $|C(v)| = w(v)$ and $\forall e \in E, where\ \ e = (u, v),\ C(u) \cap C(v) = \emptyset$.

Lemma 1. [**Simplification lemma**]
Let vertex $v \in V$ be such that $\sum_{u \in Adj(v)} w(u) \leq K - w(v)$. Let $G' = G - v$ be the subgraph obtained by removing v and its incident edges from G. If G' can be K-multi-colored, then so can G.

Proof: Let C be the K-multi-coloring of G'. Let $S_v = \bigcup_{u \in Adj(v)} C(u)$. By definition of C, $|C(u)| = w(u)$. This implies that $|S_v| \leq \sum_{u \in Adj(v)} w(u)$. Given that $\sum_{u \in Adj(v)} w(u) \leq K - w(v)$, it implies that $|S_v| \leq K - w(v)$. Consider $C_v = \{r | r \in \{1, \ldots, K\}, r \notin C(u)\}\}$. Clearly, $|C_v| \geq K - w(v)$. Let $C_v^{w(v)}$ be any $w(v)$ sized subset of C_v. Let

$$C'(p) = \begin{cases} C(p) & \text{if } p \in V(G') \\ C_v^{w(v)} & \text{if } p = v \end{cases}$$

Then, C' is a valid *K-multi-coloring* of G.

A vertex $v \in V$ such that $\sum_{u \in Adj(v)} w(u) \leq K - w(v)$ then the vertex is called *unconstrained* else it is referred to as an *constrained* node. Pseudo-code for the Graph Multi-coloring Configuration Allocator (GMCA) based on the simplification lemma is presented in Algorithm 2.

Phases of graph multi-coloring configuration allocator.

1. *Build:* Live ranges are computed and the interference graph is constructed during this phase.
2. *Coalesce:* The coalesce step removes any unnecessary move (copy) instructions effectively merging the live ranges corresponding to the values connected by the move instruction.
3. *Simplify:* The *simplification lemma* is the basis for this step. *Unconstrained* nodes are selected and pushed onto the *color_stack* and removed from the interference graph. Since the *simplification lemma* guarantees that unconstrained nodes can always be colored if the reduced graph is colorable, they are removed from the graph. This step in turn might enable other *constrained* nodes to become *unconstrained*. If so, then it is repeated until either the graph is empty or all nodes are *constrained*.
4. *Prioritize:* Priorities are assigned to constrained live ranges. Live ranges are selected for coloring in order of their priority. Priority functions capture the expected benefit of allocating the live range to on chip resources as opposed to external memory.
5. *ProcessNode:* The highest priority node is selected for coloring. For each node, just as in [10], a *Forbidden* set is maintained which indicates the set of colors (resources) that have already been allocated and hence cannot be used

Algorithm 2 Graph multi-coloring configuration allocator

```
function GMCA(CFG cfg) {
    G ← Build(cfg);
    while (G ≠ ∅) {
        while (∃v|v is unconstrained) {
            S.push(v);
            G ← G − {v};
        }
        if (G ≠ ∅) {
            ComputePriorities(G, h);
            v ← HighestPriorityNode(G);
            if (IsColorable(v)) {
                Color(v);
            } else {
                G ← Split(G, v);
            }
        }
    }
    ProcessStack(S);
}
function IsColorable(v) {
    return (v.ColorReq ≥ (K − ∑_{0<i≤K} v.Forbidden[i]));
}
function Color(v) {
    availColors ← ¬v.Forbidden;
    P ← Min_{i|0<i≤K}{∑_{0<k≤i} availColors[k] = v.ColorReq};
    mask ← 1^P 0^{N−P};
    v.Assignment ← v.availColors ∧ mask;
    ∀u|u ∈ Adj(v) : u.Forbidden ← u.Forbidden ∨ v.Assignment;
}
```

for the current node. If there are enough available colors that can satisfy the color requirement for the current node, the node is colored and the *Forbidden* sets of its neighbors are updated to reflect the allocation. If the set of available colors cannot satisfy the demand for this node, then the node is either *spilled* or the live range *split* depending on which one is most beneficial.

6. *ProcessStack:* Unconstrained nodes eliminated (pushed onto the color_stack) during the simplify phase are colored in the reverse order in which they were removed from the graph. The original graph is incrementally reconstructed by adding one node at a time from the top of the *color_stack* and is assigned the color vector. The *simplification lemma* guarantees that enough colors are available to color the inserted node.

5 Conclusion

In this paper, we defined the problems encountered in each of the phases of the compilation path for AEPIC. In some cases we proposed techniques that may be used to solve these compilation problems. Substantial research still needs to be done and we do not make any claims on whether any of these techniques are sufficient to fully exploit the capabilities of AEPIC processors.

References

1. Anant Agarwal, Saman Amarasinghe, Rajeev Barua, Matthew Frank, Walter Lee, Vivek Sarkar, Devabhaktuni Srikrishna, , and Michael Taylor. The RAW compiler project. In *Proceedings of the Second SUIF Compiler Workshop*, pages 21–23, Stanford, CA, August 1997.
2. P. M. Athanas and H. F. Silverman. Processor reconfiguration through instruction-set metamorphosis. *IEEE Computer*, 26(3):11–18, March 1993.
3. P. Bertin, D. Roncin, and J. Vuillemin. Introduction to programmable active memories. In J. McCanny, J. McWhirther, and E. Swartslander Jr., editors, *Systolic Array Processors*, pages 300–309. Prentice Hall, 1989.
4. P. Bertin, D. Roncin, and J. Vuillemin. Programmable active memories: a performance assessment. In G. Borriello and C. Ebeling, editors, *Research on Integrated Systems: Proceedings of the 1993 Symposium*, pages 88–102, 1993.
5. Srihari Cadambi, Jeffrey Weener, Seth Copen Goldstein, Herman Schmit, and Donald E. Thomas. Managing pipeline-reconfigurable fpgas. In *Proceedings ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays*, February 1998.
6. D. Callahan and B. Koblenz. Register allocation via hierarchical graph coloring. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, volume 26, pages 192–203, Toronto, Ontario, Canada, June 1991.
7. G. Chaitin, M. Auslander, A. Chandra, J. Cocke, M. Hopkins, and P. Markstein. Register allocating via coloring, 1981.
8. Gregory J. Chaitin. Register allocation and spilling via graph coloring. *SIGPLAN Notices (Proceedings of the SIGPLAN '82 Symposium on Compiler Construction, Boston, Mass.)*, 17(6):98–105, 1982.
9. F. Chow, K. Knobe, A. Meltzer, R. Morgan, and K. Zadeck. Register allocation.
10. Fred C. Chow and John L. Hennessy. Register allocation by priority–based coloring. In *Proceedings of the ACM SIGPLAN 84 Symposium on Compiler Construction*, pages 222–232, New York, NY, 1984. ACM.
11. G. Estrin. Organization of computer systems – the fixed plus variable structure computer. In *Proceedings of the Western Joint Computer Conference*, pages 33–40, 1960.
12. M. Gokhale, W. Holmes, A. Kopser, S. Lucas, R. Minnich, D. Sweely, and D. Lopresti. Building and using a highly parallel programmable logic array. *IEEE Computer*, 24(1):81–89, January 1991.
13. C. Ebeling D. C. Green and P. Franklin. RaPiD – reconfigurable pipelined datapath. In R. W. Hartenstein and M. Glesner, editors, *Field-Programmable Logic: Smart Applications, New Paradigms, and Compilers. 6th International Workshop on Field-Programmable Logic and Applications*, pages 126–135, Darmstadt, Germany, September 1996. Springer-Verlag.

14. S. Hauck, T. W. Fry, M. M. Hosler, and J. P. Kao. The chimaera reconfigurable functional unit. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 87–96, 1997.
15. S. Hauck, M. M. Hosler, and T. W. Fry. High-performance carry chains for fpgas. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 223–233, 1998.
16. John R. Hauser and John Wawrzynek. GARP: A MIPS processor with a reconfigurable coprocessor. In J. Arnold and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 12–21, Napa, CA, April 1997.
17. A. Kempe. The geographical problem of the four colors. *Amer. J. Math. 2, 193–200.*, 1879.
18. H. Kim and A. Leung. Frequency based live range splitting. *Technical report, ReaCT-ILP Laboratory, New York University*, 1999.
19. Walter Lee, Rajeev Barua, Devabhaktuni Srikrishna, Jonathan Babb, Vivek Sarkar, Saman Amarasinghe, and Anant Agarwal. Space-time scheduling of instruction-level parallelism on a RAW machine. *MIT/LCS Technical Memo TM-572*, December 1997.
20. S. Muchnick. Advanced compiler design and implementation, 1997.
21. R. Razdan and M. D. Smith. A high-performance microarchitecture with hardware-programmable functional units. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 172–80. IEEE/ACM, November 1994.
22. Rahul Razdan. *PRISC: Programmable Reduced Instruction Set Computers.* PhD thesis, Harvard University, May 1994.
23. Mario R. Schaffner. Processing by data and program blocks. *IEEE Transactions on Computers*, 27(11):1015–1028, November 1978.
24. M. Schlansker and B. Rau. EPIC: An architecture for instruction-level parallel processors. *Technical report HPL-1999-111, Hewlett-Packard Laboratories, Technical Publications Department, 1501 Page Mill Road, Palo Alto, CA 94304.*, 2000.
25. S. Talla. *Adaptive Explicitly Parallel Instruction Computing.* PhD thesis, New York University, 2000.
26. Stephen M. Trimberger. *Field-Programmable Gate Array Technology.* Kluwer Academic Publishers, 1994.
27. Triscend Corp., Mountain View, U.S.A. *Triscend A7 Configurable System-on-a-Chip Family Data Sheet*, 2001.
28. John Villasenor and William H. Mangione-Smith. Configurable computing. *Scientific American*, pages 66–71, June 1997.
29. J. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. Touati, and P. Boucard. Programmable active memories: Reconfigurable systems come of age. *IEEE Transactions on VLSI Systems*, 4(1):56–69, 1996.
30. E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring it all to software: RAW machines. *IEEE Computer*, pages 86–93, September 1997.
31. M. J. Wirthlin and B. L. Hutchings. DISC: The dynamic instruction set computer. In J. Schewel, editor, *Proceedings of the International Society of Optical Engineering (SPIE). Field Programmable Gate Arrays (FPGAs) for Fast Board Development and Reconfigurable Computing.*, volume 2607, pages 92–103, Philadephia, PA, 1995.
32. Xilinx, San Jose, CA. *The Programmable Logic Data Book*, 1994.

Embedded Software Market Transformation through Reusable Frameworks

Wolfgang Pree and Alessandro Pasetti

Software Research Lab
University of Constance, D-78457 Constance, Germany
firstname.lastname@uni-konstanz.de
www.SoftwareResearch.net

Abstract. Object-oriented frameworks are a software reuse technology that fosters reuse of entire architectures and which has made software reuse a reality in many domain areas. Like other advanced software techniques, however, framework technology has seldom been used in the embedded domain. This paper argues that its application to embedded (control) systems is technically feasible and liable to bring to them the same benefits it has already brought to other domains. The description of a prototype framework for satellite control systems corroborates the argument. It is then argued that software frameworks, when combined with other enabling technologies, have the potential to standardize various aspects of embedded software and to transform the embedded sytems market.

1 Project Culture versus Product Culture

Classical software development strategies do not focus on the reusability of software components. Bertrand Meyer [Mey89] remarks that "object-orientedness is not just a programming style but the implementation of a certain view of what software should be—this view implies profound rethinking of the software process." Thus we might discern between two views or cultures:

- The conventional culture is *project-based*: the classical software life-cycle or some of its variations have the goal of solving *one* certain problem. The primary question addressed in the analysis and design phases is "What must the system do?", followed by a stepwise refinement of a system's functions.
- The preconditions of a *product culture* are object-oriented development techniques, in particular object-oriented frameworks. This culture yields not only software systems that solve *one* certain problem, but reusable software components that are useful for a potentially large number of applications.

A set of ready-made reusable/adaptable software components also influences the system specification. In contrast to the project-based culture, where a software system is developed to satisfy specific needs, good frameworks typically capture the existing "best practices" in a domain. For example, the SAP system represents a framework that—although developed with a non-object-oriented language—standardizes

T.A. Henzinger and C.M. Kirsch (Eds.): EMSOFT 2001, LNCS 2211, pp. 274-286, 2001.
© Springer-Verlag Berlin Heidelberg 2001

significant portions of how companies conduct business, covering areas such as accounting, human resource management, production planning and manufacturing. The SAP framework can be adapted and fine-tuned to the specific needs of companies. Though the effort required to adapt SAP to a specific company is significant and some adaptations are not feasible, the defacto domain standardization is part of SAP's success story.

In other words, instead of slavishly adhering to the user's or customer's requests in the realm of a conventional custom-made system construction process, the system specification inherent in a framework most likely provides a somewhat different functionality (for example, without some nice-to-have features), which can be created by means of existing framework components. The customer has the choice between a custom-made system implemented (almost) from scratch with significantly more effort and cost, and a system built out of ready-made components by adapting a framework. The framework has the additional advantage that the quality of the resulting system in terms of reliability will likely be higher than the custom-made system. This is particularly true if the framework has been thoroughly tested and/or has already been reused several times. Some offers might be hard to refuse.

From an organizational persepective, successful application of framework technology implies a more integrated approach to software development that breaks down the traditional barrier between the application and the software specialists. The way how SAP systems are adapted to specific needs corroborates this change in the development process.

Other domains where a standarization through frameworks has lead to a product culture are graphical user interfaces and Internet applications. Fayad et al. [Fay99] present successful frameworks in various domains. We see no reason why a standardization of real-time embedded systems software should not be feasible.

2 Object-Oriented Frameworks for Reuse and Flexibility

Over the past decade object technology has gained widespread use in software development. Overall, three essential concepts comprise object technology: information hiding, inheritance/polymorphism and dynamic binding. The mixing ratio of these ingredients defines the flavors of object technology. Object-based systems stress information hiding. Object-oriented programming is often described as *programming by difference* as it adds inheritance and dynamic binding to the object-based paradigm. Nevertheless, the object-based and object-oriented paradigms do not automatically enhance reusability and exentsibility. The coupling of components[1] often manifests in the source code. Figure 1 schematically outlines the problem. If the right hand component should work with another component, its source code has to be changed.

[1] software component:= a piece of software with a programming interface; ideally deployable as unit. Classes and modules are examples of software components.

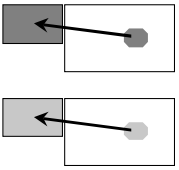


Fig. 1. Components with glue code inside.

Go the extra mile—define abstractions! Object-oriented frameworks form a special breed of object-oriented systems with extensibility as key characteristic [Font01]. The precondition for achieving extensibility and as a consequence plug&work composition capabilities are domain abstractions. They become abstract classes or Java/C# interfaces. Conceptually, domain abstractions form plugs. Polymorphism and dynamic binding allow a compositional adaptation of these components without having to change the source code. Figure 2 illustrates the difference to the previous solution: It depends on the dynamic type of the plugged in component which implementation of m1() is executed. The source code which calls m1() remains unchanged. This call-back-style of programming manifests in function/procedure parameters in conventional languages. Object-oriented languages provide polymorphism and dynamic binding for that purpose. As several methods can be grouped in object-oriented abstractions, the composition becomes more convenient if an object-oriented language is used.

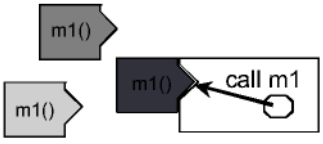


Fig. 2. Plug standardization through domain abstractions.

A framework deserves the attribute well-designed if it offers the domain-specific plugs, often referred to as variation points, to achieve the desired flexibility for adaptations. Well-designed frameworks also predefine most of the overall architecture, that is, the composition and interaction of its components. Applications built on top of a framework reuse not only source code but architecture design—which we consider as an important characteristic of a framework.

Pros and Cons of Frameworks

Besides the fact that reuse of architecture design amounts to a standardization of the application structure, frameworks offer further advantages. Adapting a framework to produce specific applications implies a significant reduction in the size of the source code that has to be written by the application programmer. Mature frameworks allow

a reduction of up to 90% [Wei89, Fay99] compared to software written with the support of a conventional function library.

More good news is that framework-centered software development is not restricted to specific domains. Actually, frameworks are well-suited for almost any commercial and technical domain as sketched in Section 1.

The bad news is that framework development requires an extraordinary development effort. The costs to develop a framework are significantly higher (in the order of three to four times) compared to the development of a specific application: If similar applications already exist, they have to be studied carefully to come up with an appropriate generic semifinished system—the framework for the particular domain. Framework development requires a thorough domain understanding. Adaptations of the resulting framework lead to an iterative redesign. So frameworks represent a long-term investment that pays off only if similar applications are developed again and again in a given domain.

Framework development and reuse is at odds with the current project culture that tries to optimize the development of specific software solutions instead of generic ones. We refer to the discussion of organizational issues by Goldberg and Rubin [Gol95].

Finally, it takes considerable effort to learn and understand a framework. The application developer cannot use a framework without knowing the basics of the functionality and interactions the framework provides. This makes essential to not only provide a high-quality framework, but also a straightforward way to learn it. Socalled cookbooks document frameworks and support their adaptations. They contain a set of recipes that provide step-by-step guidelines for typical adaptations together with sample source code. Visual, interactive composition tools may further support the configuration process.

Software Frameworks and Embedded Systems

Despite having proven its worth in various domains, framework technology is still largely shunned in the embedded world. To some extent, this is just one aspect of the wider problem of the technological backwardness of embedded software. Embedded software projects, when seen from the point of view of mainstream computer science, often appear to be taking place in a time warp: the language of choice remains C, the dominant architectural paradigm is only now shifting from procedural to object-based, software engineering tools are seldom used. Embedded systems additionally tend to be mass-produced which creates a financial incentive to limit their resources, and embedded software is consequently severely CPU- and memory-limited. The fact that object-oriented software requires some extra resources, is one reason why object-oriented frameworks have been avoided so far.

To penetrate the embedded world, framework technology must overcome at least this technological hurdle. This will require more powerful hardware and a greater willingness to see software as the main source of added value in embedded systems. Both conditions are gradually coming to be realized. On the one hand, the continuing decline in hardware costs is bringing more and more processing power and memory resources within the budgetary envelope of embedded software projects while, on the other hand, ever expanding consumer expectations are putting increasing demands on the software. Satisfying these demands will eventually force the adoption of the same

kind of technology prevalent in other domains and frameworks will undoubtedly play an important role in shortening development times and in improving product flexibility.

As part of our commitment to demonstrating the applicability of framework technology to embedded applications, we have designed and developed a prototype software framework for satellite control systems. Satellite systems are representative candidates because on-board processor resources are undergoing a rapid expansion with the traditional 16-bits CISC processors being replaced by 32-bits SPARC processors while, on the demand side, there is growing pressure to extend the capabilities of the on-board software. We think that similar trends are in evidence in other embedded fields and we believe that software frameworks represent an effective way to exploit the extra hardware resources now becoming available to meet the new user demands.

The prototype satellite framework was developed in cooperation with the European Space Agency by an interdisciplinary team that included both satellite and software expertise. Our experience certainly confirms the generally held view that this combination of software and domain skills was essential to the success of the project. Thus, we think that the satellite framework project is representative of future framework projects for embedded systems both from a technological and from an organizational point of view.

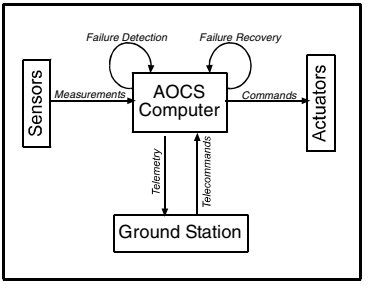


Fig. 3. General structure of an AOCS

3 Case Study: A Framework for Satellite Control Systems

The Attitude and Orbit Control System (AOCS) is usually the most complex of the subsystems on a satellite. Its main function is to process the measurements of a set of attitude and orbit sensors to generate commands for a set of actuators that are designed to stabilize the satellite's orbital position and attitude. Additionally, the AOCS must respond to commands from a ground center and must generate housekeeping data for it. Finally, like most embedded systems, the AOCS must have a high degree of autonomy and must be capable of performing failure detection tests upon itself and of carrying out emergency recovery actions in response to detected failures. The AOCS is therefore a typical embedded control system and its structure is schematically shown in figure 3.

Although the AOCS framework was initially aimed at the AOCS domain, its applicability is in fact wider because, as became clear during its development, the

framework is best seen as a collection of independent modules, the so-called *functionality managers,* that can be used either together or in isolation from each other and most of which are suitable for control systems other than the AOCS.

The AOCS framework is described in detail in [Pas01b] and on the Web site [Pas01c], which includes the freely available source code. [Pas01a] provides a summary of the framework components. Below, an overview of the principles that inspired its design is given together with a discussion of the problems that are specific to the embedded character of the target domain.

Frameworks as Domain-Specific Operating System Extensions

The typical structure of an embedded system is as shown in figure 4(a). The two intermediate layers represent the reusable part of the software whereas the top layer is application-specific and is typically re-developed from scratch for each new application. The operating system is reusable because it packages functionalities that are common to most embedded systems. The device drivers are instead reusable because they are specific to the (reusable) devices whose interface they encapsulate. Both the operating system and the device drivers are constituted by components (typically available as binary entities) that are characterized by the services they offer to their clients. They are intended to be delivered as off-the-shelf items and to be configured for use in a particular application at run-time.

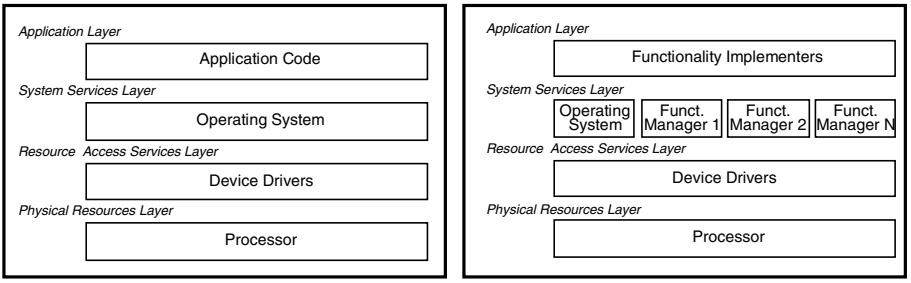


Fig. 4. Conventional (a) versus framework-based (b) structure of a control system.

The operating system represents a framework because the configuration of an operating system during application initialization can be seen as an extension of the operating system itself through the call-back style of programming. Components typically register with the operating system by passing to it pointers to functions they expose and which the operating system – which retains overall control of the main thread of execution – calls when appropriate. The scheduler, for instance, which is one of the core components of an operating system, is configured by passing to it the entry points of the tasks that it must schedule.

When looking at figure 4(a), it is natural to ask whether the operating system really represents the highest possible level of reuse in embedded systems or whether there are commonalities beyond those covered by it that can be encapsulated in reusable and reconfigurable components. If one considers embedded systems in general, the

answer to this question is no. There is simply too much variation in the generic domain "embedded systems". However, as one narrows the focus to particular categories of embedded systems, it becomes possible to identify functionalities that are common to groups of related applications and which can become the basis for domain-specific frameworks.

So far we cannot define these categories. In our project, we concentrated on satellite control systems and we tried to identify whether there were other OS-like functionalities that were specific to this domain and that might allow the OS to be extended in a domain-specific manner. In other words, the question we asked was: if we were to design an operating system *only for satellite control systems,* how would we do it? Which functions would this OS cover and which components would it offer to AOCS developers?

The OS achieves reusability by separating the *implementation* of a functionality from its *management*. Thus, for instance, the scheduler sees the tasks it schedules as abstract entities and it is exclusively responsible for deciding when they should be initialized, executed, suspended, etc. The scheduler is application-independent because it performs all these operations in a manner that is independent of how the tasks are implemented. In this sense, task management is separated from task implementation.

Analysis of the AOCS domain showed that there are several functionalities in an AOCS for which it is possible to have an analogous separation between management of the functionality and its implementation. In such cases, it then becomes possible to construct components that encapsulate the functionality management and that are completely application-independent. We have called such components the *functionality managers.* The AOCS framework offers managers for the following functionalities: telecommand and telemetry handling, failure detection, failure recovery, closed-loop controlling, as well as maneuvers and unit management.

The structure of an AOCS application instantiated from the framework then becomes as shown in figure 4(b). Comparison to figure 4(a) shows that the operating system has now been extended in a domain-specific fashion. It is worth pointing out that the functionality managers can be deployed independently of each other. There is another analogy with operating systems which are often offered as a bundle of modules of which only those needed by a particular application have to be installed.

The Real-Time Dimension

Satellite control systems—like many other embedded systems—are subject to hard real-time constraints. Framework technology has seldom been applied to hard real-time systems. This is partly for the same reasons why it has not been applied to embedded systems in general, but also because of the special problems inherent to real-timeness. Software frameworks rely heavily on dynamic binding as a behavior adaptation mechanism and dynamic binding makes static analysis of the timing properties of an application harder to perform. Additionally, many of the classical design patterns that are commonly used to model framework adaptability are typically implemented using dynamic memory allocation which is not used in real-time applications or use recursion which is again incompatible with static timing analysis.

The position adopted in the AOCS framework project is that since embedded systems are "closed" (all the components making up an application are known at

compile time), dynamic binding does not preclude timing analysis because it is always possible to put an upper bound on the execution time of a call to a dynamically bound procedure [Pas99].

As mentioned above, the AOCS framework can be seen as a collection of design patterns that have been adapted to the special needs of the AOCS domain. In many cases, this adaptation takes the form of making the pattern compatible with the real-time requirements of AOCS applications either by removing the need for dynamic memory allocation or by showing how the depth of recursion can be bounded using semantic information.

Conventional frameworks are designed to have control of application execution [Mat99]. This is a problem in the AOCS domain – and indeed in most embedded systems – where applications often differ for the scheduling policy that they adopt. The framework therefore has to be insulated from scheduling aspects. This is done by turning the functionality managers into *active components,* namely components that offer an entry point to a generic scheduler that is assumed external to the framework itself. Decoupling from the scheduling policy is achieved by designing the functionality managers in such a way that they need not make any assumptions about the frequency with which they are activated, about the source of the activation call, or about the relative ordering in which they are activated.

This decision to leave scheduling aspects outside the framework can also be justified by noting that there already exist excellent reusable solutions to the scheduling problem, provided by commercial operating systems. Hence, a framework should concentrate on addressing issues that are not yet covered, while providing an interface to the components that address the scheduling problem.

All of the solutions we have implemented in the AOCS framework to tackle the real-time dimension could, at least in principle, be carried over to other embedded domains and therefore we believe that our project demonstrates that the framework approach can be successfully applied to this class of embedded systems.

The Framework and Autocoding Tools

Control engineers are becoming more and more accustomed to using environments such as Matlab that offers tools to design controllers and to simulate their operation. (We expect Matlab to prevail in the future. The discussion below applies equally well to similar tools, in particular XMath). Increasingly, such environments come with autocoding facilities that allow code to be generated that implements the controller defined by the user. Since the AOCS framework targets a control domain, the question naturally arises as to whether the approach it promotes to the development of an application is alternative, complementary to, or just different from, that promoted by autocoding tools [Pas99].

The main point to note in this respect is that the Matlab tool is aimed at facilitating the synthesis of control laws and that the implementation of the control algorithms usually represents a small part of the total application code. In the AOCS domain, it is probably around 20-30%. The remainder of the code covers tasks such as data handling or failure detection and recovery for which the autocoding tools do not offer any specific abstractions. Although, technically, it would be possible to generate an entire AOCS from Matlab, this approach would not foster reuse since understanding a complex Matlab model can be as difficult as understanding a complex piece of code.

Reusing it is as hard as reusing poorly structured code. The AOCS framework, on the other hand, is specifically designed to be portable across projects and to have a structure that facilitates maintainability, reusability and understandability.

On the other hand, the ease of designing control algorithms in Matlab and the convenience of being able to generate code that is guaranteed to match the models that were verified by simulation is too valuable to give up. The AOCS framework therefore takes the view that the framework and the autocoding approach are complementary. The framework defines the overall application architecture but it also offers wrappers for code generated by Matlab. The intention is that Matlab covers the design and implementation of the control algorithms whereas the framework provides the architectural skeleton for the application and covers all other aspects of an AOCS application.

4 Towards a Product Culture — The Way Ahead

The AOCS framework project is only the beginning of mid- to long-term research and development activities to improve the software process for embedded control systems. The objective is that control engineers can develop new applications within a particular domain with a minimum amount of manual coding by adapting semifinished frameworks mainly through visual/interactive composition (see figure 5). Conceptually, two stages can be identified in the application development process: the definition of the *logical architecture* and the definition of the *physical architecture*. The logical architecture covers the realization of the functional aspects whereas the physical architecture covers scheduling, communication and distribution issues.

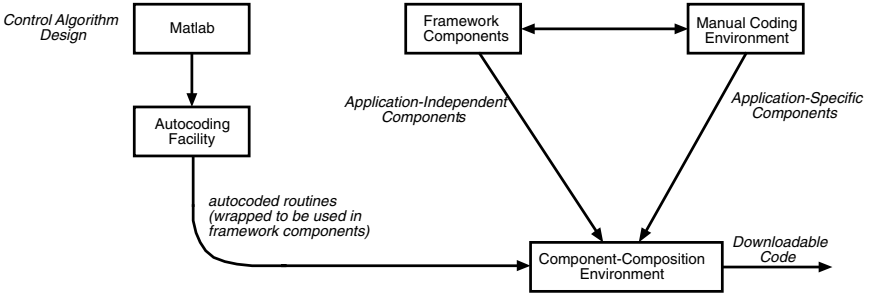


Fig. 5. Development of embedded control systems out of framework components.

The first actor in the definition process for the logical architecture is a tool such as Matlab which allows the control engineers to define and validate by simulation the control algorithms. An autocoding facility generates the corresponding code. The second actor in the definition of the logical architecture is an adequate framework providing a configurable architectural skeleton for the application and a set of default components representing common implementations of the functionalities. Inevitably, some manual development of components is required. Compliance with the framework facilitates the development of the new components and it preserves the

architectural integrity and high-level uniformity of the application by ensuring that the similar problems in different parts of the application receive similar solutions.

The components then need to be configured to form a complete application. This process is typically done by manually writing the code to glue together the components, but could be automated through visual/interactive tools analogous to commonly used GUI editors.

The final step in the application development process is the imposition of a physical architecture onto the logical architecture. This means that tasks must be allocated to processors in the distributed case or to processes in the single-processor case. Furthermore, scheduling policies must be defined and the schedulability must be checked. The communication infrastructure must be put in place to allow the tasks to exchange information among themselves and with the external world. We believe that logical and physical architectures should be kept separate and should be definable independently of each other. Within the object-oriented paradigm, this is possible if a middleware is available that sustains the "illusion of local action" [Ast01] by allowing components to interact as if they resided in the same address space. The logical architecture is defined in this fictitious but uniform space and the successive definition of a physical architecture leads to its splitting into intercommunicating subspaces possibly located on different nodes in a distributed network. Distribution and communication issues should again be handled automatically by dedicated tools.

Realizing the vision of figure 5 requires significant work. Nevertheless, the main planks of the direction outlined above are already in place. In particular, Matlab is a mature commercial product with autocoding facilities. It represents a defacto standard in the embedded community that is tested and widely applied. The AOCS framework provides wrappers for the generated Matlab code. The areas where work needs to concentrate in the future are: the generalization of the AOCS framework to embedded control systems, the development of visual/interactive composition tools, and the automation of the instantiation of logical architecture to the physical architecture of an embedded control application. For each of these areas, we are trying to provide proof-of-concepts solutions that are entirely built on existing technology.

A Set of Frameworks for Control Systems

The AOCS framework is targeted at satellite control systems. Although the structure of such systems is representative of that of generic embedded control systems, to be usable in other contexts, the AOCS framework needs to be modified and extended. This can be a gradual process resulting probably in a set of frameworks for different categories of control systems. Quite likely, additional abstractions have to be added and specific components have to be created. As the AOCS framework is made up of independently deployable functionality managers its extension can occur naturally.

An interesting aspect is whether the AOCS framework imposes a certain overall architecture or favors a real-time programming paradigm. It is unclear whether a focus on the time-triggered paradigm, which is considered superior for saftey-critcial applications [Kop97] would imply a different framework design.

The analogous question has to be raised for heavily distributed embedded systems as opposed to centralized architectures. This concerns, for example, the handling of external sensors. At present, the framework assumes "dumb" external units (sensors and actuators) of the kind currently used in satellite systems. These are conceptually

simple data acquisition devices with little or no internal processing capabilities. The trend is towards "intelligent" units which are processor-based and capable to interact with the central controller in complex ways. In the case of the AOCS, for instance, GPS receivers and star trackers with autonomous star pattern recognition capabilities are becoming common. They are endowed with software that rivals in complexity the software of the central AOCS computer. New concepts need to be developed to handle the interaction between a central computer and these intelligent peripherals.

A related problem concerns the definition of the boundaries of the control software. Once units start having their own software it is no longer clear whether one should consider the control software as separate from that of the external units. It may be more sensible to regard the entire control system as a distributed system and, in the spirit of figure 5, to design it as an integrated whole at the logical level and to leave its partitioning over central computer and external units to the second stage where the physical architecture of the system is defined.

Developing a Visual/Interactive Composition Tool

JavaBeans [Sun01] form the basis of component composition in Java development environments. The AOCS framework is currently being ported from C++ to Java and tested on an embedded target using a real-time version of Java. As part of the porting, all framework components are being turned into JavaBeans.

The advantage of transforming the framework into a set of beans and of modelling its instantiation as a sequence of bean operations is that it becomes possible to visually/interactively manipulate them in off-the-shelf Java development tools. For complex framework adaptations specific bean configuration editors might have to be provided.

Mapping the Logical Architecture to the Physical Architecture

The implementation of the logical architecture upon a physical platform requires ideally a description of the physical characteristics of the platform and a tool that can automatically map the logical architecture to the system at hand. An important property of such a tool should be its ability to take into account the real-time nature of most embedded control systems.

As already mentioned above, our philosophy is to reuse existing tools wherever possible and integrate them within the approach we are proposing. In this case, we found that Giotto [Hen00, Hen01] matches most of our requirements. Giotto can be seen as a middleware that offers a tool-supported design methodology for implementing embedded control systems on platforms of possibly distributed sensors, actuators, CPU's, and networks. It is specifically intended to decouple the software design (the logical functionality) from implementation concerns (scheduling, communication, and mapping to physical resources) which makes it compatible with our distinction between logical and physical architectures.

In order to be used for the skechted purpose, Giotto must be supplemented with a middleware layer to support the illusion of local action. We assess existing middleware such as DCOM or CORBA and its real-time extension as too complex and thus as unsuitable for real-time applications. Instead we propose a lean

middleware concept that we call the delegate object mechanism [Bro01]. This concept is tailored to distributed real-time control system and is designed to be implemented upon a Giotto infrastructure.

We believe that the Giotto toolset combined with the delegate object mechanism automates the mapping of the functional architecture to the physical architecture, generating as an output the executable components ready to be downloaded on the embedded target.

5 Conclusions

Table 1 sketches possible means for automating the development of embedded systems. We view framework technology as key factor for coming up with reusable components for the functional aspects aside of control algorithms. A lean middleware, such as the delegate object mechanism together with Giotto could automate the mapping to a physical system and provide timing determinism. Due to the fact that the development of significant portions of embedded control system can be automated as sketched in the previous sections, we expect a thorough transformation of the embedded software market over the next decades similar to the transformation of the commercial software market through SAP since the 1970s.

Table 1. Means for automating embedded software development.

Logical System	Control Algorithms	Automation Approach
	20-30% of the logical system	reuse of models
		autocoding tools
	Management	
	70-80% of the logical system	reuse of framework components
Physical System	**Timing Determinism**	formal methods and tools
	Distribution	middleware based on formal methods and tools

References

[Ast01] M. Astley, D.C. Sturman, and G.A. Agha, *Object-based Middleware*, sidebar in Customizable Middleware for Modular Distributed Software, Communications of the ACM, Vol. 44, No. 5, May 2001.

[Bro01] T. Brown, A. Pasetti, W. Pree, T. Henzinger, C. Kirsch, *A Reusable and platform-independent framework for distributed control systems*, Proceedings of the 20-th Digital Avionics Systems Conference, Daytona Beach, FL, 14-18 October 2001

[Fay99] M. Fayad, D. Schmidt, R. Johnson, *Building Application Frameworks*, Wiley Computing Publishing, 1999

[Font01] M. Fontoura, W. Pree, B. Rumpe. The UML-F Profile for Framework Architectures. Addison-Wesley/Pearson Education, 2001.

[Gol95] A. Goldberg and K. Rubin. *Succeeding with Objects: Decision Frameworks for Project Management.* Addison-Wesley, 1995.

[Hen00] T.A. Henzinger, B. Horowitz, and C.M. Kirsch, *Giotto: A Time-triggered Language for Embedded Programming*, Technical report: UCB//CSD-00-1121, University of California, Berkeley, 2000.

[Hen01] T.A. Henzinger, B. Horowitz, and C.M. Kirsch, *Embedded Control Systems Development with Giotto*, Proceedings of LCTES 2001, ACM SIGPLAN Notices, June 2001.

[Kop97] H. Koptez *Real-Time Systems : Design Principles for Distributed Embedded Applications*, Kluwer Academic Publisher

[Mat99] M. Mattsson, J. Bosch, *Composition Problems, Causes, and Solutions,* p.467-487, in [Fay99]

[Mey89] The New Culture of Software Development: Reflections on the Practice of Object-Oriented Design; in Proceedings of Tools Europe 89, Paris, France.

[Pas99] A. Pasetti, W. Pree, *A Component Framework for Satellite On-Board Software*, Proceedings of the 18-th Digital Avionics Systems Conference, St. Louis (USA), Oct. 99

[Pas01a] A. Pasetti, *et al, An Object-Oriented Component-Based Framework for On-Board Software*, Proceedings of the Data Systems In Aerospace Conference, Nice, May 2001,

[Pas01b] A. Pasetti, *A Software Framework for Satellite Control Systems – Methodology and Development*, PhD Thesis, University of Konstanz, 2001, to be published in the Springer-Verlag LNCS monograph series.

[Pas01c] http://www.SoftwareResearch.net/AocsFrameworkProject/ProjectOverview.html

[Sun01] http://java.sun.com/products/javabeans/

[Wei89] Weinand A., Gamma E. and Marty R. Design and Implementation of ET++, a Seamless Object-Oriented Application Framework. *Structured Programming,* **10**(2), Springer Verlag, 1989.

An End-to-End Methodology for Building Embedded Systems

Ragunathan (Raj) Rajkumar

Real-Time and Multimedia Systems Laboratory
Departments of ECE and CS
Carnegie Mellon University
raj@ece.cmu.edu

Abstract. Embedded systems have been and will be deployed on a wide scale. Their applications range from bread toasters with 4-bit microcontrollers to large distributed systems on a large factory floor or a battleship. These systems also operate under a wide spectrum of constraints: size constraints, power consumption needs, processing power and communication bandwidth. Multiple domains of expertise in control systems, signal processing, communications, human-computing interfaces, fault-tolerance, testing and systems integration also come into play necessitating several system models, perspectives and sets of terms. As a result, solutions and techniques for building embedded systems have tended to be very specialized, piece-meal and fragmented in nature. Thanks to Moore's Law and advances in software engineering, products and standards, future embedded systems can now be based on common frameworks and approaches.

The IMAGES (Integrated Modeling for Analysis and Generation of Embedded Software) project at Carnegie Mellon University is developing an integrated framework and toolset that can capture, model and analyze all steps of an end-to-end methodology to design embedded systems ranging from cost-constrained automotive control systems to large-scale avionics applications. The integrated toolset will interface with both commercial tools such as Rational Rose, TimeWiz from TimeSys Corporation, Matlab from MathWorks as well as research prototypes developed at Carnegie Mellon University. The framework enables multiple capabilities that have been done in compartmentalized ways, if at all, in a single coherent inter-operable manner. These capabilities include analyses (timing analysis, fault management analysis, Quality of Service analysis), modeling (hybrid systems modeling, resource consumption, event dependencies, software architecture capture), software reusability (using customizable component libraries), run-time services (RTOS and middleware abstractions, code-generation, target-specific optimizations), and verification (model checking, test-vector generation). The integration of the various system models is enabled by the use of a consistent and well-understood run-time scheduling framework based on event-driven priority-based preemptive scheduling and resource reservations.

System modeling, analysis and run-time data can be stored in a common data repository, which allows the interoperability of multiple tools operating on different models of the system. Model checking can be used

T.A. Henzinger and C.M. Kirsch (Eds.): EMSOFT 2001, LNCS 2211, pp. 287–288, 2001.
© Springer-Verlag Berlin Heidelberg 2001

to obtain the worst-case execution times of code segments, as well as to verify logical properties of the embedded application. Both design-time and run-time attributes of software components can be manipulated rendering embedded software components both customizable and reusable. Real-time UML and UML models can also be used to perform traditional modeling of the software components. Hybrid systems techniques are used to verify both the continuous and discrete dynamics of the physical plant being controlled. Application-specific code generation can be automatically interfaced to work with communications glue code in a location-independent fashion by the use of run-time abstractions. The target environment can be rich, with real-time operating systems ranging from real-time versions of Linux to OSEK, and middleware ranging from none to Real-Time Java to Real-Time CORBA.

An Implementation of Scoped Memory for Real-Time Java

William S. Beebee, Jr. and Martin Rinard

MIT Laboratory for Computer Science
Massachusetts Institute of Technology, Cambridge MA, 02139
wbeebee@alum.mit.edu, rinard@lcs.mit.edu

Abstract. This paper presents our experience implementing the memory management extensions in the Real-Time Specification for Java. These extensions are designed to given real-time programmers the control they need to obtain predictable memory system behavior while preserving Java's safe memory model. We describe our implementation of certain dynamic checks required by the Real-Time Java extensions. In particular, we describe how to perform these checks in a way that avoids harmful interactions between the garbage collector and the memory management system. We also found that extensive debugging support was necessary during the development of Real-Time Java programs. We therefore used a static analysis and a dynamic debugging package during the development of our benchmark applications.

1 Introduction

Java is a relatively new and popular programming language. It provides a safe, garbage-collected memory model (no dangling references, buffer overruns, or memory leaks) and enjoys broad support in industry. The goal of the Real-Time Specification for Java [3] is to extend Java to support key features required for writing real-time programs. These features include support for real-time scheduling and predictable memory management.

This paper presents our experience implementing the Real-Time Java memory management extensions. The goal of these extensions is to preserve the safety of the base Java memory model while giving the real-time programmer the additional control that he or she needs to develop programs with predictable memory system behavior. In the base Java memory model, all objects are allocated out of a single garbage-collected heap, raising the issues of garbage-collection pauses and unbounded object allocation times.

Real-Time Java extends this memory model to support two new kinds of memory: *immortal memory* and *scoped memory*. Objects allocated in immortal memory live for the entire execution of the program. The garbage collector scans objects allocated in immortal memory to find (and potentially change) references into the garbage collected heap but does not otherwise manipulate these objects.

Each scoped memory conceptually contains a preallocated region of memory that threads can enter and exit. Once a thread enters a scoped memory, it can

T.A. Henzinger and C.M. Kirsch (Eds.): EMSOFT 2001, LNCS 2211, pp. 289–305, 2001.
© Springer-Verlag Berlin Heidelberg 2001

allocate objects out of that memory, with each allocation taking a predictable amount of time. When the thread exits the scoped memory, the implementation deallocates all objects allocated in the scoped memory without garbage collection. The specification supports nested entry and exit of scoped memories, which threads can use to obtain a stack of active scoped memories. The lifetimes of the objects stored in the inner scoped memories are contained in the lifetimes of the objects stored in the outer scoped memories. As for objects allocated in immortal memory, the garbage collector scans objects allocated in scoped memory to find (and potentially change) references into the garbage collected heap but does not otherwise manipulate these objects.

The Real-Time Java specification uses dynamic access checks to prevent dangling references and ensure the safety of using scoped memories. If the program attempts to create either 1) a reference from an object allocated in the heap to an object allocated in a scoped memory or 2) a reference from an object allocated in an outer scoped memory to an object allocated in an inner scoped memory, the specification requires the implementation to throw an exception.

1.1 Threads and Garbage Collection

The Real-Time Java thread and memory management models are tightly intertwined. Because the garbage collector may temporarily violate key heap invariants, it must be able to suspend any thread that may interact in any way with objects allocated in the garbage-collected heap. Real-Time Java therefore supports two kinds of threads: real-time threads, which may access and refer to objects stored in the garbage-collected heap, and no-heap real-time threads, which may not access or refer to these objects. No-heap real-time threads execute asynchronously with the garbage collector; in particular, they may execute concurrently with or suspend the garbage collector at any time. On the other hand, the garbage collector may suspend real-time threads at any time and for unpredictable lengths of time.

The Real-Time Java specification uses dynamic heap checks to prevent interactions between the garbage collector and no-heap real-time threads. If a no-heap real-time thread attempts to manipulate a reference to an object stored in the garbage-collected heap, the specification requires the implementation to throw an exception. We interpret the term "manipulate" to mean read or write a memory location containing a reference to an object stored in the garbage collected heap, or to execute a method with such a reference passed as a parameter.

1.2 Implementation

The primary complication in the implementation is potential interactions between no-heap real-time threads and the garbage collector. One of the basic design goals in the Real-Time Java specification is that the presence of garbage collection should never affect the ability of the no-heap real-time thread to run. We devoted a significant amount of time and energy working with our design to convince ourselves that the interactions did in fact operate in conformance with the specification.

1.3 Debugging

We found it difficult to use scoped and immortal memories correctly, especially in the presence of the standard Java libraries, which were not designed with the Real-Time Specification for Java in mind. We therefore found it useful to develop some debugging tools. These tools included a static analysis which finds incorrect uses of scoped memories and a dynamic instrumentation system that enabled the implementation to print out information about the sources of dynamic check failures.

2 Programming Model

Because of the proliferation of different kinds of memory areas and threads, Real-Time Java has a fairly complicated programming model.

2.1 Entering and Exiting Memory Areas

Real-Time Java provides several kinds of memory areas: scoped memory, immortal memory, and heap memory. Each thread maintains a stack of memory areas; the memory area on the top of the stack is the thread's default memory area. When the thread creates a new object, it is allocated in the default memory area unless the thread explicitly specifies that the object should be allocated in some other memory area. If a thread uses this mechanism to attempt to allocate an object in a scoped memory, the scoped memory must be present in the thread's stack of memory areas. No such restriction exists for objects allocated in immortal or heap memory.

Threads can enter and exit memory areas. When a thread enters a memory area, it pushes the area onto its stack. When it exits the memory area, it pops the area from the stack. There are two ways to enter a memory area: start a parallel thread whose initial stack contains the memory area, or sequentially execute a run method that executes in the memory area. The thread exits the memory area when the run method returns.

The programming model is complicated somewhat by the fact that 1) a single thread can reenter a memory area multiple times, and 2) different threads can enter memory areas in different orders. Assume, for example, that we have two scoped memories A and B and two threads T and S. T can first enter A, then B, then A again, while S can first enter B, then A, then B again. The objects in A and B are deallocated only when T exits A, then B, then A again, and S exits B, then A, then B again. Note that even though the programming model specifies nested entry and exit of memory areas, these nested entries and exits do not directly translate into a hierarchical inclusion relationship between the lifetimes of different memory areas.

2.2 Scoped Memories

Scoped memories, in effect, provide a form of region-based memory allocation. They differ somewhat from other forms of region-based memory allocation [2]

in that each scoped memory is associated with one or more computations (each computation is typically a thread, but can also be the execution of a sequentially invoked run method), with all of the objects in the scoped memory deallocated when all of its associated computations terminate.

The primary issue with scoped memories is ensuring that their use does not create dangling references, which are references to objects allocated in scoped memories that have been deallocated. The basic strategy is to use dynamic access checks to prevent the program from creating a reference to an object in a scoped memory from an object allocated in either heap memory, immortal memory, or a scoped memory whose lifetime encloses that of the first scoped memory. Whenever a thread attempts to store a reference to a first object into a field in a second object, an access check verifies that:

> If the first object is allocated in a scoped memory, then the second object must also be allocated in a scoped memory whose lifetime is contained in the lifetime of the scoped memory containing the first object.

The implementation checks the containment by looking at the thread's stack of scoped memories and checking that either 1) the objects are allocated in the same scoped memory, or 2) the thread first entered the scoped memory of the second object before it first entered the scoped memory of the first object. If this check fails, the implementation throws an exception.

Let's consider a quick example to clarify the situation. Assume we have two scoped memories A and B, two objects O and P, with O allocated in A and P allocated in B, and two threads T and S. Also assume that T first enters A, then B, then A again, while S first enters B, then A, then B again. Now T can store a reference to O in a field of P, but cannot store a reference to P in a field of O. For S, the situation is reversed: S cannot store a reference to O in a field of P, but can store a reference to P in a field of O.

2.3 No-Heap Real-Time Threads

No-heap real-time threads have an additional set of restrictions; these restrictions are intended to ensure that the thread does not interfere with the garbage collector. Specifically, the Real-Time Specification for Java states that a no-heap real-time thread, which can run asynchronously with the garbage collector, "is never allowed to allocate or reference any object allocated in the heap nor is it even allowed to manipulate the references to objects in the heap." Our implementation uses five runtime heap checks to ensure that a no-heap real-time thread does not interfere with garbage collection by manipulating heap references. The implementation uses three of these types of checks, **CALL**, **METHOD**, and **NATIVECALL** to guard against poorly implemented native methods or illegal compiler calls into the runtime. These three checks can be removed if all native and runtime code is known to operate correctly.

- **CALL**: A native method invoked by a no-heap real-time thread cannot return a reference to a heap allocated object.

- **METHOD**: A Java method cannot be passed a heap allocated object as an argument while running in a no-heap real-time thread.

- **NATIVECALL**: A compiler-generated call into the runtime implementation from a no-heap real-time thread cannot return a reference to a heap allocated object.

- **READ**: A no-heap real-time thread cannot read a reference to a heap allocated object.

- **WRITE**: As part of the execution of an assignment statement, a no-heap real-time thread cannot overwrite a reference to a heap allocated object.

3 Example

We next present an example that illustrates some of the features of the Real-Time Specification for Java. Figure 1 presents a sample program written in Real-Time Java. This program is a version of the familiar "Hello World" program augmented to use the Real-Time Java features. It first creates a scoped memory with a worst-case Linear Time allocation scheme (`LTMemory`) with a size of 1000 bytes. It then runs the code of the `run` method in this new scope. The `run` method creates a new variable time allocation scoped memory (the `VTMemory` object) and a new `Worker NoHeapRealtimeThread`. Both of these objects are allocated in the `LTMemory` scoped memory. The `run` method then starts the `Worker` thread and executes its `join` method, which will return when the `Worker` finishes.

The `Worker` thread runs in the new `VTMemory`. The `Worker`'s `run` method allocates a new `String`[1] in `ImmortalMemory` and stores a reference to this string in the static `results` field of the `Main` class, which was previously initialized to `null`. The `Worker` then creates a new `String`, "Hello World!", to place in the array. The worker then finishes, and the implementation deallocates all of the objects allocated in the `VTMemory`. Back in the main thread, the `join` method returns, and the main thread returns back out of its `run` method. The implementation deallocates all of the objects allocated in the `LTMemory`. Finally, the main thread prints "Hello World", the first element of the `results` array, to the screen.

Note that the `LTMemory` and `VTMemory` constructors differ slightly from the constructors described in the Realtime Java specification. We implemented these constructors in addition to the specified constructors to provide additional flexibility and convenience for the programmer.

This Hello World program is a legal program using our system. However, any of the following changes would make it an illegal program:

1. Replace the `im.newInstance...` with `''Hello World!''` and there would be an illegal reference from an `ImmortalMemory` to a `ScopedMemory`.
2. Replace the `im.newArray...` with `new String[1]` and there would be an illegal static reference to a `ScopedMemory`.

```
class Worker extends NoHeapRealtimeThread {
  Worker(MemoryArea ma) { super(ma); }
  public void run() {
    ImmortalMemory im = ImmortalMemory.instance();
    try {
      Main.results = (String[])im.newArray(String.class, new int[] { 1 });
      Main.results[0] = (String)im.newInstance(String.class,
                                               new Class[] { String.class },
                                               new Object[] { ``Hello World!'' });
    } catch (Exception e) { System.exit(-1); }
  }
}
public class Main {
  public static String[] results = null;
  public static void main(String args[]) {
    LTMemory lt = new LTMemory(1000);
    lt.enter(new Runnable() {
      public void run() {
        Worker w = new Worker(new VTMemory());
        w.start();
        try { w.join(); } catch (Exception e) { System.out.println(e); }
      }
    });
    System.out.println(results[0]);
  }
}
```

Fig. 1. A Real-Time Java Example Program

3. Replace the `ImmortalMemory.instance()` with `HeapMemory.instance()`, and there would be an illegal heap reference in a `NoHeapRealtimeThread` (**READ**).
4. Replace the `null` with a `new String[1]` and the `NoHeapRealtimeThread` would be illegally destroying a heap reference by assigning `Main.results` (**WRITE**).
5. Place the `Worker w` in the `main` method and the assignment `w = new Worker...` would illegally create a reference from the heap to a `ScopedMemory`.
6. Place the `System.out` in the `NoHeapRealtimeThread` and the `NoHeapRealtimeThread` would be illegally reading from the heap. `System.out` is initialized in the initial `MemoryArea` at the start of the program, the `HeapMemory` (**READ**) As a consequence, the `NoHeapRealtimeThread` cannot `System.out.println` the message from the exception.
7. Place the entire `Worker w = new Worker(new VTMemory());` outside the `LTMemory` scope, and the `this` pointer of the `NoHeapRealtimeThread` would illegally point to the heap (**METHOD**).

4 Implementation

Our discussion of the implementation focuses on three aspects: implementing the heap and access checks, implementing the additional scoped immortal memory functionality, and ensuring the absence of interactions between no-heap real-time threads and the garbage collector.

4.1 Heap Check Implementation

The implementation must be able to take an arbitrary reference to an object and determine the kind of memory area in which it is allocated. To support this functionality, our implementation adds an extra field to the header of each object. This field contains a pointer to the memory area in which the object is allocated.

One complication with this scheme is that the garbage collector may violate object representation invariants during collection. If a no-heap real-time thread attempts to use the field in the object header to determine if an object is allocated in the heap, it may access memory rendered invalid by the actions of the garbage collector. We therefore need a mechanism which enables a no-heap real-time thread to differentiate between heap references and other references without attempting to access the memory area field of the object.

We first considered allocating a contiguous address region for the heap, then checking to see if the reference falls within this region. We decided not to use this approach because of potential interactions between the garbage collector and the code in the no-heap real-time thread that checks if the reference falls within the heap. Specifically, using this scheme would force the garbage collector to always maintain the invariant that the current heap address region include all previous heap address regions. We were unwilling to impose this restriction on the collector.

We then considered a variety of other schemes, but eventually settled on the (relatively simple) approach of setting the low bit of all heap references. The generated code masks off this bit before dereferencing the pointer to access the object. With this approach, no-heap real-time threads can simply check the low bit of each reference to check if the reference points into the heap or not.

Our current system uses the memory area field in the object header to obtain information about objects allocated in scoped memories and immortal memory. The basic assumption is that the objects allocated in these kinds of memory areas will never move or have their memory area field temporarily corrupted or invalidated.

Figure 2 presents the code that the compiler emits for each heap check; Figure 3 presents the code that determines if the current thread is a no-heap real-time thread. Note that the emitted code first checks to see if the reference is a heap reference — our expectation is that most Real-Time Java programs will manipulate relatively few references to heap-allocated objects. This expectation holds for our benchmark programs (see Section 6).

READ	WRITE	CALL
use of `*refExp` in `exp`	`*refExp = exp;`	`refExp = call(args);`
becomes:	becomes:	becomes:

`heapRef = *refExp;`
`if (heapRef&1)`
`  heapCheck(heapRef);`
`[*heapRef/*refExp] exp`
	`heapRef = *refExp;`
	`if (heapRef&1)`
	`  heapCheck(heapRef);`
	`refExp = exp;`
		`heapRef = call(args);`
		`if (heapRef&1)`
		`  heapCheck(heapRef);`
		`refExp = heapRef;`

NATIVECALL	METHOD
`refExp = nativecall(args);`	`method(args) { body }`
becomes:	becomes:

`heapRef = nativecall(args);`
`if (heapRef&1)`
`  heapCheck(heapRef);`
`refExp = heapRef;`
	`method(args) {`
	`  for arg in args:`
	`  if (arg&1)`
	`  heapCheck(arg);`
	`body }`

Fig. 2. Emitted Code For Heap Checks

```
#ifdef DEBUG
  void heapCheck(unwrapped_jobject* heapRef, const int source_line,
                 const char* source_fileName, const char* operation) {
#else       /* operation = READ, WRITE, CALL, NATIVECALL, or METHOD */
  void heapCheck(unwrapped_jobject* heapRef) {
#endif
    JNIEnv* env = FNI_GetJNIEnv();
    /* determine if in a NoHeapRealtimeThread */
    if (((struct FNI_Thread_State*)env)->noheap) {
       /* optionally print helpful debugging info */
       /* throw exception */
    }
  }
```

Fig. 3. The `heapCheck` function

4.2 Access Check Implementation

The access checks must be able to determine if the lifetime of a scoped memory area A is included in the lifetime of another scoped memory area B. The implementation searches the thread's stack of memory areas to perform this check. It first searches for the occurrence of A closest to the start of the stack (recall that A may occur multiple times on the stack). It then searches to check if there is

an occurrence of B between that occurrence of A and the start of the stack. If so, the access check succeeds; otherwise, it fails.

The current implementation optimizes this check by first checking to see if A and B are the same scoped memory area. Figure 4 presents the emitted code for the access checks, while Figure 5 presents some of the run-time code that this emitted code invokes.

New Object (or Array):

```
obj = new foo(); (or obj = new foo()[1][2][3];)
```

becomes:

```
ma = RealtimeThread.currentRealtimeThread().getMemoryArea();
obj = new foo(); (or obj = new foo()[1][2][3];)
obj.memoryArea = ma;
```

Access check:

```
obj.foo = bar;
```

becomes:

```
ma = MemoryArea.getMemoryArea(obj); /* or ma = ImmortalMemory.instance(),
ma.checkAccess(bar);                   if a static field) */
obj.foo = bar;
```

Fig. 4. Emitted Code for Access Checks

4.3 Operations on Memory Areas

The implementation needs to perform three basic operations on scoped and immortal memory areas: allocate an object in the area, deallocate all objects in the area, and provide the garbage collector with the set of all heap references stored in the memory area. Note a potential interaction between the garbage collector and no-heap real-time threads. The garbage collector may be in the process of retrieving the heap references stored in a memory area when a no-heap real-time thread (operating concurrently with or interrupting the garbage collector) allocates objects in that memory area. The garbage collector must operate correctly in the face of the resulting changes to the underlying memory area data structures. The system design also cannot involve locks shared between the no-heap real-time thread and the garbage collector (the garbage collector is not allowed to block a no-heap real-time thread). But the garbage collector may assume that the actions of the no-heap real-time thread do not change the set of heap references stored in the memory area.

Each memory area may have its own object allocation algorithm. Because the same code may execute in different memory areas at different times, our

In `MemoryArea`:

```
public void checkAccess(Object obj) {
  if ((obj != null) && (obj.memoryArea != null) && obj.memoryArea.scoped) {
    /* Helpful native method prints out all debugging info. */
    throwIllegalAssignmentError(obj, obj.memoryArea);
  }
}
```

Overridden in `ScopedMemory`:

```
public void checkAccess(Object obj) {
  if (obj != null) {
    MemoryArea target = getMemoryArea(obj);
    if ((this != target) && target.scoped &&
        (!RealtimeThread.currentRealtimeThread()
         .checkAccess(this, target))) {
        throwIllegalAssignmentError(obj, target);
    }
  }
}
```

In `RealtimeThread`:

```
boolean checkAccess(MemoryArea source, MemoryArea target) {
  MemBlockStack sourceStack = (source == getMemoryArea()) ?
    memBlockStack : memBlockStack.first(source);
  return (sourceStack != null) && (sourceStack.first(target) != null);
}
```

Fig. 5. Code for performing access checks

implementation is set up to dynamically determine the allocation algorithm to use based on the current memory area. Whenever a thread allocates an object, it looks up a data structure associated with the memory area. A field in this structure contains a pointer to the allocation function to invoke. This structure also contains a pointer to a function that retrieves all of the heap references from the area, and a function that deallocates all of the objects allocated in the area.

4.4 Memory Area Reference Counts

As described in the Real-Time Java Specification, each memory area maintains a count of the number of threads currently operating within that region. These counts are (atomically) updated when threads enter or exit the region. When the count becomes zero, the implementation deallocates all objects in the area.

Consider the following situation. A thread exits a memory area, causing its reference count to become zero, at which point the implementation starts to invoke finalizers on the objects in the memory area as part of the deallocation

process. While the finalizers are running, a no-heap real-time thread enters the memory area. According to the Real-Time Java specification, the no-heap real-time thread blocks until the finalizers finish running. There is no mention of the priority with which the finalizers run, raising the potential issue that the no-heap real-time thread may be arbitrarily delayed. A final problem occurs if the no-heap real-time thread first acquires a lock, a finalizer running in the memory area then attempts to acquire the lock (blocking because the no-heap real-time thread holds the lock), then the no-heap real-time thread attempts to enter the memory area. The result is deadlock — the no-heap real-time thread waits for the finalizer to finish, but the finalizer waits for the no-heap real-time thread to release the lock.

4.5 Memory Allocation Algorithms

We have implemented two simple allocators for scoped memory areas: a stack allocator and a `malloc`-based allocator. The current implementation uses the stack allocator for instances of `LTMemory`, which guarantee linear-time allocation, and the `malloc`-based allocator for instances of `VTMemory`, which provide no time guarantees.

The stack allocator starts with a fixed amount of available free memory. It maintains a pointer to the next free address. To allocate a block of memory, it increments the pointer by the size of the block, then returns the old value of the pointer as a reference to the newly allocated block. Our current implementation uses this allocation strategy for instances of the `LTMemory` class, which guarantees a linear time allocation strategy.

There is a complication associated with this implementation. Note that multiple threads can attempt to concurrently allocate memory from the same stack allocator. The implementation must therefore use some mechanism to ensure that the allocations take place atomically. Note that the use of lock synchronization could cause an unfortunate coupling between real-time threads, no-heap real-time threads, and the garbage collector. Consider the following scenario. A real-time thread starts to allocate memory, acquires the lock, is suspended by the garbage collector, which is then suspended by a no-heap real-time thread that also attempts to allocate memory from the same allocator. Unless the implementation does something clever, it could either deadlock or force the no-heap real-time thread to wait until the garbage collector releases the real-time thread to complete its memory allocation.

Our current implementation avoids this problem by using a lock-free, non-blocking atomic exchange-and-add instruction to perform the pointer updates. Note that on an multiprocessor in the presence of contention from multiple threads attempting to concurrently allocate from the same memory allocator, this approach could cause the allocation time to depend on the precise timing behavior of the atomic instructions. We would expect some machines to provide no guarantee at all about the termination time of these instructions.

The `malloc`-based allocator simply calls the standard `malloc` routine to allocate memory. Our implementation uses this strategy for instances of `LTMemory`.

To provide the garbage collector with a list of heap references, our implementation keeps a linked list of the allocated memory blocks and can scan these blocks on demand to locate references into the heap.

Our design makes adding a new allocator easy; the `malloc`-based allocator required only 25 lines of C code and only 45 minutes of coding, debugging, and testing time. Although the system is flexible enough to support multiple dynamically-changing allocation routines, `VTMemory`s use the linked-list allocator, while `LTMemory`s use the stack-allocator.

4.6 Garbage Collector Interactions

References from heap objects can point both to other heap objects and to objects allocated in immortal memory. The garbage collector must therefore recognize references to immortal memory and treat objects allocated in immortal memory differently than objects allocated in heap memory. In particular, the garbage collector cannot change the objects in ways that that would interact with concurrently executing no-heap real-time threads.

Our implementation handles this issue as follows. The garbage collector first scans the immortal and scoped memories to extract all references from objects allocated in these memories to heap allocated objects. This scan is coded to operate correctly in the presence of concurrent updates from no-heap real-time threads. The garbage collector uses the extracted heap references as part of its root set.

During the collection phase, the collector does not trace references to objects allocated in immortal memory. If the collector moves objects, it may need to update references from objects allocated in immortal memory or scoped memories to objects allocated in the heap. It performs these updates in such a way that it does not interfere with the ability of no-heap real-time threads to recognize such references as referring to objects allocated in the heap. Note that because no-heap real-time threads may access heap references only to perform heap checks, this property ensures that the garbage collector and no-heap real-time threads do not inappropriately interfere.

5 Debugging Real-Time Java Programs

An additional design goal becomes extremely important when actually developing Real-Time Java programs: ease of debugging. During the development process, facilitating debugging became a primary design goal. In fact, we found it close to impossible to develop error-free Real-Time Java programs without some sort of assistance (either a debugging system or static analysis) that helped us locate the reason for our problems using the different kinds of memory areas. Our debugging was especially complicated by the fact that the standard Java libraries basically don't work at all with no-heap real-time threads.

5.1 Incremental Debugging

During our development of Real-Time Java programs, we found the following incremental debugging strategy to be useful. We first stubbed out all of the Real-Time Java heap and access checks and special memory allocation strategies, in effect running the Real-Time Java program as a standard Java program. We used this version to debug the basic functionality of the program. We then added the heap and access checks, and used this version to debug the memory allocation strategy of the program. We were able to use this strategy to divide the debugging process into stages, with a manageable amount of bugs found at each stage.

It is also possible to use static analysis to verify the correct use of Real-Time Java scoped memories [5]. We had access to such an analysis when we were implementing our benchmark programs, and the analysis was very useful for helping us debug our use of scoped memories. It also dramatically increased our confidence in the correctness of the final program, and enabled a static check elimination optimization that improved the performance of the program.

5.2 Additional Runtime Debugging Information

Heap and access checks can be used to help detect mistakes early in the development process, but additional tools may be necessary to understand and fix those mistakes in a timely fashion. We therefore augmented the memory area data structure to produce a debugging system that helps programmers understand the causes of object referencing errors.

When a debugging flag is enabled, the implementation attaches the original Java source code file name and line number to each allocated object. Furthermore, with the use of macros, we also obtain allocation site information for native methods. We store this allocation site information in a list associated with the memory area in which the object is allocated. Given any arbitrary object reference, a debugging function can retrieve the debugging information for the object. Combined with a stack trace at the point of an illegal assignment or reference, the allocation site information from both the source and destination of an illegal assignment or the location of an illegal reference can be instrumental in quickly determining the exact cause of the error and the objects responsible. Allocation site information can also be displayed at the time of allocation to provide a program trace which can help determine control flow, putting the reference in a context at the time of the error.

6 Results

We implemented the Real-Time Java memory extensions in the MIT Flex compiler infrastructure.[1] Flex is an ahead-of-time compiler for Java that generates both native code and C; it can use a variety of garbage collectors. For these

[1] Available at `www.flexc.lcs.mit.edu`

Table 1. Number of Objects Allocated In Different Memory Areas

Benchmark	Heap	Scoped	Immortal	Total
Array	13	4	0	17
Tree	13	65,534	0	65,547
Water	406,895	3,345,711	0	3,752,606
Barnes	16,058	4,681,708	0	4,697,766

Table 2. Number of Arrays Allocated In Different Memory Areas

Benchmark	Heap	Scoped	Immortal	Total
Array	36	4	0	40
Tree	36	0	0	36
Water	405,943	13,160,641	0	13,566,584
Barnes	14,871	4,530,765	0	4,545,636

experiments, we generated C and used the Boehm-Demers-Weiser conservative garbage collector.

We obtained several benchmark programs and used these programs to measure the overhead of the heap checks and access checks. Our benchmarks include Barnes, a hierarchical N-body solver, and Water, which simulates water molecules in the liquid state. Initially these benchmarks allocated all objects in the heap. We modified the benchmarks to use scoped memories whenever possible. We also present results for two synthetic benchmarks, Tree and Array, that use object field assignment heavily. These benchmarks are designed to obtain the maximum possible benefit from heap and access check elimination.

Table 1 presents the number of objects we were able to allocate in each of the different kinds of memory areas. The goal is to allocate as many objects as possible in scoped memory areas; the results show that we were able to modify the programs to allocate the vast majority of their objects in scoped memories. Java programs also allocate arrays; Table 2 presents the number of arrays that we were able to allocate in scoped memories. As for objects, we were able to allocate the vast majority of arrays in scoped memories.

Table 3 presents the number and type of access checks for each benchmark. Recall that there is a check every time the program stores a reference. The different columns of the table break down the checks into categories depending on the target of the store and the memory area that the stored reference refers to. For example, the Scoped to Heap column counts the number of times the program stored a reference to heap memory into an object or array allocated in a scoped memory. Table 4 presents the running times of the benchmarks. We report results for six different versions of the program. The first three versions all have both heap and access checks, and vary in the memory area they use for objects that we were able to allocate in scoped memory. The Heap version allocates all objects in the heap. The VT version allocates scoped-memory objects in instances of

`VTMemory` (which use `malloc`-based allocation); the LT version allocates scoped-memory objects in instances of `LTMemory` (which use stack-based allocation). The next three versions use the same allocation strategy, but the compiler generates code that omits all of the checks. For our benchmarks, our static analysis is able to verify that none of the checks will fail, enabling the compiler to eliminate all of these checks [5].

Table 3. Access Check Counts

Benchmark	Heap to Heap	Heap to Immortal	Scoped to Heap	Scoped to Scoped	Scoped to Immortal	Immortal to Heap	Immortal to Immortal
Array	14	8	0	400,040,000	0	0	0
Tree	14	8	0	65,597,532	65,601,536	0	0
Water	409,907	0	17,836	9,890,211	844	3	1
Barnes	90,856	80,448	9,742	4,596,716	1328	0	0

Table 4. Execution Times of Benchmark Programs

	With Checks			Without Checks		
Benchmark	Heap	VT	LT	Heap	VT	LT
Array	28.1	43.2	43.1	7.8	7.7	8.0
Tree	13.2	16.6	16.6	6.9	6.9	6.9
Water	58.2	47.4	37.8	52.3	40.2	30.2
Barnes	38.3	22.3	17.2	34.7	19.5	14.4

These results show that checks add significant overhead for all benchmarks. But the use of scoped memories produces significant performance gains for Barnes and Water. In the end, the use of scoped memories without checks significantly increases the overall performance of the program. To investigate the causes of the performance differences, we instrumented the run-time system to measure the garbage collection pause times. Based on these measurements, we attribute most of the performance differences between the versions of Water and Barnes with and without scoped memories to garbage collection overheads. Specifically, the use of scoped memories improved every aspect of the garbage collector: it reduced the total garbage collection overhead, increased the time between collections, and significantly reduced the pause times for each collection.

For Array and Tree, there is almost no garbage collection for any of the versions and the versions without checks all exhibit basically the same performance. With checks, the versions that allocate all objects in the heap run faster than the versions that allocate objects in scoped memories. We attribute this performance difference to the fact that heap to heap access checks are faster than scope to scope access checks.

7 Related Work

Christiansen and Velschow suggested a region-based approach to memory management in Java; they called their system RegJava[4]. They found that fixed-size regions have better performance than variable-sized regions and that region allocation has more predictable and often better performance than garbage collection. Static analysis can be used to detect where region annotations should be placed, but the annotations often need to be manually modified for performance reasons. Compiling a subset of Java which did not include threads or exceptions to C++, the RegJava system does not allow regions to coexist with garbage collection. Finally, the RegJava system permits the creation of dangling references.

Gay and Aiken implemented a region-based extension of C called C@ which used reference counting on regions to safely allocate and deallocate regions with a minimum of overhead[1]. Using special region pointers and explicit `deleteregion` calls, Gay and Aiken provide a means of explicitly manipulating region-allocated memory. They found that region-based allocation often uses less memory and is faster than traditional malloc/free-based memory management. Unfortunately, counting escaping references in C@ can incur up to 16% overhead. Both Christiansen and Velschow and Gay and Aiken explore the implications of region allocation for enhancing locality.

Gay and Aiken also produced RC [2], an explicit region allocation dialect of C, and an improvement over C@. RC uses heirarchically structured regions and `sameregion`, `traditional`, and `parentptr` pointer annotations to reduce the reference counting overhead to at most 11% of execution time. Using static analysis to reduce the number of safety checks, RC demonstrates up to a 58% speedup in programs that use regions as opposed to garbage collection or the typical malloc and free. RC uses 8KB aligned pages to allocate memory and the runtime keeps a map of pages to regions to resolve `regionof` calls quickly. Regions have a partial order to facilitate `parentptr` checks.

Region analysis seems to work best when the programmer is aware of the analysis, indicating that explicitly defined regions which give the programmer control over storage allocation may lead to more efficient programs. For example, the Tofte/Talpin ML inference system required that the programmer be aware of the analysis to guard against excessive memory leaks [6]. Programs which use regions explicitly may be more hierarchically structured with respect to memory usage by programmer design than programs intended for the traditional, garbage-collected heap. Therefore, Real-Time Java uses hierarchically-structured, explicit, reference-counted regions that strictly prohibit the creation of dangling references.

Our research is distinguished by the fact that Real-Time Java is a strict superset of the Java language; any program written in ordinary Java can run in our Real-Time Java system. Furthermore, a Real-Time Java thread which uses region allocation and/or heap allocation can run concurrently with a thread from any ordinary Java program, and we support several kinds of region-based allocation and allocation in a garbage collected heap in the same system.

8 Conclusion

The Real-Time Java Specification promises to bring the benefits of Java to programmers building real-time systems. One of the key aspects of the specification is extending the Java memory model to give the programmer more control over the memory management. We have implemented these extensions. We found that the primary implementation complication was ensuring a lack of interference between the garbage collector and no-heap real-time threads, which execute asynchronously with respect to the design. We also found debugging tools necessary for the effective development of programs that use the Real-Time Java memory management extensions. We used both a static analysis and a dynamic debugging system to help locate the source of incorrect uses of these extensions.

Acknowledgements. This research was supported in part by DARPA/AFRL Contract F33615-00-C-1692 as part of the Program Composition for Embedded Systems program. The authors would like to acknowledge Scott Ananian for a large part of the development of the MIT Flex compiler infrastructure and the Precise-C backend. Karen Zee implemented stop and copy and mark and sweep garbage collectors, which facilitated the development of no-heap real-time threads. Hans Boehm, Alan Demers, and Mark Weiser implemented the conservative garbage collector which was used for all of the listed benchmarks. Alex Salcianu tailored his escape analysis to verify the correctness of our Real-Time Java benchmarks with respect to scoped memory access violations. Brian Demsky implemented a user-threads package for the Flex compiler which improved the performance of some benchmarks.

References

1. Aiken, A., Gay, D.: Memory Management with Explicit Regions. Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation. Montreal, Canada, June 1998.
2. Aiken, A., Gay, D.: Language Support for Regions. Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation. Snowbird, Utah, June 2001.
3. Bollella, G., Brosgol, B., Dibble, P., Furr, S., Gosling, J., Hardin, D., Turnbull, M.: The Real-Time Specification for Java. Addison-Wesley, Reading, Massachusetts, 2000.
4. Christiansen, M., Velschow, P.: Region-Based Memory Management in Java. Master's thesis, Department of Computer Science (DIKU), University of Copenhagen, May 1998.
5. Salcianu, A., Rinard, M.: Pointer and Escape Analysis for Multithreaded Programs. Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. Snowbird, Utah, June 2001.
6. Tofte, M., Talpin, J.: Region-Based Memory Management. Information and Computation, 132(2):109–176 (1997)

Bus Architectures for Safety-Critical Embedded Systems*

John Rushby

Computer Science Laboratory
SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025, USA
rushby@csl.sri.com

Abstract. Embedded systems for safety-critical applications often integrate multiple "functions" and must generally be fault-tolerant. These requirements lead to a need for mechanisms and services that provide protection against fault propagation and ease the construction of distributed fault-tolerant applications. A number of bus architectures have been developed to satisfy this need. This paper reviews the requirements on these architectures, the mechanisms employed, and the services provided. Four representative architectures (SAFEbus™, SPIDER, TTA, and FlexRay) are briefly described.

1 Introduction

Embedded systems generally operate as closed-loop control systems: they repeatedly sample sensors, calculate appropriate control responses, and send those responses to actuators. In safety-critical applications, such as fly- and drive-by-wire (where there are no direct connections between the pilot and the aircraft control surfaces, nor between the driver and the car steering and brakes), requirements for ultra-high reliability demand fault tolerance and extensive redundancy. The embedded system then becomes a distributed one, and the basic control loop is complicated by mechanisms for synchronization, voting, and redundancy management.

Systems used in safety-critical applications have traditionally been *federated*, meaning that each "function" (e.g., autopilot or autothrottle in an aircraft, and brakes or suspension in a car) has its own fault-tolerant embedded control system with only minor interconnections to the systems of other functions. This provides a strong barrier to fault propagation: because the systems supporting different functions do not share resources, the failure of one function has little effect on the continued operation of others. The federated approach is expensive, however (because each function has its own replicated system), so recent applications are moving toward more integrated solutions in which some resources are shared across different functions. The new danger here is that faults

* This research was supported by the DARPA MOBIES and NEST programs through USAF Rome Laboratory contracts F33615-00-C-1700 and F33615-01-C-1908, and by NASA Langley Research Center under contract NAS1-20334 and Cooperative Agreement NCC-1-377 with Honeywell Incorporated.

T.A. Henzinger and C.M. Kirsch (Eds.): EMSOFT 2001, LNCS 2211, pp. 306–323, 2001.
© Springer-Verlag Berlin Heidelberg 2001

may propagate from one function to another; *partitioning* is the problem of restoring to integrated systems the strong defenses against fault propagation that are naturally present in federated systems. A dual issue is that of *strong composability*: here we would like to take separately developed functions and have them run without interference on an integrated system platform with negligible integration effort.

The problems of fault tolerance, partitioning, and strong composability are challenging ones. If handled in an ad-hoc manner, their mechanisms can become the primary sources of faults and of *un*reliability in the resulting architecture [10]. Fortunately, most aspects of these problems are independent of the particular functions concerned, and they can be handled in a principled and correct manner by generic mechanisms implemented as an architecture for distributed embedded systems.

One of the essential services provided by this kind of architecture is communication of information from one distributed component to another, so a (physical or logical) communication bus is one of its principal components, and the protocols used for control and communication on the bus are among its principal mechanisms. Consequently, these architectures are often referred to as *buses* (or *databuses*), although this term understates their complexity, sophistication, and criticality. In truth, these architectures are the safety-critical core of the applications built above them, and the choice of services to provide to those applications, and the mechanisms of their implementation, are issues of major importance in the construction and certification of safety-critical embedded systems.

In this paper, I survey some of the issues in the design of bus architectures for safety-critical embedded systems. I hope this will prove useful to potential users of these architectures and will alert others to the benefits of building on such well-considered foundations. My presentation is derived from a review of four representative architectures: two of these were primarily designed for aircraft applications and two for automobiles. The economies of scale make the automobile buses quite inexpensive—which then renders them attractive in certain aircraft applications. The aircraft buses considered are the Honeywell SAFEbus [1, 7] (the top-level avionics bus used in the Boeing 777) and the NASA SPIDER [11] (an architecture being developed as a demonstrator for certification under the new DO254 guidelines [15]); the automobile buses considered are the TTTech Time-Triggered Architecture (TTA) [24, 8], recently adopted by Audi and Volkswagen for automobile applications, and by Honeywell for avionics and aircraft controls functions, and FlexRay [3], which is being developed by a consortium of BMW, DaimlerChrysler, Motorola, and Philips. A detailed comparison of these four architectures, along with more extended discussion of the issues, is available as a technical report [17].

The paper is organized as follows: Section 2 examines general issues in time-triggered systems and bus architectures, Section 3 examines the fault hypotheses under which they operate, and Section 4 describes the services that they provide; Section 5 briefly describes the four representative architectures, and conclusions are provided in Section 6.

2 Time-Triggered Buses

The architectures considered here are called “buses” because multicast or broadcast communication is one of the services that they provide, and their implementations are

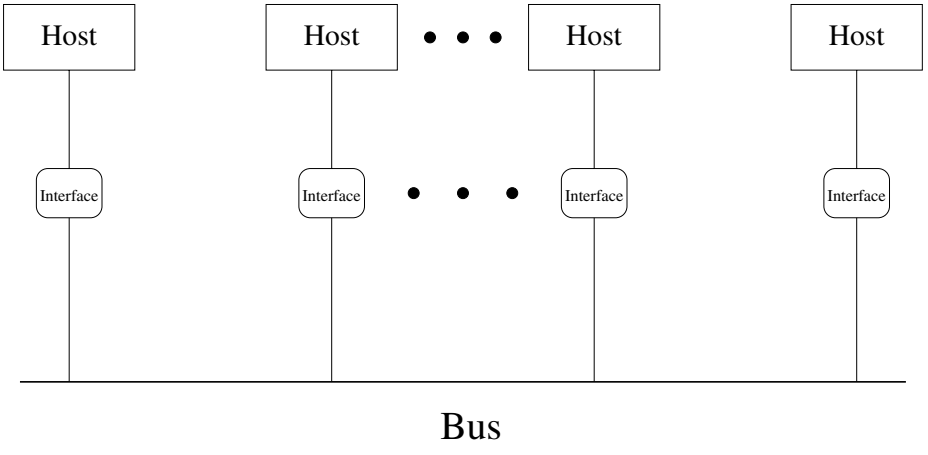


Fig. 1. Bus Interconnect

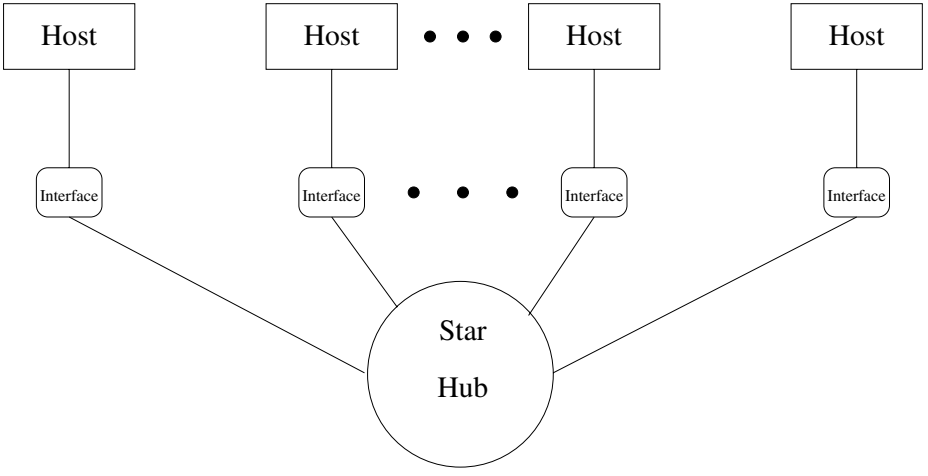


Fig. 2. Star Interconnect

based on a logical or physical bus. In a generic bus architecture, application programs run in *host* computers, and sensors and actuators are also connected to the hosts; an *interconnect* medium provides broadcast communications, and *interface* devices connect the hosts to the interconnect. The interfaces and interconnect comprise the bus; the combination of a host and its interface(s) is referred to as a *node*. Realizations of the interconnect may be a physical (passive) bus, as shown in Figure 1, or a centralized (active) hub, as shown in Figure 2. The interfaces may be physically proximate to the hosts, or they may form part of a more complex central hub. Many of the components will be replicated for fault tolerance.

All four of the buses considered here are primarily *time triggered*; this is a fundamental design choice that influences many aspects of their architectures and mechanisms, and sets them apart from fundamentally *event-triggered* buses such as Byteflight, CAN, Ethernet, LonWorks, or Profibus. The time-triggered and event-triggered approaches to systems design find favor in different application areas, and each has strong advocates; for integrated, safety-critical systems, however, the time-triggered approach is generally preferred. "Time triggered" means that all activities involving the bus, and often those involving components attached to the bus, are driven by the passage of time ("if it is 20 ms since the start of the frame, then read the sensor and broadcast its value"); this is distinguished from "event triggered," which means that activities are driven by the occurrence of events ("if the sensor reading changes, then broadcast its new value"). A prime contrast between these two approaches is their locus of control: a time-triggered system controls its own activity and interacts with the environment according to an internal schedule, whereas an event-triggered system is under the control of its environment and must respond to stimuli as they occur.

Event-triggered systems allow flexible allocation of resources and this is attractive when demands are highly variable. However, in safety-critical applications it is necessary to guarantee some basic quality of service to all participants, even (or especially) in the presence of faults. Because the clients of the bus architecture are real-time embedded control systems, the required guarantees include predictable communications with low latency and low jitter (assured bandwidth is not enough). The problem with event-driven buses is that events arriving at different nodes may cause them to contend for access to the bus, so some form of media access control (i.e., a distributed mutual exclusion algorithm) is needed to ensure that each node eventually is able to transmit without interruption. The important issue is how predictable is the access achieved by each node, and how strong is the assurance that the predictions remain true in the presence of faults.

Buses such as Ethernet resolve contention probabilistically and therefore can provide only probabilistic guarantees of timely access, and no assurance at all in the presence of faults. Buses for non-safety-critical embedded systems such as CAN, LonWorks, or Profibus use various priority, preassigned slot, or token schemes to resolve contention deterministically. In CAN, for example, the message with the lowest number always wins the arbitration and therefore has to wait only for the current message to finish, while other messages must also wait for any lower-numbered messages. Thus, although contention is resolved deterministically, latency increases with load and can be bounded with only probabilistic guarantees—and these can be quite weak in the presence of faults (e.g., the current message may be retransmitted in the case of transmission failure, thereby delaying the next message, even if this has higher priority). Furthermore, faulty nodes may not adhere to expected patterns of use and may make excessive demands for service, thereby reducing that available to others. Event-triggered buses for safety-critical applications add various mechanisms to limit such demands. ARINC 629 (an avionics data bus used in the Boeing 777), for example, uses a technique sometimes referred to as "minislotting" that requires each node to wait a certain period after sending a message before it can contend to send another. Even here, however, latency is a function of load, so the Byteflight automobile protocol developed by BMW extends this mechanism with guaranteed, preallocated slots for critical messages. But even preallocated slots provide

no protection against a faulty node that fails to recognize them. The worst manifestation of this kind of fault is the so-called "babbling idiot" failure where a faulty node transmits constantly, thereby compromising the operation of the entire bus.

In a time-triggered bus, there is a static preallocation of communication bandwidth in the form of a global schedule: each node knows the schedule and knows the time, and therefore knows when it is allowed to send messages, and when it should expect to receive them. Thus, contention is resolved at design time (as the schedule is constructed), when all its consequences can be examined, rather than at run time. A static schedule makes possible the control of the babbling idiot failure mode. This is achieved by interposing an independent component, called a bus *guardian*, that allows each node to transmit on the bus only when it is allowed to do so. The guardian must know when its node is allowed to access the bus, which is difficult to achieve in an event-triggered system but is conceptually simple in a time triggered system: the guardian has an independent clock and independent knowledge of the schedule and allows its node to broadcast only when indicated by the schedule.

Because all communication is triggered by the global schedule, there is no need to attach source or destination addresses to messages sent over a time-triggered bus: each node knows the sender and intended recipients of each message by virtue of the time at which it is sent. Elimination of the address fields not only reduces the size of each message, thereby greatly increasing the message bandwidth of the bus (messages are typically short in embedded control applications), but it also eliminates a potential source of serious faults: namely, the possibility that a faulty node may send messages to the wrong recipients or, worse, may masquerade as a sender other than itself.

Fault-tolerant clock synchronization is a fundamental requirement for a time-triggered bus architecture: the abstraction of a global clock is realized by each node having a local clock that is closely synchronized with the clocks of all other nodes. Tightness of the bus schedule, and hence the throughput of the bus, is strongly related to the quality of global clock synchronization that can be achieved—and this is related to the quality of the clock oscillators local to each node, and to the algorithm used to synchronize them. There are two basic classes of algorithm for clock synchronization: those based on averaging and those based on events. Averaging works by each node measuring the skew between its clock and that of each other node (e.g., by comparing the arrival time of each message with its expected value) then setting its clock to some "average" value. A simple average (e.g., the mean or median) over all clocks may be affected by wild readings from faulty clocks (which might provide different, or missing, readings to different observers), so we need a "fault-tolerant average" that is largely insensitive to a certain number of readings from faulty clocks. Schneider [18] gives a general description that applies to all averaging clock synchronization algorithms; these algorithms differ only in their choice of fault-tolerant average. The Welch-Lynch algorithm [25] is a popular choice that is characterized by use of the "fault-tolerant midpoint" as its averaging function. Event-based algorithms rely on nodes being able to sense events directly on the interconnect: each node broadcasts a "ready" event when it is time to synchronize and sets its clock when it has seen a certain number of events from other nodes. Depending on the fault model, additional waves of "echo" or "accept" events may be needed to make this fault tolerant. The number of faulty nodes that can be tolerated, and the quality of

synchronization that can be achieved, depend on the details of the algorithm, and on the fault hypothesis under which it operates. The event-based algorithm of Srikanth and Toueg [21] is particularly attractive because it achieves optimal accuracy.

3 Fault Hypotheses and Fault Containment Units

Safety-critical aerospace functions are generally required to have failure rates less than 10^{-9} per hour [5], and an architecture that is intended to support several such functions should provide assurance of failure rates better than 10^{-10} per hour. Similar requirements apply to cars (although higher rates of loss are accepted for individual cars than aircraft, there are vastly more of them, so the required failure rates are similar). Consumer-grade electronics devices have failure rates many orders of magnitude worse than this, so redundancy and fault tolerance are essential elements of a bus architecture. Redundancy may include replication of the entire bus, of the interconnect and/or the interfaces, or decomposition of those elements into smaller subcomponents that are then replicated.

Fault tolerance takes two forms in these architectures: first is that which ensures that the bus itself does not fail, second is that which eases the construction of fault-tolerant applications. Each of these mechanisms must be constructed and validated against an explicit *fault hypothesis*, and must deliver specified *services* (that may be specified to degrade in acceptable ways in the presence of faults). The fault hypothesis must describe the *modes* (i.e., kinds) of faults that are to be tolerated, and their maximum *number* and *arrival rate*. The fault hypothesis must also identify the different *fault containment units* (FCUs) in the design: these are the components that can *independently* be afflicted by faults. The division of an architecture into separate FCUs needs careful justification: there must be no propagation of faults from one FCU to another, and no "common mode failures" where a single physical event produces faults in multiple FCUs. Only physical faults (those caused by damage to, defects in, or aging of the devices employed, or by external disturbances such as cosmic rays, and electromagnetic interference) are consider in this analysis: design faults must be excluded, and must be shown to be so by stringent assurance and certification processes.

The assumption that failures of separate FCUs are independent must be ensured by careful design and assured by stringent analysis. True independence generally requires that different FCUs are served by different power supplies, and are physically and electrically isolated from each other. Providing this level of independence is expensive and it is generally undertaken only in aircraft applications. In cars, it is common to make some small compromises on independence: for example, the guardians may be fabricated on the same chip as the interface (but with their own clock oscillators), or the interface may be fabricated on the same chip as the host processor. It is necessary to examine these compromises carefully to ensure that the loss in independence applies only to fault modes that are benign, extremely rare, or tolerated by other mechanisms.

A fault *mode* describes the kind of behavior that a faulty FCU may exhibit. The same fault may exhibit different modes at different levels of a protocol hierarchy: for example, at the electrical level, the fault mode of a faulty line driver may be that it sends an intermediate voltage (one that is neither a digital 0 nor a digital 1), while at the message level the mode of the same fault may be "Byzantine," meaning that different receivers

interpret the same message in different ways (because some see the intermediate voltage as a 0, and others as a 1). Some protocols can tolerate Byzantine faults, others cannot; for those that cannot, we must show that the fault mode is controlled at the underlying electrical level.

The basic dimensions that a fault can affect are value, time, and space. A *value* fault is one that causes an incorrect value to be computed, transmitted, or received (whether as a physical voltage, a logical message, or some other representation); a *timing* fault is one that causes a value to be computed, transmitted, or received at the wrong time (whether too early, too late, or not at all); a *spatial proximity* fault is one where all matter in some specified volume is destroyed (potentially afflicting multiple FCUs). Bus-based interconnects of the kind shown in Figure 1 are vulnerable to spatial proximity faults: all redundant buses necessarily come into close proximity at each node, and general destruction in that space could sever or disrupt them all. Interconnect topologies with a central hub are far more resilient in this regard: a spatial proximity fault that destroys one or more nodes does not disrupt communication among the others (the hub may need to isolate the lines to the destroyed nodes in case these are shorted), and destruction of a hub can be tolerated if there is a duplicate in another location.

There are many ways to classify the effects of faults in any of the basic dimensions. One classification that has proved particularly effective in analysis of the types of algorithms that underlie the architectures considered here is the *hybrid* fault model of Thambidurai and Park [23]. In this classification, the effect of a fault may be *manifest*, meaning that it is reliably detected (e.g., a fault that causes an FCU to cease transmitting messages), *symmetric* meaning that whatever the effect, it is the same for all observers (e.g., an off-by-1 error), or *arbitrary*, meaning that it is entirely unconstrained. In particular, an arbitrary fault may be asymmetric or *Byzantine*, meaning that its effect is perceived differently by different observers (as in the intermediate voltage example).

The great advantage to designs that can tolerate arbitrary fault modes is that we do not have to justify assumptions about more specific fault modes: a system is shown to tolerate (say) two arbitrary faults by proving that it works in the presence of two faulty FCUs with *no assumptions whatsoever* on the behavior of the faulty components. A system that can tolerate only specific fault modes may fail if confronted by a different fault mode, so it is necessary to provide assurance that such modes cannot occur. It is this *absence* of assumptions that is the attraction, in safety-critical contexts, of systems that can tolerate arbitrary faults. This point is often misunderstood and such systems are derided as being focused on asymmetric or Byzantine faults, "which never arise in practice." Byzantine faults are just one manifestation of arbitrary behavior, and cannot simply be asserted not to occur (in fact, they have been observed in several systems that have been monitored sufficiently closely). One situation that is likely to provoke asymmetric manifestations is a *slightly out of specification* (SOS) fault, such as the intermediate electrical voltage mentioned earlier. SOS faults in the timing dimension include those that put a signal edge very close to a clock edge, or that have signals with very slow rise and fall times (i.e., weak edges). Depending on the timing of their own clock edges, some receivers may recognize and latch such a signal, others may not, resulting in asymmetric or Byzantine behavior.

FCUs may be active (e.g., a processor) or passive (e.g., a bus); while an arbitrary-faulty active component can do anything, a passive component may change, lose, or delay data, but it cannot spontaneously create a new datum. Keyed checksums or digital signatures can sometimes be used to reduce the fault modes of an active FCU to those of a passive one. (An arbitrary-faulty active FCU can always create its own messages, but it cannot create messages purporting to come from another FCU if it does not know the key of that FCU; signatures need to be managed carefully for this reduction in fault mode to be credible.)

Any fault-tolerant architecture will fail if subjected to too many faults; generally speaking, it requires more redundancy to tolerate an arbitrary fault than a symmetric one, which in turn requires more redundancy than a manifest fault. The most effective fault-tolerant algorithms make this tradeoff automatically between number and difficulty of faults tolerated. For example, the clock synchronization algorithm of [16] can tolerate a arbitrary faults, s symmetric, and m manifest ones simultaneously provided n, the number of FCUs, satisfies $n > 3a+2s+m$. It is provably impossible (i.e., it can be proven that no algorithm can exist) to tolerate a arbitrary faults in clock synchronization with fewer than $3a + 1$ FCUs (unless digital signatures are employed—which is equivalent to reducing the severity of the arbitrary fault mode).

Because it is algorithmically much easier to tolerate simple failure modes, some architectures (e.g., SAFEbus) arrange FCUs (the "Bus Interface Units" in the case of SAFEbus) in self-checking pairs: if the members of a pair disagree, they go offline, ensuring that the effect of their failure is seen as a manifest fault (i.e., one that is easily tolerated). Most architectures also employ substantial self-checking in each FCU; any FCU that detects a fault will shut down, thereby ensuring that its failure will be manifest. (This kind of operation is often called *fail silence*). Even with extensive self-checking and pairwise-checking, it may be possible for some fault modes to "escape," so it is generally necessary to show either that the mechanisms used have complete coverage (i.e., there will be no violation of fail silence), or to design the architecture so that it can tolerate the "escape" of at least one arbitrary fault.

Some architectures can tolerate only a single fault at a time, but can reconfigure to exclude faulty FCUs and are then able to tolerate additional faults. In such cases, the *fault arrival rate* is important: faults must not arrive faster than the architecture can reconfigure. The architectures considered here operate according to static schedules, which consist of "rounds" or "frames" that are executed repeatedly in a cyclic fashion. The acceptable fault arrival rate is often then expressed in terms of faults per round (or the inverse). It is usually important that every node is scheduled to make at least one broadcast in every round, since this is how fault status is indicated (and hence how reconfiguration is triggered).

Historical experience and analysis must be used to show that the hypothesized modes, numbers, and arrival rate are realistic, and that the architecture can indeed operate correctly under those hypotheses for its intended mission time. But sometimes things go wrong: the system may experience many simultaneous faults (e.g., from unanticipated high-intensity radiated fields (HIRF)), or other violations of its fault hypothesis. We cannot guarantee correct operation in such cases (otherwise our fault hypothesis was too conservative), but safety-critical systems generally are constructed to a "never give up"

philosophy and will attempt to continue operation in a degraded mode. The usual method of operation in "never give up" mode is that each node reverts to local control of its own actuators using the best information available (e.g., each brake node applies braking force proportional to pedal pressure if it is still receiving that input, and removes all braking force if not), while at the same time attempting to regain coordination with its peers. Although it is difficult to provide assurance of correct operation during these upsets, it may be possible to provide assurance that the system returns to normal operation once the faults cease (assuming they were transients) using the ideas of self-stabilization [20].

Restart during operation may be necessary if HIRF or other environmental influences lead to violation of the fault hypothesis and cause a complete failure of the bus. Notice that this failure must be detected by the bus, and the restart must be automatic and very fast: most control systems can tolerate loss of control inputs for only a few cycles—longer outages will lead to loss of control. For example, Heiner and Thurner estimate that the maximum transient outage time for a steer-by-wire automobile application is 50ms [6].

Restart is usually initiated when an interface detects no activity on any bus line for some interval; that interface will then transmit some "wake up" message on all lines. Of course, it is possible that the interface in question is faulty (and there was bus activity all along but that interface did not detect it), or that two interfaces decide simultaneously to send the "wake up" call. The first possibility must be avoided by careful checking, preferably by independent units (e.g., both interfaces of a pair, or an interface and its guardian); the second requires some form of collision detection and resolution: this should be deterministic to guarantee an upper bound on the time to reach resolution (that will allow a single interface can send an uninterrupted "wake up" message) and, ideally, should not depend on collision detection (because this cannot be done reliably). Notice that it must be possible to perform startup and restart reliably even in the presence of faulty components.

4 Services

The essential basic purpose of these bus architectures is to make it *possible* to build reliable distributed applications; a desirable purpose is to make it *straightforward* to build such applications. The basic services provided by the bus architectures considered here comprise clock synchronization, time-triggered activation, and reliable message delivery. Some of the architectures provide additional services; their purpose is to assist straightforward construction of reliable distributed applications by providing these services in an application-independent manner, thereby relieving the applications of the need to implement these capabilities themselves. Not only does this simplify the construction of application software, it is sometimes possible to provide *better* services when these are implemented at the architecture level, and it is also possible to provide strong assurance that they are implemented correctly.

Applications that perform safety-critical functions must generally be replicated for fault tolerance. There are many ways to organize fault-tolerant replicated computations, but a basic distinction is between those that use *exact* agreement, and those that use *approximate* agreement. Systems that use approximate agreement generally run several

copies of the application in different nodes, each using its own sensors, with little coordination across the different nodes. The motivation for this is a "folk belief" that it promotes fault tolerance: coordination is believed to introduce the potential for common mode failures. Because different sensors cannot be expected to deliver exactly the same readings, the outputs (i.e., actuator commands) computed in the different nodes will also differ. Thus, the only way to detect faulty outputs is by looking for values that differ by "a lot" from the others. Hence, these systems use some form of selection or threshold voting to select a good value to send to the actuators, and similar techniques to identify faulty nodes that should be excluded. A difficulty for applications of the kind considered here is that hosts accumulate state that diverges from that of others over time (e.g., velocity and position as a result of integrating acceleration), and they execute mode switches that are discrete decisions based on local sensor values (e.g., change the gain schedule in the control laws if the altitude, or temperature, is above a specific value). Thus small differences in sensor readings can lead to major differences in outputs and this can mislead the approximate selection or voting mechanisms into choosing a faulty value, or excluding a nonfaulty node. The fix to these problems is to attempt to coordinate discrete mode switches and periodically to bring state data into convergence. But these fixes are highly application specific, and they are contrary to the original philosophy that motivated the choice of approximate agreement—hence, there is a good chance of doing them wrong. There are numerous examples that justify this concern; several that were discovered in flight tests are documented by Mackall and colleagues [10]. The essential points of Mackall's data is that all the failures observed in flight test were due to bugs in the design of the fault tolerance mechanisms themselves, and all these bugs could be traced to difficulties in organizing and coordinating systems based on approximate agreement.

Systems based on exact agreement face up to the fact that coordination among replicated computations is necessary, and they take the necessary steps to do it right. If we are to use exact agreement, then every replica must perform the same computation on the same data: any disagreement on the outputs then indicates a fault; comparison can be used to detect those faults, and majority voting to mask them. A vital element in this approach to fault tolerance is that replicated components must work on the same data: thus, if one node reads a sensor, it must distribute that reading to all the redundant copies of the application running in other nodes. Now a fault in that distribution mechanism could result in one node getting one value and another a different one (or no value at all). This would abrogate the requirement that all replicas obtain identical inputs, so we need to employ mechanisms to overcome this behavior.

The problem of distributing data consistently in the presence of faults is variously called *interactive consistency*, *consensus*, *atomic broadcast*, or *Byzantine agreement* [12, 9]. When a node transmits a message to several receivers, interactive consistency requires the following two properties to hold.

Agreement: All nonfaulty receivers obtain the same message (even if the transmitting node is faulty).

Validity: If the transmitter is nonfaulty, then nonfaulty receivers obtain the message actually sent.

Algorithms for achieving these requirements in the presence of arbitrary faults necessarily involve more than a single data exchange (basically, each receiver must compare the value it received against those received by others). It is provably impossible to achieve interactive consistency in the presence of a arbitrary faults unless there are at least $3a+1$ FCUs, $2a+1$ disjoint communication paths between them, and $a+1$ levels (or "rounds") of communication. The number of FCUs and the number of disjoint paths required, but not the number of rounds, can be reduced by using digital signatures.

The problem might seem moot in architectures that employ a physical bus, since a bus surely cannot deliver values inconsistently (so the agreement property is achieved trivially). Unfortunately, it can—though it is likely to be a very rare event. The scenarios involving SOS faults presented earlier exemplify some possibilities.

Dealing properly with very rare events is one of the attributes that distinguishes a design that is fit for safety-critical systems from one that is not. It follows that either the application software must perform interactive consistency for itself (incurring the cost of n^2 messages to establish consistency across n nodes in the presence of a single arbitrary fault), or the bus architecture must do it.

The first choice is so unattractive that it vitiates the whole purpose of a fault-tolerant bus architecture. Most bus architectures therefore provide some type of interactively consistent message broadcast as a basic service. In addition, most architectures take steps to reduce the incidence of asymmetric transmissions (i.e., those that appear as one value to some receivers, and as different values, or the absence of values, to others). As noted, SOS faults are among the most plausible sources of asymmetric transmissions. SOS faults that cause asymmetric transmissions can arise in either the value or time domains (e.g., intermediate voltages, or weak edges, respectively). In those architectures that employ a bus guardian in a central hub or "in series" with each interface, the bus guardians are a possible point of intervention for the control of SOS faults: a suitable guardian can reshape, in both value and time domains, the signal sent to it by the controller. Of course, the guardian could be faulty and may make matters worse—so this approach makes sense only when there are independent guardians on each of two (or more) replicated interconnects. Observe that for credible signal reshaping, the guardian must have a power supply that is independent of that of the controller (faults in power supply are the most likely cause of intermediate voltages and weak edges).

Interactively consistent message broadcast provides the foundation for fault tolerance based on exact agreement. There are several ways to use this foundation. One arrangement, confusingly called the *state machine* approach [19], is based on majority voting: application replicas run on a number of different nodes, exchange their output values, and deliver a majority vote to the actuators.

Another arrangement is based on self-checking (either by individuals or pairs) so that faults result in fail-silence. This will be detected by other nodes, and some backup application running in those other nodes can take over. The architecture can assist this master/shadow arrangement by providing services that support the rollover from one node to another. One such service automatically substitutes a backup node for a failed master (both the master and the backup occupy the same slot in the schedule, but the backup is inhibited from transmitting unless the master is failed). A variant has both master and backup operating in different slots, but the backup inhibits itself unless it is

informed that the master has failed. A further variation, called *compensation*, applies when different nodes have access to different actuators: none is a direct backup to any other, but each changes its operation when informed that others have failed (an example is car braking: separate nodes controlling the braking force at each wheel will redistribute the force when informed that one of their number has failed).

The variations on master/shadow described above all depend on a "failure notification," or equivalently a "membership" service. The crucial requirement on such a service is that it must produce *consistent* knowledge: that is, if one nonfaulty node thinks that a particular node has failed, then all other nonfaulty nodes must hold the same opinion—otherwise, the system will lose coordination, with potentially catastrophic results (e.g., if the nodes controlling braking at different wheels make different adjustments to their braking force based on different assessments of which others have failed). Notice that this must also apply to a node's knowledge of its *own* status: a naïve view might assume that a node that is receiving messages and seeing no problems in its own operation should assume it is in the membership. But if this node is unable to transmit, all other nodes will have removed it from their memberships and will be making suitable compensation on the assumption that this node has entered its "blackout" mode (and is, for example, applying no force to its brake). It could be catastrophic if this node does not adopt the consensus view and continues operation (e.g., applying force to its brake) based on its local assessment of its own health.

A membership service operates as follows. Each node maintains a private *membership* list, which is intended to comprise all and only the nonfaulty nodes. Since it can take a while to diagnose a faulty node, we have to allow the common membership to contain at most one faulty node. Thus, a membership service must satisfy the following two requirements.

Agreement: The membership lists of all nonfaulty nodes are the same.
Validity: The membership lists of all nonfaulty nodes contain all nonfaulty nodes and at most one faulty node.

These requirements can be achieved only under benign fault hypotheses (it is provably impossible to diagnose an arbitrary-faulty node with certainty). When unable to maintain accurate membership, the best recourse is to maintain agreement, but sacrifice validity (nonfaulty nodes that are not in the membership can then attempt to rejoin). This weakened requirement is called "clique avoidance" [2].

Note that it is quite simple to achieve consistent membership on top of an interactively consistent message service: each node broadcasts its own membership list to every other node, and each node runs a deterministic resolution algorithm on the (identical, by interactive consistency) lists received. Conversely, a membership and clique-avoidance service can assist the construction of an interactively consistent message service: simply exclude from the membership any node that receives a message different than the majority (TTA does this).

5 Practical Implementations

Here, we provide sketches of four bus architectures that provide concrete solutions to the requirements and design challenges outlined in the previous sections. More details

are available in a companion report to this paper [17]. All four buses support the time-triggered model of computation, employ fault-tolerant distributed clock synchronization, and use bus guardians or some equivalent mechanism to protect against babbling idiot failure modes. They differ in their fault hypotheses, mechanisms employed, services provided, and in their assurance, performance, and cost.

SAFEbus. Honeywell developed SAFEbus™ (the principal designers are Kevin Driscoll and Ken Hoyme [7]) to serve as the core of the Boeing 777 Airplane Information Management System (AIMS) [22], which supports several critical functions, such as flight management and cockpit displays. The bus has been standardized as ARINC 659 [1] and variations on Honeywell's implementation are being used or considered for other avionics and space applications. It uses a bus interconnect similar to that shown in Figure 1; the interfaces (they are called Bus Interface Units, or BIUs) are duplicated, and the interconnect bus is quad-redundant. Most of the functionality of SAFEbus is implemented in the BIUs, which perform clock synchronization and message scheduling and transmission functions. Each BIU of a pair is a separate FCU and acts as its partner's bus guardian by controlling its access to the interconnect.

Each BIU of a pair drives a different pair of interconnect buses but is able to read all four; the interconnect buses themselves each comprise two data lines and one clock line and operate at 30MHz. The bus lines and their drivers have the electrical characteristics of OR-gates (i.e., if several different BIUs drive the same line at the same time, the resulting signal is the OR of the separate inputs). Some of the protocols exploit this property; in particular, clock synchronization is achieved using an event-based algorithm.

The paired BIUs at sender and receiver, and the quad-redundant buses, provide sufficient redundancy for SAFEbus to provide interactively consistent message broadcasts (in the Honeywell implementation) using an approach similar to that described by Davies and Wakerly [4] (this remarkably prescient paper anticipated many of the issues and solutions in Byzantine fault tolerance by several years). It also supports application-level fault tolerance (based on self-checking pairs) by providing automatic rapid rollover from masters to shadows.

Its fault hypothesis includes arbitrary faults, faults in several nodes (but only one per node), and a high rate of fault arrivals. It never gives up and has a well-defined restart and recovery strategy from fault arrivals that exceed its fault hypothesis. It tolerates spatial proximity faults in the AIMS application by duplicating the entire system SAFEbus is certified for use in passenger aircraft and has extensive field experience in the Boeing 777. The Honeywell implementation is supported by an in-house tool chain.

SAFEbus is the most mature of the four buses considered, and makes the fewest compromises. But because each of its major components is paired (and its bus requires separate lines for clock and data), it is the most expensive of those available for commercial use (typically, a few hundred dollars per node).

TTA. The Time Triggered Architecture (TTA) was developed by Hermann Kopetz and colleagues at the Technical University of Vienna [8]. Commercial development of the architecture is undertaken by TTTech and it is being deployed for safety-critical applications in cars by Audi and Volkswagen, and for flight-critical functions in aircraft and aircraft engines by Honeywell.

Current implementations of TTA use a bus interconnect similar to that shown in Figure 1. The interfaces (they are called *controllers*) implement the TTP/C protocol [24] that is at the heart of TTA, providing clock synchronization, and message sequencing and transmission functions. The interconnect bus is duplicated and each controller drives both of them through partially independent bus guardians. TTA uses an averaging clock synchronization algorithm based on that of Lundelius and Lynch [25]. This algorithm is implemented in the controllers, but requires too many resources to be replicated in the bus guardians. The guardians, which have independent clocks, therefore rely on their controllers for a "start of frame" signal. This compromises their independence somewhat (they also share the power supply and some other resources with their controllers), so forthcoming implementations of TTA use a star interconnect similar to that shown in Figure 2. Here, the guardian functionality is implemented in the central hub which is fully independent of the controllers: the hubs and controllers comprise separate FCUs in this implementation. Hubs are duplicated for fault tolerance and located apart to withstand spatial proximity faults. They also perform signal reshaping to reduce the incidence of SOS faults.

TTA employs algorithms for group membership and clique avoidance [2]; these enable its clock synchronization algorithm to tolerate multiple faults (by reconfiguring to exclude faulty members) and combine with its use of checksums (which can be considered as digital signatures) to provide a form of interactively consistent message broadcasts. The membership service supports application-level fault tolerance based on master-backup or compensation. Proposed extensions provide state machine replication in a manner that is transparent to applications.

The fault hypothesis of TTA includes arbitrary faults, and faults in several nodes (but only one per node), provided these arrive at least two rounds apart (this allows the membership algorithm to exclude the faulty node). It never gives up and has a well-defined restart and recovery strategy from fault arrivals that exceed this hypothesis.

The prototype implementations of TTA have been subjected to extensive testing and fault injections, and deployed in experimental vehicles. Several of its algorithms have been formally verified [14, 13], and aircraft applications under development are planned to lead to FAA certification. It is supported by an extensive tool suite that interfaces to standard CAD environments (e.g., Matlab/Simulink and Beacon). Current implementations provide 25 Mbit/s data rates; research projects are designing implementations for gigabit rates. TTA controllers and the star hub (which is basically a modified controller) are quite simple and cheap to produce in volume.

Of the architectures considered here, TTA is unique in being used for both automobile applications, where volume manufacture leads to very low prices, and aircraft, where a mature tradition of design and certification for flight-critical electronics provides strong scrutiny of arguments for safety.

SPIDER. A Scalable Processor-Independent Design for Electromagnetic Resilience (SPIDER) is being developed by Paul Miner and colleagues at the NASA Langley Research Center as a research platform to explore recovery strategies for radiation-induced (HIRF/EMI) faults, and to serve as a case study to exercise the recent design assurance guidelines for airborne electronic hardware (DO-254) [15].

SPIDER uses a star configuration similar to that shown in Figure 2, in which the interfaces (called BIUs) may be located either with their hosts or in the centralized hub, which also contains active elements called Redundancy Management Units, or RMUs.

Clock synchronization and other services of SPIDER are achieved by novel distributed algorithms executed among the BIUs and RMUs [11]. The services provided include interactively consistent message broadcasts, and identification of failed nodes (from which a membership service can easily be synthesized). SPIDER's fault hypothesis uses a hybrid fault model, which includes arbitrary faults, and allows some combinations of multiple faults. Its algorithms are novel and highly efficient and are being formally verified.

SPIDER is an interesting design that uses a different topology and a different class of algorithms from the other buses considered here. However, it is a research project whose design and implementation are still in progress and so it cannot be compared directly with the commercial products.

FlexRay. A consortium including BMW, DaimlerChrysler, Motorola, and Philips, is developing FlexRay for powertrain and chassis control in cars. It differs from the other buses considered here in that its operation is divided between time-triggered and event-triggered activities. Published descriptions of the FlexRay protocols and implementation are sketchy at present [3] (see also the Web site `www.flexray-group.com`).

FlexRay can use either an "active" star interconnect similar to that shown in Figure 2, or a "passive" bus similar to that shown in Figure 1. In both cases, duplication of the interconnect is optional. The star configuration of FlexRay (and also that of TTA) can also be deployed in distributed configurations where subsystems are connected by hub-to-hub links. Each FlexRay interface (it is called a communication controller) drives the lines to its interconnects through separate bus guardians located with the interface. (This means that with two buses, each node has three clocks: one for the controller and one for each of the two guardians; this differs from the bus configuration of TTA where there is one clock for the controller and both guardians share a second clock.) Like the bus configuration of TTA, the guardians of FlexRay are not fully independent of their controllers.

FlexRay aims to be more flexible than the other buses considered here, and this seems to be reflected in the choice of its name. As noted, one manifestation of this flexibility is its combination of time- and event-triggered operation. FlexRay partitions each time cycle into a "static" time-triggered portion, and a "dynamic" event-triggered portion. The division between the two portions is set at design time and loaded into the controllers and bus guardians. Communication during the event-driven portion of the cycle uses the Byteflight protocol. Unlike SAFEbus and TTA, FlexRay does not install the full schedule for the time-triggered portion in each controller. Instead, this portion of the cycle is divided into a number of slots of fixed size and each controller and its bus guardians are informed only of those slots allocated to their transmissions (nodes requiring greater bandwidth are assigned more slots than those that require less). Controllers learn the full schedule only when the bus starts up. Each node includes its identity in the messages that it sends; during startup, nodes use these identifiers to label their input buffers as the schedule reveals itself (e.g., if the messages that arrive in slots 1 and 7 carry identifier 3, then all nodes will thereafter deliver the contents of buffers 1 and

7 to the task that deals with input from node 3). There appears to be a vulnerability here: a faulty node could masquerade as another (i.e., send a message with the wrong identifier) during startup and thereby violate partitioning for the remainder of the mission. It is not clear how this fault mode is countered.

Like TTA, FlexRay uses the Lundelius-Welch clock synchronization algorithm but, unlike TTA, it does not use a membership algorithm to exclude faulty nodes. FlexRay provides no services to its applications beyond best-efforts message delivery; in particular, it does not provide interactively consistent message broadcasts. This means that all mechanisms for fault-tolerant applications must be provided by the applications programs themselves. Published descriptions of FlexRay do not specify its fault hypothesis, and it appears to have no mechanisms to counter certain fault modes (e.g., SOS faults or other sources of asymmetric broadcasts, and masquerading on startup). A never-give-up strategy has not been described, nor have systematic or formal approaches to assurance and certification.

FlexRay is interesting because of its mixture of time- and event-triggered operation, and potentially important because of the industrial clout of its developers. Currently, it is the slowest of the commercial buses, with a claimed data rate of no more than 10 Mbit/s.

6 Summary and Conclusion

A safety-critical bus architecture provides certain properties and services that assist in construction of safety-critical systems. As with any system framework or middleware package, these buses offer a tradeoff to system developers: they provide a coherent collection of services, with strong properties and highly assured implementations, but developers must sacrifice some design freedom to gain the full benefit of these services. For example, all these buses use a time-triggered model of computation, and system developers must build their applications within that framework. In return, the buses are able to guarantee strong partitioning: faults in individual components or applications ("functions" in avionics terms) cannot propagate to others, nor can they bring down the entire bus (within the constraints of its fault hypothesis).

Partitioning is the minimum requirement, however. It ensures that one failed function will not drag down others, but in many safety-critical systems the failure of even a single function can be catastrophic, so the individual functions must themselves be made fault tolerant. Accordingly, most of the buses provide mechanisms to assist the development of fault-tolerant applications. The key requirement here is interactively consistent message transfer: this ensures that all masters and shadows (or masters and monitors), or all members of a voting pool, maintain consistent state. Three of the buses considered here provide this basic service; some of them do so in association with other services, such master/shadow rollover or group membership, that can be provided with much increased efficiency and much reduced latency when implemented at a low level. FlexRay, alone, provides none of these services. In their absence, all mechanisms for fault-tolerance must be implemented in applications programs. Thus, application programmers, who may have little experience in the subtleties of fault-tolerant systems, become responsible for the design, implementation, and assurance of very delicate mechanisms with no support from the underlying bus architecture. Not only does this increase the cost and difficulty

of making sure that things are done right, it also increases their computational cost and latency. For example, in the absence of an interactively consistent message service provided by the architecture, applications programs must explicitly transmit the multiple rounds of cross-comparisons that are needed to implement this service at a higher level, thereby substantially increasing the message load. Such a cost will invite inexperienced developers to seek less expensive ways to achieve fault tolerance—in probable ignorance of the impossibility results in the theoretical literature, and the history of intractable "Heisenbugs" (rare, unrepeatable, failures) encountered by practitioners who pushed for 10^{-9} with inadequate foundations.

It is unlikely that any single bus architecture will satisfy all needs and markets, so it is possible that FlexRay's lack of application-level fault-tolerant services will find favor in some areas. It is also to be expected that new or modified architectures will emerge to satisfy new markets and requirements. (For example, it is proposed that TTA could match FlexRay's ability to support event-triggered as well as time-triggered communications by allocating certain time slots to a simulation of CAN; the simulation is actually faster than a real CAN bus, while retaining all the safety attributes of TTA.) I hope that the description provided here will help potential users to evaluate existing architectures against their own needs, and that it will help designers of new architectures to learn from and build on the design choices made by their predecessors.

References

1. *ARINC Specification 659: Backplane Data Bus*. Aeronautical Radio, Inc, Annapolis, MD, December 1993. Prepared by the Airlines Electronic Engineering Committee.
2. Günther Bauer and Michael Paulitsch. An investigation of membership and clique avoidance in TTP/C. In *19th Symposium on Reliable Distributed Systems*, Nuremberg, Germany, October 2000.
3. Joef Berwanger et al. FlexRay–the communication system for advanced automotive control systems. In *SAE 2001 World Congress*, Society of Automotive Engineers, Detroit, MI, April 2001. Paper number 2001-01-0676.
4. Daniel Davies and John F. Wakerly. Synchronization and matching in redundant systems. *IEEE Transactions on Computers*, C-27(6):531–539, June 1978.
5. *System Design and Analysis*. Federal Aviation Administration, June 21, 1988. Advisory Circular 25.1309-1A.
6. Günnter Heiner and Thomas Thurner. Time-triggered architecture for safety-related distributed real-time systems in transportation systems. In *Fault Tolerant Computing Symposium 28*, pages 402–407, IEEE Computer Society, Munich, Germany, June 1998.
7. Kenneth Hoyme and Kevin Driscoll. SAFEbus™. In *11th AIAA/IEEE Digital Avionics Systems Conference*, pages 68–73, Seattle, WA, October 1992.
8. Hermann Kopetz and Günter Grünsteidl. TTP—a protocol for fault-tolerant real-time systems. *IEEE Computer*, 27(1):14–23, January 1994.
9. Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.
10. Dale A. Mackall. Development and flight test experiences with a flight-crucial digital control system. NASA Technical Paper 2857, NASA Ames Research Center, Dryden Flight Research Facility, Edwards, CA, 1988.

11. Paul S. Miner. Analysis of the SPIDER fault-tolerance protocols. In C. Michael Holloway, editor, *LFM 2000: Fifth NASA Langley Formal Methods Workshop*, NASA Langley Research Center, Hampton, VA, June 2000. Slides available at `http://shemesh.larc.nasa.gov/fm/Lfm2000/Presentations/lfm2000-spider/`.
12. M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, April 1980.
13. Holger Pfeifer. Formal verification of the TTA group membership algorithm. In Tommaso Bolognesi and Diego Latella, eds., *Formal Description Techniques and Protocol Specification, Testing and Verification FORTE XIII/PSTV XX 2000*, pages 3–18, Pisa, Italy, Oct. 2000.
14. Holger Pfeifer, Detlef Schwier, and Friedrich W. von Henke. Formal verification for time-triggered clock synchronization. In Charles B. Weinstock and John Rushby, eds., *Dependable Computing for Critical Applications—7*, Volume 12 of IEEE Computer Society *Dependable Computing and Fault Tolerant Systems*, pages 207–226, San Jose, CA, Jan. 1999.
15. *DO254: Design Assurance Guidelines for Airborne Electronic Hardware*. Requirements and Technical Concepts for Aviation, Washington, DC, April 2000.
16. John Rushby. A formally verified algorithm for clock synchronization under a hybrid fault model. In *Thirteenth ACM Symposium on Principles of Distributed Computing*, pages 304–313, Association for Computing Machinery, Los Angeles, CA, August 1994. Also available as NASA Contractor Report 198289.
17. John Rushby. A comparison of bus architectures for safety-critical embedded systems. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, June 2001. Available at `http://www.csl.sri.com/~rushby/papers/buscompare.pdf`.
18. Fred B. Schneider. Understanding protocols for Byzantine clock synchronization. Technical Report 87-859, Department of Computer Science, Cornell University, Ithaca, NY, Aug. 1987.
19. Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.
20. Marco Schneider. Self stabilization. *ACM Computing Surveys*, 25(1):45–67, March 1993.
21. T. K. Srikanth and Sam Toueg. Optimal clock synchronization. *Journal of the ACM*, 34(3):626–645, July 1987.
22. William Sweet and Dave Dooling. Boeing's seventh wonder. *IEEE Spectrum*, 32(10):20–23, October 1995.
23. Philip Thambidurai and You-Keun Park. Interactive consistency with multiple failure modes. In *7th Symposium on Reliable Distributed Systems*, pages 93–100, IEEE Computer Society, Columbus, OH, October 1988.
24. *Specification of the TTP/C Protocol*. Time-Triggered Technology TTTech Computertechnik AG, Vienna, Austria, July 1999.
25. J. Lundelius Welch and N. Lynch. A new fault-tolerant algorithm for clock synchronization. *Information and Computation*, 77(1):1–36, April 1988.

Using Multiple Levels of Abstractions in Embedded Software Design

Jerry R. Burch[1], Roberto Passerone[1], and Alberto L. Sangiovanni-Vincentelli[2]

[1] Cadence Berkeley Laboratories, Berkeley CA 94704, USA
[2] Department of EECS, University of California at Berkeley, Berkeley CA 94720, USA

Abstract. The methodologies that are in use today for software development rely on representations and techniques appropriate for the applications (compilers, business applications, CAD, etc.) that have been traditionally implemented on programmable processors. Embedded software is different: by virtue of being embedded in a surrounding system, the software must be able to continuously react to stimula in the desired way. Verifying the correctness of the system requires that the model of the software be transformed to include (refine) or exclude (abstract) information to retain only what is relevant to the task at hand. In this paper, we outline a framework that we inted to use for studying the problems of abstraction and refinement in the context of embedded software for hybrid systems.

1 Introduction

Embedded Software (ESW) design is one, albeit critical, aspect of the more general problem of Embedded System Design (ESD or just ES). ESD is about the implementation of a set of functionalities satisfying a number of constraints ranging from performance to cost, emissions, power consumption and weight. The choice of implementation architecture implies which functionality will be implemented as a hardware component or as software running on a programmable component. In recent years, the functionalities to be implemented in ES have grown in number and complexity so much so that the development time is increasingly difficult to predict and control. The complexity increase coupled with the constantly evolving specifications has forced designers to look at implementations that are intrinsically flexible, i.e., that can be changed rapidly. Since hardware-manufacturing cycles do take time and are expensive, the interest in software-based implementation has risen to previously unseen levels. The increase in computational power of processors and the corresponding decrease in size and cost have allowed moving more and more functionality to software. However, this move corresponds to increasing problems in verifying design correctness, a critical aspect of ESD since several application domains, such as transportation and environment monitoring, are characterized by safety considerations that are certainly not necessary for the traditional PC-like software applications. In addition to this aspect, little attention has been traditionally

T.A. Henzinger and C.M. Kirsch (Eds.): EMSOFT 2001, LNCS 2211, pp. 324–343, 2001.
© Springer-Verlag Berlin Heidelberg 2001

paid to hard constraints on reaction speed, memory footprint and power consumption of software. This is of course crucial for ES. These considerations point to the fact that ESW is really an implementation choice for a functionality that can be indifferently implemented as a hardware component, and that we cannot abstract away hard characteristics of software as we have done in the traditional software domain. No wonder then that we are witnessing a crisis in the ES domain for ESW design. This crisis is not likely to be resolved going about business as usual but we need to focus at the root of the problems.

Our vision for ESW is to change radically the way in which ESW is developed today by: 1) linking ESW upwards in the abstraction layers to system functionality; 2) linking ESW to the programmable platforms that support it thus providing the much needed means to verify whether the constraints posed on ES are met.

ESW today is written using low level programming languages such as C or even Assembler to cope with the tight constraints on performance and cost typical of most embedded systems. The tools available for creating and debugging software are no different than the ones used for standard software: compilers, assemblers and cross-compilers. If any difference can be found, it is that most tools for ESW are rather primitive when compared to analogous tools for richer platforms. On the other hand, ESW needs hardware support for debugging and performance evaluation that in general is not a big issue for traditional software. In most embedded software, operating systems were application dependent and developed in house. Once more, performance and memory requirements forced this approach.

When embedded software was simple, there was hardly need for a more sophisticated approach. However, with the increased complexity of ES application, this rather primitive approach has become the bottleneck and most system companies have decided to enhance their software design methodology to increase productivity and product quality. However, we do not believe that the real reasons for such a sorry state are well understood. We have seen a flurry of activities towards the adoption of object-oriented approaches and other syntactically driven methods that have certainly value in cleaning the structure and the documentation of embedded software but have barely scratched the surface in terms of quality assurance and time-to-market. Along this line, we also saw a growing interest towards standardization of Real-Time Operating Systems either de facto, see for example the market penetration of WindRiver in this domain, or through standard bodies such as the OSEK committee established by the German automotive industry. Quite small still is the market for development tools even though we do believe this is indeed the place where much productivity gain can be had. There is an interesting development in some application areas, where the need to capture system specifications at higher levels of abstraction is forcing designers to use tools that emphasize mathematical descriptions, making software code an output of the tool rather than an input. The leader in this market is certainly the Matlab tool set developed by MathWorks. In this case, designers develop their concept in this friendly environment where they can assemble their designs quickly and simulate their behaviour. While this approach

is definitely along the right direction, we have to raise the flag and say that the mathematical models supported by Matlab and more pertinently by Simulink, the associated simulator, are somewhat limited and do not cover the full spectrum of embedded system design. The lack of data flow support is critical. The lack of integration between the FSM capture tool (State Flow) and Simulink is also a problem. As we will detail later, this area is of key interest for our vision. It is at this level that we are going to have the best results in terms of functional correctness and error free refinement to implementation. The understanding of the mathematical properties of the embedded system functionality must be a major emphasis of our approach.

We have advocated the introduction of rigorous methodologies for system-level design for years, but we feel that there is still much to do. Recently we have directed our efforts to a new endeavor that tries to capture the requirements of present day embedded system design: the Metropolis project.

The *Metropolis* project, supported by the Gigascale Silicon Research Center, started two years ago and involves a large number of people in different research institutions. It is based on the following principles:

1. **Orthogonalization of concerns**: In Metropolis, behavior is clearly separated from implementation. Communication and computation are orthogonalized. Communication is recognized today as the main difficulty in assembling systems from basic components. Errors in software systems can often be traced to communication problems. Metropolis was created to deal with communication problems as the essence of the new design methodology. Communication-based design will allow the composition of either software or hardware blocks at any layer of abstraction in a controlled way. If the blocks are correct, the methodology ensures that they communicate correctly.

2. **Solid theoretical foundations that provide the necessary infrastructure for a new generation of tools**: The tools used in Metropolis will be interoperable and will work at different levels of abstraction, they will verify, simulate, and map designs from one level of abstraction to the next, help choose implementations that meet constraints and optimize the criteria listed above. The theoretical framework is necessary to make our claims of correctness and efficiency true. Metropolis will deal with both embedded software and hardware designs since it will intercept the design specification at a higher level of abstraction. The design specifications will have precise semantics. The semantics is essential to be able to: (i) reason about designs, (ii) identify and correct functional errors, (iv) initiate synthesis processes.

Several formal models have been proposed over the years (see e.g. [6]) to capture one or more aspects of computation as needed in embedded system design. We have been able to compare the most important models of computations using a unifying theoretical framework introduced recently by Lee and Sangiovanni-Vincentelli [9]. However, this denotational framework has only helped us to identify the sources of difficulties in combining different models of computation that are certainly needed when complex systems are being designed. In this case, the partition of the functionality of the design into different models of computation

is somewhat arbitrary as well as arbitrary are the communication mechanisms used to connect the "ports" of the different models. We believe that it is possible to optimize across model-of-computation boundaries to improve performance and reduce errors in the design at an early stage in the process.

There are many different views on how to accomplish this. There are two essential approaches: one is to develop encapsulation techniques for each pair of models that allow different models of computation to interact in a meaningful way, i.e., data produced by one object are presented to the other in a consistent way so that the object "understands" [4,5]. The other is to develop an encompassing framework where all the models of importance "reside" so that their combination, re-partition and communication happens in the same generic framework and as such may be better understood and optimized. While we realize that today heterogeneous models of computation are a necessity, we believe that the second approach is possible and will provide designers a powerful mechanism to actually select the appropriate models of computation, (e.g., FSMs, Data-flow, Discrete-Event, that are positioned in the theoretical framework in a precise order relationship so that their interconnection can be correctly interpreted and refined) for the essential parts of their design.

In this paper, we focus on this very aspect of the approach: a framework where formal models can be rigorously defined and compared, and their interconnections can be unambiguously specified. We use a kind of abstract algebra to provide the underlying mathematical machinery. We believe that this framework is essential to provide the foundations of an intermediate format that will provide the Metropolis infrastructure with a formal mechanism for interoperability among tools and specification methods.

This framework is a work in progress. Our earlier work [2,3] does not provide a sufficiently general notion of sequential composition, which is essential for modeling embedded software. The three models of computation described in this paper (which all include sequential composition) are examples of the kinds of models that we want to have fit into the framework. After the framework has been thoroughly tested on a large number of different models of computation, we plan to publish a complete description of the framework.

2 Overview

This section presents the basic framework we use to construct semantic domains, which is based on trace algebras and trace structure algebras. The concepts briefly described here will be illustrated by examples later in the paper. This overview is intended to highlight the relationships between the concepts that will be formally defined later.

More details of these algebras can be found in our earlier work [2,3]. Note, however, that our definitions of these algebras do not include sequential composition. For this reason, and other more technical reasons, the models of computation used in this paper do not fit into our earlier framework.

We maintain a clear distinction between models of processes (a.k.a. agents) and models of individual executions (a.k.a. behaviors). In different models of computation, individual executions can be modeled by very different kinds of mathematical objects. We always call these objects *traces*. A model of a process, which we call a trace structure, consists primarily of a set of traces. This is analogous to verification methods based on language containment, where individual executions are modeled by strings and processes are modeled by sets of strings. However, our notion of trace is quite general and so is not limited to strings.

Traces often refer to the externally visible features of agents: their actions, signals, state variables, etc. We do not distinguish among the different types, and we refer to them collectively as a set of *signals* W. Each trace and each trace structure is then associated with an *alphabet* $A \subseteq W$ of the signals it uses.

We make a distinction between two different kinds of behaviors: *complete* behaviors and *partial* behaviors. A complete behavior has no endpoint. A partial behavior has an endpoint; it can be a prefix of a complete behavior or of another partial behavior. Every complete behavior has partial behaviors that are prefixes of it; every partial behavior is a prefix of some complete behavior. The distinction between a complete behavior and a partial behavior has only to do with the length of the behavior (that is, whether or not it has an endpoint), not with what is happening during the behavior; whether an agent does anything, or what it does, is irrelevant.

Complete traces and *partial traces* are used to model complete and partial behaviors, respectively. A given object can be both a complete trace and a partial trace; what is being represented in a given case is determined from context. For example, a finite string can represent a complete behavior with a finite number of actions, or it can represent a partial behavior.

In our framework, the first step in defining a model of computation is to construct a trace algebra. The trace algebra contains the universe of partial traces and the universe of complete traces for the model of computation. The algebra also includes three operations on traces: *projection*, *renaming* and *concatenation*. Intuitively, these operations correspond to encapsulation, instantiation and sequential composition, respectively. Concatenation can be used to define the notion of a prefix of a trace. We say that a trace x is a prefix of a trace z if there exists a trace y such that z is equal to x concatenated with y.

The second step is to construct a trace structure algebra. Here each element of the algebra is a trace structure, which consists primarily of a set of traces from the trace algebra constructed in the first step. Given a trace algebra, and the set of trace structures to be used as the universe of agent models, a trace structure algebra is constructed in a fixed way. Thus, constructing a trace algebra is the creative part of defining a model of computation. Constructing the corresponding trace structure algebra is much easier.

A *conservative approximation* is a kind of mapping from one trace structure algebra to another. It can be used to do abstraction, and it maintains a precise relationship between verification results in the two trace structure algebras. The two trace structure algebras do not have to be based on the same trace alge-

bra. Thus, conservative approximations are a bridge between different models of computation. Conservative approximations have inverses, which can be used to embed an abstract model of computation into a more detailed one. Conservative approximations can be constructed from homomorphisms between trace algebras.

3 Trace Algebras for Embedded Software

In this section we will present the definition of three trace algebras at progressively higher levels of abstraction. The first trace algebra, called *metric time*, is intended to model exactly the evolutions (the flows and the jumps) of a hybrid system as a function of global real time. With the second trace algebra we abstract away the metric while maintaining the total order of occurence of events. This model is used to define the untimed semantics of embedded software. Finally, the third trace algebra further abstracts away the information on the event occurences by only retaining initial and final states and removing the intermediate steps. This simpler model can be used to describe the semantics of some programming language constructs. The next section will then present the definition of the homomorphisms that we use to approximate a more detailed trace algebra with the more abstract ones.

3.1 Metric Time

A typical semantics for hybrid systems includes continuous *flows* that represent the continuos dynamics of the system, and discrete *jumps* that represent instantaneous changes of the operating conditions. In our model we represent both flows and jumps with single piece-wise continuous functions over real-valued time. The flows are continuous segments, while the jumps are discontinuities between continuous segments. In this paper we assume that the variables of the system take only real or integer values and we defer the treatment of a complete type system for future work. The sets of real-valued and integer valued variables for a given trace are called $V_{\mathcal{R}}$ and $V_{\mathcal{N}}$, respectively.

Traces may also contain actions, which are discrete events that can occur at any time. Actions do not carry data values. For a given trace, the set of input actions is M_I and the set of output actions is M_O.

Each trace has a signature γ which is a 4-tuple of the above sets of signals:

$$\gamma = (V_{\mathcal{R}}, V_{\mathcal{N}}, M_I, M_O).$$

The sets of signals may be empty, but we assume they are disjoint. The *alphabet* of γ is

$$A = V_{\mathcal{R}} \cup V_{\mathcal{N}} \cup M_I \cup M_O.$$

The set of partial traces for a signature γ is $B_P(\gamma)$. Each element of $B_P(\gamma)$ is as a triple $x = (\gamma, \delta, f)$. The non-negative real number δ is the *duration* (in time) of the partial trace. The function f has domain A. For $v \in V_{\mathcal{R}}$, $f(v)$ is a

function in $[0, \delta] \to \mathcal{R}$, where $\mathcal{R}$ is the set of real numbers and the closed interval $[0, \delta]$ is the set of real numbers between 0 and δ, inclusive. This function must be piece-wise continuous and right-hand limits must exist at all points. Analogously, for $v \in V_{\mathcal{N}}$, $f(v)$ is a piece-wise constant function in $[0, \delta] \to \mathcal{N}$, where $\mathcal{N}$ is the set of integers. For $a \in M_I \cup M_O$, $f(a)$ is a function in $[0, \delta] \to \{0, 1\}$, where $f(a)(t) = 1$ iff action a occurs at time t in the trace.

The set of complete traces for a signature γ is $B_C(\gamma)$. Each element of $B_C(\gamma)$ is as a double $x = (\gamma, f)$. The function f is defined as for partial traces, except that each occurence of $[0, \delta]$ in the definition is replaced by $R^{/}_{-}$the set of non-negative real numbers.

To complete the definition of this trace algebra, we must define the operations of projection, renaming and concatenation on traces. The projection operation $proj(B)(x)$ is defined iff $M_I \subseteq B \subseteq A$. The trace that results is the same as x except that the domain of f is restricted to B. The renaming operation $x' = rename(r)(x)$ is defined iff r is a one-to-one function from A to some $A' \subseteq W$. If x is a partial trace, then $x' = (\gamma', \delta, f')$ where γ' results from using r to rename the elements of γ and $f' = r \circ f$.

The definition of the concatenation operator $x_3 = x_1 \cdot x_2$, wherer x_1 is a partial trace and x_2 is either a partial or a complete trace, is more complicated. If x_2 is a partial trace, then x_3 is defined iff $\gamma_1 = \gamma_2$ and for all $a \in A$,

$$f_1(a)(\delta_1) = f_2(a)(0)$$

(note that δ_1, δ_2, etc., are components of x_1 and x_2 in the obvious way). When defined, $x_3 = (\gamma_1, \delta_3, f_3)$ is such that $\delta_3 = \delta_1 + \delta_2$ and for all $a \in A$

$$f_3(a)(\delta) = f_1(a)(\delta) \text{ for } 0 \leq \delta \leq \delta_1$$
$$f_3(a)(\delta) = f_2(a)(\delta - \delta_1) \text{ for } \delta_1 \leq \delta \leq \delta_3.$$

Note that concatenation is defined only when the end points of the two traces match. The concatenation of a partial trace with a complete trace yields a complete trace with a similar definition. If $x_3 = x_1 \cdot x_2$, then x_1 is a *prefix* of x_3.

3.2 Non-metric Time

In the definition of this trace algebra we are concerned with the order in which events occur in the system, but not in their absolute distance or position. This is useful if we want to describe the semantics of a programming language for hybrid systems that abstracts from a particular real time implementation.

Although we want to remove real time, we want to retain the global ordering on events induced by time. In particular, in order to simplify the abstraction from metric time to non-metric time described below, we would like to support the case of an uncountable number of events[1]. Sequences are clearly inadequate given our requirements. Instead we use a more general notion of a partially

[1] In theory, such Zeno-like behavior is possible, for example, for an infinite loop whose execution time halves with every iteration

ordered multiset to represent the trace. We repeat the definition found in [12], and due to Gischer, which begins with the definition of a labeled partial order.

Definition 1 (Labeled partial order). *A* labeled partial order *(lpo) is a 4-tuple* $(V, \Sigma, \leq, \mu)$ *consisting of*

1. *a* vertex set V*, typically modeling* events*;*
2. *an* alphabet Σ *(for symbol set), typically modeling* actions *such as the arrival of integer 3 at port Q, the transition of pin 13 of IC-7 to 4.5 volts, or the disappearance of the 14.3 MHz component of a signal;*
3. *a* partial order $\leq$ *on* V*, with* $e \leq f$ *typically being interpreted as event* e *necessarily preceding event* f *in time; and*
4. *a* labeling funciton $\mu : V \to \Sigma$ *assigning symbols to vertices, each labeled event representing an* occurence *of the action labeling it, with the same action possibly having multiple occurence, that is,* μ *need not be injective.*

A *pomset* (partially ordered multiset) is then the isomorphism class of an lpo, denoted $[V, \Sigma, \leq, \mu]$. By taking lpo's up to isomorphism we confer on pomsets a degree of abstractness equivalent to that enjoyed by strings (regarded as finite linearly ordered labeled sets up to isomorphism), ordinals (regarded as well-ordered sets up to isomorphism), and cardinals (regarded as sets up to isomorphism).

This representation is suitable for the above mentioned infinte behaviors: the underlying vertex set may be based on an uncountable total order that suits our needs. For our application, we do not need the full generality of pomsets. Instead, we restrict ourselves to pomsets where the partial order is total, which we call *tomsets*.

Traces have the same form of signature as in metric time:

$$\gamma = (V_{\mathcal{R}}, V_{\mathcal{N}}, M_I, M_O).$$

Both partial and complete traces are of the form $x = (\gamma, L)$ where L is a tomset. When describing the tomset L of a trace, we will in fact describe a particular lpo, with the understanding that L is the isomorphism class of that lpo. An action $\sigma \in \Sigma$ of the lpo is a function with domain A such that for all $v \in V_{\mathcal{R}}$, $\sigma(v)$ is a real number (the value of variable v resulting from the action σ); for all $v \in V_{\mathcal{N}}$, $\sigma(v)$ is an integer; and for all $a \in M_I \cup M_O$, $\sigma(v)$ is 0 or 1. The underlying vertex set V, together with its total order, provides the notion of time, a space that need not contain a metric. For both partial and complete traces, there must exist a unique minimal element $min(V)$. The action $\mu(min(V))$ that labels $min(V)$ should be thought of as giving the initial state of the variables in $V_{\mathcal{R}}$ and $V_{\mathcal{N}}$. For each partial trace, there must exist a unique maximal element $max(V)$ (which may be indentical to $min(V)$).

Notice that, as defined above, the set of partial traces and the set of complete traces are not disjoint. It is convenient, in fact, to extend the definitions so that traces are labeled with a bit that distinguishes partial traces from complete traces, although we omit the details.

By analogy with the metric time case, it is straightforward to define projection and renaming on actions $\sigma \in \Sigma$. This definition can be easily extended to lpo's and, thereby, traces.

The concatenation operation $x_3 = x_1 \cdot x_2$ is defined iff x_1 is a partial trace, $\gamma_1 = \gamma_2$ and $\mu_1(max(V_1)) = \mu_2(min(V_2))$. When defined, the vertex set V_3 of x_3 is a disjoint union:

$$V_3 = V_1 \uplus (V_2 - min(V2))$$

ordered such that the orders of V_1 and V_2 are preserved and such that all elements of V_1 are less than all elements of V_2. The labeling function is such that for all $v \in V_3$

$$\mu_3(v) = \mu_1(v) \text{ for } min(V_1) \leq v \leq max(V_1)$$
$$\mu_3(v) = \mu_2(v) \text{ for } max(V_1) \leq v.$$

3.3 Pre-post Time

The third and last trace algebra is concerned with modeling non-interactive constructs of a programming language. In this case we are interested only in an agents possible final states given an initial state. This semantic domain could therefore be considered as a denotational representation of an axiomatic semantics.

We cannot model communication actions at this level of abstraction, so signatures are of the form $\gamma = (V_{\mathcal{R}}, V_{\mathcal{N}})$ and the alphabet of γ is $A = V_{\mathcal{R}} \cup V_{\mathcal{N}}$. A non-degenerate state s is a function with domain A such that for all $v \in V_{\mathcal{R}}$, $s(v)$ is a real number (the value of variable v in state s); and for all $v \in V_{\mathcal{N}}$, $s(v)$ is an integer. We also have a degenerate, undefined state $\perp_*$.

A partial trace $B_P(\gamma)$ is a triple (γ, s_i, s_f), where s_i and s_f are states. A complete trace $B_C(\gamma)$ is of the form $(\gamma, s_i, \perp_\omega)$, where $\perp_\omega$ indicates non-termination. This trace algebra is primarily intended for modeling terminating behaviors, which explains why so little information is included on the traces that model non-terminating behaviors.

The operations of projection and renaming are built up from the obvious definitions of projection and renaming on states. The concatenation operation $x_3 = x_1 \cdot x_2$ is defined iff x_1 is a partial trace, $\gamma_1 = \gamma_2$ and the final state of x_1 is identical to the initial state of x_2. As expected, when defined, x_3 contains the initial state of x_1 and the final state of x_2.

3.4 Trace Structure Algebras

The basic relationship between trace algebras, trace structures and trace structure algebras was described earlier (see section 2). This section provides a few more details. A trace algebra provides a set of signatures and a set of traces for each signature.

A trace structure over a given trace algebra is a pair (γ, P), where γ is a signature and P is a subset of the traces for that signatures. The set P represents the set of possible behaviors of an agent.

A trace structure algebra contains a set of trace structures over a given trace algebra. Operations of projection, renaming, parallel composition and serial composition on trace structures are defined using the operations of the trace algebra, as follows.

Project and renaming are the simplest operations to define. When they are defined depends on the signature of the trace structure in the same way that definedness for the corresponding trace algebra operations depends on the signatures of the traces. The signature of the result is also analogous. Finally, the set of traces of the result is defined by naturally extending the trace algebra operations to sets.

Sequential composition is defined in terms of concatentation in an analogous way. The only difference from projection and renaming is that sequential composition requires two traces structures as arguments, and concatenation requires two traces as arguments.

Parallel composition of two trace structures is defined only when all the traces in the structures are complete traces. Let trace structure T'' be the parallel composition of T and T'. Then the components of T'' are as follows (M_I and M_O are omitted in pre-post traces):

$$\begin{aligned} V''_{\mathcal{R}} &= V_{\mathcal{R}} \cup V'_{\mathcal{R}} \\ V''_{\mathcal{N}} &= V_{\mathcal{N}} \cup V'_{\mathcal{N}} \\ M''_O &= M_O \cup M'_O \\ M''_I &= (M_I \cup M'_I) - M''_O \\ P'' &= \{x \in \mathcal{B}_C(\gamma'') : proj(A)(x) \in P \wedge \\ &\qquad proj(A')(x) \in P'\}. \end{aligned}$$

4 Homomorphisms

The three trace algebras defined above cover a wide range of levels of abstraction. The first step in formalizint the relationships between those levels is to define homomorphims between the trace algebras. As mentioned in section 2, trace algebra homomorphisms induce corresponding conservative approximations between trace structure algebras.

4.1 From Metric to Non-metric Time

A homomorphism from metric time trace algebra to non-metric time should abstract away detailed timing information. This requires characterizing events in metric time and mapping those events into a non-metric time domain. Since metric time trace algebra is, in part, value based, some additional definitions are required to characterize events at that level of abstraction.

Let x be a metric trace with signature γ and alphabet A such that

$$\begin{aligned} \gamma &= (V_{\mathcal{R}}, V_{\mathcal{N}}, M_I, M_O) \\ A &= V_{\mathcal{R}} \cup V_{\mathcal{N}} \cup M_I \cup M_O. \end{aligned}$$

We define the homomorphism h by defining a non-metric time trace $y = h(x)$. This requires building a vertex set V and a labeling function μ to construct an lpo. The trace y is the isomorphism class of this lpo. For the vertex set we take all reals such that an event occurs in the trace x, where the notion of event is formalized in the next several definitions.

Definition 2 (Stable function). *Let f be a function over a real interval to $\mathcal{R}$ or $\mathcal{N}$. The function is* stable at t *iff there exists an $\epsilon > 0$ such that f is constant on the interval $(t - \epsilon, t]$.*

Definition 3 (Stable trace). *A metric time trace x is* stable at t *iff for all $v \in V_{\mathcal{R}} \cup V_{\mathcal{N}}$ the function $f(v)$ is stable at t; and for all $a \in M_I \cup M_O$, $f(a)(t) = 0$.*

Definition 4 (Event). *A metric time trace x has an* event at $t > 0$ *if it is not stable at t. Because a metric time trace doesn't have a left neighborhood at $t = 0$, we always assume the presence of an event at the beginning of the trace. If x has an event at t, the* action label σ *for that event is a function with domain A such that for all $v \in A$, $\sigma(a) = f(a)(t)$, where f is a component of x as described in the definition of metric time traces.*

Now we construct the vertex set V and labeling function μ necessary to define y and, thereby, the homomorphsim h. The vertex set V is the set of reals t such that x has an event at t. While it is convenient to make V a subset of the reals, remember that the tomset that results is an isomorphism class. Hence the metric defined on the set of reals is lost. The labeling function μ is such that for each element $t \in V$, $\mu(t)$ is the action label for the event at t in x.

Note that if we start from a partial trace in the metric trace we obtain a trace in the non-metric trace that has an initial and final event. It has an initial event by definition. It has a final event because the metric trace either has an event at δ (the function is not constant), or the function is constant at δ but then there must be an event that brought the function to that constant value (which, in case of identically constant functions, is the initial event itself).

To show that h does indeed abstract away information, consider the following situation. Let x_1 be a metric time trace. Let x_2 be same trace where time has been "stretched" by a factor of two (i.e., forall $v \in A_1$, $x_1(a)(t) = x_2(a)(2t)$). The vertex sets generated by the above process are isomorphic (the order of the events is preserved), therefore $h(x_1) = h(x_2)$.

4.2 From Non-metric to Pre-post Time

The homomorphism h from the non-metric time traces to pre-post traces requires that the signature of the trace structure be changed by removing M_I and M_O. Let $y = h(x)$. The initial state of y is formed by restricting $\mu(min(V))$ (the initial state of x) to $V_{\mathcal{R}} \cup V_{\mathcal{N}}$. If x is a complete trace, then the final state of y is $\perp_\omega$. If x is a complete trace, and there exists $a \in M_I \cup M_O$ and time t such that $f(a)(t) = 1$, the final state of y is $\perp_*$. Otherwise, the final state of y is formed by restricting $\mu(max(V))$.

5 Conservative Approximations

Trace algebras and trace structure algebras are convenient tools for constructing models of agents. We are interested in relating different models that describe systems at different levels of abstraction. Let $\mathcal{A}$ and $\mathcal{A}'$ be two trace structure algebras. A *conservative approximation* is a pair of functions (Ψ_l, Ψ_u) that map the trace structures in $\mathcal{A}$ into the trace structures in $\mathcal{A}'$. Intuitively, the trace structure $\Psi_u(T)$ in $\mathcal{A}'$ is an upper bound of the behaviors contained in T (i.e. it contains all abstracted behaviors of T plus, possibly, some more). Similarly, the trace structure $\Psi_l(T)$ in $\mathcal{A}'$ represents a lower bound of T (it contains only abstract behaviors of T, but possibly not all of them). As a result,

$$\Psi_u(T_1) \subseteq \Psi_l(T_2) \text{ implies } T_1 \subseteq T_2.$$

Thus, a verification problem that involves checking for refinement of a specification can be done in $\mathcal{A}'$, where it is presumably more efficient than in $\mathcal{A}$. The conservative approximation guarantees that this will not lead to a false positive result, although false negatives are possible.

5.1 Homomorphisms and Conservative Approximations

A conservative approximation can be derived from a homomorphism between two trace algebras. A homomorphism h is a function between the domains of two trace algebras that commutes with projection, renaming and concatenation. Consider two trace algebras $\mathcal{C}$ and $\mathcal{C}'$. Intuitively, if $h(x) = x'$ the trace x' is an abstraction of any trace y such that $h(y) = x'$. Thus, x' can be thought of as representing the set of all such y. Similarly, a set X' of traces in $\mathcal{C}'$ can be thought of as representing the largest set Y such that $h(Y) = X'$, where h is naturally extended to sets of traces. If $h(X) = X'$, then $X \subseteq Y$, so X' represents a kind of upper bound on the set X. Hence, if $\mathcal{A}$ and $\mathcal{A}'$ are trace structure algebras over $\mathcal{C}$ and $\mathcal{C}'$ respectively, we use the function Ψ_u that maps an agent P in $\mathcal{A}$ into the agent $h(P)$ in $\mathcal{A}'$ as the upper bound in a conservative approximation. A sufficient condition for a corresponding lower bound is: if $x \not/ P$, then $h(x)$ is not in the set of possible traces of $\Psi_l(T)$. This leads to the definition of a function $\Psi_l(T)$ that maps P into the set $h(P) - h(\mathcal{B}(A) - P)$. The conservative approximation $\Psi = (\Psi_l, \Psi_u)$ is an example of a *conservative approximation induced by* h. A slightly tighter lower bound is also possible (see [2]).

It is straightforward to take the general notion of a conservative approximation induced by a homomorphism, and apply it to specific models. Simply construct trace algebras $\mathcal{C}$ and $\mathcal{C}'$, and a homomorphism h from $\mathcal{C}$ to $\mathcal{C}'$. Recall that these trace algebras act as models of individual behaviors. One can construct the trace structure algebras $\mathcal{A}$ over $\mathcal{C}$ and $\mathcal{A}'$ over $\mathcal{C}'$, and a conservative approximation Ψ induced by h. Thus, one need only construct two models of individual behaviors and a homomorphism between them to obtain two trace structure models along with a conservative approximation between the trace structure models.

This same approach can be applied to the three trace algebras, and the two homomorphisms between them, that were defined in section 3, giving conservative approximations between process models at three different levels of abstraction.

5.2 Inverses of Conservative Approximations

Conservative approximations represent the process of abstracting a specification in a less detailed semantic domain. Inverses of conservative approximations represent the opposite process of refinement.

Let $\mathcal{A}$ and $\mathcal{A}'$ be two trace structure algebras, and let Ψ be a conservative approximation between $\mathcal{A}$ and $\mathcal{A}'$. Normal notions of the inverse of a function are not adequate for our purpose, since Ψ is a pair of functions. We handle this by only considering the T in $\mathcal{A}$ for which $\Psi_u(T)$ and $\Psi_l(T)$ have the same value T'. Intuitively, T' represents T exactly in this case, hence we define $\Psi_{inv}(T') = T$. When $\Psi_u(T) \not= \Psi_l(T)$ then Ψ_{inv} is not defined.

The inverse of a conservative approximation can be used to embed a trace structure algebra at a higher level of abstraction into one at a lower level. Only the agents that can be represented exactly at the high level are in the image of the inverse of a conservative approximation. We use this as part of our approach for reasoning about embedded software at multiple levels of abstraction.

6 Embedded Software

This section outlines our approach for using multiple levels of abstraction to analyze embedded software. Our motivating example is a small segment of code used for engine cutoff control [1]. This example is particularly interesting to us because the solution proposed in [1] includes the use of a hybrid model to describe the torque generation mechanism.

6.1 Cutoff Control

The behaviors of an automobile engine are divided into regions of operation, each characterized by appropriate control actions to achieve a desired result. The cutoff region is entered when the driver releases the accelerator pedal, thereby requesting that no torque be generated by the engine. In order to minimize power train oscillations that result from suddenly reducing torque, a closed loop control damps the oscillations using carefully timed injections of fuel. The control problem is therefore hybrid, consisting of a discrete (the fuel injection) and a continuous (the power train behavior) systems tightly linked. The approach taken in [1] is to first relax the problem to the continuous domain, solve the problem at this level, and finally abstract the solution to the discrete domain.

Figure 1 shows the top level routine of the control algorithm. Although we use a C-like syntax, the semantics are simplified, as described later. The controller is activated by a request for an injection decision (this happens every full engine

cycle). The algorithm first reads the current state of the system (as provided by the sensors on the power train), predicts the effect of injecting or not injecting on the future behavior of the system, and finally controls whether injection occurs. The prediction uses the value of the past three decisions to estimate the position of the future state. The control algorithm involves solving a differential equation, which is done in the call to `compute_sigmas` (see [1] for more details). A nearly optimal solution can be achieved without injecting intermediate amounts of fuel (i.e., either inject no fuel or inject the maximum amount). Thus, the only control inputs to the system are the actions `action_injection` (maximum injection) and `action_no_injection` (zero injection).

```
void control_algorithm( void ) {
  // state definition
  struct state { double x1; double x2; double omega_c; } current_state;
  // Init the past three injections (assume injection before cutoff)
  double u1, u2, u3 = 1.0;
  // Predictions
  double sigma_m, sigma_0;

  loop forever {
    await( action_request );
    read_current_state( current_state );
    compute_sigmas( sigma_m, sigma_0, current_state, u1, u2, u3 );
    // update past injections
    u1 = u2;
    u2 = u3;
    // compute next injection signal
    if ( sigma_m < sigma_0 ) {
      action_injection( );
      u3 = 1.0;
    } else {
      action_no_injection( );
      u3 = 0.0;
    }
  }
}
```

Fig. 1. The control algorithm

6.2 Using Pre-post Traces

One of the fundamental features of embedded software is that it interacts with the physical world. Conventional axiomatic or denotational semantics of sequential programming languages only model initial and final states of terminating

programs. Thus, these semantics are inadequate to fully model embedded software. However, much of the code in an embedded application does computation or internal communication, rather than interacting with the physical world. Such code can be adequately modeled using conventional semantics, as long as the model can be integrated with the more detailed semantics necessary for modeling interaction. Pre-post trace structures are quite similar to conventional semantics. As described earlier, we can also embed pre-post trace structures into more detailed models. Thus, we can model the non-interactive parts of an embedded application at a high level of abstraction that is simpler and more natural, while also being able to integrate accurate models of interaction, real-time constraints and continuous dynamics.

This subsection describes the semantics of several basic programming language constructs in terms of pre-post trace structures. The following two subsections describe how these semantics can be integrated into more detailed models.

The semantics of each statement is given by a trace structure. To simplify the semantics, we assume that inter-process communication is done through shared actions rather than shared variables. A pre-post trace structure has a signature γ of the form $(V_{\mathcal{R}}, V_{\mathcal{N}})$. For the semantics of a programming language statement, γ indicates the variables accessible in the scope where the statement appears. For a block that declares local variables, the trace structure for the statement in the block includes in its signature the local variables. The trace structure for the block is formed by projecting away the local variables from the trace structure of the statement.

The sequential composition of two statements is defined as the concatenation of the corresponding trace structures: the definition of concatenation ensures that the two statements agree on the intermediate state. The traces in the trace structure for an assignment to variable v are of the form (γ, s_i, s_f), where s_i is an arbitrary initial state, and s_f is identical to s_i except that the value of v is equal to the value of the right-hand side of the assignment statement evaluated in state s_i (we assume the evaluation is side-effect free).

The semantics of a procedure definition is given by a trace structure with an alphabet $\{v_1, \ldots, v_r\}$ where v_k is the k-th argument of the procedure (these signal names do not necessarily correspond to the names of the formal variables). We omit the details of how this trace structure is constructed from the text of the procedure definition. More relevant for our control algorithm example, the semantics of a procedure call `proc(a, b)` is the result of renaming $v_1 \to a$ and $v_2 \to b$ on the trace structure for the definition of `proc`. The parameter passing semantics that results is *value-result* (i.e. no aliasing or references) with the restriction that no parameter can be used for both a value and result. More realistic (and more complicated) parameter passing semantics can also be modeled.

To define the semantics of `if-then-else` and `while` loops we define a function $\mathit{init}(x, c)$ to be true if and only if the predicate c is true in the initial state of trace x. The formal definition depends on the particular trace algebra being used. In particular, for pre-post traces, $\mathit{init}(x, c)$ is false for all c if x has $\perp_*$ as its initial state.

For the semantics of `if-then-else`, let c be the conditional expression and let P_T and P_E be the sets of possible traces of the `then` and `else` clauses, respectively. The set of possible traces of the `if-then-else` is

$$P = \{x \in P_T : \mathit{init}(x, c)\} \cup \{x \in P_E : \neg \mathit{init}(x, c)\}$$

Notice that this definition can be used for any trace algebra where $\mathit{init}(x, c)$ has been defined, and that it ignores any effects of the evaluation of c not being atomic.

In the case of `while` loops we first define a set of traces E such that for all $x \in E$ and traces y, if $x \cdot y$ is defined then $x \cdot y = y$. For pre-post traces, E is the set of all traces with identical initial and final states. If c is the condition of the loop, and P_B the set of possible traces of the body, we define $P_{T,k}$ and $P_{N,k}$ to be the set of terminating and non-terminating traces, respectively, for iteration k, as follows:

$$\begin{aligned} P_{T,0} &= \{x \in E : \neg \mathit{init}(x, c)\} \\ P_{N,0} &= \{x \in E : \mathit{init}(x, c)\} \\ P_{T,k+1} &= P_{N,k} \cdot P_B \cdot P_{T,0} \\ P_{N,k+1} &= P_{N,k} \cdot P_B \cdot P_{N,0} \end{aligned}$$

The concatenation of $P_{T,0}$ and $P_{N,0}$ at the end of the definition ensures that the final state of a terminating trace does not satisfy the condition c, while that of a non-terminating trace does. Clearly the semantics of the loop should include all the terminating traces. For non-terminating traces, we need to introduce some additional notation. A sequence $Z =< z_0, \ldots >$ is a non-terminating execution sequence of a loop if, for all k, $z_k \in P_{N,k}$ and $z_{k+1} \in z_k \cdot P_B$. This sequence is a chain in the prefix ordering. The initial state of Z is defined to be the initial state of z_0. For pre-post traces, we define $P_{N,\omega}$ to be all traces of the form $(\gamma, s, \perp_\omega)$ where s is the initial state of some non-terminating execution sequence Z of the loop. The set of possible traces of the loop is therefore

$$P = (\bigcup_k P_{T,k}) \cup P_{N,\omega}.$$

6.3 Using Non-metric Time Traces

Using an inverse conservative approximation, as described earlier, the pre-post trace semantics described in the previous subsection can be embedded into non-metric time trace structures. However, this is not adequate for two of the constructs used in figure 1: `await` and the non-terminating loop. These constructs must be describe directly at the lower level of abstraction provided by non-metric time traces.

As used used in figure 1, the `await(a)` simply delays until the external action `a` occurs. Thus, the possible partial traces of `await` are those where the values

of the state variables remain unchanged and the action a occurs exactly once, at the endpoint of the trace. The possible complete traces are similar, except that the action a must never occur.

To give a more detailed semantics for non-terminating loops, we define the set of extensions of a non-terminating execution sequence Z to be the set $ext(Z) = \{x \in \mathcal{B}(\gamma) : \forall k[z_k \in pref(x)]\}$. For any non-terminating sequence Z, we require that $ext(Z)$ be non-empty, and have a unique maximal lower bound contained in $ext(Z)$, which we denote $\lim(Z)$. In the above definition of the possible traces of a loop, we modify the definition of the set of non-terminating behaviors $P_{N,\omega}$ to be the set of $\lim(Z)$ for all non-terminating execution sequences Z.

6.4 Using Metric Time Traces

Analogous to the embedding discussed in the previous subsection, non-metric time traces structures can be embedded into metric-time trace structures. Here continuous dynamics can be represented, as well as timing assumptions about programming language statements. Also, timing constraints that a system must satisfy can be represented, so that the system can be verified against those constraints.

7 Conclusions and Comparisons with Other Approaches

It was not our goal to construct a single unifying semantic domain, or even a parameterized class of unifying semantic domains. Instead, we wish to construct a formal framework that simplifies the construction and comparison of different semantic domains, including semantic domains that can be used to unify specific, restricted classes of other semantic domains.

There is a tradeoff between two goals: making the framework general, and providing structure to simplify constructing semantic domains and understanding their properties. While our framework is quite general, we have formalized several assumptions that must be satisfied by our semantic domains. These include both axioms and constructions that build process models (and mappings between them) from models of individual behaviors (and their mappings). These assumptions allow us to prove many generic theorems that apply to all semantic domains in our framework. In our experience, having these theorems greatly simplifies constructing new semantic domains that have the desired properties and relationships.

Process Spaces [10,11] are an extremely general class of concurrency models. However, because of their generality, they do not provide much support for constructing new semantic domains or relationships between domains. For example, by proving generic properties of broad classes conservative approximations, we remove the need to reprove these properties when a new conservative approximation is constructed.

Similarly, our notion of conservative approximation can be described in terms of abstract interpretations. However, abstraction interpretations are such a gen-

eral concept that they do not provide much insight into abstraction and refinment relationships between different semantic domains.

Many are the models that have been proposed to represent the behavior of hybrid systems. Most of them share the same view of the behavior as composed of a sequence of steps; each step is either a continuous evolution (a flow) or a discrete change (a jump). Different models varies in the way they represent the sequence. One example is the Masaccio model ([7,8]) proposed by Henzinger et alii. In Masaccio the representation is based on components that communicate with other components through variables and locations. During an execution the flow of control transitions from one location to another according to a state diagram that is obtained by composing the components that constitute the system. The underlying semantic model is based on sequences. The behavior of each component is characterized by a set of finite executions, each of them composed of an entry location and a sequence of steps that can be either jumps or flows. An exit location is optional. The equations associated with the transitions in the state diagram define the legal jumps and flows that can be taken during the sequence of steps.

The operation of composition in Masaccio comes in two flavors: parallel and serial. The parallel composition is defined on the semantic domain as the conjunction of the behaviors: each execution of the composition must also match an execution of the individual components. Conversely, serial composition is defined as the disjunction of the behaviors: each execution of the composition need only match the execution of one of the components. Despite its name, this operation doesn't serialize the behaviors of the two components. Instead, a further operation of *location hiding* is required to string together the possible executions of a disjunction.

In our framework we talk about hybrid models in terms of the semantic domain only (which is based on functions of a real variable rather than sequences). This is a choice of emphasis: in Masaccio the semantic domain is used to describe the behavior of a system which is otherwise represented by a transition system. In our approach the semantic domain is the sole player and we emphasize results that abstract from the particular representation that is used. It's clear, on the other hand, that a concrete representation (like a state transition system) is extremely important in developing applications and tools that can generate or analyze an implementation of a system.

In our paper we presented three models for semantic domains. Masaccio compares to our more detailed model. In our approach we have decided to model the flows and the jumps using a single function of a real variable: flows are the continuous segments of the functions, while jumps are the points of discontinuity. This combined view of jumps and flows is possible in our framework because we are not constrained by a representation based on differential equations, and hence we do not require the function to be differentiable. Another difference is that different components are allowed to concurrently execute a jump and a flow, as long as the conditions imposed by the operation of parallel composition are satisfied.

Because in Masaccio the operations of composition are defined on the semantic domain and not on the representation it is easy to do a comparison with our framekwork. Parallel composition is virtually identical (both approaches use a projection operation). On the other hand we define serial composition in quite different terms: we introduce a notion of concatenation that is difficult to map to the sequence of steps that include serial composition and location hiding. In fact, it appears that the serial composition so obtained might contain side-effects that are intuitively not intended in a proper sequential composition of behaviors (because of the projection during the serial composition, a behavior of the compound component might include executions that were not originally present in the components themselves). We believe this could simply be an artifact of the representation based on state transitions that requires the identification of the common points where the control can be transferred.

The concept of *refinement* in Masaccio is also based on the semantic domain. Masaccio extends the traditional concept of trace containment to a prefix relation on trace sets. In particular, a component A refines a component B either if the behavior of A (its set of executions) is contained in the behavior of B, or if the behaviors of A are suffixes of behaviors of B. In other words, B could be seen as the prefix of all legal behaviors.

In our framework we must distinguish between two notions of refinement. The first is a notion of refinement within a semantic domain: in our framework this notion is based on pure trace containment. We believe this notion of refinement is sufficient to model the case of sequential systems as well: it is enough to require that the specification include all possible continuations of a common prefix.

The second notion of refinement that is present in our framework has to do with changes in the semantic domain. This notion is embodied in the concept of conservative approximation that relates models at one level of abstraction to models at a different level of abstraction. There is no counterpart of this notion in the Masaccio model.

References

1. A. Balluchi, M. D. Benedetto, C. Pinello, C. Rossi, and A. Sangiovanni-Vincentelli. Cut-off in engine control: a hybrid system approach. In *IEEE Conf. on Decision and Control*, 1997.
2. J. R. Burch. *Trace Algebra for Automatic Verification of Real-Time Concurrent Systems*. PhD thesis, School of Computer Science, Carnegie Mellon University, Aug. 1992. Technical Report CMU-CS-92-179.
3. J. R. Burch, R. Passerone, and A. Sangiovanni-Vincentelli. Overcoming heterophobia: Modeling concurrency in heterogeneous systems. In M. Koutny and A. Yakovlev, editors, *Application of Concurrency to System Design*, 2001.
4. J. Davis II, M. Goel, C. Hylands, B. Kienhuis, E. A. Lee, J. Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Reekie, N. Smyth, J. Tsay, and Y. Xiong. Overview of the ptolemy project. ERL Technical Report UCB/ERL No. M99/37, Dept. EECS, University of California, Berkeley, July 1999.

5. J. Davis II, C. Hylands, B. Kienhuis, E. A. Lee, J. Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Tsay, B. Vogel, and Y. Xiong. Heterogeneous concurrent modeling and design in java. Technical Memorandum UCB/ERL M01/12, EECS, University of California, Berkeley, Mar. 2001.
6. S. Edwards, L. Lavagno, E. Lee, and A. Sangiovanni-Vincentelli. Design of embedded systems: Formal models, validation, and synthesis. *Proceedings of the IEEE*, 85(3):366–390, Mar. 1997.
7. T. Henzinger. Masaccio: a formal model for embedded components. In J. van Leeuwen, O. Watanabe, M. Hagiya, P. Mosses, and T. Ito, editors, *TCS 00: Theoretical Computer Science*, volume 1872 of *Lecture Notes in Computer Science*, pages 549–563. Springer-Verlag, 2000.
8. T. Henzinger, M. Minea, and V. Prabhu. Assume-guarantee reasoning for hierarchical hybrid systems. In M. di Benedetto and A. Sangiovanni-Vincentelli, editors, *HSCC 00: Hybrid Systems—Computation and Control*, volume 2034 of *Lecture Notes in Computer Science*, pages 275–290. Springer-Verlag, 2001.
9. E. A. Lee and A. L. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Trans. Comput.-Aided Design Integrated Circuits*, 17(12):1217–1229, Dec. 1998.
10. R. Negulescu. *Process Spaces and the Formal Verification of Asynchronous Circuits*. PhD thesis, University of Waterloo, Canada, 1998.
11. R. Negulescu. Process spaces. In C. Palamidessi, editor, *CONCUR*, volume 1877 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.
12. V. R. Pratt. Modelling concurrency with partial orders. *International Journal of Parallel Programming*, 15(1):33–71, Feb. 1986.

Hierarchical Approach for Design of Multi-vehicle Multi-modal Embedded Software

T. John Koo, Judy Liebman, Cedric Ma, and S. Shankar Sastry

Department of Electrical Engineering and Computer Sciences,
University of California at Berkeley
Berkeley, CA 94704,
koo,judithl,cedricma,sastry@eecs.berkeley.edu,

Abstract. Embedded systems composed of hardware and software components are designed to interact with a physical environment in real-time in order to fulfill control objectives and system specifications. In this paper, we address the complex design challenges in embedded software by focusing on predictive and systematic hierarchical design methodologies which promote system verification and validation. First, we advocate a mix of top-down, hierarchical design and bottom-up, component-based design for complex control systems. Second, it is our point of view that at the level closest to the environment under control, the embedded software needs to be time-triggered for guaranteed safety; at the higher levels, we advocate an asynchronous hybrid controller design. We briefly illustrate our approach through an embedded software design for the control of a group of autonomous vehicles.

1 Introduction

Embedded software is designed to process information to and fro between the information and physical worlds. Advances in software, computation, communication, sensing and actuation have enabled the rapid realization of high-performance and sophisticated multi-vehicle systems. The rapidly growing demand for high-confidence embedded software that is required in order to control new generations of autonomous vehicles for collectively delivering high levels of mission reliability, is putting tremendous pressure on control and software designers in industry.

A high-confidence system should have the following characteristics: correctness by construction and fault-tolerance, and resistance to information attack. In fact, the cost of system development is mainly due to prohibitively expensive embedded software integration and testing techniques that rely almost exclusively on exhaustively testing of more or less complete versions of complex systems. Formal methods are hence introduced for the synthesis and verification of embedded software in order to guarantee that the system is correct by construction with respect to a given set of specifications. The system design should also be fault-tolerant so that it can handle all anticipated faults that might occur in the system and possibly recover from them after the faults have been detected and

T.A. Henzinger and C.M. Kirsch (Eds.): EMSOFT 2001, LNCS 2211, pp. 344–360, 2001.
© Springer-Verlag Berlin Heidelberg 2001

identified. Furthermore, the design should also ensure that the system is not vulnerable to attack from the information world by taking into account models of attack. While this last item is important we do not address it in this paper since it would require a lengthy discussion of models of attack of embedded systems.

In this paper, we address this bottleneck by focusing on systematic hierarchical design methodologies. We briefly illustrate our approach through an embedded software design for the control of a group of autonomous vehicles. Consider that the task consists of flying a group of autonomous vehicles in a prespecified formation. We assume that each vehicle is equipped with the necessary sensing, communication, and computation capabilities in order to perform a set of predetermined tasks. The control of the multi-vehicle systems can be organized as a distributed hierarchical system. The meaning of a distributed system refers to a system comprised of several subsystems which are spatially distributed. Large-scale systems ranging from automated highway systems (AHS) [1], air traffic management systems (ATMS) [2], and power distribution networks are typical examples of distributed systems. However, large-scale systems are systems of very high complexity. Complexity is typically reduced by imposing a hierarchical structure on the system architecture. In a such a structure, systems of higher functionality reside at higher levels of the hierarchy and are therefore unaware of lower-level details. A component-based design provides a clean way to integrate different models by hierarchical nesting of parallel and serial composition of heterogeneous components. This hierarchical composition also allows one to manage the complexity of a design by information hiding and by reusing components.

To cope with these complex design challenges, we advocate a mix of top-down, hierarchical design and bottom-up, component-based design for complex control systems. Hierarchical design begins with the choice and evaluation of an overall distributed, multi-layered system architecture; component-based design begins with the derivation of robust mathematical control laws and decision-making procedures.

Hybrid systems, in general, are defined as systems built from atomic discrete and continuous components by parallel and serial composition, which is arbitrarily nested. Hybrid systems refers to the distinguishing fundamental characteristics of embedded control systems, namely, the tight coupling and interaction of discrete with continuous phenomena. The coupling is inherent to embedded systems since every digital software/hardware implementation of a control design is ultimately a discrete approximation that interacts through sensors and actuators with a continuous physical environment. There has been a large and growing body of work on formal methods for hybrid systems: mathematical logics, computational models and methods, and automated reasoning tools supporting the formal specification and verification of performance requirements for hybrid systems, and the design and synthesis of control programs for hybrid systems that are provably correct with respect to formal specifications.

Depending on the level of abstraction and domain-specificity of design practice, different models of computation (MOCs) [6] which govern the interaction

among components can be chosen and mixed for composing hybrid systems. There are a rich variety of models of computation that deal with concurrency and time in different ways. As mentioned in [7], MOCs such as continuous-time (CT), synchronous data flow (SDF), finite-state machine (FSM), synchronous/ reactive (SR) [8,9,10,11], discrete-event (DE), and time triggered (TT) [13,14] are useful for the design of embedded systems.

Hybrid systems have been used in solving synthesis and verification problems of some high-profile and safety-critical applications such as conflict resolution [27] for multi-vehicle platforms, and multi-modal control [20] and envelope protection [17] for single vehicle platforms. In the above words, the hybrid system is a composition of CT and FSM. Any transition in system discrete states can occur only when the transition guard, which is a function of discrete/continuous states and inputs, becomes true. The continuous state in a discrete location evolves continuously according to differential equations, as long as the location's invariant remains true. Each transition is asynchronous since it can happen any time as long as the guard condition is satisfied.

Embedded systems composed of hardware and software components are designed to interact with a physical environment in real-time in order to fulfill control objectives and system specifications. In the real-time computing literature, the word *time* means that the correctness of the system depends not only on the logical result of the computation but also on the time at which the results are produced. In the implementation of control laws using digital computers, the necessary computational activities for the implementation of control laws are decomposed and translated into periodic tasks. Hence, there is a *hard* deadline in time for the task to meet in each period of execution. Failure in meeting the deadline may cause catastrophic consequences on the environment under control. The word *real* indicates that the reaction of the systems to external events must occur during their evolution. Notice that the notion of time is not an intrinsic property of a control system but it is strictly related to the environment under control. For example, the design of a real-time embedded system for flight control has to take the timing characteristics of a vehicle into consideration.

It is our point of view that at the level closest to the physical systems under control, the embedded software needs to be *time-triggered* for guaranteed safety. At higher levels, we advocate *asynchronous* decision-making procedures. In Section 2, a hierarchical architecture which allows modular verification will be introduced for the construction of embedded systems. Next, in Section 3, we will show the state-of-the-art in verification and synthesis of control laws and decision-making algorithms based on formal methods. Motivation and implementation of the control design in embedded software using a time-triggered framework will be presented in Section 4. Finally, we conclude our work in Section 5.

2 Hierarchical Architecture for Multi-vehicle Multi-modal Systems

Imposing a hierarchical structure on the system architecture has been used for solving the control problem of large-scale systems. A desired hierarchical structure should not only provide manageable complexity but also promote verification. There are several approaches to understanding a hierarchy depending on the design perspective. In particular, two distinct approaches have been shown in [3] for the design and analysis of AHS [1]. One approach to the meaning of hierarchy is to adopt *one-world* semantics, and the other approach is referred to as *multi-world* semantics.

In one-world semantics for hierarchical systems, a higher-level expression is interpreted in a process called *semantic flattening*: the expression is first compiled into lower-level expression and then interpreted. In other words, an interpretation at each level is semantically complied into a single interpretation at the lowest-level in the *imperative* world. Furthermore, semantic flattening implies that checking any high-level truth-claim can be performed by an automatic procedure if there is a facility for automatically verifying the lowest-level interpretation. This approach provides a unique interpretation to system description and verification. The main drawback to one-world semantics is that higher-level syntax and truth-claims have to be reformulated if there are some changes at any of the lower levels. On the other hand, in multi-world semantics for hierarchical systems, an expression at each level is interpreted at the same level. Therefore, checking the truth-claim at that level is performed in its own *declarative* world. Moreover, this approach conforms with common system design practice. However, relating these separate worlds together is a nontrivial task.

To take advantage of the two approaches, a decision structure has been suggested in [3] for providing connections between the multiple world of declarative semantics and the single world of imperative semantics. The construction of the structure is based on the following ideas: 1. Ideal compilation - a higher-level expression is complied into an idealized lower-level expression and then interpreted; 2. Usage of Invariants - higher-level truth-claims are conditional lower-level truth-claims; 3. Modal decomposition - Splitting multi-world semantics into multiple-frameworks, each dealing with identified faults, should be done if one-world interpretation leads to falsification of higher-level claims that are true in multi-world semantics.

Ideal compilation and usage of invariants are enablers for performing formal verification of the truth-claim at each level of hierarchy. Modal decomposition suggests that the system should be modeled as a multi-modal system which has multiple modes of operation. *Abstraction* is a key concept for providing ways to connect disjointed worlds together. In [4,5], the notions of abstraction, or aggregation, refer to grouping the system states into equivalence classes. Depending on the cardinality of the resulting quotient space, there may be discrete or continuous abstractions. Truth-claims made at each world are then based on a consistent notion of abstraction in order to have a well-defined definition. Furthermore, at different levels of abstraction, the definitions of environment could

also be different. In general, the environment is defined as the entities which could not be designed.

Here, we describe the construction of a hierarchical architecture which keeps the advantages of the multiple world of declarative semantics and the single world of imperative semantics. Consider the control laws and decision-making procedures as the basic components for the construction of the hierarchical system for controlling the environment. Any changes in the scenario are triggered by a fault in the lower level. Each fault, defined in each world, triggers a change in scenario. The construction is formalized by using component-based design approach.

A component-based design provides a clean way to integrate different models by hierarchical nesting of parallel and serial composition of heterogeneous components. This hierarchical composition also allows one to manage the complexity of a design by information hiding and reusing components. In addition to the syntactic issues mentioned in above, for interpreting the syntax, semantic issues have to be addressed in the design. A semantics gives meaning to components and their interconnections. A collection of semantics models which are useful for embedded software design have been codified in [6] as *models of computation* (MOCs). Thus, in the design, the selection of a MOC governing interactions between components depends on whether its properties match the application domain.

CT models represented by differential equations are excellent for modeling physical systems. Execution in FSM, which is a strictly ordered sequence of state transitions that are not concurrent, and FSM models, are amenable to in-depth formal analysis. In DE models of computation, an event consists of a value and time stamp. There is no global clock tick in DE, but there is a globally consistent notion of time. This model has been realized in a large number of simulation environments. In TT, systems with timed events are driven by clocks, so that signals with events are repeated indefinitely with a fixed period. Time-triggered architecture (TTA) [13] is a hardware architecture supporting this highly regular style of computation. The Giotto programming language [14] provides a time-triggered software abstraction which, unlike the TTA, is hardware independent. It is intended for embedded software systems where periodic events dominate.

As defined in [7], a model is a hierarchical aggregation of components, which are called *actors*. Actors encapsulate an atomic execution and provide communication interfaces to other actors. An actor can be *atomic* or *composite*. A composite actor can not only contain other actors but can also be contained by another composite actor. Hence, hierarchies can be arbitrarily nested. A MOC associated with a composite actor is implemented as a *domain*. A domain defines the communication semantics and the execution order among actors. Consider the multi-modal system [12] for single vehicle control depicted in Figure 1. It is constructed by hierarchically nesting parallel and serial compositions of heterogeneous components. There are two levels of hierarchy introduced. At the first level, in the CT domain, the multi-modal controller is modeled as a composite component and the vehicle dynamics is modeled as ordinary differential

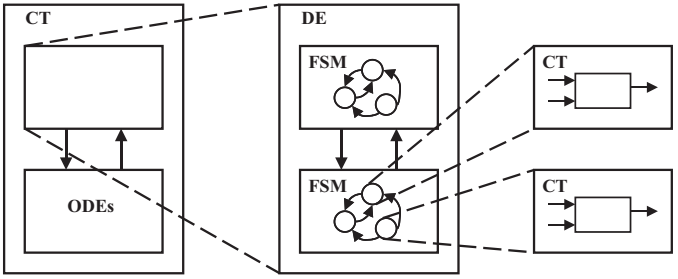


Fig. 1. A multi-modal control system

equations (ODEs). The multi-modal controller is hierarchically refined by two components in the DE domain due to the asynchronous interaction between them. At the second level, the component on the top, models the mode switching logic and the other component, on the bottom, models the control switches by FSM. In each discrete state of the FSM, the component is further refined to model the continuous-time control laws. Notice that domain DE is chosen due to the asynchronous behaviors exhibited by the models. The model assumes a fault-free scenario.

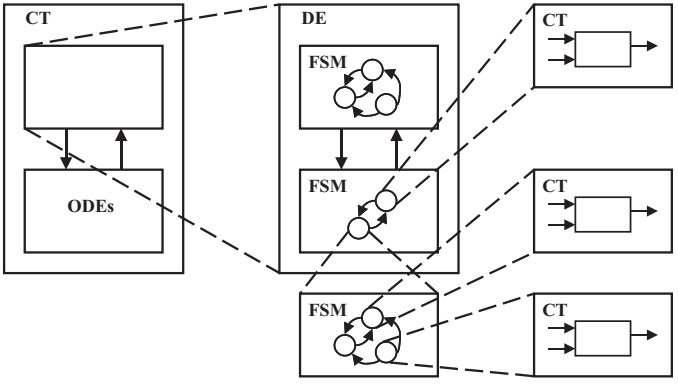


Fig. 2. A multi-modal control system with fault handling

In order to handle anticipated faults such as failures in sensors and actuators or even changes in dynamics, the single world has to be split into two modes of operation in order to handle different scenarios: a non-faulty mode and a faulty mode. The switching between framework is caused by two events generated after fault detection and fault recovery. Therefore, FSM domain is introduced for handling the execution. As shown in Figure 2, a FSM is introduced and there

are two discrete states representing the two modes. In the non-faulty mode, the component is refined by a FSM for modeling control switches, and then further refined by continuous-time control laws. While in the fault mode, the component is refined by a continuous-time controller for handling the anticipated fault.

In order to to reconcile different world views, we propose a hierarchical architecture for the control of multi-vehicle multi-modal systems. It is shown in 3. On the lowest level, the dynamics of the vehicles are combined and modeled by ODEs and the composite components represent the controllers for the vehicles. Consider the hierarchy inside the controller for vehicle 1. At each level of hierarchy, there is a non-fault mode and fault modes. The operation of the non-fault mode assumes that no fault happened in the lower level. The transitions among modes are governed by FSM. Then, the component of each discrete state of FSM can further be refined by the control laws. DE is chosen for the domain to govern the asynchronous communications between levels. In DE domain, although the execution is asynchronous, there is still a globally consistent notion of clock. Similarly, one can construct more levels by further defining and refining the composite components.

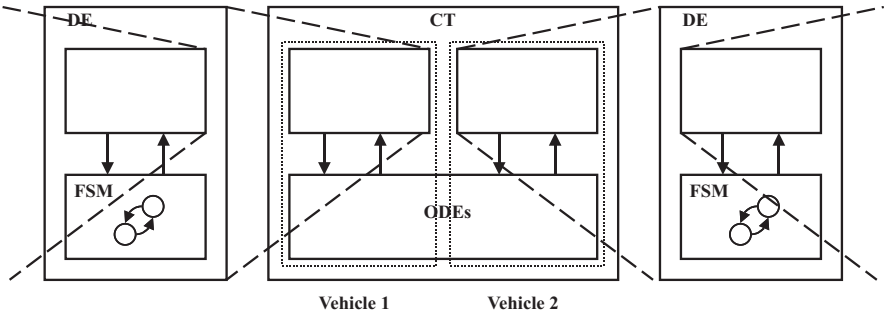


Fig. 3. Proposed hierarchical architecture for multi-vehicle multi-modal systems

3 Design of Multi-vehicle Multi-modal System Components

A hierarchical architecture for multi-vehicle multi-modal systems has been proposed in the previous section. In this section, the components with which the system is composed are presented. The task assigned to the system is decomposed and then executed by the components, according to the system architecture. Each component has to be designed so that it provides a guarantee on its performance under certain assumptions. Consequently the truth-claims of the whole system can be inferred. Formal methods for the synthesis and verification of the components are presented.

Reconsider that a task for a group of autonomous vehicles is to keep flying in a prespecified formation. In the group, one vehicle is assigned to be the leader of the group. The leader is responsible for making decisions on the route and formation pattern for the group. In a bigger picture, leaders from different groups can exchange information and negotiate with other group leaders via a global communication network to perform a mission collectively, for example a pursuit-evasion game [32,33]. Among group members, assume that there exits a local communication network for distributing information for performing the task.

According to the mission along with the information and assumptions made about current environment, the leader has to compute the best possible route and formation for the group with the maneuverability of the vehicles taken into consideration. The set of maneuvers by which a vehicle can perform depends on the available controllers being designed and coded in the embedded software.

As in most of the designs of large-scale systems for autonomous vehicle control, the task is decomposed into four subtasks which are responsible for the control of the vehicle, control mode switching, maneuver sequence generation and route selection. Furthermore, possible faults in each subtask due to the violation of the underlying assumptions of the design and the corresponding fault-handling procedures are presented.

3.1 Control Law Synthesis

Modern control design methodologies [15,16] have enable the synthesis of robust and high-performance control systems. Consider each vehicle dynamics modeled by differential equations of the form

$$\dot{x}(t) = f(x(t)) + g(x(t))u(t), \quad x(t_0) = x_0, \quad t \geq t_0 \tag{3.1}$$

where $x \in \mathbb{R}^n$, $u \in \mathbb{R}^p$, $f(x) : \mathbb{R}^n \to \mathbb{R}^n$ and $g(x) : \mathbb{R}^n \to \mathbb{R}^n \times \mathbb{R}^p$. The system is assumed to be as smooth as needed. The controllers are synthesized with a given set of performance specifications and design criteria with assumptions on the vehicle dynamics, sensor and actuator models which describe sensor noise, actuator dynamics and anticipated structured disturbance. Therefore, the validity of the claim about the control system, at a higher-level, depends on the validity of the assumptions made on the vehicle dynamics, sensor, and actuator models at a lower-level. However, in the presence of sensor and actuator faults or unanticipated disturbance, in order to keep the vehicle in safe operating condition, least restrictive control is introduced in [17] by keeping the vehicle states within the flight envelope so that no catastrophic consequences can happen. The controllers *guarantee* the safe operation of the vehicle under multiple anticipated fault scenarios. In each scenario, game theoretic approach is used to synthesize the so-called *least restrictive control laws*, and the *maximal safe set* for keeping the vehicle state safe is obtained by solving the corresponding Hamilton-Jacobi equation. The Hamilton-Jacobi equation is a partial differential equation of the form

$$\frac{\partial J}{\partial t}(x,t) = -H^*(x, \frac{\partial J}{\partial x}(x,t)) \tag{3.2}$$

where H^* is the so-called Hamiltonian determined from the appropriate sets and the game between the controllable and uncontrollable actions. A computational tool is designed by [18] for computing the reachable set based on a level set technique developed by [19], which computes the viscosity solution to the Hamilton-Jacobi equation, ensuring that discontinuities are preserved.

3.2 Control Mode Switching Synthesis

Given a specific set of controllers of satisfactory performance, a variety of high-level tasks can be accomplished by appropriately switching between low-level controllers. Here, we define a control mode as the operation of the system under a controller that is *guaranteed* to track a certain class of output trajectories. Formally, we have:

A control mode, labeled by q_i where $i \in \{1, \ldots, N\}$, is the operation of the nonlinear system (3.1) under a closed-loop feedback controller of the form

$$u(t) = k_i(x(t), r_i(t)) \tag{3.3}$$

associated with an output $y_i(t) = h_i(x(t))$ such that $y_i(t)$ shall track $r_i(t)$ where $y_i(t), r_i(t) \in \mathbb{R}^{m_i}$, $h_i : \mathbb{R}^n \to \mathbb{R}^{m_i}$, $k_i : \mathbb{R}^n \times \mathbb{R}^{m_i} \to \mathbb{R}^p$ for each $i \in \{1, \ldots, N\}$. We assume that $r_i \in \mathcal{R}_i$, the class of output trajectories associated with the control mode q_i, when the initial condition of the system (3.1) starts in the set $S_i(r_i) \subseteq X_i$, output tracking is guaranteed and the state satisfies a set of state constraints $X_i \subseteq \mathbb{R}^n$.

Consider a reachability task as reaching a desired final control mode from an initial control mode. For this reachability task, [20] has derived a formal method for the synthesis of *control mode graph*. Therefore, if there exists at least one path between an initial control mode to a desired final control mode on the control mode graph, the reachability problem is solvable with a finite number of switching of control modes and, furthermore, the switching conditions can be derived. Switching of controllers [17] with multiple objectives can also be formulated within the same framework, and the partial orders on objectives can be used to bias the search for feasible solutions. The approach consists of extracting a finite graph which refines the original collections of control modes, but is consistent with the physical system. Thus, proof of "control mode switching problem is solvable" is conducted in the semantics of directed graphs. Computational tools based on hybrid system theory have been developed for computing exactly, or approximately reachable sets [17,21,22,23,24] for solving the reachability problem.

However, the design is valid under the free-space assumption in the proximity of the vehicle. Failure in sensing the rapid unanticipated changes in the environment may lead to the vehicle colliding with the environment. Since avoiding obstacles can be formulated as a reachability problem, obstacle avoidance algorithms should be designed to reuse the existing low-level controllers.

3.3 Maneuver Sequence Synthesis

With the availability of a local communication network, vehicles can communicate with each other to perform tasks together. Reconsider the problem of formation. As shown in [25], with consideration on different levels of centralization for formation, one can determine differential geometric conditions that *guarantee* formation feasibility given the individual vehicle kinematics. Information requirements for keeping a formation while maintaining mesh stability, *i.e.* any transient errors dampen out as they travel away from the source of errors within the formation, has been studied in [26].

While there is a communication failure within the network, the group vehicles may not be able to maintain *minimum separation* with each other. Any conflict resolution algorithm must use available, probably local, information to generate maneuvers that resolve conflicts as they are predicted. In [27], given a set of flight modes which are defined as an abstraction of the control modes by considering the kinematic motion of the closed-loop system, the conflict resolution could synthesize the parameters of the maneuver, *i.e.* a sequence of flight modes, and the condition for switching between them. The result would be a maneuver that is proven to be safe within the limits of the model used. The algorithm is based on game-theoretic methods and optimal control. The solution is obtained by performing a hybrid game that is a multi-player structure in which the players have both discrete and continuous moves. Hybrid games have been classified with respect to the complexity of the laws that govern the evolution of the variables and with respect to the winning conditions for the players. This has been studied in the timed game [28,29] for constant differential equations of the form $\dot{x} = c$, the rectangular games [30,31] for rectangular differential inclusions of the form $c_i \leq \dot{x}_i \leq d_i$ for $i \in \{1, \ldots, n\}$, and the nonlinear hybrid games [27] for nonlinear differential equations with inputs.

Regardless of the availability of communication, the formations and the maneuvers should be generated with the consideration of anticipated, static obstacles in the environments.

3.4 Routing Sequence Verification

Given a discretized map which can be represented as a directed graph under the assumption made on its maneuverability, the existence of a route for going from an initial node to a desired final node can be verified by using discrete reachability computation. This is important for performing tasks such as pursuit-evasion games [32,33] over graphs. Again, the validity of the claim depends upon the compiled ideal-typical behavior at the lowest-level.

4 Embedded Software Design

Given a system architecture and the components for composing the system, the high-level truth-claims rely heavily on the ideal-typical behaviors of the lower

levels. For this reason the *correct* execution of the control laws at the lowest-level must be enforced to avoid any catastrophic consequences from occurring. In this section we discuss the embedded software that is close to the environment under control.

Control laws are decomposed and translated into sets of *periodic* computational tasks in order to be implemented in embedded software. Along with functionality specifications, timing specifications are also given for the execution of the tasks in order to ensure that the implementation is consistent with the design. There are precedence relations not only defined among computational tasks but also specified for the logical ordering of event executions: sensing, control, and actuation.

In such an embedded system, processors, memory, inputs/outputs (I/O), sensors, and actuators are considered hardware components. We assume that the hardware components and some software components, such as the operating system (OS) and I/O drivers are given and are not parts of the design. Hence, the embedded software that is referred to here is made up of software components which are supported by the services provided by the OS. Therefore, the embedded software layer must work within the timing and scheduling framework of the underlying OS.

Another given in the system is that the basic design for an embedded controller includes a reliance on I/O and I/O rates in order to run any computational tasks. The system as a whole is then assumed to be driven by various local clocks and data processes that may occur at differing rates. Hardware components, such as sensors and actuators, exchange data with the embedded software via the I/O handled by the OS.

Operating systems, including real-time operating systems (RTOS), are designed under the same basic assumptions made in timesharing systems, where tasks are considered as unknown asynchronous activities activated at random instants. Except for the priority, no other parameters are provided to the system. The problem of scheduling a set of independent and preemptable periodic tasks has been solved under fixed and dynamic priority assignments [34], using the Rate-Monotonic algorithm and Earliest Deadline First algorithm. Given the precedence constraints and the Worst Case Execution Time (WCET), an efficient solution is provided by [35] for classes of the static scheduling problem. However, in order to implement the embedded software correctly, domain expertise is required for solving the scheduling problems on these heterogeneous computing platforms.

In the following, we discuss an embedded software example where the above configuration is utilized, and where the expected timing and ordering issues arise. We then continue and suggest adding a middle layer to the embedded software setup that eliminates these problems with, and dependencies on, any particular operating system. Consider a typical system configuration of a flight control system as depicted in Figure 4. This is motivated by our design experience on the development of embedded software for our helicopter-based unmanned aerial vehicles developed at the University of California, Berkeley. In the sys-

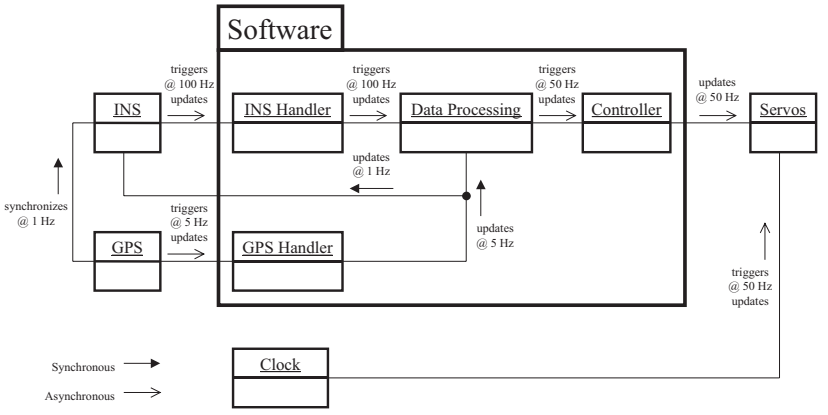


Fig. 4. Current configuration of system components is illustrated in Component Diagram. Communication between components via asynchronous and synchronous methods are shown. Notice that the software components are highlighted by grouping them together.

tem, there are hardware and software components running concurrently. Notice that the sensors, namely the Inertial Navigation System (INS) and the Global Positioning System (GPS), push the data asynchronously to separate software components. Because the INS and GPS each have their own internal clock, it is difficult to predict the times at which each of these two software processes will execute. Not only do these components have their own clocks, but they also must run at disparate rates as dictated by the design: the GPS at 5Hz, the INS at 100Hz, and the Servos at 50Hz. The rate of the Servos dictates that the control laws have to be applied at the same rate. The combination of the differing rates and the inherent variations of the clock edges creates random lags in the overall system. Furthermore, the execution of most of the software components relies on the timing of other software components. For example, the Data Processing task waits until it is triggered with data sent from the INS Handler, and the Controller waits until it is triggered by the Data Processing task. Due to the distinct clocks and rates of different software components, the interdependency of components allows for drastic uncertainty of the deadlines for each task. Consider that the Rate-Monotonic scheduling scheme is used for assigning priorities among tasks. Since the scheduling scheme does not take into account these dependencies between tasks, no guarantee can be made on the execution of the software components. Consequently, jitter may occur and it affects the regularity in time for computing the control results.

In order to mitigate these undesirable conditions with which the embedded software must work, we suggest reorganizing the system to include a middle layer of software, middleware, that establishes a time-triggered architecture. The middleware should provide a suitable level of abstraction for specifying functionality

and timing requirements of periodic tasks and ensure the correct execution of the tasks in the lower level provided that the scheduling problem is feasible to be solved. Next, in order to eradicate the problem of RTOS scheduling,

A middleware, named Giotto, is being developed by [14] in order to provide a time-triggered software abstraction for the execution of the embedded control software to meet hard real-time deadlines. Giotto, like other middlewares, is designed to provide the isolation between the embedded control software and the underlying computing platforms. In the example, the embedded control software refers to the software components, the Data Processing and the Controller. Hence, the scheduling of processing recourses and I/O handling is handled by the middleware. This modeling language is entirely time-triggered, and does not allow for any event triggered occurrences.

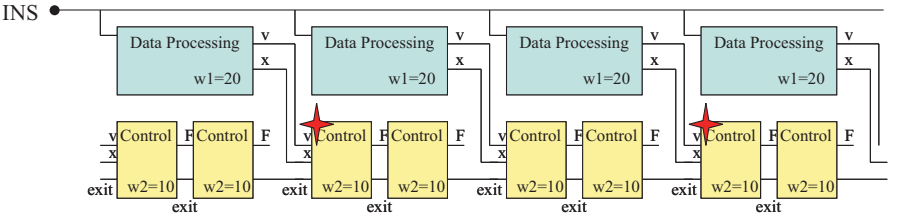


Fig. 5. The time line for the invocation of tasks in Giotto.

However, several changes are necessary to implement on the embedded system in order to allow for the usage of the time-triggered middleware. The most drastic change is that the entire system must now be comprised of synchronous elements in both the hardware and the software. For this effect, we propose two main changes in the hardware configuration. First, several local clocks in different components have to be collapsed into one global clock to avoid any phase differences. Second, due to the fact that the sensors are the basis of the current event driven paradigm, the sensors have to be reconfigured into a pull configuration so that the sensors would be synchronously queried for data by the software components. These two changes would translate the hardware system from an event-driven domain into a time-triggered one.

The software components must be slightly modified to fit into the Giotto language semantics. Since the embedded software is originally made up of periodic tasks that run concurrently and have desired end times associated with them, fitting the system into the framework of Giotto is relatively intuitive. The only complication with this change in the software specifications is the introduction of an increased, though bounded, delay. This delay is a function of the time-triggered paradigm. Instead of acting on data immediately, as in the event-driven scenario, a time-triggered architecture acts on data at a set time and, in this way, introduces a stale data phenomenon. In order to curtail the added

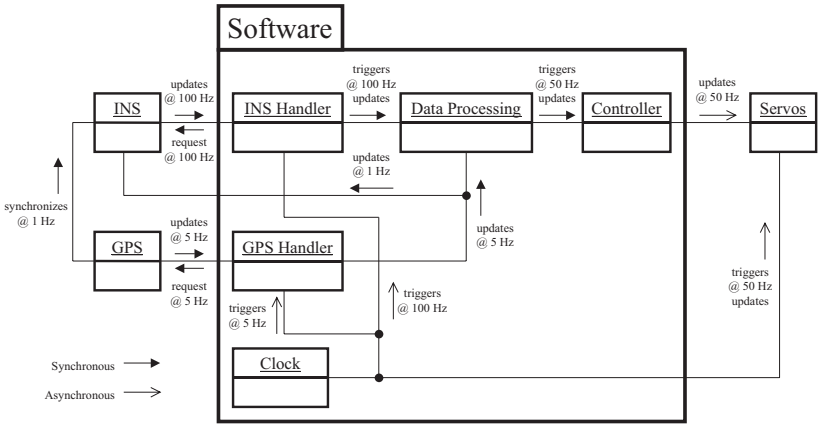


Fig. 6. Proposed configuration of system components is illustrated in Component Diagram. Synchronous communication methods are used between components, except the clocks, for communication.

delay in our modified example, the deadline times for the control computation task are shortened. Since shortening the execution time allowed for a block necessarily bounds the delay to the length of the block, the designer can regulate the added delay in this manner. Figure 5 displays the embedded software design under Giotto of the INS handler and the controller blocks. In order to bound the delay in data usage without computing the control more than is necessary, the control block deadline time is shortened but the control is only computed at the rate necessary (on the blocks marked with a star). In this manner, the software can be restructured to make use of the benefits of a time triggered middle layer with a minimum of added bounded delay.

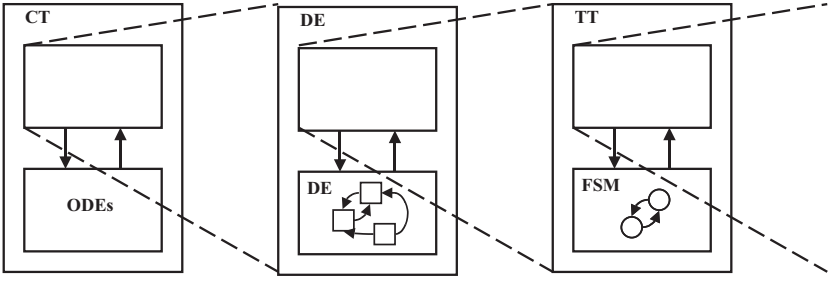


Fig. 7. Proposed embedded system hierarchy for a single vehicle

The resulting *synchronous* hybrid system as shown in Figure 6does have a predictable behavior and therefore is able to guarantee safety critical control performance. The drawback to this approach is that there is an increase in delay over the original unpredictable system. However, as long as the timing deadlines set are within the tolerances of the control design, the performance of the overall system can be guaranteed. Since Giotto also supports multi-modal behavior, the hierarchy of the system for a single vehicle can be further constructed and the idea of the design is depicted in Figure 7. The hardware and software components below the level of Giotto, labeled as TT domain, are represented by a single component in DE domain to model the asynchronous behaviors of the component.

5 Conclusion

In this paper, we have considered the design problem of embedded software for multi-vehicle multi-modal systems. A hierarchical architecture which promotes verification is presented for the construction of embedded systems. Hybrid control design techniques are an important design tool for rapid prototyping of system components for real-time embedded systems. Motivated by our design experience on the development of embedded software for our helicopter-based unmanned aerial vehicles which are composed of heterogeneous components, we believe that at the level closest to the environment under control, the embedded software needs to be time-triggered for guaranteed safety; at the higher levels, we advocate a asynchronous hybrid controller design. Current work focuses on the realization of the system for hardware-in-the-loop (HIL) simulation. This is especially challenging since it will demand the development of formal methodologies for the integration of multiple MOCs and for the analysis of the resultant hybrid system.

Acknowledgments. The authors would like to thank Benjamin Horowitz, Jie Liu, Xiaojun Liu, Hyunchul Shim, and Ron Tal for stimulating discussions and valuable comments. This work is supported by the DARPA SEC grant, F33615-98-C-3614.

References

1. P. Varaiya. Smart Cars on Smart Roads: Problems of Control, *IEEE Transactions on Automatic Control*, 38(2):195-207, February 1993.
2. C. Tomlin, G. Pappas, J. Lygeros, D. Godbole, and S. Sastry. Hybrid Control Models of Next Generation Air Traffic Management, *Hybrid Systems IV*, volume 1273 of Lecture Notes in Computer Science, pages 378-404, Springer Verlag, Berlin, Germany, 1997.
3. P. Varaiya. A Question About Hierarchical Systems, *System Theory: Modeling, Analysis and Control*, T. Djaferis and I. Schick (eds), Kluwer, 2000.
4. P. Caines, and Y. J. Wei, Hierarchical Hybrid Control Systems : A lattice theoretic formulation, *IEEE Transactions on Automatic Control*, 43(4), 1998.

5. G. J. Pappas, G. Lafferriere, and S. Sastry. Hierarchically Consistent Control Systems, *IEEE Transactions on Automatic Control*, 45(6):1144-1160, June 2000.
6. E. A. Lee and A. Sangiovanni-Vincentelli. A Framework for Comparing Models of Computation, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(12):1217-1229, December 1998.
7. E. A. Lee. Overview of the Ptolemy Project, *Technical Memorandum UCB/ERL*, M01/11, University of California, Berkeley, March 2001.
8. G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation, *Science of Computer Programming*, 19(2):87-152, 1992.
9. A. Benveniste and P. Le Guernic. Hybrid Dynamical Systems Theory and the SIGNAL Language, *IEEE Transactions on Automatic Control* 35(5):525-546, May 1990.
10. P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE: A Declarative Language for Programming Synchronous Systems, *Conference Record of the 14th Annual ACM Symp. on Principles of Programming Languages*, Munich, Germany, January 1987.
11. F. Maraninchi. The Argos Language: Graphical Representation of Automata and Description of Reactive Systems, in *Proceedings of the IEEE Workshop on Visual Languages*, Kobe, Japan, October 1991.
12. J. Liu, X. Liu, T. J. Koo, B. Sinopoli, S. S. Sastry, and E. A. Lee. Hierarchical Hybrid System Simulation. In *Proceedings of the 38th Conference on Decision and Control*, Phoenix, Arizona. December 1999.
13. H. Kopetz. The Time-Triggered Architecture, in *Proceedings of the First International Symposium on Object-Oriented Real-Time Distributed Computing*, Kyoto, Japan, April 1998.
14. T.A. Henzinger, B. Horowitz, and C. M. Kirsch. Embedded Control Systems Development with Giotto, in *Proceedings of LCTES 2001*, Snowbird, Utach, June 2001.
15. A. Isidori. Nonlinear Control Systems, Springer-Verlag, New York, 1995.
16. S. S. Sastry. Nonlinear Systems: Analysis, Stability, and Control, Springer-Verlag, New York, 1999.
17. J. Lygeros, C. Tomlin, and S. Sastry. Controllers for Reachability Specifications for Hybrid Systems, Automatica, Volume 35, Number 3, March 1999.
18. I. Mitchell, and C. Tomlin. Level Set Methods for Computation in Hybrid Systems, *Hybrid Systems: Computation and Control*, Lecture Notes in Computer Science, Springer Verlag, 2000.
19. S. Osher, and J. A. Sethian. Fronts Propagating with Curvature-dependent Speed: Algorithms based on Hamilton-Jacobi Formulations, *J. Computat. Phys.*, vol. 79, pages 12-49, 1988.
20. T. J. Koo, G. Pappas, and S.Sastry. Mode Switching Synthesis for Reachability Specifications, *Hybrid Systems: Computation and Control*, Lecture Notes in Computer Science, Springer Verlag, 2001.
21. G. Lafferriere, G.J. Pappas, S. Yovine. Reachability Computation for Linear Hybrid Systems, In Proceedings of the 14th IFAC World Congress, volume E, pages 7-12, Beijing, 1999.
22. E. Asarin, O. Bournez, T. Dang, O. Maler, and A. Pnueli, Effective Synthesis of Switching Controllers for Linear Systems, Proceedings of the IEEE, 88(2):1011-1025.

23. A.B. Kurzhanski, P.Varaiya, Ellipsoidal Techniques for Reachability Analysis, Hybrid Systems : Computation and Control, Lecture Notes in Computer Science, 2000.
24. A. Chutinan, B.H. Krogh, Verification of polyhedral-invariant hybrid systems using polygonal flow pipe approximations, Hybrid Systems : Computation and Control, Lecture Notes in Computer Science, 1999.
25. P. Tabuada, G. Pappas, and P. Lima. Feasible Formations of Multi-Agent Systems, in *Proceedings of American Control Conference*, pages 56-61, Arlington, Virginia, June, 2001.
26. A. Pant, P. Seiler, T. J. Koo, and J. K. Hedrick. Mesh Stability of Unmanned Aerial Vehicle Clusters, in *Proceedings of American Control Conference*, pages 62-68, Arlington, Virginia, June, 2001.
27. C. Tomlin, J. Lygeros, and S. Sastry. A Game Theoretic Approach to Controller Design for Hybrid Systems, in Proceedings of the IEEE, pages 949-970, Volume 88, Number 7, July 2000.
28. O. Maler, A. Puneli, and J. Sifakis. On the Synthesis of Discrete Controllers for Timed Systems, in *STAC 95: Theoretical Aspects of Computer Science*, E.W. Mayr and C. Puech (eds). Munich, Germany: Springer-Verlag, 1995, vol. 900, Lectures Notes in Computer Science, pages 229-242.
29. E. Asarin, O. Maler, and A. Puneli. Symbolic Controller Synthesis for Discrete and Timed Systems, in *Proceedings of Hybrid Systems II*, P. Antsaklis, W. Kohn, A. Nerode, and S. Sastry (eds). Berlin, Germany: Springer-Verlag, 1995, vol. 999, Lectures Notes in Computer Science.
30. M. Heymann, F. Lin, and G. Meyer. Control Synthesis for a Class of Hybrid Systems subject to Configuration-Based Safety Constraints, in *Hybrid and Real Time Systems*, O. Maler (ed). Berlin, Germany: Springer-Verlag, 1997, vol. 1201, Lectures Notes in Computer Science, pages 376-391.
31. H. Wong-Toi. The Synthesis of Controllers for Linear Hybrid Automata, in *Proceedings of IEEE Conference on Control and Decision*, San Diego, CA, 1997.
32. T. D. Parsons. Pursuit-Evasion in a Graph, *Theory and Application of Graphs*, pages 426-441, Y. Alani and D. R. Lick (eds), Springer-Verlag, 1976.
33. J. P. Hespanha, H. J. Kim, and S. Sastry. Multiple-Agent Probabilistic Pursuit-Evasion Games, in *Proceedings of IEEE Conference on Decision and Control*, pages 2432-2437, Phoenix, Arizona, December 1999.
34. G. C. Buttazzo. *Hrad Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*, Kluwer, 1997.
35. F. Balarin, L. Lavagno, P. Murthy, A. Sangiovanni-Vincentelli. Scheduling for Embedded Real-Time Systems, *IEEE Design and Test of Computers*, pages 71-82, January 1998.

Adaptive and Reflective Middleware for Distributed Real-Time and Embedded Systems

Douglas C. Schmidt

Electrical & Computer Engineering Dept.
University of California, Irvine
Irvine, CA 92697-2625, USA
schmidt@uci.edu

Abstract. Software has become strategic to developing effective distributed real-time and embedded (DRE) systems. Next-generation DRE systems, such as total ship computing environments, coordinated unmanned air vehicle systems, and national missile defense, will use many geographically dispersed sensors, provide on-demand situational awareness and actuation capabilities for human operators, and respond flexibly to unanticipated run-time conditions. These DRE systems will also increasingly run unobtrusively and autonomously, shielding operators from unnecessary details, while communicating and responding to mission-critical information at an accelerated operational tempo. In such environments, it's hard to predict system configurations or workloads in advance. This paper describes the need for adaptive and reflective middleware systems (ARMS) to bridge the gap between application programs and the underlying operating systems and network protocol stacks in order to provide reusable services whose qualities are critical to DRE systems. ARMS middleware can adapt in response to dynamically changing conditions for the purpose of utilizing the available computer and network infrastructure to the highest degree possible in support of mission needs.

Motivation

New and planned distributed real-time and embedded (DRE) systems are inherently network-centric "systems of systems." DRE systems have historically been developed via *multiple technology bases*, where each system brings its own networks, computers, displays, software, and people to maintain and operate it. Unfortunately, not only are these "stove-pipe" architectures proprietary, but they tightly couple many functional and non-functional DRE system aspects, which impedes their

1. *Assurability,* which is needed to guarantee efficient, predictable, scalable, and dependable quality of service (QoS) from sensors to shooters
2. *Adaptability*, which is needed to (re)configure DRE systems dynamically to support varying workloads or missions over their lifecycles and
3. *Affordability*, which is needed to reduce initial non-recurring DRE system acquisition costs and recurring upgrade and evolution costs.

The affordability of certain types of systems, such as logistics and mission planning, can often be enhanced by using commercial-off-the-shelf (COTS) technologies. However, today's efforts aimed at integrating COTS into mission-critical DRE systems have largely failed to support affordability *and* assurability and adaptability

T.A. Henzinger and C.M. Kirsch (Eds.): EMSOFT 2001, LNCS 2211, pp. 361-372, 2001.
© Springer-Verlag Berlin Heidelberg 2001

effectively since they focus mainly on initial non-recurring acquisition costs and do not reduce recurring software lifecycle costs, such as "COTS refresh" and subsetting military systems for foreign military sales. Likewise, many COTS products lack support for controlling key QoS properties, such as predictable latency, jitter, and throughput; scalability; dependability; and security. The inability to control these QoS properties with sufficient confidence compromises DRE system adaptability and assurability, *e.g.,* minor perturbations in conventional COTS products can cause failures that lead to loss of life and property.

Historically, conventional COTS software has been particularly unsuitable for use in mission-critical DRE systems due to its either being:

1. Flexible and standard, but incapable of guaranteeing stringent QoS demands, which restricts assurability or
2. Partially QoS-enabled, but inflexible and non-standard, which restricts adaptability and affordability.

As a result, the rapid progress in COTS software for mainstream business information technology (IT) has not yet become as broadly applicable for mission-critical DRE systems. Until this problem is resolved effectively, DRE system integrators and warfighters will be unable to take advantage of future advances in COTS software in a dependable, timely, and cost effective manner. Thus, developing the new generation of assurable, adaptable, and affordable COTS software technologies is an important R&D goal.

Key Technical Challenges and Solutions

Some of the most challenging IT requirements for new and planned DRE systems can be characterized as follows:

- Multiple QoS properties must be satisfied in real-time
- Different levels of service are appropriate under different configurations, environmental conditions, and costs
- The levels of service in one dimension must be coordinated with and/or traded off against the levels of service in other dimensions to meet mission needs and
- The need for autonomous and time-critical application behavior necessitates a flexible distributed system substrate that can adapt robustly to dynamic changes in mission requirements and environmental conditions.

Standards-based COTS software available today cannot meet all of these requirements simultaneously for the reasons outlined in Section *Motivation*. However, contemporary economic and organizational constraints—along with increasingly complex requirements and competitive pressures—are also making it infeasible to built complex DRE system software entirely from scratch. Thus, there is a pressing need to develop, validate, and ultimately standardize a new generation of *adaptive and reflective middleware systems* (ARMS) technologies that can support stringent DRE system functionality and QoS requirements.

Middleware [Sch01a] is reusable service/protocol component and framework software that functionally bridges the gap between

1. the end-to-end functional requirements and mission doctrine of applications and
2. the lower-level underlying operating systems and network protocol stacks.

Middleware therefore provides capabilities whose quality and QoS are critical to DRE systems.

Adaptive middleware [Loy01] is software whose functional and QoS-related properties can be modified either

- *Statically*, *e.g.,* to reduce footprint, leverage capabilities that exist in specific platforms, enable functional subsetting, and minimize hardware and software infrastructure dependencies or
- *Dynamically*, *e.g.,* to optimize system responses to changing environments or requirements, such as changing component interconnections, power-levels, CPU/network bandwidth, latency/jitter, and dependability needs.

In DRE systems, adaptive middleware must make these modifications dependably, *i.e.,* while meeting stringent end-to-end QoS requirements.

Reflective middleware [Bla99] goes a step further to permit automated examination of the capabilities it offers, and to permit automated adjustment to optimize those capabilities. Thus, reflective middleware supports more advanced adaptive behavior, *i.e.,* the necessary adaptations can be performed autonomously based on conditions within the system, in the system's environment, or in DRE system doctrine defined by operators and administrators.

The Structure and Functionality of Middleware

Networking protocol stacks can be decomposed into multiple layers, such as the physical, data-link, network, transport, session, presentation, and application layers. Similarly, middleware can be decomposed into multiple layers, such as those shown in Figure 1.

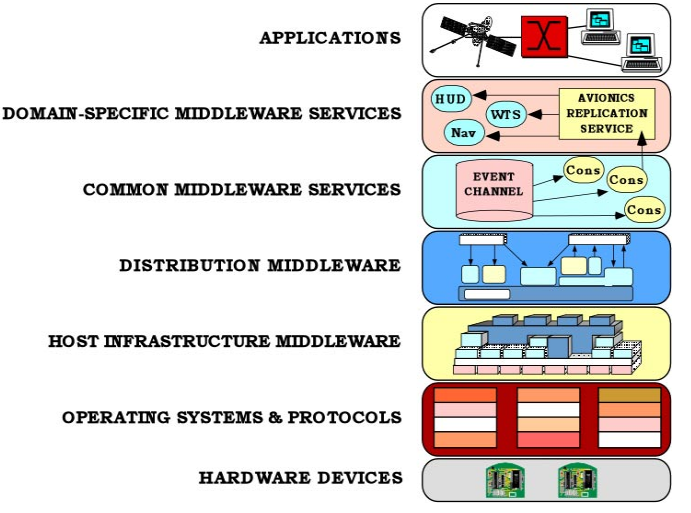


Fig. 1. Layers of Middleware and Their Surrounding Context

Below, we describe each of these middleware layers and outline some of the COTS technologies in each layer that are suitable (or are becoming suitable) to meet the stringent QoS demands of DRE systems.

Host infrastructure middleware encapsulates and enhances native OS communication and concurrency mechanisms to create portable and reusable network programming components, such as reactors, acceptor-connectors, monitor objects, active objects, and component configurators [Sch00b]. These components abstract away the accidental incompatibilities of individual operating systems, and help eliminate many tedious, error-prone, and non-portable aspects of developing and maintaining networked applications via low-level OS programming API, such as Sockets or POSIX Pthreads. Examples of COTS host infrastructure middleware that are relevant for DRE systems include:

- *The ADAPTIVE Communication Environment* (ACE) [Sch01], which is a highly portable and efficient toolkit written in C++ that encapsulates native operating system (OS) network programming capabilities, such as connection establishment, event demultiplexing, interprocess communication, (de)marshaling, static and dynamic configuration of application components, concurrency, and synchronization. ACE has been used in a wide range of commercial and military DRE systems, including hot rolling mill control software, surface mount technology for "pick and place" systems, missile control, avionics mission computing, software defined radios, and radar systems.
- *Real-time Java Virtual Machines* (RT-JVMs), which implement the Real-time Specification for Java (RTSJ) [Bol00]. The RTSJ is a set of extensions to Java that provide a largely platform-independent way of executing code by encapsulating the differences between real-time operating systems and CPU architectures. The key features of RTSJ include scoped and immortal memory, real-time threads with enhanced scheduling support, asynchronous event handlers, and asynchronous transfer of control between threads. Although RT-JVMs based on the RTSJ are in their infancy, they have generated tremendous interest in the R&D and integrator communities due to their potential for reducing software development and evolution costs.

Distribution middleware defines higher-level distributed programming models whose reusable APIs and mechanisms automate and extend the native OS network programming capabilities encapsulated by host infrastructure middleware. Distribution middleware enables developers to program distributed applications much like stand-alone applications, *i.e.*, by invoking operations on target objects without hard-coding dependencies on their location, programming language, OS platform, communication protocols and interconnects, and hardware characteristics.

At the heart of distribution middleware are QoS-enabled object request brokers, such as the Object Management Group's (OMG) *Common Object Request Broker Architecture* (CORBA) [Omg00]. CORBA is distribution middleware that allows objects to interoperate across networks regardless of the language in which they were written or the OS platform on which they are deployed. In 1998 the OMG adopted the Real-time CORBA (RT-CORBA) specification [Sch00a], which extends CORBA with features that allow DRE applications to reserve and manage CPU, memory, and networking resources. RT-CORBA implementations have been used in dozens of DRE

systems, including telecom network management and call processing, online trading services, avionics mission computing, submarine DRE systems, signal intelligence and C4ISR systems, software defined radios, and radar systems.

Common middleware services augment distribution middleware by defining higher-level domain-independent components that allow application developers to concentrate on programming application logic, without the need to write the "plumbing" code needed to develop distributed applications by using lower level middleware features directly. Whereas distribution middleware focuses largely on managing end-system resources in support of an object-oriented distributed programming model, common middleware services focus on allocating, scheduling, and coordinating various end-to-end resources throughout a distributed system using a component programming and scripting model. Developers can reuse these services to manage global resources and perform recurring distribution tasks that would otherwise be implemented in an *ad hoc* manner by each application or integrator.

Examples of common middleware services include the OMG's CORBAServices [Omg98b] and the CORBA Component Model (CCM) [Omg99], which provide domain-independent interfaces and distribution capabilities that can be used by many distributed applications. The OMG CORBAServices and CCM specifications define a wide variety of these services, including event notification, logging, multimedia streaming, persistence, security, global time, real-time scheduling, fault tolerance, concurrency control, and transactions. Not all of these services are sufficiently refined today to be usable off-the-shelf for DRE systems. The form and content of these common middleware services will continue to mature and evolve, however, to meet the expanding requirements of DRE.

Domain-specific middleware services are tailored to the requirements of particular DRE system domains, such as avionics mission computing, radar processing, weapons targeting, or command and decision systems. Unlike the previous three middleware layers—which provide broadly reusable "horizontal" mechanisms and services—domain-specific middleware services are targeted at vertical markets. From a COTS perspective, domain-specific services are the least mature of the middleware layers today. This immaturity is due in part to the historical lack of distribution middleware and common middleware service *standards*, which are needed to provide a stable base upon which to create domain-specific middleware services. Since they embody knowledge of a domain, however, domain-specific middleware services have the most potential to increase the quality and decrease the cycle-time and effort that integrators require to develop particular classes of DRE systems.

A mature example of domain-specific middleware services is the Boeing Bold Stroke architecture [Sha98]. Bold Stroke uses COTS hardware, operating systems, and middleware to produce an open architecture for mission computing avionics capabilities, such as navigation, heads-up display management, weapons targeting and release, and airframe sensor processing. The domain-specific middleware services in Bold Stroke are layered upon COTS processors (PowerPC), network interconnects (VME), operating systems (VxWorks), infrastructure middleware (ACE), distribution middleware (Real-time CORBA), and common middleware services (the CORBA Event Service).

Recent Progress

Significant progress has occurred during the last five years in DRE middleware research, development, and deployment, stemming in large part from the following trends:

- ***Years of research, iteration, refinement, and successful use*** – The use of middleware and DOC middleware is not new [Sch86]. Middleware concepts emerged alongside experimentation with the early Internet (and even its predecessor ARPAnet), and DOC middleware systems have been continuously operational since the mid 1980's. Over that period of time, the ideas, designs, and most importantly, the software that incarnates those ideas have had a chance to be tried and refined (for those that worked), and discarded or redirected (for those that didn't). This iterative technology development process takes a good deal of time to get right and be accepted by user communities, and a good deal of patience to stay the course. When this process is successful, it often results in *standards* that codify the boundaries, and *patterns and frameworks* that reify the knowledge of how to apply these technologies, as described in the following bullets.
- ***The maturation of standards*** – Over the past decade, middleware standards have been established and have matured considerably with respect to DRE requirements. For instance, the OMG has adopted the following specifications in the past three years:
  - *Minimum CORBA*, which removes non-essential features from the full OMG CORBA specification to reduce footprint so that CORBA can be used in memory-constrained embedded systems.
  - *Real-time CORBA*, which includes features that allow applications to reserve and manage network, CPU, and memory resources predictably end-to-end.
  - *CORBA Messaging*, which exports additional QoS policies, such as timeouts, request priorities, and queueing disciplines, to applications.
  - *Fault-tolerant CORBA*, which uses entity redundancy of objects to support replication, fault detection, and failure recovery.

  Robust implementations of these CORBA capabilities and services are now available from multiple vendors. Moreover, emerging standards such as Dynamic Scheduling Real-Time CORBA, the Real-Time Specification for Java, and the Distributed Real-Time Specification for Java are extending the scope of open standards for a wider range of DRE applications.
- ***The dissemination of patterns and frameworks*** – A substantial amount of R&D effort during the past decade has also focused on the following means of promoting the development and reuse of high quality middleware technology:
  - *Patterns* codify design expertise that provides time-proven solutions to commonly occurring software problems that arise in particular contexts [Gam95]. Patterns can simplify the design, construction, and performance tuning of DRE applications by codifying the accumulated expertise of developers who have successfully confronted similar problems before. Patterns also elevate the level of discourse in describing software development activities to focus on strategic architecture and design issues, rather than just the tactical programming and representation details.

- *Frameworks* are concrete realizations of groups of related patterns [John97]. Well-designed frameworks reify patterns in terms of functionality provided by the middleware itself, as well as functionality provided by an application. Frameworks also integrate various approaches to problems where there are no *a priori*, context-independent, optimal solutions. Middleware frameworks can include strategized selection and optimization patterns so that multiple independently-developed capabilities can be integrated and configured automatically to meet the functional and QoS requirements of particular DRE applications.

Historically, the knowledge required to develop predictable, scalable, efficient, and dependable mission-critical DRE systems has existed largely in programming folklore, the heads of experienced researchers and developers, or buried deep within millions of lines of complex source code. Moreover, documenting complex systems with today's popular software modeling methods and tools, such as the Unified Modeling Language (UML), only capture *how* a system is designed, but do not necessarily articulate *why* a system is designed in a particular way. This situation has several drawbacks:

- Re-discovering the rationale for complex DRE system design decisions from source code is expensive, time-consuming, and error-prone since it's hard to separate essential QoS-related knowledge from implementation details.
- If the insights and design rationale of expert system architects are not documented they will be lost over time, and thus cannot help guide future DRE system evolution.
- Without proper guidance, developers of mission-critical DRE software face the Herculean task of engineering and assuring the QoS of complex DRE systems from the ground up, rather than by leveraging proven solutions.

Middleware patterns and frameworks are therefore essential to help capture DRE system design expertise in a more readily accessible and reusable format.

Much of the pioneering R&D on middleware patterns and frameworks was conducted in the DARPA ITO Quorum program [DARPA99]. This program focused heavily on CORBA open systems middleware and yielded many results that transitioned into standardized service definitions and implementations for the Real-time [Sch98] and Fault-tolerant [Omg98a] CORBA specification and productization efforts. Quorum is an example of how a focused government R&D effort can leverage its results by exporting them into, and combining them with, other on-going public and private activities that also used a common open middleware substrate. Prior to the viability of standards-based COTS middleware platforms, these same R&D results would have been buried within custom or proprietary systems, serving only as an existence proof, rather than as the basis for realigning the R&D and integrator communities.

Looking Ahead

Due to advances in COTS technologies outlined earlier, host infrastructure middleware and distribution middleware have now been successfully demonstrated and deployed in a number of mission-critical DRE systems, such as avionics mission computing, software defined radios, and submarine information systems. Since COTS middleware technology has not yet matured to cover the realm of large-scale, dynamically changing systems, however, DRE middleware has been applied to relatively small-scale and statically configured embedded systems. To satisfy the highly application- and mission-specific QoS requirements in network-centric "system of system" environments, DRE middleware—particularly common middleware services and domain-specific services—must be enhanced to support the management of individual and aggregate resources used by multiple system components at multiple system levels in order to:

- *Manage communication bandwidth*, *e.g.,* network level resource capability and status information services, scalability to 10^2 subnets and 10^3 nodes, dynamic connections with reserved bandwidth, aggregate policy-controlled bandwidth reservation and sharing, incorporation of non-network resource status information, aggregate dynamic network resource management strategies, and managed bandwidth to enhance real-time predictability.
- *Manage distributed real-time scheduling and allocation of DRE system artifacts (such as CPUs, networks, UAVs, missiles, radar, illuminators, etc)*, *e.g.,* fast and predictable queueing time properties, timeliness assurances for end-to-end activities based on priority/deadlines, admission controlled request insertion based on QoS parameters and global resource usage metrics, and predictable behavior over WANs using bandwidth reservations.
- *Manage distributed system dependability*, *e.g.,* group communication-based replica management, dependability manager maintaining aggregate levels of object replication, run-time switching among dependability strategies, policy-based selection of replication options, and understanding and tolerating timing faults in conjunction with real-time behavior.
- *Manage distributed security*, *e.g.,* object-level access control, layered access control for adaptive middleware, dynamically variable access control policies, and effective real-time, dependability, and security interactions.

Ironically, there is currently little or no scientific underpinning for QoS-enabled resource management, despite the demand for it in most distributed systems. Today's system designers and mission planners develop concrete plans for creating global, end-to-end functionality. These plans contain high-level abstractions and doctrine associated with resource management algorithms, relationships between these, and operations upon these. There are few techniques and tools, however that enable *users*, *i.e.,* commanders, administrators, and operators, *developers*, *i.e.,* systems engineers and application designers and/or *applications* to express such plans systematically, reason about and refine them, and have these plans enforced automatically to manage resources at multiple levels in network-centric DRE systems.

Systems today are built in a highly static manner, with allocation of processing tasks to resources assigned at design time. For systems that never change, this is an adequate approach. Large and complex military DRE combat systems change and evolve over their lifetime, however, in response to changing missions and operational environments. Allocation decisions made during initial design often become obsolete over time, necessitating expensive and time-consuming redesign. If the system's requisite end-to-end functionality becomes unavailable due to mission and environment changes, there are no standard tools or techniques to diagnose configuration or run-time errors automatically. Instead, designers and operators write down their plans on paper and perform such reasoning, refinement, configuration generation, and diagnosis manually. This *ad hoc* process is clearly inadequate to manage the accelerated operational tempo characteristic of network-centric DRE combat systems.

To address these challenges, the R&D community needs to discover and set the technical approach that can significantly improve the effective utilization of networks and endsystems that DRE systems depend upon by creating middleware technologies and tools that can automatically allocate, schedule, control, and optimize customizable—yet standards-compliant and verifiably correct—software-intensive systems. To promote a *common technology base*, the interfaces and (where appropriate) the protocols used by the middleware should be based on established or emerging industry or military standards that are relevant for DRE systems. However, the protocol and service *implementations* should be customizable—statically and dynamically—for specific DRE system requirements.

To achieve these goals, middleware technologies and tools need to be based upon some type of layered architecture, such as the one shown in Figure 2 [Loy01]. This architecture decouples DRE middleware and applications along the following two dimensions:

- *Functional paths,* which are flows of information between client and remote server applications. In distributed systems, middleware ensures that this information is exchanged efficiently, predictably, scalably, dependably, and securely between remote peers. The information itself is largely application-specific and determined by the functionality being provided (hence the term "functional path").
- *QoS paths*, which are responsible for determining how well the functional interactions behave end-to-end with respect to key DRE system QoS properties, such as
  1. How and when resources are committed to client/server interactions at multiple levels of distributed systems
  2. The proper application and system behavior if available resources do not satisfy the expected resources and
  3. The failure detection and recovery strategies necessary to meet end-to-end dependability requirements.

In next-generation DRE systems, the middleware—rather than operating systems or networks in isolation—will be responsible for separating non-functional DRE system QoS properties from the functional application properties. Middleware will also coordinate the QoS of various DRE system and application resources end-to-end. The

architecture in Figure 2 enables these properties and resources to change independently, *e.g.,* over different distributed system configurations for the same applications.

The architecture in Figure 2 is based on the expectation that non-functional QoS paths will be developed, configured, monitored, managed, and controlled by a different set of specialists (such as systems engineers, administrators, operators, and perhaps someday automated agents) and tools than those customarily responsible for programming functional paths in DRE systems. The middleware is therefore responsible for collecting, organizing, and disseminating QoS-related meta-information needed to

1. Monitor and manage how well the functional interactions occur at multiple levels of DRE systems and
2. Enable the adaptive and reflective decision-making needed to support non-functional QoS properties robustly in the face of rapidly changing mission requirements and environmental conditions.

These middleware capabilities are crucial to ensure that the aggregate behavior of complex network-centric DRE systems is dependable, despite local failures, transient overloads, and dynamic functional or QoS reconfigurations.

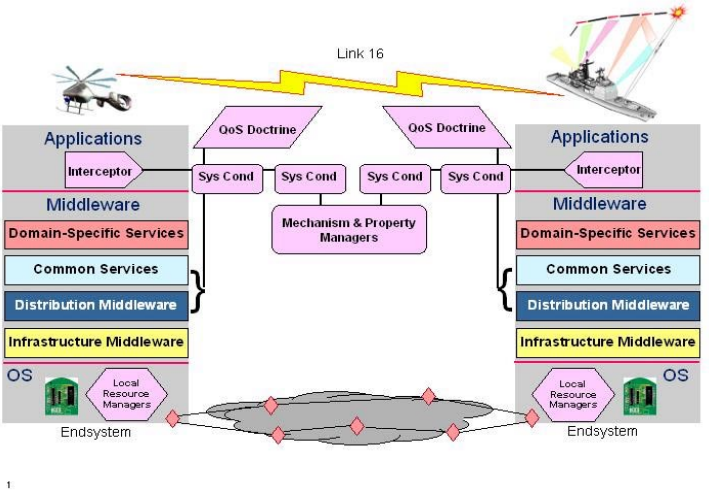


Fig. 2. Decoupling Functional and QoS Paths

To simultaneously enhance assurability, adaptability, *and* affordability, the middleware techniques and tools developed in future R&D programs increasingly need to be application-independent, yet customizable within the interfaces specified by a range of open standards, such as

- The OMG Real-time CORBA specifications and The Open Group's QoS Forum
- The Java Expert Group Real-time Specification for Java (RTSJ) and the Distributed RTSJ
- The DMSO/IEEE High-level Architecture Run-time Infrastructure (HLA/RTI) and
- The IEEE Real-time Portable Operating System (POSIX) specification.

Concluding Remarks

Advances in wireless networks and COTS hardware technologies are enabling the lower level aspects of network-centric DRE systems. The emerging middleware software technologies and tools are likewise enabling the higher level distributed real-time and embedded (DRE) aspects of network-centric DRE systems, making them tangible and affordable by controlling the hardware, network, and endsystem mechanisms that affect mission, system, and application QoS tradeoffs.

The economic benefits of middleware stem from moving standardization up several levels of abstraction by maturing DRE software technology artifacts, such as middleware frameworks, protocol/service components, and patterns, so that they are readily available for COTS acquisition and customization. This middleware focus is helping to lower the total ownership costs of DRE systems by leveraging common technology bases so that complex and DRE functionality need not be re-invented repeatedly or reworked from proprietary "stove-pipe" architectures that are inflexible and expensive to evolve and optimize.

Adaptive and reflective middleware systems (ARMS) are a key emerging theme that will help to simplify the development, optimization, validation, and integration of middleware in DRE systems. In particular, ARMS will allow researchers and system integrators to develop and evolve complex DRE systems assurably, adaptively, *and* affordably by:

- Standardizing COTS at the middleware level, rather than just at lower hardware/networks/OS levels and
- Devising optimizers, meta-programming techniques, and multi-level distributed dynamic resource management protocols and services for ARMS that will enable DRE systems to customize standard COTS interfaces, without the penalties incurred by today's conventional COTS software product implementations.

Many DRE systems require these middleware capabilities. Additional information on DRE middleware is available at `www.ece.uci.edu/~schmidt`.

References

[Bla99] Blair, G.S., F. Costa, G. Coulson, H. Duran, et al, "The Design of a Resource-Aware Reflective Middleware Architecture", *Proceedings of the 2nd International Conference on Meta-Level Architectures and Reflection*, St.-Malo, France, Springer-Verlag, LNCS, Vol. 1616, 1999.

[Bol00] Bollella, G., Gosling, J. "The Real-Time Specification for Java," *Computer*, June 2000.

[DARPA99] DARPA, *The Quorum Program*, http://www.darpa.mil/ito/research/ quorum/index.html, 1999.

[Gam95] Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[John97] Johnson R., "Frameworks = Patterns + Components", *Communications of the ACM*, Volume 40, Number 10, October, 1997.

[Loy01] Loyall JL, Gossett JM, Gill CD, Schantz RE, Zinky JA, Pal P, Shapiro R, Rodrigues C, Atighetchi M, Karr D. "Comparing and Contrasting Adaptive Middleware Support in Wide-Area and Embedded Distributed Object Applications". *Proceedings of the 21st IEEE International Conference on Distributed Computing Systems (ICDCS-21)*, April 16-19, 2001, Phoenix, Arizona.

[Omg98a] Object Management Group, "Fault Tolerance CORBA Using Entity Redundancy RFP", OMG Docu-ment orbos/98-04-01 edition, 1998.

[Omg98b] Object Management Group, "CORBAServcies: Common Object Service Specification," OMG Technical Document formal/98-12-31.

[Omg99] Object Management Group, "CORBA Compon-ent Model Joint Revised Submission," OMG Document orbos/99-07-01.

[Omg00] Object Management Group, "The Common Object Request Broker: Architecture and Specification Revision 2.4, OMG Technical Document formal/00-11-07", October 2000.

[Sch86] Schantz, R., Thomas R., Bono G., "The Architecture of the Cronus Distributed Operating System", *Pro-ceedings of the 6th IEEE International Conference on Distributed Computing Systems (ICDCS-6)*, Cambridge, Massachusetts, May 1986.

[Sch98] Schmidt D., Levine D., Mungee S. "The Design and Performance of the TAO Real-Time Object Request Broker", *Computer Communications Special Issue on Building Quality of Service into Distributed Systems,* 21(4), 1998.

[Sch00a] Schmidt D., Kuhns F., "An Overview of the Real-time CORBA Specification," *IEEE Computer Magazine*, June, 2000.

[Sch00b] Schmidt D., Stal M., Rohnert H., Buschmann F., *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*, Wiley and Sons, 2000.

[Sch01] Schmidt D., Huston S., *C++ Network Programming: Resolving Complexity with ACE and Patterns*, Addison-Wesley, Reading, MA, 2001.

[Sch01a] Schantz R., Schmidt D., "Middleware for Distributed Systems: Evolving the Common Structure for Network-centric Applications," Encyclopedia of Software Engineering, Wiley & Sons, 2001.

[Sha98] Sharp, David C., "Reducing Avionics Software Cost Through Component Based Product Line Development", *Software Technology Conference*, April 1998.

Modeling Real-Time Systems – Challenges and Work Directions

Joseph Sifakis

Verimag, 2 rue Vignate, 38610 Gières, France Joseph.Sifakis@imag.fr

1 Introduction

1.1 Advanced Real-Time Systems

The evolution of information sciences and technologies is characterized by the extensive integration of embedded components in systems used in various application areas, from telecommunications to automotive, manufacturing, medical applications, e-commerce etc. In most cases, embedded components are real-time systems that continuously interact with other systems and the physical world. Integration and continuous interaction of software and hardware components makes the assurance of global quality a major issue in system design. The failure of a component may have catastrophic consequences on systems performance, security, safety, availability etc.

Building embedded real-time systems of guaranteed quality, in a cost-effective manner, raises challenging scientific and technological problems. Existing theory, techniques and technology are of little help as they fail to provide a global framework relating various design parameters to system dynamics and its properties. Contrary to conventional real-time systems, the development of advanced real-time systems, must take into account a variety of requirements about:

- Cost-effectiveness and time to market. These requirements are certainly the most important for advanced real-time systems usually embedded in mass market products. It is possible to improve quality by increasing costs and this has been often the case for conventional real-time applications. For example, the cost of the control equipment in a commercial aircraft is (still) a small percentage of the cost of the whole. On the contrary, for cellular phones even minimal optimizations of resources such as memory and energy or of time to market is of paramount importance.
- Fast evolving environments with rich dynamics e.g. in multimedia and telecommunication systems.
- Combination of hard and soft real-time activities which implies the possibility to apply dynamic scheduling policies respecting optimality criteria. Soft real-time is indeed harder than hard real-time as it requires that when necessary, some timing constraints are relaxed in some optimal manner, provided quality of service remains acceptable.
- Behavior which is dynamically adaptive, reconfigurable, reflexive, intelligent and "any fashionable buzzword used to qualify properties meaning that systems behave less stupidly than they actually do". Building systems meeting

T.A. Henzinger and C.M. Kirsch (Eds.): EMSOFT 2001, LNCS 2211, pp. 373–389, 2001.
© Springer-Verlag Berlin Heidelberg 2001

such properties is essential for quality assurance if we want to increase system interactivity and autonomy. Inventing new buzzwords does not help solving problems which are intrinsically hard. In fact, it is easy to understand that building systems enjoying such desirable properties amounts to designing controllers and thus, advanced controller design techniques for complex and heterogeneous systems are needed.
- Dependability covering in particular reliability, security, safety and availability. The dynamic nature and heterogeneity of advanced real-time systems makes most dependability evaluation techniques partial or obsolete.

Advanced real-time system developers lack theoretical and practical tools and enabling technology for dependable and affordable products and services. The emergence of such enabling technology requires tight and long term cooperation between researchers and practitioners. From a theoretical point of view, it raises foundational problems about systems modeling, design, analysis and control. The concept of control appears to be a key concept in advanced real-time systems engineering.

1.2 The Role of Modeling

Modeling plays a central role in systems engineering. The use of models can profitably replace experimentation on actual systems with incomparable advantages such as,

- enhanced modifiability of the model and its parameters
- ease of construction by integration of models of heterogeneous components,
- generality by using genericity, abstraction, behavioral non determinism
- enhanced observability and controllability especially, avoidance of probe effect and of disturbances due to experimentation
- finally, possibility of analysis and predictability by application of formal methods.

Building models which faithfully represent complex systems is a non trivial problem and a prerequisite to the application of formal analysis techniques. Usually, modeling techniques are applied at early phases of system development and at high abstraction level. Nevertheless, the need of a unified view of the various lifecycle activities and of their interdependencies, motivated recently, the so called model-based approaches [Ves97,BALS99,HHK01,L$^+$01] which heavily rely on the use of modeling methods and tools to provide support and guidance for system development and validation. Modeling systems in the large is an important trend in software and systems engineering today.

Currently, validation of real-time systems is done by experimentation and measurement on specific platforms in order to adjust design parameters and hopefully achieve conformity to QoS requirements. The existence of modeling techniques for real-time systems is a basis for rigorous design and should drastically ease their validation.

The paper unifies results developed at Verimag over the past four years into a methodological framework for modeling advanced real-time systems. Most of the ideas are applicable to arbitrary systems modeling. Current trends in advanced real-time systems foreshadow trends in general systems as integration and interactivity increase.

We consider modeling as an activity integrated in the system development process, strongly related to design and synthesis. Based on this view, we identify some challenging problems and a related research agenda. *A central thesis is that a dynamic model of a real-time system can be obtained by adequately restricting the behavior of its application software with timing information.* We present a composition/decomposition methodology and experimental results illustrating the thesis. The methodology raises some very basic and challenging problems about relating functional to non functional (time dependent) aspects of the behavior by application of composition techniques. Its application requires building models in the large by composition of software components and is illustrated by results obtained within the Taxys project.

2 Challenges and Open Problems

We identify main challenging problems and work directions for their solution.

2.1 The Divide between Application Software and Real-Time System

It is generally agreed that a main obstacle to the application of rigorous development techniques is the lack of methodology for relating application software and functional design to physical architecture and implementation.

At functional design level, a system is specified as a set of interacting components. The functional architecture adopts a decomposition which may be different from the one adopted by physical architecture. High level real-time languages such as ADA, Esterel and SDL, very often use simplifying assumptions about components behavior and their cooperation e.g. instantaneous and perfect communication of components, synchrony of interaction with the external environment or atomicity of actions. These are very useful abstractions that drastically simplify description. The lack of associated methods for correct implementation of the high level constructs and concepts may be a serious limitation for the effective use of high level languages.

The deep distinction between real-time application software and a corresponding real-time system resides in the fact that the former is immaterial and thus untimed. Its real-time properties depend on the speed of the underlying platform and the interaction with external environment. By the operational semantics of the language, it represents a reactive machine triggered by external stimuli. Even if there are actions depending on time e.g. timeouts, time is external and provided by the execution platform. Expiration of a timeout is an event that is treated in the same manner as any external event such as hitting

an obstacle. The study of a real time system requires the use of timed models, that can describe the combined effect of both actions of the application software and time progress.

The transition from application software to implementation involves steps and choices that determine the dynamic behavior of the real-time system. These steps consist in

- partitioning the application software into parallel tasks or threads and mapping them to the physical architecture.
- ensuring resource management and task synchronization by means of synchronization primitives offered by the underlying platform
- finding adequate scheduling policies so that given quality of service requirements (non functional properties) are met; this requires in principle, taking into account the dynamics of the execution platform e.g. WCET for atomic actions and the dynamics of the external environment.

There exist today reasonably good methods, tools and technology for supporting functional system design and application software development activities. Nevertheless, the rigorous transition to implementation comes up against several problems, such as

- relating properties of the functional design with properties of the implementation e.g. it is desirable that functional properties are preserved by the implementation methods
- modeling the dynamics of an implementation for simulation, analysis and validation purposes
- evaluating the influence of design choices on non functional properties.

2.2 Synchronous vs. Asynchronous Real-Time

Current practice in real-time systems design follows two well-established paradigms.

The synchronous paradigm based on a synchronous execution model has been developed in order to better control reaction times and interaction with the external environment. It assumes that a system interacts with its environment by performing global computation steps. In a step, the system computes its reaction to environment stimuli by propagating their effects through its components. The synchrony assumption says that system reaction is fast enough with respect to its external environment. This practically means that environment changes occurring during a step are treated at the next step and implies that responsiveness and precision are limited by step duration. Hardware description languages and languages such as Esterel [BG92], Lustre [HCRP91], and Signal [BLJ91] adopt the synchronous paradigm. For correct implementation of these languages care should be taken to meet the synchrony assumption by guaranteeing not only termination of the actions performed in a step but also that their execution times have known upper bounds.

Synchronous programs are implemented by adopting scheduling policies that guarantee that within a step all the tasks make some progress even though high priority tasks get a larger share of CPU. They are used in signal processing, multimedia and automatic control. Safety critical applications are often synchronous and are implemented as single sequential tasks on single processors. Synchronous language compilers generate sequential code by adopting very simple scheduling principles.

The asynchronous paradigm does not impose any notion of global computation step in program or system execution. Asynchronous real-time is a still unexplored area, especially for distributed computing systems that are inherently dynamic and must adapt to accommodate workload changes and to counter uncertainties in the system and its environment.

A simple instance of the asynchronous paradigm is the one developed around the ADA 95 [Whe96] language and associated implementation techniques, essentially RMA techniques [HKO+93,GKL91]. The latter provide a collection of quantitative methods that allow to estimate response times of a system composed of periodic or sporadic tasks with fixed priorities.

In asynchronous implementations, tasks are granted resources according to criteria based on the use of priorities. This is usually implemented by using RTOS and schedulers.

There exists a basic difference between the two paradigms. The synchronous one guarantees, due to the existence of global computation steps, that "everybody gets something". The asynchronous paradigm applies the principle that the "winner takes all", the winner being elected by application of scheduling criteria. This implies that in asynchronous implementations, a high priority external stimulus can be taken into account "as soon as possible" while in synchronous implementations reaction time is bounded by the execution time of a step. Nevertheless, "immediate" reaction to high priority external stimuli does not imply satisfaction of global real-time properties. Existing theory based on the use of analytic models, allows to guarantee simple real-time properties, typically meeting deadlines and is applicable only to simple task arrival models [LL73].

For advanced real-time applications it is desirable to combine the synchronous and asynchronous paradigm at both description and implementation levels. We need programming and specification languages combining the two description styles as some applications have loosely coupled subsystems composed of strongly synchronized components.

Even in the case where purely synchronous or asynchronous programming languages are used, it is interesting to mix synchronous and asynchronous implementation to cope with inherent limitations of each paradigm. For instance, for synchronous languages it is possible to get more efficient implementations that respect the abstract semantics, by scheduling components execution within a step and thus making the system sensitive to environment state changes within a step. Furthermore, for synchronous software it is possible to relax synchrony at implementation level by mapping components solicited at different rates to different non pre-emptible tasks coordinated by a runtime system.

Proposals of real-time versions of object-based languages such as Java [Gro00] and UML [Gro01], provide concepts and constructs allowing to mix the two paradigms and even to go beyond the distinction synchronous/asynchronous. In principle, it is possible to associate with objects general scheduling constraints to be met when they are executed. The concept of dynamic scheduling policy should allow combining the synchronous and asynchronous paradigms or most importantly, finding intermediate policies corresponding to tradeoffs between these two extreme policies. The development of technology enabling such a practice is certainly an important work direction.

3 Modeling Real-Time Systems

3.1 Component-Based Modeling

Definition: The purpose of modeling is to build models of software and systems which satisfy given requirements. We assume that models are built by composing components which are model units (building blocks) fully characterized by their interface. We use the notation $\|$ to denote an arbitrary composition operation including simple composition operations a la CCS [Mil89] or CSP [Hoa85], protocols or any kind of "glue" used in an integration process: $C1\|C2$ represents a system composed of two components $C1$ and $C2$. We assume that the meaning of $\|$ can be defined by operational semantics rules determining the behavior of the composite system from the behavior of the components.

The modeling problem: Given a component C and a property P find a composition operation $\|$ and a component C' such that the system $C\|C'$ satisfies P.

Notice that by this definition, we consider that modeling does not essentially differ from design or synthesis. In practice, the property P is very often implicit as it may express the conjunction of all the requirements to be met by the composite system. The above definition provides a basis for hierarchical or incremental modeling as composite systems can themselves be considered as components. Thus, complex systems can be obtained by iterative application.

Incremental modeling: To cope with complexity, it is desirable that the model of a complex system is obtained incrementally by further restricting some initial model. This can be done in two manners:

- By integration of components, that is building a system $C1\|C2\ldots\|Cn$ by adding to $C1$ interacting components $C2\ldots Cn$ so that the obtained system satisfies a given property. We want, throughout the integration process, already established properties of components to be preserved. *Composability*, means that if a property P holds for a component C then this property holds in systems of the form $C\|C'$ obtained by further integration. For example, if C is deadlock-free then it remains deadlock-free by integration. Composability is essential for building models which are by construction correct. Unfortunately, time dependent properties are non composable, in general [AGS00,AGS01,Lee00].

– By refinement, that is by getting from an abstract description C a more concrete (restricted) one C' in the sense that the behaviors of C' are "contained" in the behaviors of C. Refinement relations are captured as simulation relations modulo some observability criterion establishing a correspondence between concrete and abstract states and/or actions. We want, throughout the modeling process, refinement to be preserved by composition operators. That is, if components of a system are replaced by components refining them, then the obtained system is a refinement of the initial system. This property called *refinement compositionality*, is essential for relating models at different abstraction levels. Refinement compositionality should play an important role in relating application software to its implementations. Existing results are not satisfactory in that they deal only with preservation of safety properties. The application of constructive techniques requires stronger results e.g. preservation of progress properties.

3.2 About Timed Models

A real-time system is a layered system consisting of the application software implemented as a set of interacting tasks, and the underlying execution platform. It continuously interacts with an external environment to provide a service satisfying requirements, usually called QoS requirements. The requirements characterize essential properties of the dynamics of the interaction.

Models of real-time systems should represent faithfully the system's interactive behavior taking into account relevant implementation choices related to resource management and scheduling as well as execution speed of the underlying hardware. They are timed models as they represent the dynamics of the interaction not only in terms of actions but also in terms of time. Building such models is clearly a non trivial problem.

Timed models can be defined as extensions of untimed models by adding time variables which are state variables used to measure the time elapsed. They can be represented as machines that perform two kinds of state changes: transitions and time steps. Transitions are timeless state changes that represent the effect of actions of the untimed system; their execution may depend on and modify time variables. Time steps represent time progress and increase uniformly only time variables. They are specified by time progress conditions [BS00]: time can progress from a state by t if the time progress condition remains true in all intermediate states reached by the system. During time steps state components of the untimed model remain unchanged. There exists a variety of timed formalisms extensions of Petri nets [Sif77], process algebras [NS91] and automata [AD94]. Any executable untimed description e.g. application software, can be extended into a timed one by adding explicitly time variables or other timing constraints about action execution times, deadlines etc.

Timed models use a notion of logical time. Contrary to physical time, logical time progress can block especially as a result of inconsistency of timing constraints. The behavior of a timed model is characterized by the set of its runs,

that is the set of maximal sequences of consecutive states reached by performing transitions or time steps. The time elapsed between two states of a run is computed by summing up the durations of all the time steps between them. For a timed model to represent a system, it is necessary that it is well-timed in the sense that in all runs time diverges.

As a rule, in timed models there exist states from which time cannot progress. If time can progress from any state of a timed model, then it is always possible to wait and postpone the execution of actions which means that it is not possible to model action urgency. Such models represent degenerated timed systems which are in fact untimed systems as time can be uniformly abstracted away. Action urgency at a state is modeled by disallowing time progress. This possibility of stopping time progress goes against our intuition about physical time and constitutes a basic difference between the notions of physical and logical time. It has deep consequences on timed systems modeling by composition of timed components.

Composition of timed models can be defined as extensions of untimed composition. They compose actions exactly as untimed composition. Furthermore, for time steps, a synchronous composition rule is applied as a direct consequence of the assumption about a global notion of time. For a time step of duration t to occur in a timed system, all its components should allow time progress by t. Well-timedness is not composable in general, especially when components have urgent synchronization actions.

3.3 Building the Timed Model

We present a methodology for building timed models of real-time systems as layered descriptions composed of

- Models of the tasks
- A synchronization layer
- A scheduler that controls execution so as to meet QoS requirements.

Timed models of tasks. The application of the methodology requires compilation and WCET analysis tools. We discuss the principle of construction of the timed model without addressing efficiency or even effectiveness issues. We assume that code for tasks (execution units) is obtained by partitioning the application software and separate compilation. Furthermore, we assume that for a given implementation platform and by using analysis tools, the following information can be obtained about each task:

- Which sequences of statements are atomic during execution
- For atomic sequences of statements, estimates of execution times e.g. WCET or interval timing constraints with lower and upper bounds.

The code of a task C with this information constitutes its timed model C_T. The latter can perform the actions of C - which are atomic sequences of statements of C - and time steps.

Synchronization of timed tasks. We assume that task synchronization is described in terms of synchronization primitives of the underlying platform such as semaphores and monitors, to resolve task cooperation and resource management issues. In principle, untimed tasks code with their synchronization constraints is a refinement of the application software.

We want to get compositionally the timed model corresponding to the system of synchronized tasks. For this, it is necessary to extend the meaning of synchronization operations on timed behavior. For example, if $C1$ and $C2$ is the code of two tasks and $C\|C2$ represents the system after application of synchronization constraints, then we need a timed extension $\|_T$ of $\|$ to compose the timed models $C1_T$ and $C2_T$ of $C1$ and $C2$. $C1_T\|_T C2_T$ is the timed model of $C1\|C2$. In [BS00], it has been shown that untimed composition operators $\|$ admit various timed extensions $\|_T$ depending on the way urgency constraints (deadlines) of timed tasks are composed.

Extending untimed composition operators to timed ones is an important problem for the construction of the global timed model. It is desirable that essential properties of the timed components such as deadlock-freedom and well-timedness are preserved by timed composition. We have shown that usual composition operators for timed models do not preserve well-timedness [BGS00, BS00]. The existence of methodology and theory relating properties of untimed application software to the associated timed model is instrumental for getting correct implementations. Usually, correct implementation of the operator $\|$ used in the software development language, requires a fairly complex definition of $\|_T$. The reasons for this are twofold.

1. Whenever the interaction between untimed components is supposed to be instantaneous, the application of the same composition principle to timed components may lead to inconsistency such as deadlocks or timelocks. The obvious reason for this is that in the timed model (the implementation) component reactions to external stimuli do take time. When a component computes a reaction, its environment state changes must be recorded to be taken into account later. So, contrary to instantaneous untimed composition, timed composition needs memory to store external environment events.
2. In many composition operations for untimed systems, independent actions of components may interleave. Interleaving implicitly allows indefinite waiting of a component before achieving synchronization. It may happen that an untimed system is deadlock-free and the corresponding timed system has deadlocks. Adding timing constraints may restrict waiting times and destroy smooth cooperation between components (see example in [BST98,BS00]).

Composition of timed components should preserve the main functional properties of the corresponding untimed system. Otherwise, most of the benefit from formal verification of functional properties is lost. To our knowledge, this is a problem not thoroughly investigated and which cannot be solved by directly transposing usual untimed composition concepts. We need results for correct implementation of untimed interaction primitives relying on the concepts of protocol or architecture.

Scheduler modeling. The real-time system model must be "closed" by a timed model of the external environment. We assume that this model characterizes the expected QoS from the system. That is, instead of using extra notation such as temporal logic or diagrams to express the QoS requirements, we represent them as a timed model. It can be shown that their satisfaction boils down as for untimed systems, to absence of deadlock in the product system obtained by composition of the environment and the real-time system models. This method applied to verify safety properties for untimed systems can be used to verify also liveness properties if the timed models are well-timed and this property is preserved by parallel composition.

Scheduler modeling is a challenging problem. Schedulers coordinate the execution of system activities so that requirements about their timed behavior are met, especially QoS requirements. We have shown that schedulers can be considered as controllers of the system model composed of its timed tasks with their synchronization and of a timed model of the external environment [AGS00, AGS01]. They apply execution strategies which satisfy, on the one hand timing constraints resulting from the underlying implementation, essentially execution times, and on the other hand QoS requirements represented by a timed model of the environment. Under some conditions, correct scheduling can be reduced to a deadlock avoidance problem.

More concretely, a scheduler monitors the state of the timed model and selects among pending requests for using common resources. The role of scheduler can be specified by requirements expressed as a constraint K, set of timed states of the scheduled system. The scheduler keeps the system states in a control invariant K' subset of K, set of states from which the constraint K can be maintained in spite of "disturbances" of the environment and of internal actions of the tasks. The existence of a scheduler maintaining K depends on the existence of non empty control invariants contained in K [MPS95,AGP+99]. Control invariants can be characterized as fixpoints of monotonic functions (predicate transformers) representing the transition relation of the timed system to be scheduled. There exists a scheduler maintaining K iff there exist non empty fixpoints implying K. Computing such fixpoints is a non trivial problem of prohibitive complexity when it is decidable - essentially, for scheduling without preemption. This relation of scheduler modeling to controller synthesis explains the inherent difficulties in the application of model-based techniques to scheduling.

A methodology based on composability results has been studied in [AGS01] to circumvent the difficulties in scheduler modeling. It consists in decomposing the constraint K into a conjunction of two constraints $K = K_{sched} \wedge K_{pol}$. K_{sched} specifies the fact that timing requirements are satisfied. K_{pol} specifies scheduling policies for resource allocation and can be expressed as the conjunction of resource allocation constraints K^r, $K_{pol} = \bigwedge_{r \in R} K^r$, where R is the set of the shared resources. The constraint K^r can again be decomposed into two types of constraints $K^r = K^r_{resolve} \wedge K^r_{admit}$. $K^r_{resolve}$ specifies how conflicts for the acquisition of r are resolved while K^r_{admit} characterizes admission control policies and says when a request for r is taken into account by the scheduler.

According to results presented in [AGS01] such a decomposition allows to simplify the modeling problem of a scheduler maintaining $K_{sched} \wedge K_{pol}$ by proceeding in two steps. First, getting a scheduler that maintains K_{pol}; this does not require the application of synthesis algorithms. The second step aims at finding by restriction of this scheduler, a scheduler which maintains K_{sched} and requires in general, the application of synthesis or verification techniques.

The construction of the scheduler maintaining K_{sched} heavily relies on the use of **dynamic priority rules**. It uses composability results allowing to obtain incrementally the scheduler. A priority rule is a pair consisting of a condition and of an order relation on task actions. When the condition is true, the associated priority order is applied and its effect is to prevent actions of low priority to occur when actions of higher priority are enabled. It has been shown that non trivial scheduling policies and algorithms can be modeled, by using priority rules. Furthermore, there exist composabilty results that guarantee preservation of absence of deadlock and in some cases of liveness, by application of dynamic priority rules [BGS00].

Interesting research directions in scheduler modeling are:

- Composability: An important observation is that scheduling methods are not composable, in the sense that if a scheduler maintains $K1$ and a scheduler maintains $K2$ then even if they control access to disjoint sets of resources, their application does not in general maintain $K1 \wedge K2$. In [AGS00] a sufficient condition for scheduler composability is given. There is a lot to be done in that direction.
- Connections to the controller synthesis paradigm: This is a very interesting research direction, especially because controller synthesis provides a general theoretical framework that allows better understanding the scheduling problem and the inherent difficulties. Controller synthesis provides also the general framework for tackling more pragmatic and methodological questions.
- Combining model-based scheduling and scheduling theory: Scheduling theory proposes sufficient conditions for schedulability of simple tasks, usually periodic or sporadic, for specific scheduling policies. It is based on the use of analytic models that lead to more tractable analysis techniques at the price of approximation [ABR91,HKO+93,EZR+99]. Some combination and unification of the two approaches seems feasible and should give results of highest interest.

3.4 Application to Modeling Embedded Telecommunication Systems

The Taxys project, in collaboration with France Telecom and Alcatel Business Systems uses the principle given in 3.3 to build timed models of real-time systems. The purpose of this project is the analysis of embedded real-time applications used in cellular phones, to check QoS requirements. The latter are expressed as properties of the form “the time distance between an input signal

and the associated reaction is bounded by some delay" or "the time distance between two consecutive output signals is within a given interval".

The application software is written in Esterel extended with C functions and data. The design environment comprises a compiler used to generate C code from application software and a parametric code generator used to generate executable code for DSP's. The implementation consists of one task and does not use OS or scheduler. Scheduling issues are resolved by the compiler which determines causally correct orders of execution between processes.

The C code is structured as a reactive machine whose transitions are triggered by signals provided by the external environment. A transition occurrence changes atomically control state and data state by executing an associated C function. To build a timed model of the application, the C code is instrumented by adding timing constraints about execution times of C functions. These constraints are lower and upper bounds of execution times estimated by using an execution time analysis tool. The global timed model of the embedded real-time system is obtained by composition of the timed model of the C code with a timed model of the environment and an external events handler. The latter is used to store signals emitted by the environment. It is important to notice that while according to Esterel semantics, the interaction between an Esterel program and its environment is instantaneous, the composition between the timed application software and the timed environment requires the use of memory. In fact, instantaneous actions of the untimed model take time when executed and during this time the environment changes must be registered to be taken into account in the next application cycle. This is an instance of the general problem evoked in 3.3. Global timed models of the real-time system, are analyzed by using the Kronos [DOTY96,Yov97,BSS97] timing analysis tool.

The Esterel compiler has been engineered to generate the instrumented C code from pragmas describing timing constraints. Thus, the timed models of both the application software and of the environment can be described in Esterel appropriately extended with pragmas to express timing constraints. The obtained results are very encouraging [BPS00,CPP+01,TY01b,TY01a] and we plan to apply this principle to modeling real-time systems with multitasking.

4 Discussion

Modeling is a central activity in systems development intimately related to design, programming and specification. We consider only modeling by using executable languages. Although we do not deny the interest of using notations based on logic and declarative languages, we do not consider them as essential in control dominated modeling. Some notation encompassing abstraction such as the various kinds of charts (message charts, sequence charts) is certainly useful for requirements expression. Nevertheless, we consider that the main challenges concern modeling by using executable languages at different levels from specification to implementation and for different execution models.

We believe that an important trend is relating implementation to models at different abstraction levels. For this, it seems inevitable that models are built by using or rather reusing components written in languages such as C and C++. We need modeling environments which support methodologies and tools for building models from heterogeneous components, synchronous and asynchronous. The use of C and C++ code in such environments requires methodology and tools for monitoring and instrumenting code execution, in particular to change abstraction, granularity of execution and execution model. Modeling systems in the large is an important and practically relevant work direction.

4.1 For a Common Conceptual Basis

Building general modeling environments requires a common conceptual basis for component based modeling, a unifying notation with clean and natural operational semantics. Such a notation should distinguish 3 basic and complementary ingredients in models: components, interaction model and execution model. A model is a layered description consisting of a set of components, on which are applied successively an interaction model and an execution model. Interaction and execution models describe two complementary aspects of an architecture model.

Components are system units characterized by their interface specified as a set of interaction points. An interaction point corresponds to a shared action (event-based approach) or a shared variable (state-based approach). An interaction point can be conceived as a pair consisting of a name (port, signal, input, output) and a list parameters.

A system is modeled as a set of interacting components. A component state determines which interaction points can participate in an interaction and the values of their parameters. For given component states, an interaction between a set of components can be specified as a subset of compatible interacting points. The compatibility relation may be statically determined or may depend on the states of the components involved in the interaction. Usually, we distinguish between the initiator of the interaction and the other participating components as in some interaction models it is necessary to take into account the flow of causality. Interactions may be n-ary, in general. For general modeling languages there is no reason for restricting to interactions of particular arity.

The interaction model describes the effect of interactions on the states of the participating components. We need languages encompassing the following different interaction types.

- **Blocking and non blocking interaction:** Blocking interactions are interactions that cannot take place unless all the involved interaction points are enabled for instance, synchronous message passing in CSP and CCS. Non blocking interactions can take place when the initiator of the interaction is enabled and involve all the enabled participants. Synchronous languages and hardware description languages use non blocking interaction.

– **Atomic and non atomic interaction:** Atomic interaction means that interaction is completed without interference of other system events. Clearly, non atomic interaction can be modeled in terms of atomic interaction by using memory such as shared variables and fifo queues. It is desirable that modeling languages offer primitives for the direct description of non atomic interaction mechanisms.

Some papers distinguish also between event-based and state-based interaction. We do not consider this distinction to be foundationally relevant although in practice it can be convenient to use rather one than the other type of interaction.

The execution model contains information about component concurrency control in a given architecture. It further constrains the behavior obtained by application of an interaction model by reducing non determinism inherent to interactive behavior. Its effect can be represented by adding extra components that restrict the behavior of the interacting components. For methodological reasons, it is very important to separate component interaction from execution control. Component interaction deals with satisfaction of functional properties, essentially safety properties, such as mutual exclusion. The execution model deals with component progress and sharing computing power. It determines the computation threads and is strongly related to fairness, scheduling, run to completion, time step properties. Usually, in existing languages, the execution model is strongly related to their interaction model: atomic interaction for synchronous languages and non atomic interaction for asynchronous languages. Nevertheless, it is desirable to dissociate interaction from execution aspects. An important trend in proposals for real-time versions of object-based languages, is associating with objects priorities and scheduling constraints.

4.2 Relating Functional to Non Functional Models

Relating untimed to timed, functional to non functional models by using as much as possible composability and compositionality is a very important work direction. Existing theory on system modeling and composition such as process algebra, has been developed on low level models e.g. transitions systems, and badly resists to extensions with data and time. For programs, we have compositionality results for simple sequential programs e.g. Hoare logic whose extension to concurrent systems leads to complicated methodologies. The concepts of composition studied so far, consider essentially composition as intersection of behaviors. They lend themselves to theoretical treatment but are not appropriate for component-based modeling. For the latter, we need composition concepts that modulo some assumptions about component properties, guarantee properties of the whole system. Results using assume/guarantee rules seem difficult to apply. We believe that for component-based modeling, we need "flexible" composition rules that compose the components behavior by preserving essential component properties such as deadlock-freedom. Contrary to usual composition, flexible composition operators seek consensus (avoid "clashes") between interacting components. A

notion of flexible composition for timed systems have been studied in [BS00]. The basic idea is not to compose separately timing requirements and actions which may easily lead to inconsistency: a deadlock is reached if a component offers an urgent synchronization action that cannot be accepted by its environment without letting time pass. Instead, flexible composition adopts a "non Newtonian" or "relativistic" view of time by assuming that time is "bent by the action space". To avoid deadlock in the previous example, time is allowed to pass as long as no action is enabled. This may lead to violation of the individual timing constraints of components but guarantees well-timedness of the composed system.

Existing implementation theories do not preserve properties such as deadlock-freedom of components, or more general progress properties, especially when going from functional to non functional (timed) models. Usually, for timed models obtained by adding timing requirements to an untimed model, component deadlock-freedom is not preserved in either way. Timing can introduce deadlocks to deadlock-free untimed systems and deadlocks of untimed systems can be removed by adding timing constraints. In absence of any theoretical results about preservation of properties other than safety properties, it is questionable whether it is worthwhile verifying at functional model level properties which are not preserved by implementation techniques such as deadlock-freedom, liveness and fairness properties. We badly lack implementation theories for high level languages.

4.3 Scheduler and Controller Modeling

The distinction between interaction and execution models determines two kinds of modeling problems of uneven difficulty. Modeling interaction seems to be an easier and in any case a better understood problem than modeling execution. We consider the latter to be a still unexplored problem, especially in the case of timed models. To describe execution models, we need languages with powerful constructs allowing compositional description of dynamic priorities, preemption and urgency. The distinction between synchronous and asynchronous execution should be captured by using execution models.

Schedulers are components that implement execution models. Their modeling deserves special attention as their role is crucial for correct implementation. Scheduler modeling can be considered as an instance of a more general problem, that of finding controllers which maintain given constraints. The development of design and modeling methodologies for controllers leads to explore connections to control theory. This is certainly a very interesting research direction confirmed by trends in some areas such as multimedia where control algorithms are used to solve traffic management problems. Connection to controller synthesis problems may also bring solutions to all the hard and still not very well understood problems of building systems enjoying "magic" properties such as adaptivity, reflectivity, survivability, and why not, intelligence.

Acknowledgment. I would like to thank my colleagues of Verimag and J. Pulou of France Télécom R&D for constructive comments and criticism.

References

[ABR91] N.C. Audsley, A. Burns, and M.F. Richardson. Hard real-time scheduling: the deadline monotonic approach. In *Workshop on real-time operating systems and software*, 1991.

[AD94] R. Alur and D.L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[AGP+99] K. Altisen, G. Gößler, A. Pnueli, J. Sifakis, S. Tripakis, and S. Yovine. A framework for scheduler synthesis. In *Proc. RTSS 1999*, pages 154–163. IEEE Computer Society Press, 1999.

[AGS00] K. Altisen, G. Gößler, and J. Sifakis. A methodology for the construction of scheduled systems. In M. Joseph, editor, *proc. FTRTFT 2000*, volume 1926 of *LNCS*, pages 106–120. Springer-Verlag, 2000.

[AGS01] K. Altisen, G. Gößler, and J. Sifakis. Scheduler modeling based on the controller synthesis paradigm. *Journal of Real-Time Systems, special issue on "control-theoretical approaches to real-time computing" (to appear)*, 2001.

[BALS99] H. Ben-Abdallah, I. Lee, and O. Sokolsky. Specification and analysis of real-time systems with PARAGON. In *Annals of Software Engineering*, 1999.

[BG92] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[BGS00] S. Bornot, G. Gößler, and J. Sifakis. On the construction of live timed systems. In S. Graf and M. Schwartzbach, editors, *Proc. TACAS 2000*, volume 1785 of *LNCS*, pages 109–126. Springer-Verlag, 2000.

[BLJ91] A. Benveniste, P. LeGuernic, and Ch. Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Science of Computer Programming*, 16:103–149, 1991.

[BPS00] V. Bertin, M. Poize, and J. Sifakis. Towards validated real-time software. In *Proc. 12th Euromicro Conference on Real Time Systems*, pages 157–164, 2000.

[BS00] S. Bornot and J. Sifakis. An algebraic framework for urgency. *Information and Computation*, 163:172–202, 2000.

[BSS97] A. Bouajjani, S.Tripakis, and S.Yovine. On-the-fly symbolic model-checking for real-time systems. In *Proc. IEEE Real-Time Systems Symposium, RTSS'97*. IEEE Computer Society Press, 1997.

[BST98] S. Bornot, J. Sifakis, and S. Tripakis. Modeling urgency in timed systems. In *Proc. COMPOS'97*, volume 1536 of *LNCS*. Springer-Verlag, 1998.

[CPP+01] E. Closse, M. Poize, J. Pulou, J. Sifakis, P. Venier, D. Weil, and S. Yovine. TAXYS = ESTEREL + KRONOS. a tool for the development and verification of real-time embedded systems. In *Proc. CAV'01*, volume 2102 of *LNCS*, pages 391–395. Springer-Verlag, 2001.

[DOTY96] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In *Hybrid Systems III, Verification and Control*, volume 1066 of *LNCS*, pages 208–219. Springer-Verlag, 1996.

[EZR+99] R. Ernst, D. Ziegenbein, K. Richter, L. Thiele, and J. Teich. Hardware/software codesign of embedded systems — the SPI workbench. In *Proc. Int. Workshop on VLSI, Orlando, Florida*, 1999.

[GKL91] M. Gonzáles, M. Klein, and J. Lehoczky. Fixed priority scheduling of periodic tasks with varying execution priorities. In *Proc. RTSS 1991*, 1991.

[Gro00] Real-Time Java Working Group. International j consortium specification. Technical report, International J Consortium, 2000.

[Gro01] OMG Working Group. Response to the omg rfp for schedulability, performance, and time. Technical Report ad/2001-06-14, OMG, June 2001.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[HHK01] T. A. Henzinger, B. Horowitz, and C. Meyer Kirsch. Embedded control systems development with Giotto. In *Proc. LCTES 2001 (to appear)*, 2001.

[HKO+93] M.G. Härbour, M.H. Klein, R. Obenza, B. Pollak, and T. Ralya. *A Practitioner's Handbook for Real-Time Analysis*. Kluwer, 1993.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[L+01] E. A. Lee et al. Overview of the Ptolemy project. Technical Report UCB/ERL M01/11, University of California at Berkeley, 2001.

[Lee00] E.A. Lee. What's ahead for embedded software. *Computer, IEEE*, pages 18–25, September 2000.

[LL73] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1), 1973.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[MPS95] O. Maler, A. Pnueli, and J. Sifakis. On the synthesis of discrete controllers for timed systems. In E.W. Mayr and C. Puech, editors, *STACS'95*, volume 900 of *LNCS*, pages 229–242. Springer-Verlag, 1995.

[NS91] X. Nicollin and J. Sifakis. An overview and synthesis on timed process algebras. In *Proc. CAV'91*, volume 575 of *LNCS*. Springer-Verlag, July 1991.

[Sif77] J. Sifakis. Use of petri nets for performance evaluation. In *Proc. 3rd Intl. Symposium on Modeling and Evaluation*, pages 75–93. IFIP, North Holland, 1977.

[TY01a] S. Tripakis and S. Yovine. Modeling, timing analysis and code generation of PATH's automated vehicle control application software. In *Proc. CDC'01 (to appear)*, 2001.

[TY01b] S. Tripakis and S. Yovine. Timing analysis and code generation of vehicle control software using taxys. In *Proc. Workshop on Runtime Verification, RV'01*, volume 55, Paris, France, July 2001. Elsevier.

[Ves97] S. Vestal. MetaH support for real-time multi-processor avionics. In *Proc. IEEE Workshop on Parallel and Distributed Real-Time Systems*, pages 11–21, 1997.

[Whe96] D. Wheeler. *ADA 95. The Lovelace Tutorial*. Springer-Verlag, 1996.

[Yov97] S. Yovine. KRONOS: A verification tool for real-time systems. *Software Tools for Technology Transfer*, 1(1+2):123–133, 1997.

VEST – A Toolset for Constructing and Analyzing Component Based Embedded Systems*

John A. Stankovic

Department of Computer Science
University of Virginia
Charlottesville, VA 22906
stankovic@cs.virginia.edu

Abstract. Embedded systems are proliferating at an amazing rate. To be successful these systems must be tailored to meet many cost, time to market, performance, and dependability constraints. Building and tailoring embedded systems by using domain specific components has promise. However, in composing components the most difficult issues are ensuring that non-functional properties such as real-time performance and dependability are being met. The VEST toolkit is aimed at providing a rich set of dependency checks based on aspects to support embedded system development via components.

1 Introduction

Embedded systems are proliferating at an amazing rate with no end in sight. In 1998 only 2% of all processors where used for general purpose workstations and 98% for embedded systems. The percentage of processors used for workstations is rapidly approaching 0% for the year 2001. Success of embedded systems depends on low cost, a high degree of tailorability, quickness to market, cost-effective variations in the product, and sometimes flexible operation of the product. The reliability of these products and the degree of configurability are paramount concerns. Currently, there is no efficient way to build software for these systems. The use of component based software for constructing and tailoring these systems has promise. However, most components are too heavyweight and don't explicitly address real-time, memory, power and cost constraints. What is required is a new type of component that is lightweight and relates to the physical and real-time properties of embedded systems.

The first part of the solution is developing the new components themselves. This problem is where most people have spent their energies. While this is a necessary step, it is the easiest step and it ignores fully addressing how the components interact with other components or how they fit into a component

* This work was supported, in part, by the DARPA PCES program under grant F33615-00-C-3048.

T.A. Henzinger and C.M. Kirsch (Eds.): EMSOFT 2001, LNCS 2211, pp. 390–402, 2001.
© Springer-Verlag Berlin Heidelberg 2001

infrastructure. The second problem is that while significant work has been done in developing CORBA [36], DCOM [24], Jini [1] components, much less has been done with components suitable for embedded systems. Third, most tools available for the configuration process provide little more than linking or an extended "make" capability. It is the designer who has to know everything about the components and their (often hidden) constraints. For example, a designer may know that the combination of 2 seemingly unrelated components leads to unpredictable delays in network protocol processing. What is needed are tools that support the specification of embedded system requirements followed by knowledgeable and helpful construction of the embedded system with careful analysis of component dependencies as well as the time, memory, power, and cost constraints. The final product must offer as many guarantees as possible along many dimensions including correctness, performance and reliability. Our work is focusing on the development of effective composition mechanisms, and the associated dependency and non-functional analyses for real-time embedded systems. Our solutions are embodied within a toolkit called VEST (Virginia Embedded Systems Toolkit).

We are attempting to develop techniques that are general enough to construct and perform analysis checks on any of the following types of systems:

- a static and fixed embedded system at low cost (e.g., software for a wrist watch),
- networked and long lived embedded systems that can be made more evolvable by hot swapping both hardware and software (e.g., air traffic control), and
- a reflective runtime environment to support a high degree of adaptability (e.g., air traffic control or even a set top box that sometimes acts to download movies, at other times act as a PC, or to run games).

The main purposes of this paper are threefold: to present the design of VEST, to highlight various types of dependency checks including aspects, and to illustrate the special issues involved with composing embedded real-time systems.

2 The Design of VEST

VEST (Virginia Embedded Systems Toolkit) is an integrated environment for constructing and analyzing component based embedded and real-time systems. Generally speaking, the VEST environment is designed to help people select or create passive components (collections of code fragments, functions, classes, HW, etc), compose them into a system/product, map them onto runtime (active) structures such as processes or threads, map them onto specific hardware, and perform dependency checks and non-functional analyses to offer as many guarantees as possible along many dimensions including real-time performance, and reliability[1]. The VEST environment is comprised of the following main parts:

[1] VEST does not support formal proof of correctness.

- DOME (Domain Modeling Environment) [17], a tool-set which is an extensible collection of integrated model editing and analysis tools supporting a Model-Based Development approach to system/software engineering.
- A library system composed of 5 libraries: domain application components, middleware components, OS components, aspects, and the product under development library.
- An interactive composition and analysis tool which supports:
  - Tailored infrastructure and product creation via domain specific software/hardware composition
  - Mapping of passive software components to active runtime structures (processes/threads)
  - Mapping of processes/threads to hardware platforms
  - A set of dependency and aspect checks
  - Ability to invoke off-the shelf or home grown analysis tools such as real-time and reliability analyses.

Using VEST the designer can

- Create a new component which entails not only adding the source code, but also adding a significant amount of reflective information. The reflective information for source code objects include categories such as interface information, linking information, location of source code itself, graphical information for displaying/representing the source object on the VEST graphical editor canvas, and other reflective information of the source object needed to analyze cross cutting dependencies. Examples of this latter category are presented in section 3.
- Choose a component from one of the libraries.
- Graphically compose components so that a hierarchy of components can exist.
- Use Glue Code to connect components, if necessary.
- Map components to processes so that the active part of a composed system can be designed and analyzed.
- Bind processes to HW. Only after this step can we truly do the real-time analysis since execution times are highly platform dependent.
- Perform dependency checks including aspects. This is one area where we believe VEST makes a major contribution. Previous systems did not have enough support for cross-cutting dependencies and this is one specific goal for VEST.
- Invoke off-the-shelf analysis tools to analyze a configured system.

To better support real-time and embedded systems, the VEST component library contains both software and hardware objects. The software objects can themselves be further divided into two subgroups: source objects that describe a specific code written in a traditional language like C, and higher-level objects that describe groupings of source objects. The hardware objects can also be divided into descriptions of a specific piece of hardware and descriptions of groupings of hardware components. We are supporting the following hardware

devices: processors, memories, caches, DSP, busses, A/D and D/A, actuators and sensors. Sets of reflective information exist for each of these device types. Processes and threads are separate entities in VEST and also have reflective information describing their attributes.

To briefly provide some idea of the graphical interface the tool buttons available to a designer include invoking/using/composing: Functions, Classes, Components, Threads, Processes, Devices, Composed Devices, Glue Code, Select and Move, and Dependency Connectors are provided for users to describe entities' dependency graphically (we provide both a Precedence Connector and a Call-Graph Connector). Attributes can be added to each connector, e.g., attributes can be added to a call-graph connector such as 1) Call instance: Which call instance this connector stands for; 2) Return type: Return parameter type; and 3) Input parameter list. A Mapping/Binding connector is used to map a passive component to a thread/process or bind a thread/process to a piece of hardware.

From the VEST graphical interface users can create a new library, add/delete components to a library, browse for a component, and search for a component by name or by keyword. Finally, through a menu item they can invoke the off-line analysis tools.

3 Dependency Checks and Aspects

One goal of VEST is to provide significant support for various types of dependency checking. Dependency checks are invoked to establish certain properties of the composed system. This is a critical part of embedded and real-time system design and implementation. For purposes of doing research, we have separated dependency checks into 4 types:

- Factual
- Inter-component
- Aspect
- General

Factual. The simplest, component-by-component dependency check is called factual dependency checking. Each component has factual (reflective) information about itself including:

- WCET (worst case execution time)
- memory needs
- data requirements
- interface assumptions
- importance
- deadline or other time constraints
- jitter requirements
- power requirements (if a hardware component),
- initialization requirements
- environment requirements such as

  - must be able to disable interrupts
  - requires virtual memory
  - cannot block
- the code itself

The above list is extensible depending on the application area. A factual dependency check can include using this information across all the components in a configuration. For example, a dependency check might involve determining if there is enough memory. This can be very easy; add up the memory requirements of each component and compare to the memory provided by the hardware memory component. If memory needs are variable, a slightly more sophisticated check is needed that accounts for worst case memory needs. The greater number of factual dependency requirements that are checked by VEST the more likely it is that simple mistakes are avoided.

Inter-component. Inter-component dependencies refer to pairwise component checks. This includes:

- call graphs
- interface requirements including parameters and types of those parameters
- precedence requirements
- exclusion requirements
- version compatibility
- software to hardware requirements, and
- this component is included in another.

The above list is extensible. Given a set of components configured for some purpose the inter-component checks also tend to be easy. For example, that modules conform to interface definitions can easily checked. That correct versions are being used together and that no two components that must exclude each other are included, also can be easily checked. The greater the number of inter-component dependency requirements that are checked, the more likely that simple mistakes are avoided.

Aspects. Aspects [19] are defined as those issues which cannot be cleanly encapsulated in a generalized procedure. They usually include issues which affect the performance or semantics of components. This includes many real-time, concurrency, synchronization, and reliability issues. For example, changing one component may affect the end-to-end deadlines of many components that are working together. Many open questions exist here. How can we specify the aspects? Which aspects are specific for a given application or application domain and which are general? How does the tool perform the checks? Are the checks feasible for certain problems?

It is our belief that aspects include an open ended (unbounded) set of issues. Therefore, we cannot hope to be complete, rather we need to identify key aspects for embedded systems and create the specification and tools to address as many of these issues as possible. The more aspects that can be checked, the more

confidence in the resulting composed system we will have. However, by no means do we claim that the system is *correct*, only that certain specific checked errors are not present.

Consider an example of aspects relating to concurrency. In looking into this issue, we have divided the problem into 4 parts: what are the concurrency concepts that need to be expressed, what mechanisms exist for supporting those concepts (the components), how can we express the concepts and components, and how are the dependency checks themselves performed?

Concurrency concepts include: synchronous versus asynchronous operation, mutual exclusion, serialization, parallelism, deadlock, atomicity, state (running, ready, blocked), events (triggers), broadcasts, coordinated termination, optimistic concurrency control, and multiple readers - single writer. This list can be extended, perhaps indefinitely.

Concurrency mechanisms include: locks, binary and counting semaphores, wait-list management, timers, messages, monitors, spin locks, fork-join, guards, conditional critical regions, and precedence graphs. At this time we have not developed any language for expressing the concurrency concepts or mechanisms.

Related to aspects, dependency checks seem to come in two types. One is an explicit check across components in the current configuration. Examples include: suppose a given system has only periodic tasks and a change is made to add aperiodic tasks. A particular dependency check may be able to identify all those components that had previously assumed that no aperiodics would exist. Another example involves a system that has no kill primitive. Adding a kill primitive imposes a key requirement on all other tasks that they cannot be killed while they are in the middle of a critical section.

The second type of aspect checking is actually deferred to associated analysis tools. For example, checking that real-time constraints are met is best done by collecting the right information and passing it to a more general analysis tool. Four such tools are described in section 4.

General. The fourth type of dependency check we refer to as a *general* dependency check. It is not clear that this category is needed and it may eventually be folded into the other three. However, global properties such as that the system should not experience deadlock, or that hierarchical locking rules must be followed, or a degree of protection is required seem different enough that, for now, we are treating these checks as a separate category. Note that it is not important what category a certain dependency check falls into, only that such a check is made. However, each category may require fundamentally different solutions. Aspects, while they can be global, seem to focus more on how code changes affect other modules, while the general dependency category focuses on global requirements. For example, a dependency check for the absence of deadlock might rely on a Petri-net analysis (see also non-functional analyses section below).

3.1 Examples of Dependency Checks

The current version of VEST implements the following dependency checks.

A Memory Size Check: Browse through a system (all the graphical icons on the editing pane or reachable from that pane), adding up the memory requirements. Browse through the system, adding up the total memory provided by the hardware memory modules. Compare the memory required with the available memory and inform users of the result. Note: Besides modules whose memory requirement values are available, the number of modules whose memory requirement values are not available is also counted and reported to users.

Power Consumption Check: Browse through a system (all the graphical icons on the editing pane or reachable from the pane), adding up the power requirements and the available power, and summarizing the number of modules whose power requirement value are not available. Display the power consumption result.

Interface Compatibility Check: For each call graph connector in the system, check whether its interface is compatible with its calling component's interface. For those mismatched call graph connectors, mark them in red and inform the users.

All Call-Graphs Check: Browse through a system to check all of its components' calling dependencies. For a component's calling dependency list, look through the whole system to see if components it calls are missing from current system. If some components are missing from current system, inform users.

Exclusion Constraints Check: This name-based checking will check any generic exclusion violations among the system. Browse through a system to check all of its components' exclusion constraints. For a component's exclusion list, look through the whole system to see if there is an exclusion violation, i.e., other components that it excludes also compose current system. If an exclusion happens in current system, inform users.

No Dynamic MM Check: This checks a specific exclusion constraint, i.e., a dynamic memory management constraint. Users can use it to check if dynamic memory management is excluded by a system (i.e., some components in the system exclude the use of any dynamic memory management components).

Buffer Size: Browse through a system, for each buffer, add up its total buffer consumption based on its rate of execution. Browse through the system, for each buffer, add up its total buffer production based on its rate of execution. Compare each buffer's total production with its total consumption. If its total production is larger than total consumption, messages might be lost. Inform users.

4 Non-functional Analysis Tools

Our configuration tool needs to scan the newly composed system, extracting the appropriate reflective information needed for a particular analysis. It then passes that information to the analysis tool in the correct format. A tool may be an off-the-shelf tool or one supplied with our system. To make this clear, consider the following four examples.

Consider that we want to perform a real-time analysis that takes exclusive access to shared resources into account. See Figure 1. To perform such an analysis requires making many decisions such as we must know all the components (hardware and software), the mapping of software to runtime active structures, the mapping of software to hardware, the scheduling algorithm being used, and the workload. The component reflective information must include worst case execution times for that software on specific hardware as well as what data that software needs to share in exclusive mode. All of this information can be part of the configuration tool set. Given knowledge of all these things it is then possible to use the Spring scheduling off-line algorithm [45], designated as H() in the figure, to compute a feasible schedule where all deadlines and exclusion constraints are met, or be told that no such schedule was found. See Figure 2. In figure 2 it is shown that task A is scheduled on one CPU, while tasks B and C are on a second CPU. Resources R1 through R5 are additional resources required by these tasks and they are allocated for time periods that *avoid* conflicts. For example, tasks A and B are scheduled to run concurrently and they use a disjoint set of resources R2 and R3, respectively. Alternatively, tasks B and C are scheduled to run sequentially because they conflict over resource R3. In this schedule all the tasks make their deadlines, designated D(i) i=A,B,C in the picture. In general, if no valid schedule is found the designer can take various actions such as adding hardware and then reiterate the analysis.

A second example would be to perform actions very similar to the above scenario except the designer makes assumptions about using the rate monotonic analysis approach with priority ceiling and invokes the schedulability analysis for that paradigm. This, for example, is being done in MetaH.

A third example involves reliability analysis. Off-the shelf tools exist for computing the reliability of a given system configuration. Such an analysis is usually quite restrictive. Yet, what ever level of sophistication that the off-the-shelf tool supports, that can be supported here. For example, if we assume fail stop hardware, we know the probability of failure of each hardware component and the HW configuration, we can input this to a tool such as Galileo and determine the reliability. This, of course, does not consider software faults.

Tools based on Petri-nets analysis exist to evaluate various concurrency properties of a system. Again, our configuration tool can extract the right information from the reflective information about the newly composed system, put it into the format expected from a Petri-net analysis tool and obtain the analysis benefits of that tool.

5 Current Status

VEST version 0 has been implemented. Version 0 has basic library support, support for functions as components, various examples of dependency checking, and the capability to invoke external off-the-shelf tools. It has basic graphic support and is built on DOME. VEST version 1 is currently being implemented. It will contain the following:

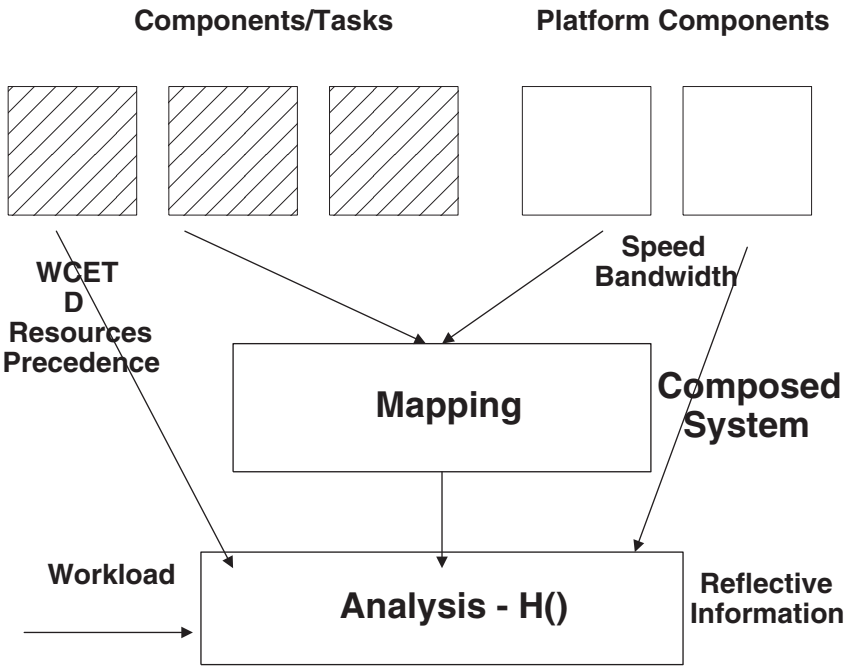


Fig. 1. Real-Time and Concurrency Example.

- Support for objects and classes including multiple interfaces.
- Support for configurable components so that a designer can instantiate a library component with specific parameters for their product.
- Support for events as first class entities and how those entities relate to aspects.
- Direct support for language independent aspects.
- Additional library support for the management of 5 libraries: base application code library, middleware library, OS library, aspect library, and new product library.
- Addition of more dependency and aspect checking.

6 State of the Art

The software engineering field has worked on component based solutions for a long time. Systems such as CORBA [36], COM [23], DCOM [24], and Jini [1] exist to facilitate object or component composition. These systems have many advantages including reusability of software and higher reliability since the components are written by domain experts. However, CORBA and COM tend to produce heavyweight components that are not well suited to embedded systems,

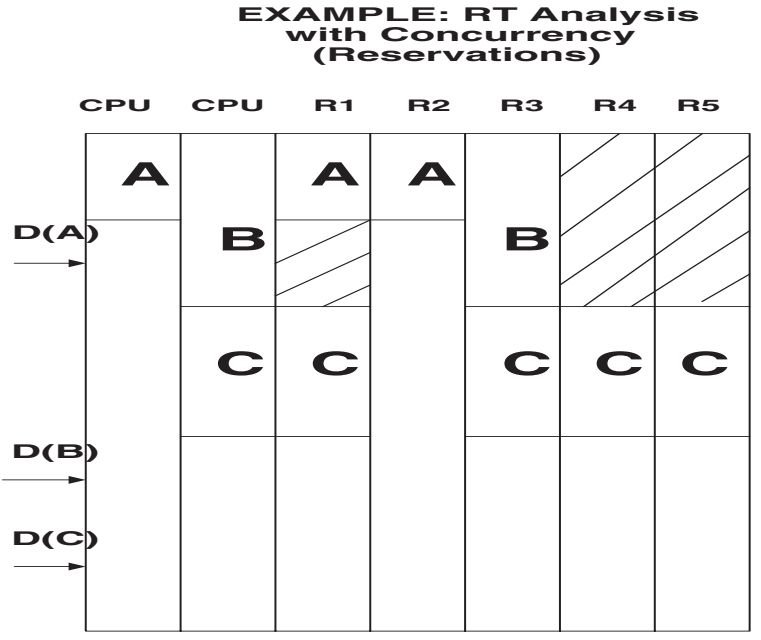


Fig. 2. Feasible Schedule.

and none of these systems have adequate analysis capabilities. Also, the generality and large size of component libraries makes these general solutions difficult to use. However, using components in more focussed domains such as found in many embedded system applications seems promising. Often a high degree of tailorability is needed to make a product successful in the marketplace. This tailorability may be required in the application code, the middleware (if any) and the OS.

Tailorability has been the focus of OS microkernel research such as in Spin [2], Vino [9], and the exo-kernel [10]. However, these systems have not addressed embedded systems issues, are not component based and have no analysis of the type being proposed here. They also are applied to general purpose timesharing systems with totally unrelated applications executing on a general purpose OS.

Real-time kernels such as VxWorks [44], VRTX32 [31], and QNX [16] all have a degree of configurability. But the configuration choices are at a high level, e.g., you can include virtual memory or network protocols or not. There are no dependency checks or analysis tools to support composition.

The KNIT [32] system is an interesting composition system for general purpose operating systems. This system is addressing a number of cross cutting concerns in composing operating systems. For example, they consider linking, initialization, and a few other dependencies. To date, it has not focussed on real-time and embedded system concerns.

For embedded systems we do find some work on component based OSs. These include: MMLite [15], Pebble [13], Pure [3], eCOS [8], icWorkshop [18] and 2K [20]. However, these might be considered first generation systems and have focussed on building the components themselves with a substantial and fixed infrastructure. These systems offer very little or no analysis tools and, for the most part, minimal configuration support. For more details on these systems see [12].

Using components in focused domains has been successful. For example, there are tools and components for building avionics systems. There are also systems for creating network protocols such as Coyote [5], the click modular router [25], and Ensemble/Nuprl [21]. The success of these systems lies in the ability to focus on a specific area. This permits better control over problems such as understanding dependencies and avoiding the explosion of the numbers and variants of components.

To date, the closest system to match our goals is the MetaH [43] system. This system consists of a collection of tools for the layout of the architecture of an embedded system and its reliability and real-time analysis. The main differences from our work include MetaH begins with active tasks as the components, assumes an underlying real-time OS, and has limited dependency checking. In contrast we propose to first create passive components (collections of code fragments, functions and objects) and then map those onto runtime structures, and we also focus on adding key dependency checks to address cross cutting dependencies among composed code.

7 Summary

The number of embedded systems and processors are growing far faster than workstations. Many of these systems are composed of networks of processors, sensors, and actuators. Software for these systems must be produced quickly and reliably. The software must be tailored for a particular application/product. A collection of components must interoperate, satisfy various dependencies, and meet non-functional requirements. These systems cannot all be built from scratch. A goal of our toolkit is to substantially improve the development, implementation and evaluation of these systems. The toolkit focuses on using language independent notions of *aspects* to deal with non-functional properties, and is geared to embedded system issues that include application domain specific code, middleware (if any), the OS, and the hardware platform. Our toolkit could have beneficial impact on products from smart toasters, to set top boxes, Internet appliances, defense systems, and avionics.

References

1. Arnold K., O'Sullivan B., Scheifler R.W., Waldo J., and Wollrath A.(1999) The Jini Specification. Addison-Wesley.
2. Bershad B., Chambers C., Eggers S., Maeda C., McNamee D., Pardyak P. Savage S., Sirer E. (1994) SPIN - An Extensible Microkernel for Application-specific Operating System Services, University of Washington Technical Report 94-03-03.

3. Beuche D., Guerrouat A., Papajewski H., Schroder-Preikschat W., Spinczyk O., and Spinczyk U. (1999) The PURE Family of Object-Oriented Operating Systems for Deeply Embedded Systems. Proceedings of 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, Saint-Malo, France.
4. Beugnard A., Jezequel J., Plouzeau N. and, Watkins D. (1999) Making Components Contract Aware. Computer, 32(7), 38-45.
5. Bhatti N., Hiltunen M., Schlichting R., and Chiu W. (1998) Coyote: A System for Constructing Fine-Grain Configurable Communication Services. ACM Transactions on Computer Systems, 16(4), 321-366.
6. Booch G. (1987) Software Components with Ada: Structures, Tools and Subsystems. Benjamin-Cummings, Redwood City, CA.
7. Campbell R., Islam N., Madany P., and Raila D. (1993) Designing and Implementing Choices: an Object-Oriented System in C++. Communications of the ACM, September 1993.
8. Cygnus (1999) eCos - Embedded Cygnus Operating System. Technical White Paper (http://www.cygnus.com/ecos).
9. Endo Y., et. al. (1994) VINO: The 1994 Fall Harvest, TR-34-84, Harvard University.
10. Engler D., Kaashoek M.F. and O'Toole J. (1995) Exokernel: An Operating System Architecture for Application-Level Resource Management. Proceedings of the 15th SOSP, Copper Mountain, CO.
11. Ford B., Back G., Benson G., Lepreau J., Lin A., and Shivers O. (1997) The Flux OSKit: A Substrate for Kernel and Language Research. Proceedings of the 16th ACM Symposium on Operating Systems Principles, Saint-Malo, France.
12. Friedrich, L., J. Stankovic, M. Humphrey, M. Marley, and J. Haskins. (2001). A Survey of Configurable, Component-Based Operating Systems for Embedded Applications, *IEEE Micro*, Vol. 21, No. 3, May-June, pp. 54-68.
13. Gabber E., Small C., Bruno J., Brustoloni J., and Silberschatz A. (1999) The Pebble Component-Based Operating System. Proceedings of the USENIX Annual Technical Conference. Monterey, California, USA.
14. Haskins J., Stankovic J., (2000), muOS, TR, University of Virginia, in preparation.
15. Helander J. and Forin A.(1998) MMLite: A Highly Componentized System Architecture. Proceedings of the Eighth ACM SIGOPS European Workshop. Sintra, Portugal.
16. Hildebrand D. (1992) An Architectural Overview of QNX. Proceedings of the USENIX Workshop on Micro-kernels and Other Kernel Architectures. Seattle, WA.
17. Honeywell DOME User Guide, http://www.htc.honeywell.com/dome/support.htm.documentation.
18. Integrated Chipware IcWorkShop (http://www.chipware.com/).
19. Lopes, C., and Kiczales G. (1997), D: A Language Framework for Distributed Programming, TR SPL97-010, Xerox Parc.
20. Kon F., Singhai A., Campbell R. H., Carvalho D., Moore R., and Ballesteros F. (1998) 2K: A Reflective, Component-Based Operating System for Rapidly Changing Environments. Proceedings of the ECOOP'98 Workshop on Reflective Object-Oriented Programming and Systems. Brussels, Belgium. July 1998.
21. Liu X., et. al. (1999), Building Reliable High-Performance Communication Systems from Components, *SOSP*, Vol. 33, No. 5.
22. Meyer B. and Mingins C. (1999) Component-Based Development: From Buzz to Spark. Computer, 32(7), 35-37.
23. Microsoft Corporation and Digital Equipment Corporation (1995) The Component Object Model Specification. Redmond, Washington.

24. Microsoft Corporation (1998) Distributed Component Object Model Protocol, version 1.0. Redmond, Washington.
25. Morris R., Kohler E., Jannotti J., and Kaashoek M. (1999), The Click Modular Router, *SOSP*, Vol. 33, No. 5.
26. Niehaus D., Stankovic J., and Ramamritham K. (1995), A Real-Time Systems Description Language, *IEEE Real-Time Technology and Applications Symposium*, pp. 104-115.
27. Nierstrasz O., Gibbs S., and Tsichritzis D. (1992) Component-oriented software development. Communications of the ACM, 35(9), 160-165.
28. Oberon Microsystems (1998) Jbed Whitepaper: Component Software and Real-time Computing. Technical Report. Zurich, Switzerland (http://www.oberon.ch).
29. Object Management Group (1997) The Common Object Request Broker: Architecture and Specification, Revision 2.0, formal document 97-02-25 (http://www.omg.org).
30. Orfali R., Harkey D., and Edwards J. (1996) The Essential Distributed Objects Survival Guide. John Wiley and Sons, New York.
31. Ready J. (1986), VRTX: A Real-Time Operating System for Embedded Microprocessor Applications, *IEEE Micro*, Vol. 6, No. 4, pp. 8-17.
32. Reid, A., M. Flatt, L. Stoller, J. Lepreau, and E. Eide. (2000) Knit: Component Composition for Systems Software. *OSDI 2000*, San Diego, Calif., pp. 347-360.
33. Samentiger J. (1997) Software Engineering with Reusable Components. Springer-Verlag, Town.
34. Saulpaugh T. and Mirho C. (1999) Inside the JavaOS Operating System. Addison Wesley, Reading, Massachusetts.
35. Short K. (1997) Component Based Development and Object Modeling. Sterling Software (http://www.cool.sterling.com).
36. Siegel J. (1998), OMG Overview: Corba abd OMA in Enterprise Computing, *CACM*, Vol. 41, No. 10.
37. Stankovic J., and Ramamritham K. (1991), The Spring Kernel: A New Paradigm for Real-Time Systems, *IEEE Software*, Vol. 8, No. 3, pp. 62-72; also in text on *Readings in Real-Time Systems*, IEEE Press, editor Mani Krishna.
38. Stankovic J. , Ramamritham K., Niehaus D., Humphrey M., and Wallace G. (1999), The Spring System: Integrated Support for Complex Real-Time Systems, special issue of *Real-Time Systems Journal*, Vol. 16, No. 2/3.
39. Stankovic J. and Ramamritham K. (1995), A Reflective Architecture for Real-Time Operating Systems, chapter in *Advances in Real-Time Systems*, Prentice Hall, pp. 23-38.
40. Sun Microsystems (1996) JavaBeans, version 1.0. (http://java.sun.com/beans).
41. Szyperski C. (1998) Component Software Beyond Object-Oriented Programming. Addison-Wesley, ACM Press, New York.
42. Takada H. (1997) ITRON: A Standard Real-Time Kernel Specification for Small-Scale Embedded Systems. Real-Time Magazine, issue 97q3.
43. Vestal, S., (1997), MetaH Support for Real-Time Multi-Processor Avionics, *Real-Time Systems Symposium*.
44. Wind River Systems, Inc. (1995) VxWorks Programmer's Guide.
45. Zhao, W., Ramamritham, K., and Stankovic, J., (1987) Preemptive Scheduling Under Time and Resource Constraints, **Special Issue** of *IEEE Transactions on Computers* on Real-Time Systems, Vol. C-36, No. 8, pp. 949-960.

Embedded Software: Challenges and Opportunities

Janos Sztipanovits and Gabor Karsai

Institute for Software Integrated Systems, Vanderbilt University, P.O. Box 1829 Sta. B. Nashville, TN 37235, USA

{sztipaj, gabor}@vanderbilt.edu

Abstract. This paper discusses challenges and opportunities in embedded system and software design. The key thesis of the paper is that embedded computing applications create fundamentally new challenges for software technology that require new ideas and approaches. The paper reviews two of the new challenges, physicality and its effects on change management. The discussion on the impacts of these challenges focuses on key components of integrated development environments, representation, synthesis and analysis, and generator technology.

1 Introduction

One of the fundamental trends in information technology is the rapidly expanding integration role of computing in physical systems. From airplanes to automobiles and from manufacturing systems to appliances, interaction and coordination of physical components increasingly involves computing. In these, so called embedded computing applications, larger and larger part of the software gets involved in closed-loop, tight interactions with physical processes. The role of the embedded software is to configure and control the operation of programmable computing devices so as to meet physical requirements at their sensor-actuator interfaces. Tight integration of physical processes with information processing has a profound impact on the complexity of embedded software and system design.

Is there any *essential* difference between embedded software development and software development in general? While no one argues that embedded software is much harder to design, the source and effects of differences have not been investigated well in the past. The most frequently mentioned differentiators, such as hardware closeness, domain specificity or the need of real-time response, capture only some, albeit interesting attributes of embedded software. However, these arguments do not explain the wholesale problems with large-scale embedded software experienced in almost all application fields, ranging from aerospace through medical to automotive applications. Reported problems with large embedded software systems suggest existing software design techniques are not suitable for building large embedded software systems. The differences are fundamental, which require a full re-thinking of basic principles.

The goals of this paper are to discuss new challenges embedded software brings about to researchers and developers, and to examine the impact of these new

T.A. Henzinger and C.M. Kirsch (Eds.): EMSOFT 2001, LNCS 2211, pp. 403-415, 2001.
© Springer-Verlag Berlin Heidelberg 2001

challenges on key aspects of Integrated Development Environments (IDE). Our primary emphasis is on the hard problems of creating domain specific IDE-s for embedded system and software. The examples for possible solutions are based on experience gained from the Model Integrated Computing effort at the Institute of Software Integrated Systems (ISIS) at Vanderbilt University [1].

The outline of the paper is the following: In Section 2, we discuss "Physicality" "Constraints and Change" as fundamental discriminators in embedded software design. Section 3 summarizes the unique requirements toward IDE-s. In Section 4, we discuss the problems of domain specific representations in IDE-s. Section 5 summarizes the problems of integrating analysis and synthesis tools in domain specific environments. In Section 6, we give a brief overview of the role of generators in creating domain specific IDE-s

2 Challenges of Embedded Software Development

The key discriminator of embedded computing is its tight integration with physical processes. Tight integration means that computing becomes direct part of physical interactions. In the following, we will summarize the most important consequences of this interaction on the design of embedded software.

2.1 Physicality of Embedded Software

Embedded computers are surrounded by physical processes: they receive their inputs from sensors and send their outputs to actuators. Accordingly, embedded computing devices, viewed from their sensor and actuator interfaces, act like physical processes, with dynamics, noise, fault, size, power and other physical characteristics. *The role of the embedded software is to "configure" the computing device so as to meet physical requirements.*

In non-embedded software, logical correctness of the computations is the primary requirement. Physical properties of their execution on a particular computing device are considered secondary or non-essential. Since most of the computer applications in the past were non-embedded, software technology has evolved with an emphasis on the logical correctness of increasingly complex applications. Introduction of higher levels of abstractions and functional composition in languages have proved to be effective. The best concepts of modern software component technologies such as procedural abstraction, objects (i.e. data abstraction, information hiding, and polymorphism), design patterns, and architectural abstractions (i.e. components and connectors) enable rapid functional composition and result in significant productivity increase.

In embedded software, logical correctness is insufficient. An embedded software component, whose logical behavior is defined in some computer language, is "instantiated into" a physical behavior on a computing device. The instantiation of logical behavior into physical behavior is complicated by the following factors:

1. Detailed physical characteristics of the devices involved (physical architecture, instruction execution speed, bus bandwidth, etc.) are required.
2. Modern processor architectures introduce complex interactions between the code and essential physical characteristics of the device (speed, power dissipation, etc.)
3. Lower layers of typical software architectures (RTOS scheduler, memory managers, middleware services) strongly interact with application code in producing the net physical behavior.
4. Physical properties of components interfere due to the use of shared resources (processors, buses, physical memory devices, etc.)

It is not surprising that using current software technology, logical/functional composability does not imply physical composability. In fact, *physical properties are not composable*, rather, they appear as cross-cutting constraints in the development process. The effects of cross-cutting constraints can be devastating for the design. Meeting specifications in one part of the system may destroy performance in others, and, additionally, many of the problems will surface at system integration time. Consequently, we need to change our approach to the design of embedded software: productivity increases must come from tools that directly address the design of the whole system with its many different physical, functional and logical aspects.

2.2 Constraints and Change in Embedded Software

Physicality is only one of the consequences of the rapidly growing integration role of embedded computing in physical systems. The other, perhaps less obvious result of this trend is the increasing challenge of achieving flexibility via software.

A common assumption in systems engineering is that embedded software will lend flexibility to physical platforms because software is easy to modify. Unfortunately, when embedded computing is used as "universal integrator" in systems, this assumption becomes invalid. The reasons are the following:

1. Software is designed with the goal of achieving an overall behavior in the integrated physical/information system by assuming specific characteristics and behaviors of the physical components.
2. Software components have implicit (e.g. architectural decisions) and explicit (e.g. parameters and models of physical components and processes) interdependencies with the embedding physical environment.
3. Deviations from nominal behavior in and among physical components are compensated for by computations.

It means that embedded software is the place where system-wide constraints tend to accumulate in a hidden fashion. The integrated system becomes rigid as a result of these accumulated constraints – which does not do anything with the ease of changing the software, as an implementation media. A particular difficulty of using current software technology in embedded systems is that languages do not support the

explicit representation of the rich interdependency among software and physical system components.

Flexibility is the result of careful design and not the byproduct of the implementation media. Flexibility can be achieved by solving a system wide constraint management problem: constraints need to be explicitly represented and effects of changes need to be propagated both in design time and run time.

3 Impacts on Integrated Development Environments

Tight integration of physical processes with information processing has profound impact on the complexity of embedded software and system design. The following challenges draw increasing attention from researchers and practitioners.

1. Composition is a key concept of modern software engineering in managing complexity. In embedded computing, where software is designed to achieve some required physical behavior, functional composition is not sufficient. Physical requirements, such as dynamics, jitter, noise, fault behavior, etc. appear as cross-cutting physical constraints, which are not composable along functional relations. One of the efficient approaches to address this problem is to establish composability on a higher level of abstraction. Instead of composing software on "code level", embedded systems (including physical and information processing components) are modeled from different aspects, composition is expressed on the level of models, and the code is automatically generated form the composed models.
2. In embedded computing, the traditional separation of system and software design is not maintainable: predictability of the design requires integrated modeling and analysis of physical processes and information processing. Since modeling of physical and information processes are based on different mathematical and physical frameworks (physical processes are usually modeled with continuous dynamics, information processes are modeled with discrete dynamics and use strongly different models of time), the requirement for integrated modeling leads to new theoretical foundations in hybrid systems [2], and in hybrid models of computation [3].
3. The design of embedded systems is inherently domain specific. The *embedding environment* establishes a unique engineering context that determines the language of the design, the essential behaviors to be analyzed and the nature of the components to be assembled. While the benefits of Integrated Development Environments (IDE) in embedded computing and in system integration are well understood, the high cost of domain-specific IDE-s represents a significant roadblock against their wider application. We need technology for rapidly *composing* design environments from reusable components for the fraction of the cost today.

Model-based IDE-s have fundamental significance in embedded system and software development. They represent a strong departure from conventional programming languages and language-based software development environments. The difference is not only in the level of abstractions (model-based software development environments are gaining momentum via the increased use of UML and UML based code generators [4]), but also in the *scope* of modeling. Model-based IDE-s for embedded systems and software extend modeling to the entire system, including both physical and information system components.

In the following Sections we will discuss challenges and opportunities in creating model-based IDE-s for embedded systems.

4 Representation

One of the fundamental problems in integrated, domain specific modeling of physical processes and information processing is *representation*. We will consider the following two closely related issues in model representation: (a) the selection of modeling languages and (b) the integration of tools supporting the modeling process.

4.1 Modeling Languages

Selection of modeling languages has significant impact on the design process. *Should we try to introduce a single, "unified" modeling language for integrated software and system modeling or should we use domain specific languages?* The single language approach has substantial advantages compared to the domain specific languages: time saving in learning the language, cost saving in developing an efficient tool infrastructure, and all the collateral benefits coming with maturation: accumulated experience, portability of model assets, and increasing trust. These arguments drive several efforts in building system-level design languages, such as Modelica [5] or Rosetta [6].

However, there are compelling arguments in favor of using multiple, domain specific modeling languages as well. Domain specific modeling languages can adopt representation formalisms and modeling constructs of established engineering disciplines – there is no need to learn yet another unified language. Domain specific formalisms offer highly efficient constructs to capture design requirements and constraints, and thus the modeler does not have to "translate" the domain specific constructs into some generic notation at modeling time. Last but not least, in rapidly progressing fields, such as embedded computing, emergence of new modeling and analysis techniques drive the progress, which makes earlier modeling approaches (and investment in dedicated tools) obsolete.

The difficulties with domain specific modeling languages can be resolved by introducing a *meta-modeling approach*. Formally, a *modeling language* can be defined as a triplet of abstract syntax, concrete syntax, and interpretation:

$$L = \langle A, C, I \rangle$$

The abstract syntax: *A* defines the *concepts, relationships, integrity constraints* and *model composition principles* available in the language. Thus the abstract syntax determines all the (syntactically) correct "sentences" (in this case models) that can be built. It is important to note, the *A* abstract syntax includes semantic elements as well. The integrity constraints, which define well formedness rules for the models, are frequently called "static semantics". The concrete syntax: *C* defines the form of representation: visual, textual or mixed. Assigning syntactic constructs to the elements of the abstract syntax specifies the concrete syntax. The interpretation: *I* defines the semantics: the meaning of the correct sentences expressed in the language. Formal specification of the semantics is a complex problem particularly in heterogeneous modeling languages. The common practice is to side step the problems and to define the semantics using English text or some semi-formal representations (see e.g. UML semantics [4]). A limited, but useful technique is to define semantics as a mapping between the abstract syntax *A* of a modeling language *L* and the abstract syntax *A'* of another modeling language *L'* having a well defined semantics *I'*, formally: $I: A \rightarrow A'$.

The precise definition of modeling languages requires the specification of all three components. It is important to note that decomposition of modeling languages to abstract syntax, concrete syntax and interpretation introduces significant flexibility: the same abstract syntax can be represented with multiple concrete syntax, and the same abstract syntax may have several interpretations. This flexibility is expected and widely used in the design of large-scale systems, where declarative modeling languages are used to describe the structure of systems and these structural models are mapped into various behavioral models using model interpretation.

Meta-modeling means the modeling of modeling languages by using *meta-modeling languages*. The relationships among meta- and domain specific modeling languages and meta-models and domain models can be seen in Figure 1 (more details can be found in [7]).

A *domain-specific modeling language* consists of domain-specific abstract syntax, concrete syntax, and interpretation(s): $L_D = <A_D, C_D, I_D>$. The syntactically and semantically correct *sentences* of L_D, which are built from instances of concepts and relationships defined in the abstract syntax of the domain A_D, and which also have a well-defined semantics in I_D, are the *domain models.* Similarly, a meta-modeling language $L_M = <A_M, C_M, I_M>$ consists of the abstract syntax used for defining (domain-level) modeling languages, the concrete syntax of those domain-language definitions, and their interpretation. The meta-models —the syntactically and semantically correct sentences of L_M built from instances of concepts and relationships defined in the abstract syntax A_M— define any arbitrary L_D in terms of $<A_M, C_M, I_M>$. This implies that a meta-language should allow us to define *abstract syntax, concrete syntax*, and *interpretation*.

A *meta-modeling language* is a modeling language that is used to define other modeling languages. Because a meta-modeling language can also be considered a domain-specific modeling language (with the domain being that of "modeling languages"), it is desirable that the meta-modeling language be powerful enough to describe itself, in a meta-circular manner. This corresponds to the *four layer meta-model language architecture* used in UML [4] or in CDIF [8].

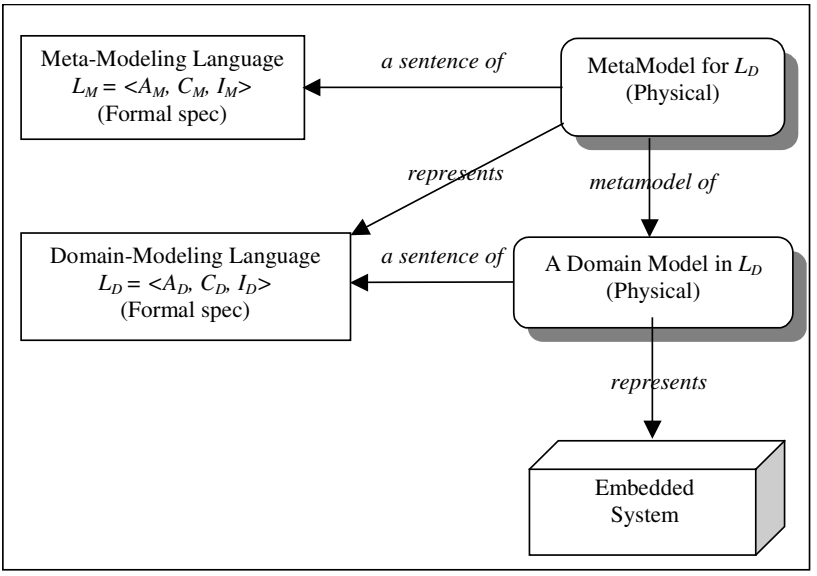


Fig. 1. Meta-modeling language architecture

4.2 Modeling Tools

The meta-model language architecture requires two distinct sets of tools: tools for the meta-modelers and tools for the domain modelers. Both sets of tools include support for building the models (graphical model editors and constraint checkers that enforce well formedness rules during modeling) and tools for storing and managing the models (model database). The relationships between these toolsets are shown in Figure 2.

The meaning (i.e. the semantics) of a meta-model is defined through a domain-modeling tool. The *interpretation* of the meta-model *configures* a generic domain modeling environment to support a domain-specific modeling language. The domain-specialized instance of the generic modeling environment allows only the creation of syntactically and semantically correct domain models, as defined by the meta-model. Interestingly, this principle and approach makes possible a very high degree of reuse in the modeling tools. The Vanderbilt Model Integrated Computing toolset uses the *same* generic modeling environment as the foundation tool for metamodeling and domain modeling. The main components of he environment (see Figure 2) are the meta-programmable Graphical Model Editor [9] and the Model Database, which adds persistent object storage to the modeling environment. The tool architecture supports meta-circularity: a meta-meta-model (defined by the meta-language) configures the environment to support meta-modeling. Thus, the meta-modeling language can be modified, although this typically necessitates changes in the generic modeling environment as well.

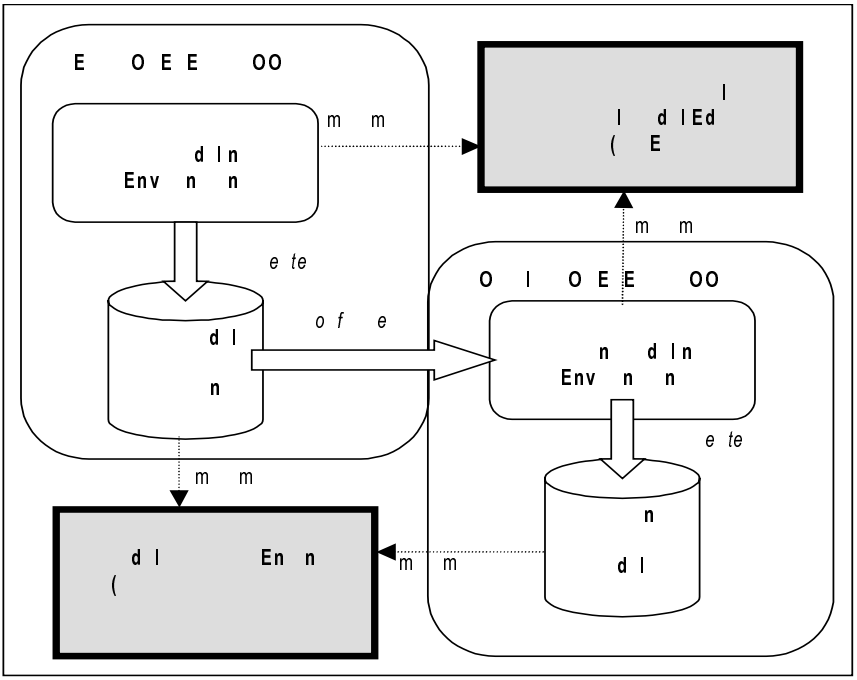


Fig. 2. Meta-programmable Modeling Tool Architecture

Currently, the toolset uses the class diagram portion of the industry standard Unified Modeling Language (UML) [4] and the Object Constraint Language (OCL) [10] to facilitate meta-modeling. This choice was made for the following reasons: (a) the UML/OCL syntax and semantics are (reasonably) well defined [7], (b) UML/OCL is an OMG standard that enjoys widespread use in the industry, (c) tools supporting UML are widely available, and last but not least, (d) familiarity with UML class diagrams helps to mitigate the conceptual difficulty of the meta-modeling language architecture. It is important to note that adopting UML for meta-modeling does not imply any commitment to use UML as domain modeling language. Further details on meta-modeling can be found in [11].

The Vanderbilt toolset can be downloaded from the ISIS web site [12].

5 Synthesis and Analysis

Heterogeneity and physicality of embedded systems requires the application of different types of models in the design process. For example, embedded controllers for automotive applications are often designed by using continuous-time dynamic models (representing the plant dynamics), discrete-time synchronous dataflow models (representing the discretized control laws to be implemented), discrete state models

(representing the modality of the plant and the controller), software architecture models, hardware architecture models, power dissipation models, reliability models, fault models and possibly other models according to the requirements of the technology involved. These models are used by analysis and synthesis tools for understanding expected system behavior and for performing optimization and/or synthesis. A crucial point is that the models are not independent: they represent interdependent aspects of a complex design problem.

5.1 Integrated Modeling

Integrated modeling means that interdependency among different modeling aspects is explicitly represented and tracked during the modeling process. The meta-model language architecture described above provides significant help to do this: relationships among modeling views can be represented as relationships among meta-models. In simple cases, the interdependency among aspects can be captured as constraints on the abstract syntax specification of modeling languages. For example, the fact that a state variable *s* in the discrete, finite state model of the plant corresponds to a discrete variable *d* in the dynamic model of the control law can be represented with defining an equivalence relation between the two concepts in the corresponding meta-models. In more complex cases, where the relationship between two modeling aspects are based on overlapping semantics, the specification of interdependence may take the form of complex transformations among the models.

5.2 Tool Integration

Domain specific IDE-s typically include a suite of modeling, analysis and synthesis tools that use their individual modeling aspects in the integrated models (See Figure 3). Development of dedicated tool suites for the many different, domain specific, integrated modeling languages is not feasible.

Embedded system developers face the following tough choices: (a) They adopt an available tool suite and start using the modeling approaches and languages of the tools – even if those do not meet their needs. The obvious risks are to get locked in a modeling approach and to loose the ability to develop or adopt new emerging techniques. (b) Use different tools without integrated modeling, which costs a lot in terms of lost consistency across interdependent design aspects.

The situation is not better for tool vendors either. They are pushed toward adopting "unified" modeling approaches that may not fit to the modeling philosophy required by their analysis or synthesis techniques, or they face significant resistance in getting new modeling and analysis methods accepted by end-users, due to the cost of integration with existing tools suits.

The desired solution is the development of open tool integration frameworks that enable inexpensive insertion of new modeling, analysis and synthesis methods in domain specific IDE-s. Based on the Model-Integrated Computing approach and using elements of the MIC toolset, ISIS has developed and implemented a tool integration framework [13] shown in Figure 3. The core idea of the approach is to capture the shared (static) semantics among the tools with an Integrated Data Model,

which defines the schema for the Integrated Model Database. The Integrated Data Model is not simply the union of the data models of the integrated tools. It is designed to include all information, which is shared across the tools – possibly structured and expressed in a domain specific form. The Integrated Model Server provides semantic translation (in the sense of static semantics) between the Integrated Data Model and the data models of the individual tools. The Tool Adapters interface the internal representation form (the concrete syntax) of the tools to the Integrated Model Server. The Common Model Interface (CMI) defines a single protocol with a tagged, canonical data representation for the interaction between the Semantic Translators and their Tool Adapters. Introduction of the CMI greatly simplifies the implementation. The semantic translators are defined by using the meta-model language architecture of MIC: their specification is captured in terms of the relationship between the meta-model of the individual tools and the Integrated Data Model. Current work focuses on the automatic generation of the semantic translators from their specification. Details of implementation and application experiences are also reported in [13].

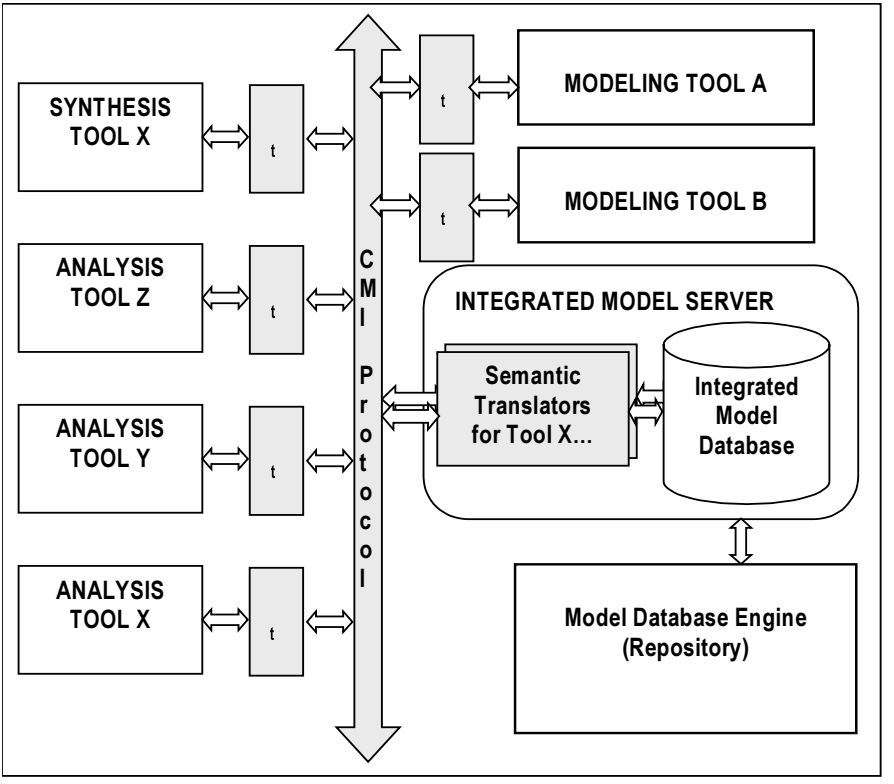


Fig. 3. Tool Integration Architecture

6 Generators

An important task for the Model-Integrated Computing technology is the generation of embedded systems from domain models. Since embedded systems frequently consist of many physical and software components their automated generation is a complex problem. Usually, a variety of artifacts, such as parameters or configurations of hardware and software components, component structure specification, glue code, etc. have to be generated using information in the integrated models.

Using a generative approach in system building is based on the recognition that typical programming languages are not suitable for capturing all the engineering issues that arise when implementing embedded systems, however, the mapping between domain specific models and specific "software" artifacts that go into the final product, is clearly understood. The task of a software generator is to facilitate this mapping in a computational form.

Generative programming [14] is a technique that supports the rapid, yet verifiable construction of software systems for specific domains, by identifying the language of the problem space (i.e. the domain), the elements of the solution space (e.g. the execution environment), and the relationship between the two, that is captured in a software tool: the generator. The generator incorporates all the "configuration knowledge": best practices, construction rules, optimizations, cross-cutting dependencies that are often exist only the head of very experienced software engineers. Undoubtedly, in embedded system development these relationships are even more complex, and arguably it may even be impossible for humans to keep track of them without tool support. All these dependencies will eventually influence the constituents of the final product: parameterized hardware and software modules, communication protocols and scheduling policies used, etc. If this influence is clearly understood, it can be captured *once* in the form of a generator, and then reused across many applications.

In the most general sense, generators implement a transformation engine that transliterates the artifacts of the input problem space (i.e. models) into artifacts of the target solution space (e.g. configurations files, code fragments, etc.) as shown on Figure 4. Generators are different from compilers in that their output is only rarely executable code, but they may perform quite sophisticated analysis and synthesis tasks on the input during generation. However, the input and the output of the rewrite engine can easily be represented as some sort of graph data structure (which offer a richer capabilities than trees typically used in compilers). If the input models are stored as networks of objects (like in the Vanderbilt modeling tools), than the input to the generator is trivial, otherwise the model text needs to be parsed and a graph built. Similarly, the output of the rewriting engine is a graph, which is constructed during the rewriting process, and which is being "printed" into the final target form.

These observations suggest that generators can be constructed using a model-based approach by modeling the "input language" and the "output language" of the engine, and the mapping implemented by the engine, and then synthesizing the generator's from these models.

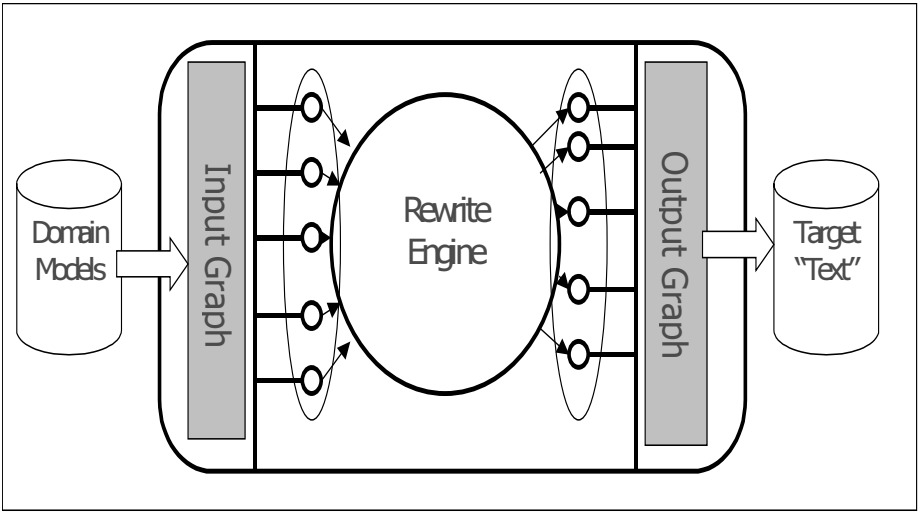


Fig. 4. Generator Architecture

The input language for the generators is defined by the meta-model of the domain specific modeling language (i.e. by its abstract syntax). Although the required outputs are highly problem and platform specific, in a large category of problems the generators translate the domain models into a well formed output model, which can also be defined in terms of their meta-model. Clearly, the tool integration architecture described above is one example for the extensive use of *model-based generators.* The semantic translators used there were designed and implemented by using an early manifestation of this model-based generator technology [13].

The second important category of applications of model-based generators is the assignment of dynamic (or execution) semantics to domain models. As it was mentioned before, a somewhat limited, but useful technique to define dynamic semantics is to provide the $I: A \rightarrow A'$ mapping between the abstract syntax A of a modeling language L and the abstract syntax A' of another modeling language L' with a well defined dynamic semantics. In this case, the interpretation of L is defined as the composition of I and I'. (For example, the execution semantics of a hierarchical dataflow modeling language may be defined by specifying its mapping on a synchronous dataflow computational model, such as given in [2].) This mapping can also be implemented by using the model-based generators technology of MIC.

The model-based development of translators is centered on modeling what the rewriting engine does. In [13] an approach based on the traversal-visitor pattern was employed. A high-level language captured the traversal sequences what the rewriting engine uses to visit the nodes of the input graph, and an imperative approach was used to construct the output graph. The technique demonstrated increase in programmer productivity, but it did not lend itself well to the formal analysis of the translation mapping (which is needed when properties of translators are to be verified). On-going research projects investigate the use of techniques based on graph rewriting.

7 Conclusion

Embedded systems are the most aggressively growing applications of computing. The design technology for embedded systems is significantly different from both "pure" software design and systems design, which creates major challenges for researchers and practitioners. Typically, a number of formal domain-specific modeling languages are needed for the precise specification and efficient design and implementation of embedded systems. The meta-model language approach to the specification of domain-specific modeling languages and modeling environment generation has several advantages. By specifying the entities, relationships, attributes, and constraints at the meta-modeling level, the domain specific modeling languages can be described with mathematical precision, can be safely evolved over time, and can be used to configure a generic modeling environment for use in designing embedded systems within a particular domain. Once configured, the domain modeling environment ensures valid model creation through the use of constraint specifications obtained from the meta-model, enforcing the formal static semantics of the domain at model editing time. Furthermore, if the meta-model captures the specification of the mapping of domain models into the "instruction-set" of an execution platform, it can be used to automatically synthesize a transformation engine to facilitate that mapping.

References

[1] J. Sztipanovits and G. Karsai: "Model-Integrated Computing," *IEEE Computer,* April, 1997 (1997) 110-112
[2] R. Alur, T. A. Henzinger, G. Laferriere, G. J. Pappas: "Discrete Abstractions of Hybrid Systems," Proc. of the IEEE, Vol. 88, No. 7, (2000) 984
[3] E. A. Lee and A. Sangiovanni-Vincentelli: "A Framework for Comparing Models of Computations," IEEE Trans. CAD Integrated Circuits and Systems, (1998) 1217-1229
[4] OMG Unified Modeling Language Specification, Version 1.3. June, 1999 (http://www.rational.com/media/uml/)
[5] Modelica Documents: http://www.modelica.org/documents.shtml
[6] Rosetta Documents: http://www.sldl.org/
[7] G. Karsai, G. Nordstrom, A. Ledeczi, J. Sztipanovits: "Towards Two-Level Formal Modeling of Computer-Based Systems," Journal of Universal Computer Science, Vol.6, No. 10, (2000) 1131-1144
[8] CDIF Meta Model documentation. http://www.metamodel.com/cdif-metamodel.html
[9] Ledeczi A., Maroti M., Karsai G., Nordstrom G.: "Metaprogrammable Toolkit for Model-Integrated Computing", Proceedings of the Engineering of Computer Based Systems (ECBS) Conference, Nashville, TN, March, 1999 (1999) 311-319
[10] Object Constraint Language Specification, ver. 1.1, Rational Software Corporation, et al., Sept. 1997. (1997)
[11] Nordstrom G., Sztipanovits J., Karsai G., Ledeczi, A.: "Metamodeling - Rapid Design and Evolution of Domain-Specific Modeling Environments", Proceedings of the IEEE ECBS'99, Nashville, TN, April, 1999. (1999) 68-75
[12] Generic Modeling Environment documents, http://www.isis.vanderbilt.edu/projects/gme/Doc.html
[13] Karsai, G., Gray,J.: "Design Tool Integration: An Exercise in Semantic Interoperability," Proceedings of the Engineering of Computer Based Systems (ECBS) Conference, Edinburgh, UK, March, 2000. (2000) 272-278
[14] Czarnecki, K. Eisenecker, U: Generative Programming - Methods, Tools, and Applications, Addison-Wesley, 2000.

Embedded Software in Network Processors – Models and Algorithms

Lothar Thiele, Samarjit Chakraborty, Matthias Gries,
Alexander Maxiaguine, and Jonas Greutert

Computer Engineering and Networks Laboratory
Swiss Federal Institute of Technology (ETH) Zürich
CH-8092 Zürich, Switzerland
{thiele, samarjit, gries, maxiagui, greutert}@tik.ee.ethz.ch
http://www.tik.ee.ethz.ch

Abstract. We introduce a task model for embedded systems operating on packet streams, such as network processors. This model along with a calculus meant for reasoning about packet streams allows a unified treatment of several problems arising in the network packet processing domain such as packet scheduling, task scheduling and architecture/algorithm explorations in the design of network processors. The model can take into account quality of service constraints such as data throughput and deadlines associated with packets. To illustrate its potential, we provide two applications: (a) a new task scheduling algorithm for network processors to support a mix of real-time and non-real-time flows, (b) a scheme for design space exploration of network processors.

1 Introduction

The need for intelligent and flexible network packet processing at high data rates, required by many emerging applications, have led to the development of a new class of devices called *network processors* (NPs). NPs are highly programmable dedicated processors optimized to perform packet processing functions, and will become critical components of next-generation networking equipments. Although there is a wide variety of application areas that are addressed by NPs, their task can generally be viewed as manipulation of packets. These include packet processing tasks (such as classification, forwarding, "deep" packet analysis, data stream manipulation, security, TCP termination, etc.) and traffic management tasks (such as bandwidth management, load balancing, admission control, etc.).

The functionality of an NP and hence its architecture and implementation depend to a very large extent on its placement in the Internet hierarchy. Those deployed in the access network are usually required to support a wide and varied range of packet processing functions, but at relatively low data rates (of around 100 Kbps per end-user for standard networks and 1-10 Mbps for networks based on emerging technologies like DSLs), and hence are ideal for a largely software based implementation. NPs placed in core or backbone networks have to handle

T.A. Henzinger and C.M. Kirsch (Eds.): EMSOFT 2001, LNCS 2211, pp. 416–434, 2001.
© Springer-Verlag Berlin Heidelberg 2001

much higher data rates (in the range of several Gbps) which therefore restricts their processing capabilities. In this case many core functionalities are implemented in software on a general purpose processor and other operations are delegated to dedicated coprocessors.

Irrespective of the type and functionality of an NP, because of flexibility reasons, software forms an integral part of it. In this direction, very recently several papers proposed different software architectures for flexible and configurable routers which can easily be programmed and extended to support new functionality rather than only routing packets (see [14,16] and the references therein). However, till date there has been no formal and unified study of this subject. All the previous papers dealt with this topic mainly from a software engineering perspective. There is a large body of literature on packet scheduling, but network processing is much more than that. In this paper we attempt to initiate a formal study of packet processing devices such as NPs. As a first step in this direction, we outline a framework for embedded systems operating on packet streams and based on it provide a unified treatment of some problems arising in this area. Our framework is motivated to some extent by some recent models used in packet scheduling (such as [7,8,12]) and primarily consists of

- a task and resource model for network processors and
- a calculus which allows to reason about packet streams and their processing.

As an application of the above, we consider two examples: the first is that of task scheduling in an embedded packet processor, and the second is related to hardware-software interactions where we illustrate the potential of our model and calculus for estimating delays and memory requirements in the context of a design space exploration of NPs. The next two sections introduce our model and the calculus, following which we describe the two examples of scheduling and design space exploration in Sections 4 and 5.

2 Model of Computation for Packet Processing

Our model for describing typical task structures and their mapping to hardware and software resources is not very different from the specification of conventional real-time systems, see e.g. [9]. Nevertheless, there are some notable differences, for example the way in which the capabilities of processing devices and sequences of events or packets are specified. These form the basis of the described unification of models for, for example packet and task scheduling.

It may be noted that here we will define a basic model only. Depending on the particular design task and the envisioned level of abstraction, additional information can easily be added.

Definition 1 (Task Structure). *The task structure of a network processor consists of a set of flows $f \in F$ and a set of tasks $t \in T$. To each flow f there is associated a connected, directed and acyclic task graph $G(f)$. It consists of a set of task nodes $T(f) \subseteq T$ and a set of directed edges $E(f) \subseteq T(f) \times T(f)$. Each task graph has a unique source node $s(f) \in T(f)$ having no incoming edges.*

Figure 1 shows a relatively simple task structure of a sample NP that will be used throughout the paper. There are twenty five tasks (denoted by t) which perform general packet processing functions and operations dedicated to encryption/decryption and voice processing. Depending on the flow to which a packet belongs, different sequences of tasks are executed, see the dotted lines. In the above terminology, we have five flows $f \in F$ and the corresponding task graphs $G(f)$ are simply chains.

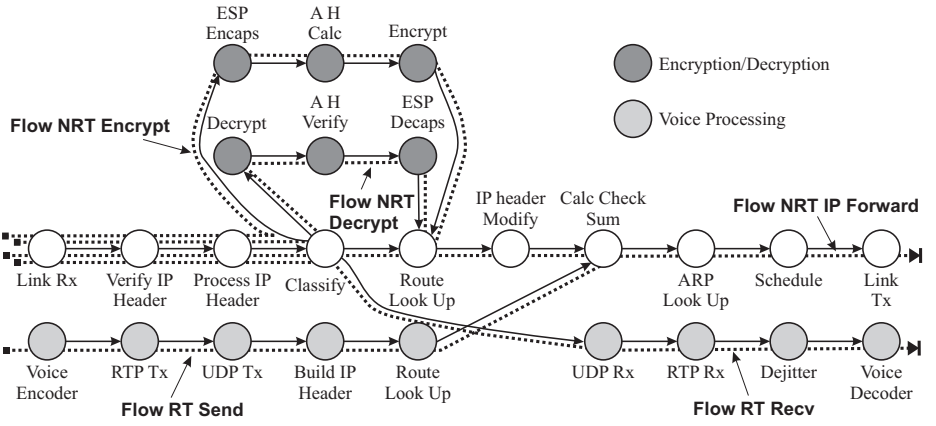


Fig. 1. Example of a task graph corresponding to a simple network processor, see Definition 1.

In addition to the above definition of the basic task structure, it is useful to define a simple resource model consisting of a set of computing resources which are able to execute the tasks defined in the task structure.

Definition 2 (Resource Structure). *Given a set of resources $s \in S$, the function cost $: S \to \mathbb{R}^{\geq 0}$ denotes the (relative) cost of the corresponding resource. The mapping relation $M \subseteq T \times S$ defines possible mappings of tasks $t \in T$ to resources, i.e. if $(t, s) \in M$ then t can be executed on s.*

In order to enable a performance analysis of the above defined resource structure, we now introduce some additional functions.

Definition 3 (Timing Properties). *To each flow $f \in F$ there is associated an end-to-end deadline $d : F \to \mathbb{R}^{\geq 0}$. If a task f can be executed on a resource s, then it creates a "request", i.e. for all $(t, s) \in M$ there exist a request $w(t, s) \in \mathbb{R}^{\geq 0}$.*

The above definitions can be interpreted as follows: To each flow f there is associated a set of tasks T. If a packet belonging to this flow arrives at the processing device, the tasks corresponding to the flow are executed respecting the

partial order defined by the edges. To each of these packets there is associated a (possibly infinite) end-to-end deadline $d(f)$. The execution of the complete task graph corresponding to the packet must be finished within at most $d(f)$ time units after the packet arrival. The resource nodes s denote the available computing resources. If a task is executed on a computing resource, it creates a request w, for example measured in number of instructions.

As an example, Figure 2 shows a part of the resource structure of our simple network processor. Only five of the eight resource nodes $s \in S$ and four of the twenty five tasks $t \in T$ are shown along with their associated costs *cost*, mapping relations $(t, s) \in M$ and requests $w(t, s)$.

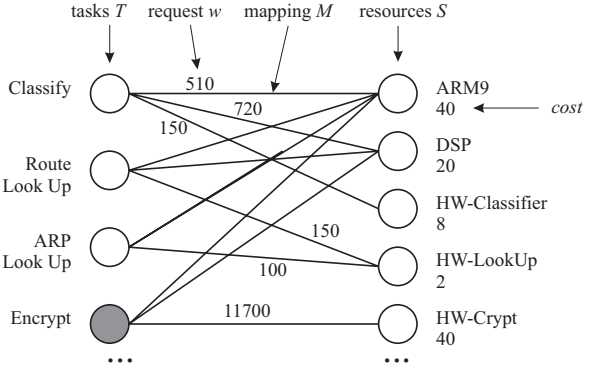


Fig. 2. Example of a resource structure as defined in Definition 2.

Associated to each flow are also some arrival curves modelling the incoming event/packet sequences, and to each resource some service curves modelling its processing capabilities; details of these will be described in the next section.

The above model abstracts important properties of a task structure or resource structure such as communication nodes, interconnection structure, memory or data nodes, power consumption, cost functions and bandwidth requirements. It should be obvious that the model can easily be extended. Finally, it may be noted that the above model closely resembles the one that was used for a design space exploration of hardware/software architectures in [6].

3 Modeling Discrete Event Streams and Systems

3.1 Basic Models

Traditionally event streams are modeled statistically. Methods that have been applied are well known in the area of queuing theory. However, If we are interested in hard bounds instead of failure probabilities, it is more suitable to use a more appropriate characterization of discrete event models and systems.

In order to unify the methods commonly used in the domain of communication networks with those used in operating systems and real-time scheduling, we adopt the concept of arrival curves and service curves, see [12]. Recently, this approach has been formalized [7] and put into a network theory context based on a linear algebra for discrete event systems, see e.g. [3]. This approach has also been used to derive efficient packet scheduling algorithms, see e.g. [1].

Definition 4 (Arrival and Service Function). *An event stream can be described by an arrival function R where $R(t)$ denotes the number of events that have arrived in the interval $[0,t)$. A computing resource can be described by a service function C where $C(t)$ denotes the number of events that could have been served in the interval $[0,t)$.*

Depending on the context in which these functions are used, the number of events may model the number of packets, the number of bytes or the number of instructions to be performed. Obviously, the functions $C(t)$ and $R(t)$ are non-decreasing and can be used to accurately describe the incoming events and the processing capabilities. The abstraction used in this paper is based on deterministic bounds on the corresponding behavior.

Definition 5 (Arrival and Service Curves). *The upper and lower arrival curves $\alpha^u(\Delta), \alpha^l(\Delta) \in \mathbb{R}^{\geq 0}$ of an arrival function $R(t)$ satisfy*

$$\alpha^l(t-s) \leq R(t) - R(s) \leq \alpha^u(t-s) \qquad \forall s,t : 0 \leq s \leq t$$

The upper and lower service curves $\beta^u(\Delta), \beta^l(\Delta) \in \mathbb{R}^{\geq 0}$ of a service function $C(t)$ satisfy

$$\beta^l(t-s) \leq C(t) - C(s) \leq \beta^u(t-s) \qquad \forall s,t : 0 \leq s \leq t$$

Similar functions have been used in network calculus [7] and its applications in hard real-time systems, see [23,24] and [29]. Therefore, we will not repeat those results here. However, it may be useful to note a simple interpretation of the above definitions. The values $\alpha^l(\Delta)$ and $\alpha^u(\Delta)$ can be interpreted as the minimum and maximum number of events arriving within any time interval of length Δ, respectively. Therefore, given R, the corresponding curves can be computed using $\alpha^l(\Delta) = \min_{u \geq 0}\{R(\Delta+u) - R(u)\}$ and $\alpha^u(\Delta) = \max_{u \geq 0}\{R(\Delta+u) - R(u)\}$. In a similar way, the service curves $\beta^l(\Delta)$ and $\beta^u(\Delta)$ can be interpreted as the minimum and maximum available computing service within any time interval of length Δ, respectively. Therefore, given the service function C, we have $\beta^l(\Delta) = \min_{u \geq 0}\{C(\Delta + u) - C(u)\}$ and $\beta^u(\Delta) = \max_{u \geq 0}\{C(\Delta + u) - C(u)\}$.

Figure 3 shows examples of upper and lower arrival curves and service curves. Referring to the task and resource structures defined in Section 2, we can now extend the characterization of flows and resource nodes by the above defined curves.

Definition 6 (Curves and Flows). *To each flow f there are associated upper and lower arrival curves $\alpha^u(\Delta)$ and $\alpha^l(\Delta)$, respectively. To each resource s there are associated upper and lower service curves $\beta^u(\Delta)$ and $\beta^l(\Delta)$, respectively.*

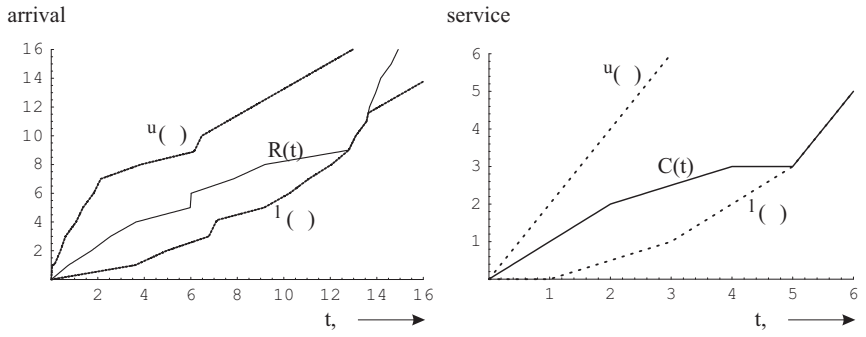


Fig. 3. The left hand side shows an arrival function that has been generated from a finite size exponentially distributed event sequence with mean 1. The right hand side may model a service curve from a resource node that processes 1 event per time unit in the interval $[0, 2]$, only $1/2$ a unit in $[2, 4]$ (e.g. because of a processor share with another task), has no computing capability in $[4, 5]$ (e.g. because of an interrupt handler), and can process 2 events per time unit starting from time 5.

In order to understand the processing of event streams using resource nodes, we start with a simple unit receiving only one event stream, see also [23], [24] and [29]. In this case, the number of events is supposed to denote the computation request of the incoming stream.

Definition 7 (Function Processing). *Given a resource node s with its corresponding service function $C(t)$ and an event stream described by the arrival function $R(t)$ being processed by s, we then have*

$$C'(t) = C(t) - R'(t)$$

$$R'(t) = \min_{0 \le u \le t} \{R(u) + C(t) - C(u)\}$$

where $C'(t)$, $R'(t)$ denote the remaining service function of the resource node and the amount of computation delivered to the processed event stream, respectively.

The first equation just states, that the remaining amount of computation $C'(t)$ (for example in terms of the number of instructions) available until time t is the initial amount reduced by the amount spent for the processed events. In order to understand the meaning of the second equation, note that $R'(t)$ is the maximum value that satisfies $R'(t) - R(u) \le C(t) - C(u)$ for any $u \le t$. The left hand side denotes the number of events that arrived after time u and are processed before time t. This is clearly smaller than the available computing resource in the interval $[u, t]$.

Figure 4 shows the processing network view of a simple resource node. As we are interested in the processing of whole classes of input streams, we will now describe the processing of event streams in terms of the upper and lower curves.

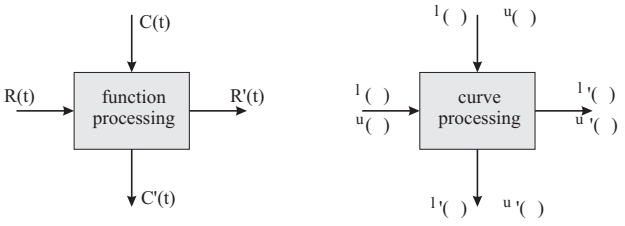


Fig. 4. Diagrams which show the processing of event streams by resource nodes.

These results substantially generalize and sharpen the bounds obtained in [29, 8], as *both* lower *and* upper bounds are involved.

Proposition 1 (Curve Processing). *Given an event stream described by the arrival curves $\alpha^l(\Delta)$ and $\alpha^u(\Delta)$ and a resource node described by the service curves $\beta^l(\Delta)$ and $\beta^u(\Delta)$, then the following expressions bound the remaining service function of the resource node and the arrival function of the processed event stream:*

$$\alpha^{l'}(\Delta) = \min_{0 \le u \le \Delta} \left\{ \alpha^l(u) + \beta^l(\Delta - u), \beta^l(\Delta) \right\}$$

$$\alpha^{u'}(\Delta) = \min_{0 \le u \le \Delta} \left\{ \max_{v \ge 0} \left\{ \alpha^u(u+v) - \beta^l(v) \right\} + \beta^u(\Delta - u), \beta^u(\Delta) \right\}$$

$$\beta^{l'}(\Delta) = \max_{0 \le u \le \Delta} \{ \beta^l(u) - \alpha^u(u) \}$$

$$\beta^{u'}(\Delta) = \max_{0 \le u \le \Delta} \{ \beta^u(u) - \alpha^l(u) \}$$

Figure 5 shows the application of Proposition 1 to the lower and upper arrival and service curves given in Figure 3.

Using well known results from the area of communication networks, see e.g. [8], the bounds derived in Proposition 1 can be used to determine the maximal delay of events and the necessary memory required to store waiting events. The number of events still waiting to be processed at time t is $R(t) - R'(t)$. The delay that an event entering the resource node at time t will experience can be given by $d(t) = \min\{\tau \ge 0 : R(t) \le R'(t+\tau)\}$. The following two equations give bounds on both quantities:

$$d(t) \le \max_{u \ge 0} \left\{ \min\{\tau \ge 0 : \alpha^u(u) \le \beta^l(u+\tau)\} \right\}$$

$$R(t) - R'(t) \le \max_{u \ge 0} \{ \alpha^u(u) - \beta^l(u) \}$$

In other words, the delay can be bounded by the maximal horizontal distance between curves $\alpha^u(\Delta)$ and $\beta^l(\Delta)$ whereas the backlog is bounded by the maximal vertical distance between them.

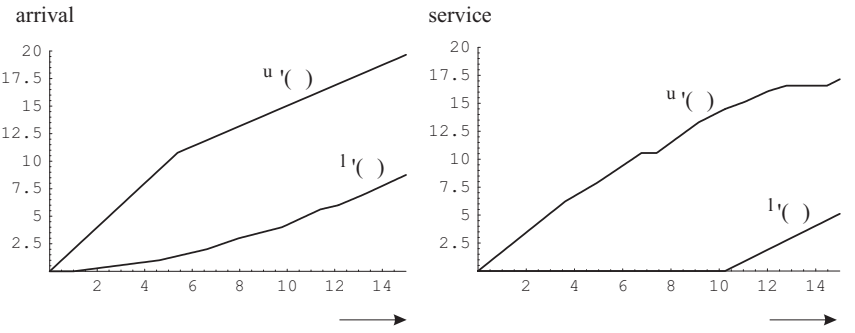


Fig. 5. Processing of the curves given in Figure 3 in accordance with Proposition 1.

In [23,24] it has been shown that the above event model generalizes some recent attempts to describe bursty tasks in real-time systems, e.g. [21,4]. In addition, the above bounds lead to schedulability tests which are equivalent to those known in case of fixed priority and earliest deadline first scheduling, e.g. [18,17,2].

Based on these results, we can now formulate different problems in the network packet processing domain on a unified basis. As a simple illustration of the above results we consider a fixed priority scheme in the next subsection.

3.2 Simple Processing Network

Following our main approach, to investigate the effect of a scheduling algorithm, we represent the "scheduling block" in the form of a network consisting of nodes which operate on event streams. In addition, there are (virtual) resource streams which model the available resources, see Figure 4. This way, it is possible to describe and analyze packet scheduling, task scheduling and hierarchical approaches. As a simple example, we consider a fixed priority scheme.

Let us start from the models defined in Definitions 1 to 5. In case of a fixed priority scheme, let us suppose that there is a set of flows $f_1, ..., f_n$ with associated event streams $\overline{R}_1(t), ...\overline{R}_n(t)$ ordered according to decreasing priority, i.e. $\overline{R}_1(t)$ has highest priority. In addition, for each event of flow f_i, a task t_i must be executed on one resource s with associated request $w(t_i, s)$ or w_i in short. The event stream associated to a flow f_i is described by arrival curves $\overline{\alpha}_i^u(\Delta)$ and $\overline{\alpha}_i^l(\Delta)$. The resource node s is characterized by the service curves $\beta_i^u(\Delta)$ and $\beta_i^l(\Delta)$.

Because of the fixed priority scheme, the resource stream serves the flows in the order of decreasing priority by the use of Definition 7 and Proposition 1. In order to have compatible units, we first have to multiply the arrival functions and arrival curves with the request for each event, namely w_i. Correspondingly, the request stream leaving the resource must be divided by w_i. If a unit using these events or packets can start only when the whole task has finished on the

preceding unit, we need to apply the floor-function to the outgoing streams, i.e. $\overline{R}_i'(t) = \lfloor R_i'(t)/w_i \rfloor$. In a similar way, the outgoing arrival curves are transformed according to $\overline{\alpha}_i^{u\prime}(\Delta) = \lceil \alpha_i^{u\prime}(\Delta)/w_i \rceil$ and $\overline{\alpha}_i^{l\prime}(\Delta) = \lfloor \alpha_i^{l\prime}(\Delta)/w_i \rfloor$. These curves correctly bound $\overline{R}_i(t)$ as one can show that $\lfloor a \rfloor - \lfloor b \rfloor \leq \lfloor a - b \rfloor$ and $\lceil a \rceil - \lceil b \rceil \geq \lceil a - b \rceil$. In addition to the relations shown in Proposition 1 (which hold for all flows $1 \leq i \leq n$), we have the following equations which describe the processing of event streams using fixed priority scheduling with preemption:

$$\alpha_i^u(\Delta) = w_i \cdot \overline{\alpha}_i^u(\Delta) \quad , \quad \alpha_i^l(\Delta) = w_i \cdot \overline{\alpha}_i^l(\Delta)$$

$$\overline{\alpha}_i^{u\prime}(\Delta) = \lceil \alpha_i^u(\Delta)/w_i \rceil \quad , \quad \overline{\alpha}_i^{l\prime}(\Delta) = \lfloor \alpha_i^l(\Delta)/w_i \rfloor$$

$$\beta_1^u(\Delta) = \beta^u(\Delta) \quad , \quad \beta_i^u(\Delta) = {\beta_{i-1}^u}'(\Delta) \quad \forall 1 < i \leq n \quad , \quad \beta^{u\prime}(\Delta) = \beta_n^{u\prime}(\Delta)$$

$$\beta_1^l(\Delta) = \beta^l(\Delta) \quad , \quad \beta_i^l(\Delta) = {\beta_{i-1}^l}'(\Delta) \quad \forall 1 < i \leq n \quad , \quad \beta^{l\prime}(\Delta) = {\beta_n^l}'(\Delta)$$

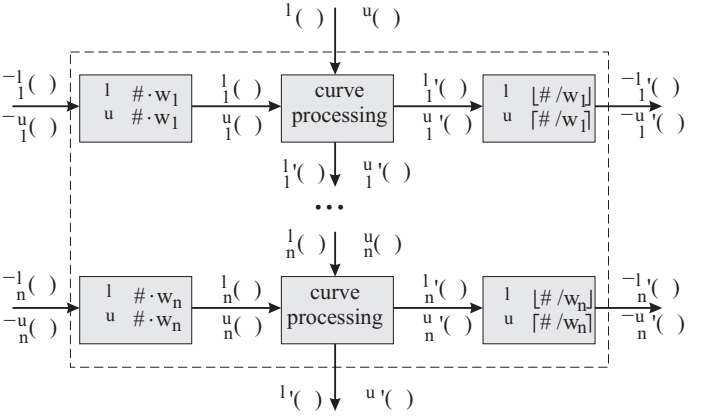


Fig. 6. Diagram showing a processing network for the processing of a set of event streams by a resource node using fixed priority preemptive scheduling.

Finally note, that the remaining resource service $\beta'(\Delta)$ can be used to service other flows with a different scheduling scheme in a hierarchical manner. The event streams ${\overline{R}_i}'(t)$ can enter further resource nodes which are responsible for executing other tasks $t \in T$ or for performing link scheduling.

In a similar way, one could also describe other scheduling schemes such as Generalized Processor Sharing, Weighted Fair Queueing, First Come First Served and Earliest Deadline First. As it has also been pointed out in the context of communication networks [8], the determination of accurate bounds in these cases is still an active area of research.

Using the results described in the previous section, we can also bound the delay caused by the processing and the necessary memory in terms of the number

packets waiting to be processed. These quantities can be used to determine the necessary memory inside a network processor and the worst-case delay which packets might experience. An application of this technique will be described in Section 5.

4 Task Scheduling in Network Processors

As a first application of the models described in the last two sections, we consider the problem of task scheduling in an NP where different packet processing functions are implemented as programs running on a general-purpose processor. A detailed formal description of the problem is given below; essentially the problem is to schedule the CPU cycles of the processor to process a mix of real-time and non-real-time packets such that all real-time packets meet their deadlines and the non-real-time packets experience the minimum processing delay. There has not been much work on issues related to task scheduling in software-based routers/NPs (see [27]), and most of the previous work on scheduling a mix of real-time and non-real-time tasks is based on CPU reservations (see [5] and the references therein). Our algorithm in this section is on the contrary based mainly on Earliest Deadline First scheduling, and is motivated by algorithms for scheduling a mix of periodic and aperiodic tasks (see [10] and the references therein) and approaches used for packet scheduling (such as [12]).

4.1 Scheduling a Mix of Real-Time and Non-real-Time Flows

Given a set of flows F, we consider it to be composed of two disjoint subsets F_{RT} and F_{NRT}. All flows $f_i \in F_{RT}$ are real-time flows having finite end-to-end deadlines $d(f_i)$ (see Definition 3). For each packet of f_i the execution of the corresponding task graph $G(f_i)$ must complete within $d(f_i)$ time units after the arrival of the packet. Real-time flows might represent traffic such as voice or video streams. Flows belonging to F_{NRT} have no time constraints (i.e. they have infinite deadlines) and are used to model packet streams corresponding to bulk data transfers such as FTP; we refer to these flows as non-real-time flows. We assume each real-time flow f_i to be constrained by an upper arrival curve α_i^u. The processing cost of each packet of a flow f_k (both real-time and non-real-time) on a single resource s (the CPU) is denoted by $w(f_k)$, where $w(f_k) = \sum_{t \in T(f_k)} w(t, s)$ (see Definitions 1 and 3).

The objective of our scheduling algorithm is multi-fold: (i) to guarantee that all real-time packets meet their associated deadlines, (ii) that the non-real-time packets experience the minimal possible delay, and (iii) we associate with each non-real-time flow f_j a weight ϕ_j and require that the remaining CPU power, given by the lower service curve $\beta^{l'}$, after processing all real-time flows is divided among the non-real-time flows in proportion to their corresponding weights i.e. the number of CPU cycles on the average allocated to flow f_j is $\frac{\phi_j}{\sum_{f_k \in F_{NRT}} \phi_k} \beta^{l'}$. This offers the provision for allocating different non-real-time flows a relative importance.

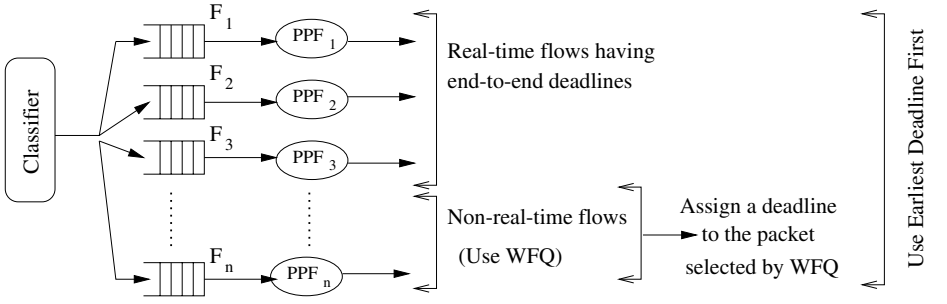


Fig. 7. A new task scheduler based on a hierarchy of WFQ and EDF.

An overview of our scheduling algorithm meeting the above mentioned goals is given in Figure 7. It is composed of a Weighted Fair Queuing (WFQ) [25] scheduling algorithm organized in a hierarchical fashion with an Earliest Deadline First (EDF) scheduler on the top. A WFQ scheduler is based on the Generalized Processor Sharing [15] scheduling algorithm and guarantees that the relative share of the CPU bandwidth among the non-real-time flows follows the proportions dictated by the respective weights associated with them. $F_1, \ldots, F_n$ indicate packet queues (which are filled by the packet classifier) corresponding to the different flows, some of which are real-time and the rest being non-real-time. $PPF_1, \ldots, PPF_n$ denote chains of packet processing functions corresponding to the different flows (as shown in Figure 1).

Given the upper arrival curve α_i^u of each real-time flow f_i, we can compute the processor demand resulting from the set of all real-time flows F_{RT}. Towards this, for each α_i^u we compute a curve $\overline{\alpha}_i^u$ given by

$$\overline{\alpha}_i^u(\Delta) = \begin{cases} 0 & \text{if } \Delta \leq d(f_i) \\ w(f_i)\alpha_i^u(\Delta - d(f_i)) & \text{otherwise} \end{cases}$$

The processor demand by the set of all real-time flows F_{RT} within any time interval of length Δ if all packets have to meet their associated deadlines is then given by a curve α_{RT} where $\alpha_{RT}(\Delta) = \sum_{f_i \in F_{RT}} \overline{\alpha}_i^u(\Delta)$. It is possible to prove (using arguments similar to those given in [11]) that the set of real-time flows F_{RT} is preemptively schedulable on a single processor (such that all packets meet their respective deadlines) if and only if $\beta^l(\Delta) \geq \alpha_{RT}(\Delta)$ for all $\Delta \geq 0$, where the lower service curve β^l is as described in Definition 5. This schedulability test can form the basis for admission control for real-time flows. Now we are in a position to state our algorithm.

Given the set of real-time flows and a lower service curve β^l, we first compute the curve α_{RT} as described above and a curve α_{NRT} which is defined as $\alpha_{NRT}(\Delta) = \min_{t \geq \Delta}\{\beta^l(t) - \alpha_{RT}(t)\}$. The WFQ scheduler shown in Figure 7 computes an ordering of the non-real-time packets according to which they should be processed to respect their relative CPU reservations.

For each packet selected by the WFQ scheduler for processing, if the packet belongs to flow f_i and has a processing requirement of $w(f_i)$ then it is assigned a deadline $d(f_i) = \min\{\Delta : \alpha_{NRT}(\Delta) \geq w(f_i)\}$. Obviously, at any instant of time there is exactly one non-real-time packet that is processed and is assigned a deadline according to the scheme just described. The top level EDF scheduler preemptively schedules this packet along with all the real-time packets (which already have associated deadlines) according to the earliest deadline first scheduling strategy.

Proposition 2 (Schedulability). *If the set of real-time flows is preemptively schedulable (i.e. there exists some scheduling algorithm using which all real-time packets can be processed within their respective deadlines) then our algorithm also schedules the real-time flows such that all deadlines are met.*

Proof. If $d(f_i)$ is the deadline associated with the non-real-time packet having a processing requirement of $w(f_i)$, then for this packet along with all the real-time flows to be schedulable, the following must hold:

$$\beta^l(\Delta) \geq \alpha_{RT}(\Delta) + w(f_i), \quad \forall \Delta \geq d(f_i)$$

which is equivalent to requiring that

$$w(f_i) \leq \beta^l(\Delta) - \alpha_{RT}(\Delta), \quad \forall \Delta \geq d(f_i)$$

The above condition is obviously satisfied if

$$w(f_i) \leq \min_{\Delta \geq d(f_i)} \{\beta^l(\Delta) - \alpha_{RT}(\Delta)\}$$

Hence the proposition follows.

The above scheduling algorithm is preemptive, meaning that the processing of each packet can be preempted at any stage of its task graph and then resumed later. In general such arbitrary preemptions might be costly for any practical implementation, and the only allowable preemption points might be at the end of each node of the task graph (i.e. a packet processing can not be preempted in the middle of executing any node of the corresponding task graph). However, assuming that the execution time of each node is small compared to the total execution time of the whole task graph, the above analysis gives a good approximation of an algorithm where preemption is allowed only at the end of each node.

4.2 Experimental Evaluation

We have implemented and evaluated our algorithm using the Moses tool-suite [22] which is used for modelling and simulation of discrete event systems. Our experimental setup consists of six flows, of which three are real-time and three others are non-real-time flows. Each flow is specified by a TSpec [28] with all its

parameters specified in terms of packets rather than bytes. A TSpec is described by a conjunction of two token buckets and an incoming packet complies with the specified profile only if there are enough tokens in both the buckets (we refer to them as the *avg. bucket* and the *peak bucket* in Table 1). The arrival curves of the different flows were chosen to reflect a typical access network scenario. In particular, the high-volume flows such as video and FTP are bounded by an average rate of around 10 MBits/s. Table 1 gives the specifications of the different flows.

Table 1. Specifications of the real-time (1-3) and non-real-time flows (4-6).

	deadline (Flows 1-3) *WFQ weight* (Flows 4-6) $[ms]$ for Flows 1-3	*avg. bucket* (burstiness, rate) $[pkts, pkts/s]$	*peak bucket* (burstiness, rate) $[pkts, pkts/s]$	*CPU demand* $[cycles]$
Flow 1	2	$(150, 300)$	$(1, 1000)$	40000
Flow 2	10	$(40, 840)$	$(1, 4200)$	600
Flow 3	1	$(3, 300)$	$(1, 1000)$	20000
Flow 4	0.5	$(400, 1000)$	$(1, 5000)$	600
Flow 5	0.2	$(50, 150)$	$(1, 700)$	4000
Flow 6	0.1	$(8, 30)$	$(1, 700)$	40000

Flows 1-3 are real-time flows and Flows 4-6 are non-real-time flows. Flow 1 represents some transactions with encryption, Flow 2 represents a video traffic and Flow 3 some voice encoding. Among the non-real-time flows, Flow 4 represents an FTP download, Flow 5 represents HTTP page downloads, and Flow 6 represents email traffic with encryption.

For the above specified flows, we have compared our algorithm with a scheduling algorithm consisting of a combination of plain EDF for real-time packets and WFQ for non-real-time packets. The ordering of the non-real-time packets is due to the WFQ scheduler and they are processed only when there are no backlogged real-time packets. The real-time packets are scheduled according to EDF scheduling. To avoid any possible confusion (because of the similar terminology and setup), we remind the reader that here we are describing a task scheduling and not a link scheduling algorithm, so the ordering of a packet stream is not used for putting them on the output link, but to execute the packet processing functions in that order.

Figure 8 shows an excerpt of a simulation, comparing the above algorithm with ours. The horizontal-axis shows the simulation time and the vertical-axis represents the delay experienced by a packet in getting processed. Any point on the horizontal-axis represents the completion time of a packet processing task and the corresponding point on the vertical-axis plots the delay experienced (completion time minus the arrival time) by this packet. Note that in the plain EDF + WFQ scheduler, packets from the real-time Flow 2 are processed much before their deadline. In our improved algorithm, the non-real-time Flows 4

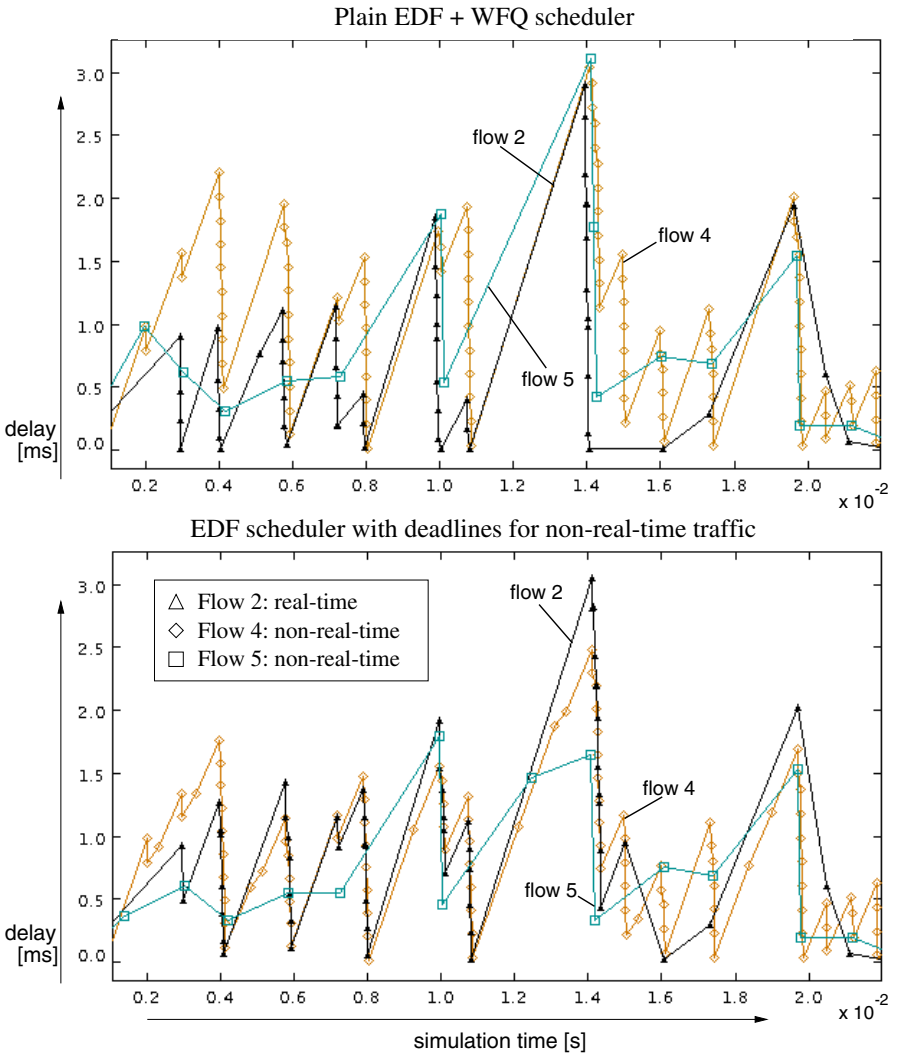


Fig. 8. Comparison of delays experienced by three selected flows.

and 5 experience shorter response times at the expense of slightly higher delays for Flow 2 (which nevertheless meets its deadlines). On the average, the response time of Flows 4 and 5 improve by around 20%, at the expense of the response time of Flows 1, 2, and 3 increasing by around 2%, 32%, and 3% respectively. Due to its high CPU demand, the response time of Flow 6 (not sketched in the figure) does not improve much since the deadlines assigned by our algorithm to packets of this flow fall behind those of the real-time packets, and therefore they are executed after the real-time packets as in the case of the plain EDF + WFQ scheduler.

5 Design Space Exploration

The final section of this paper is devoted to the design space exploration of network processors, where we show how the basic models and methods described in the previous sections can be used to perform an exploration of hardware and software architectures. It is expected that the next generation of network processors will consist of general purpose processing units and dedicated modules for executing run-time extensive functions. Therefore, the purpose of a high level exploration is to select appropriate functional units such that the performance of the processor is maximized under various constraints such as cost, packet delay and power consumption. Here we will concentrate on the following questions:

- How can we estimate the performance of a network processor?
- How can we estimate delay and memory consumption of a hardware/software architecture?

The approach taken here is influenced by our previous results in design space exploration of hardware/software architectures [30]. In particular, we will adopt the model based approach in combination with concepts of multi-objective optimization (see [19] and the references therein).

As described in the last section, we consider the set of flows F to be composed of two disjoint sets F_{RT} and F_{NRT}. The arrival and service curves associated to these flows are interpreted differently. Whereas in case of F_{RT} they describe the rates of the incoming packet streams, they are interpreted as relative rates in case of F_{NRT}. In other words, the rates of the packet streams can be described by the lower and upper arrival curves $\psi \cdot \alpha^l(f)$ and $\psi \cdot \alpha^u(f)$ for $f \in F_{NRT}$ where ψ is a positive scaling variable. The problem of design space exploration can now be formulated as the following multi-objective optimization problem:

> Allocate resource nodes $s \in S$ and bind the tasks $t \in T$ of the flows $f \in F$ to the allocated resource nodes such that ψ is maximized, other criteria such as cost, memory and power consumption are minimized and the deadlines $d(f)$ associated to the flows are satisfied.

The rationale for this performance criterion is based on the fact, that real-time flows such as video and voice streams often have a fixed rate which must be handled by the network processor. On the other hand, the throughput of other flows such as FTP, web traffic or email should be maximized. The relative arrival curves determine the relative weights for these kinds of flows.

For the network processor architecture we assume a heterogeneous set of components consisting of RISC processors, digital signal processors, micro-controllers and dedicated units for compute intensive tasks such as header parsing, table look-up and encryption/decryption. The purpose of the allocation is to select the "right" subset of this modules. In Definition 2, a function *cost* was defined which models the cost of allocating the corresponding unit. It may be noted, that much more complex measures could be added to tasks and resource nodes such as power consumption, and program and data memory. The following definition extends Definition 2 by formally defining the terms allocation and binding.

Definition 8 (Allocation and Binding). *The set $A \subseteq S$ denotes the set of allocated resource nodes $s \in A$. The binding of a task $t \in T$ to a resource $s \in S$ is a relation $B \subseteq T \times S$ where $B \subseteq M$ (see Definition 2), i.e. $(t, s) \in B$ if task t is executed on resource s.*

Based on our simple model for design space exploration of network processors, we can now determine the cost of an implementation as $Cost = \sum_{s \in A} cost(s)$. The problem of minimizing $Cost$ might involve a trade-off between several conflicting criteria such as memory and power requirements, and performance issues. Among the possible candidates for solving such a multi-objective optimization problem, where we are interested in obtaining all the trade-off or the so called Pareto points, are branch-and-bound methods and evolutionary algorithms [13].

For the purpose of illustrating the potential of the formal model introduced in this paper, we solved the design space exploration problem for the task structure shown in Figure 1 using a branch-and-bound search algorithm. It is based on a specification of the whole problem in the form of integer linear equations (see [20]). Only a portion of the resource structure used is shown in Figure 2. There are end-to-end deadlines d for the two real-time flows *RT Send* and *RT Receive*. The network processor architecture was designed for a 10-times higher rate of packets from the flow *NRT-IP-Forward* compared to each of the two other non-real-time flows. We measured *performance* by the scaling variable ψ which is maximized in the integer linear program. The total cost is the sum of costs of the allocated resource nodes and was introduced as a constraint into the ILP. An overview of the optimization setup is shown in Figure 9.

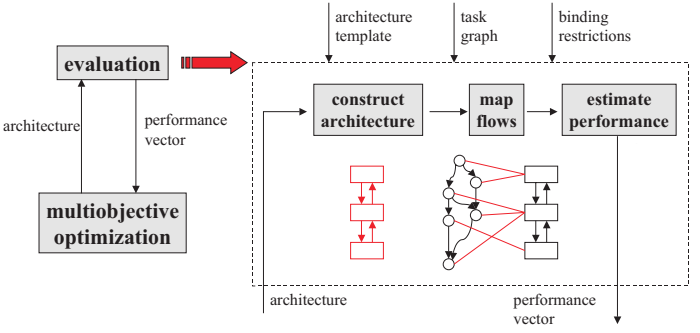


Fig. 9. Strategy of the design space exploration based on a multi-objective search algorithm and an estimation phase.

All the equations were generated using well known techniques [20,26] and the nonlinear function for taking task scheduling into account was modelled using piecewise linear approximations. Figure 10 shows the result of the design space exploration where the right hand side shows the different pareto points corresponding to the different architectures and the left hand side gives the

corresponding resource usages. Note that the hardware units are not loaded to 100% because of the end-to-end delay constraints.

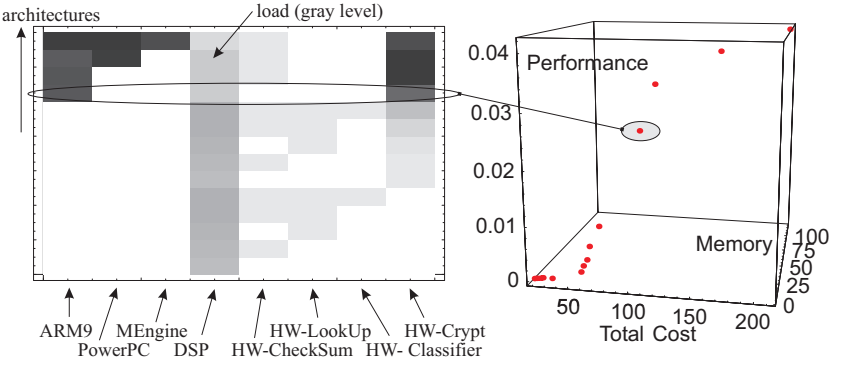


Fig. 10. Results from a design space exploration run. The left hand side shows in a diagram the hardware resources allocated for the Pareto points that have been found. The columns are related to the resource nodes and the rows correspond to the Pareto points, ordered with respect to the performance ψ. The gray level corresponds to the load of the respective hardware unit. The right hand side diagram represents the Pareto points in the performance-memory-cost coordinate system.

The above example took 521 variables, used 544 inequalities and required about 4 seconds per Pareto point on a 500 MHz Pentium III. We used the commercial branch and bound ILP solver CPLEX and Mathematica for generating the input and processing the output.

Scheduling was done in the simplest possible way, in particular using FCFS queuing for each resource, which obviously gives rough estimates only. Here it is possible to use the results obtained in Section 3.2. Let us suppose that the tasks $t \in T(f_i)$ of some flow f_i are assigned to allocated resource nodes $s \in A$, i.e. $(t, s) \in B$. In addition, let us suppose that the task graph $G(f_i)$ is a simple chain. Then we can model the flow of events and resources in form of a computation network (see Figure 11). In particular, we have two different kinds of streams through the network, namely event streams caused by the packet streams entering the system and resource streams determined by the service curves of the resource nodes. Each distinct input flow is mapped onto a directed path through the communication network. To each edge in the communication network, either lower and upper arrival curves or lower and upper service curves are assigned. Using the relations derived in Section 3.2, these quantities can be determined, unless there are no directed cycles for any flow f_i in the network. Finally, interesting performance measures such as delay and memory can be determined using the estimations on backlog and delay as given in Section 3.1. More work needs to be done in this direction to exploit the the full potential of the approach described in Section 3.2.

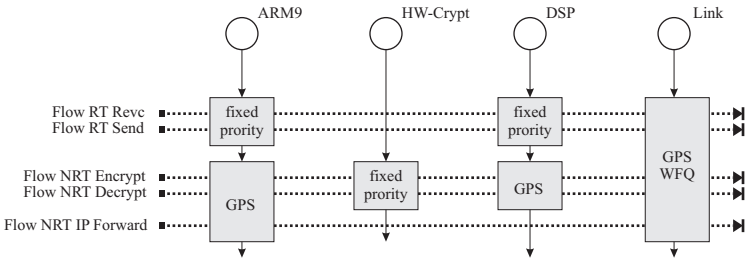


Fig. 11. Example of a simple processing network that is used to operate on packet streams such as those specified in the task graph in Figure 1. A final last resource node may model the link scheduler for the outgoing packet streams.

6 Concluding Remarks

We have shown that the design of embedded packet processing devices such as network processors pose interesting research issues related to models of computation, processing and communication networks, task and packet scheduling and design space exploration. Our approach was based on a unified modelling of these aspects.

It must be noted however, that there are still many open research issues related to the above approach. In particular, the full potential of the chosen task, resource and stream processing specification must still be explored.

References

1. R. Agrawal, R. L. Cruz, C. Okino, and R. Rajan. Performance bounds for flow control protocols. *IEEE/ACM Transactions on Networking*, 7(3):310–323, 1999.
2. N. Audsley, A. Burns, M. Richardson, and A. Wellings. Fixed priority preemptive scheduling: A historical perspective. *Real-Time Systems*, 8:173–198, 1995.
3. F. Baccelli, G. Cohen, G.J. Olsder, and J.-P. Quadrat. *Synchronization and Linearity*. John Wiley, Sons, New York, 1992.
4. S.K. Baruah. Dynamic- and static-priority scheduling of recurring real-time tasks, 2001. To appear in Real-Time Systems.
5. A. Bavier and L. Peterson. BERT: A scheduler for best effort and real-time tasks. Technical Report TR-602-99, Department of Computer Science, Princeton University, 2001. Revised in January 2001.
6. T. Blickle, J. Teich, and L. Thiele. System-level synthesis using evolutionary algorithms. *Design Automation for Embedded Systems*, 3(1):23–58, 1998.
7. J.Y. Le Boudec. Application of network calculus to guaranteed service networks. *IEEE Trans on Information theory*, 44(3), May 1998.
8. J.Y. Le Boudec and P. Thiran. *Network Calculus - A Theory of Deterministic Queuing Systems for the Internet*. LNCS 2050, Springer Verlag, 2001.
9. G.C. Buttazzo. *Hard Real-Time Computing Systems - Predictable Scheduling Algorithms and Applications*. Kluwer Academic Publishers, 1997.

10. G.C. Buttazzo and F. Sensini. Optimal deadline assignment for scheduling soft aperiodic tasks in hard real-time environments. *IEEE Transactions on Computers*, 48(10):1035–1052, 1999.
11. S. Chakraborty, T. Erlebach, and L. Thiele. On the complexity of scheduling conditional real-time code. In *Proc. 7th International Workshop on Algorithms and Data Structures (WADS)*, LNCS 2125, 2001.
12. R.L. Cruz. A calculus for network delay. *IEEE Trans. Information Theory*, 37(1):114–141, 1991.
13. K. Deb. *Multi-objective optimization using evolutionary algorithms*. John Wiley, Chichester, 2001.
14. D. Decasper, Z. Dittia, G.M. Parulkar, and B. Plattner. Router plugins: A software architecture for next-generation routers. *IEEE/ACM Transactions on Networking*, 8(1):2–15, 2000.
15. A. Demers, S. Keshav, and S. Shenkar. Analysis and simulation of a fair queueing algorithm. *Journal of Internetworking Research and Experience*, 1(1):3–26, 1990.
16. E. Kohler, R. Morris, B. Chen, J. Jannotti, and M.F. Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, 2000.
17. J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm. In *Proc. IEEE Real-Time Systems Symposium*, pages 166–171, 1989.
18. C. Liu and J. Layland. Scheduling algorithms for multiprogramming in hard real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.
19. L. Thiele M. Eisenring, E. Zitzler. Handling conflicting criteria in embedded system design. *IEEE Design & Test of Computers*, 17(2):51–59, 2000.
20. G. De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill International Editions, New York, 1994.
21. A.K. Mok and D. Chen. A multiframe model for real-time tasks. *IEEE Transactions on Software Engineering*, 23(10):635–645, 1997.
22. The Moses project homepage. http://www.tik.ee.ethz.ch/∼moses/.
23. M. Naedele, L. Thiele, and M. Eisenring. General task and resource models for processor task scheduling, 1998. TIK Report 45, ETH Zürich.
24. M. Naedele, L. Thiele, and M. Eisenring. Characterizing variable task releases and processor capacities. In *Proceedings of the 14th IFAC World Congress*, 1999.
25. A.K. Parekh and R.G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM Transactions on Networking*, 1(3):344–357, 1993.
26. S. Prakash and A.C. Parker. Synthesis of application-specific multiprocessor systems including memory components. In *Proc. IEEE Application Specific Array Processors*, 1992.
27. X. Qie, A. Bavier, L. Peterson, and S. Karlin. Scheduling computations on a software-based router. In *Proc. SIGMETRICS*, 2001.
28. S. Shenker and J. Wroclawski. General characterization parameters for integrated service network elements. RFC 2215, IETF, September 1997.
29. L. Thiele, S. Chakraborty, and M. Naedele. Real-time calculus for scheduling hard real-time systeme. In *Proc. IEEE Internation Conference on Circuits and Systems*, 2000.
30. E. Zitzler, J. Teich, and S.S. Bhattacharyya. Evolutionary algorithms for the synthesis of embedded software. *IEEE Transactions on VLSI Systems*, 8(4):452 – 456, August 2000.

Design of Autonomous, Distributed Systems*

Tunc Simsek and Pravin Varaiya

University of California, Berkeley, CA 94720, USA
{tunc,varaiya}@eecs.Berkeley.EDU

Abstract. The paper describes an approach to organizing a collection of spatially distributed entities into a system capable of carrying out different complex missions. The entities are devices (artifacts like vehicles, sensors, actuators, computing and communications equipment) or agents or controllers, which control devices or other agents. The system-level capabilities concern complicated activities such as reconnaissance and logistics, requiring optimization, planning, and real-time control. The approach is to construct a system-level capability by appropriately organizing (composing) agents that can execute a small set of tasks. Agents are organized in a hierarchy: those at upper layers have authority over those at lower layers. The capability to conduct different missions is achieved by changing the agent architecture. The hybrid system specification and simulation language SHIFT is convenient for describing devices and agents, and their interaction. The paper gives two illustrative designs, one specified in SHIFT. To deploy systems organized in this way poses several research challenges, one of which is indicated here—the need for an exception-handling facility that provides a 'fallback' mechanism if an agent is unable to complete the task assigned to it.

1 Introduction

Figure 1 is an artist's conception of a futuristic battlefield. In our imagination, we can animate this static picture with a narrative describing several missions of surveillance, logistics and target tracking—conducted through the control and coordination of a variety of equipment, including vehicles, weapons, sensors and communications gear. We can also imagine that these entities could be reorganized and dedicated to different missions; and further, that these systems are autonomous, requiring little human intervention in the real-time control of equipment.

The paper describes an approach to organizing a collection of spatially distributed entities into a *system* capable of carrying out a variety of complex missions. Conceptually, we distinguish between entities that are *devices*, and those that are *controllers* or *agents*. Devices are artifacts such as vehicles, sensors, actuators, and communications equipment, which interact with the the physical world. These interactions are continuous-time signals. Controllers are artifacts

* Research supported by ONR Contract N00014-98-1-0585 and Darpa Contract F33615-00-C-1698

T.A. Henzinger and C.M. Kirsch (Eds.): EMSOFT 2001, LNCS 2211, pp. 435–450, 2001.
© Springer-Verlag Berlin Heidelberg 2001

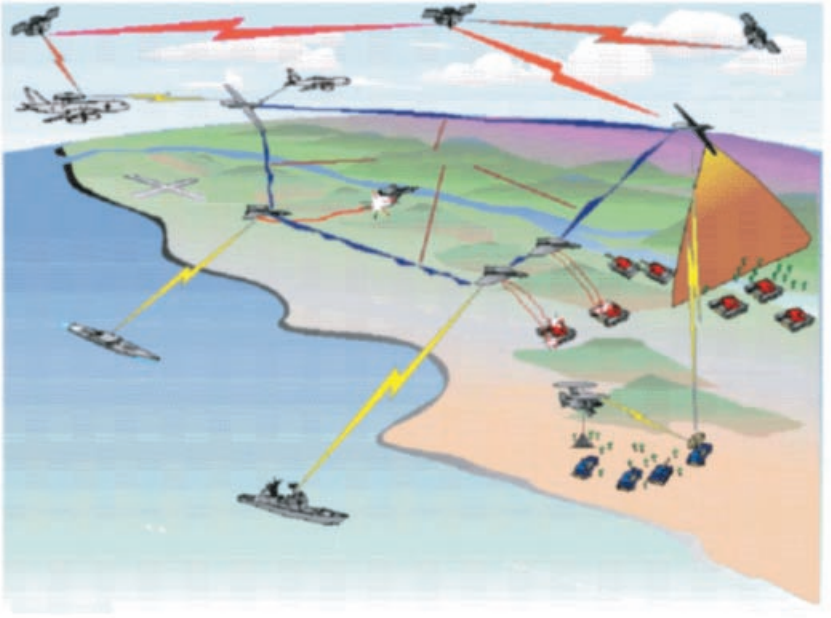


Fig. 1. Artist's conception of a future battlefield

such as algorithms, programs, and feedback control policies, which control the devices or other agents by exchanging symbolic messages. Interfaces "translate" between messages and signals. (The distinction between devices and agents is a matter of context: a vehicle may be viewed as a device or as a system comprising several devices and agents.)

To summarize, our interest is in organizing a system capable of complex missions involving optimization, planning, coordination, and control of entities over space and time. We suppose that each device has its own characteristic capability, which we cannot change. For example, a vehicle can move with a certain speed, a sensor can measure temperature, and a transmitter can broadcast a signal. On the other hand, we are free to design controllers, and to combine entities—devices and agents—into a system with a capability far superior to the capabilities of the constitutive devices. For example, the sensor, transmitter and vehicle could be organized into a system that autonomously measures, maps, and transmits the temperature profile over a designated land area. Evidently, the system-level capability to conduct a temperature survey emerges from an appropriate *organization* of the entities. The same entities can be organized in different ways to create different system-level capabilities.

Work in hierarchical system design forms two streams. The older stream of research, now called "mechanism design," began in the 1920s by asking if it is possible to organize a 'socialist' economy in which decisions $x_i \in X_i \subset R^n$ of

individual agents (enterprises and households), indexed $i = 1, \cdots, N$, could be decentralized like in a 'market' economy, but in such a way that, unlike in the market economy, the outcome of all these decisions would promote the social benefit. The simplest version of this design problem is the following.

Suppose the optimal socialist decision, $x^* = (x_1^*, \cdots, x_N^*)$, is the one that maximizes the social benefit $\sum_i U_i(x_i)$, subject to the individual constraints, $x_i \in X_i$, and the *interaction* constraint, $\sum_i x_i = 0$. (The constraint reflects, for example, the requirement that aggregate supply equal aggregate demand for each of n commodities.)

For each $\lambda \in R^n$, define the ith agent's optimal individualistic decision, $x_i(\lambda)$, as the one that maximizes $U_i(x_i) - \lambda^T x_i,\ x_i \in X_i$. (Note that once λ is given, the individual decisions are independent of one another, thereby achieving the goal of decentralization.) It is easy to see that if there exists a coordinating signal λ^* for which the optimal 'individualistic' decisions satisfy the constraint, $\sum x_i(\lambda^*) = 0$, then they coincide with the socialist decision, i.e., $x^* = (x_1(\lambda^*), \cdots, x_N(\lambda^*))$. However, the coordinating signal is not known, and two-layer, hierarchical feedback schemes were proposed to discover λ^*.

In these schemes, the agent at the upper or coordinating layer, broadcasts a coordinating signal, say λ_k, to the N individual agents who, in turn, reply with their optimal decisions, $\{x_i(\lambda_k)\}$. The coordinating agent evaluates the interaction constraint $\sum x_i(\lambda_k) = 0$, and then selects the next signal λ_{k+1}. Many algorithms for selecting the next signal were developed and analyzed to determine convergence: $\lambda_k \to \lambda^*$ as $k \to \infty$.

The invention of mathematical programming in the 1960s picked up this stream of work in the search for decomposition methods for large programs with special two-layer structures that recall the earlier formulation ([1] is an early reference). Another branch of research started from the observation that the coordinating signal above, λ, captured just the right amount of information about the interaction, and went on to study properties, such as efficiency of organizations of agents, with different organizations distinguished by the pattern of communication between agents [2]. In all of this work, the individual agents are like computer programs that take decisions based on prescribed calculations.

A third branch of research in mechanism design focused on organizations of agents whose personal objectives may be different from those of the organization. The design of the coordination problem now must take into account the discrepancy between individual and organizational objectives, and game theory is a useful setting for formulating and analyzing these designs [3]. A contemporary example is the design of the FCC auctions of spectrum for cellulary telephony to maximize social benefit of the decisions made by telephone companies who seek to maximize their own profits.

The second stream of research originates in the practice of organizing the control of large systems in a distributed hierarchy. There are good reasons for such an organization: deeper understanding facilitated by the hierarchical structure, reduction in complexity of communication and computation, modularity, adaptibility to change, robustness, and scalability. The aim is similar—the de-

composition of a large problem, into a *coordinated* set of smaller tasks, each carried out by simple agents. But the focus has shifted. Instead of seeking general theorems on mechanisms to coordinate homogeneous agents, the objective is to realize a given, concrete system objective by coordinating a heterogeneous collection of agents, each designed to carry out a narrowly specified task, using devices embedded in a physical environment. For lack of a better term, we call this work research in "intelligent systems."

In mechanism design problems, the physical environment is modeled as static resource constraints. In the design of intelligent systems, the dynamics of the physical devices and the environment are important. Accordingly, the coordination signals and the individual agents become more complex. The whole system is viewed as a hierarchical control system with many feedback loops. At the lower layers, physical devices are controlled by continuous-time signals, representing physical quantities. At the upper layer, the feedback loops carry messages, with the richer semantics needed to coordinate control agents.

In mechanism design, the agents are homogenous,and the design space of coordination mechanisms is mathematically highly structured (the set $\{\lambda \in R^n\}$ and the iteration rule from λ_k to λ_{k+1} in the example). This restricts the space of mechanism design.

In intelligent control design, the agents and the devices are represented as hybrid automata, with their continuous dynamics modeling the physics and continuous-time feedback loops, and discrete transitions representing production and consumption of coordination messages. The space of hybrid automata is vast, with little mathematical structure. Not surprisingly, there are few theorems to guide design. This paper, therefore, merely reports some design experience.

Section 2 introduces through an example the concepts of task, agent, and hierarchical organization that help describe system design. Section 3 gives a different example and shows that SHIFT is a convenient language for design specification and simulation. A SHIFT program models a *dynamic* network of hybrid automata. The example illustrates the use of SHIFT's "dynamic" features: creation and destruction of components and dynamic changes in how components interact. Section 4 offers a remark on the semantics of hierarchical control. Section 5 discusses exception-handling.

2 A Surveying System

The top of figure 2 depicts a distributed system with a 4-layer controller. Layer 0 comprises two kinds of devices, vehicles and sensors. Vehicles move over a two-dimensional terrain. Let (x, y) be the vehicle's position and ϕ its heading relative to some global coordinate system, as shown. The vehicle's movement is given by

$$\begin{aligned}\dot{x}(t) &= u(t)\cos\phi(t),\\ \dot{y}(t) &= u(t)\sin\phi(t),\\ \dot{\phi}(t) &= v(t).\end{aligned}$$

The vehicle's linear speed, $u(t) = \sqrt{\dot{x}^2(t) + \dot{y}^2(t)}$, is limited by $u(t) \in [0, 5]$, and its angular speed is limited by $v(t) \in [-1, 1]$. Therefore, the vehicle's intrinsic capacity to move is given by all curves $(x(t), y(t)), t \geq 0$, whose derivatives satisfy these constraints. This is too unstructured. The first step in the design is to impose a finite, discrete set of control laws.

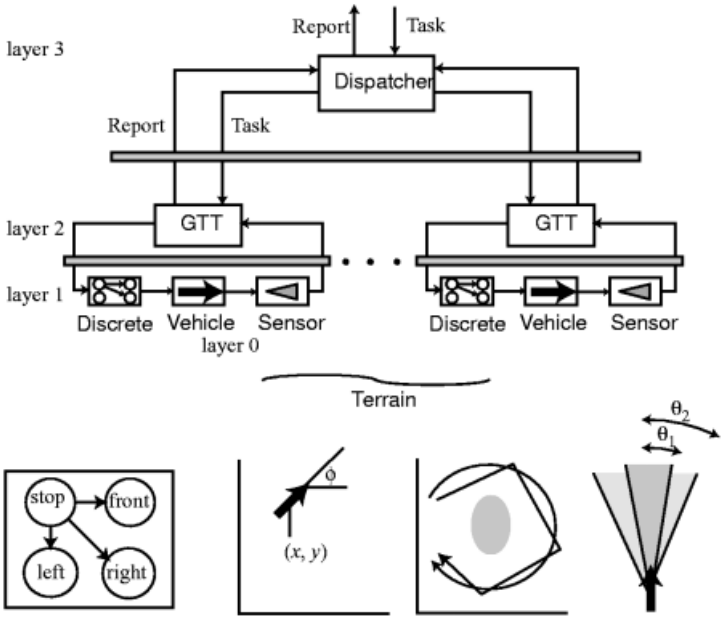


Fig. 2. A hierachical control structure, with details of Layers 0, 1

The layer 1 controller (called "Discrete" in the figure) *restricts* the inputs to four *modes*—stop, front, left, and right—as follows:

$$\begin{aligned} \text{stop} &: u(t) \equiv 0, v(t) \equiv 0, \\ \text{front} &: u(t) \equiv 5, v(t) \equiv 0, \\ \text{left} &: u(t) \equiv 0, v(t) \equiv 1, \\ \text{right} &: u(t) \equiv 0, v(t) \equiv -1. \end{aligned}$$

These modes can be selected in any sequence. So the layer 1 agent or controller is represented by a 4-state machine, each discrete state or mode representing a mode selection, and with transitions possible between any pair, as shown in the figure. (The figure only displays the transitions from "stop.") The transitions are labeled in the obvious way as "GoFront," "GoStop," "GoLeft," and "GoRight."

The restriction of all possible control inputs to a finite set of control laws is a crucial step in the design. We call this step *discretization*. (This is not to be

confused with discrete time.) The step structures the design space of the next layer, layer 2, since the agent at layer 2 can only select from the mode transitions that layer 1 offers. The restriction leads to a reduction in vehicle control and performance that layer 2 can achieve. There is a tradeoff which can be crudely expressed by saying that the larger the number of modes in the discretization step at layer 1, the more complex is the design space at layer 2, and the greater the vehicle control and performance that layer 2 can achieve. We investigate this tradeoff briefly.

The four-mode discretization limits the phase portraits of the motions that layer 2 can achieve to those that can be drawn as a sequence of straight-line segments. So layer 2 cannot achieve the phase portrait given by the circular arc, even though the vehicle is capable of it. However, since any phase portrait can be approximated by piecewise linear segments, we could say that the loss of controllability by the discretization is slight. We can be more precise. Suppose the terrain contains obstacles that the vehicle cannot penetrate. (One obstacle is represented by the shaded oval.) Then all the phase portraits that the vehicle is capable of executing is partitioned into a set of homotopy classes. The piecewise straight-line approximation implies that each homotopy class contains a phase portrait that an agent at layer 2 can execute. Therefore, if the objective of control is to move the vehicle from a starting point to a destination through a terrain "punctured" by obstacles, then this discretization leads to no loss of control.

However, the discretization definitely leads to a reduction in performance, measured as the minimum time needed to go from the starting point to the destination. This is easy to see since under the discretization, the vehicle turns (left or right modes) only with zero speed, even though the vehicle is capable of moving with speed 5.

To summarize: layer 1 offers (limits) the capability to movements straight ahead or turning without moving. Layer 2 can use this capability to achieve any motion comprising a sequence of straight-line segments. Our design will structure this capability to steering the vehicle to any target. The resulting agent or controller is called GTT—GoToTarget.

GTT uses the sensor attached to the vehicle. As indicated in the bottom right-most picture in figure 2, the sensor has two fields of view: an inner cone of angle θ_1 and an outer cone of angle θ_2. The sensor can determine whether a given target is located inside the inner cone, inside the outer cone, to the left or to the right of the outer cone.

GTT has the capability of accepting a target from layer 3 and issuing commands to layer 1 so that the vehicle reaches the target. To offer this capability GTT implements the following simple control logic:
1. If vehicle has reached target, it commands the transition GoStop to layer 1 and reports "TargetReached" to layer 3;
2. If vehicle has not reached target, it commands GoStop;
3. If target is in OuterCone, it commands GoStraight;
4. If target is Left of OuterCone, it commands GoLeft, and then when target is in InnerCone, it commands GoFront;
5. etc.

Evidently, GTT can be implemented as a finite state machine whose transitions are triggered by changes in the sensor outputs or by commands from layer 3. In figure 2, the "task" received by GTT is a target from layer 3, and it reports "TargetReached." In turn it issues commands GoStop, etc to layer 1.

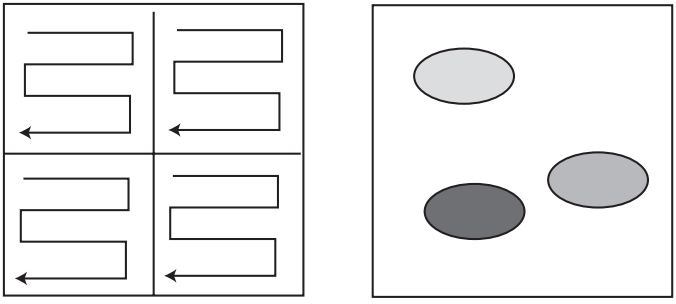


Fig. 3. The Dispatcher design

The layer 3 agent, the "Dispatcher," coordinates the movement of several vehicles by issuing tasks to their GTTs. The dispatcher offers to its superior at layer 4 (not shown), the capability to survey a specified region, using the vehicles over whose GTT the dispatcher has authority. For example, if the region is the rectangular area shown in the left of figure 3, and four vehicles are available, the dispatcher could plan the "lawnmowing" pattern as shown. If, on the other hand, a region contained obstacles or variable threats as indicated by the differently shaded ovals on the right, the paths may be selected via some optimization procedure.

With this design, beginning with the 'raw' capability embodied in the vehicles and sensors, we can construct controllers, in a hierarchical manner, with the agent at each layer offering a greater capability, building on the capabilities of the agents it controls.

In the next section, we study a different example to see designs are specified in SHIFT.

3 A Web Server

SHIFT is a programming language for specifying and simulating dynamic networks of hybrid systems [4]. The web server example illustrates useful features of SHIFT. The web server is modeled as a 2-layer control system. (For large SHIFT applications, see [5,6,7].)

The web server is a connection-based process capable of serving HTTP requests from clients. We consider a multi-threaded server:

- the server listens to connection requests,
- a client requests a connection from the server,

- if possible, the server allocates a thread and establishes a point-to-point full-duplex connection between the client and the thread,
- the server returns to listening connection requests,
- the thread services the HTTP request from the client.

The web server is a controller with total resources of R_T per unit time. An HTTP request is a structured command such as `GET /foo.txt`. There is a finite number of such requests denoted by the symbols $h \in H$. The total resource needed to serve request h is given by the function $requires : H \rightarrow Reals$.

At any time t, $N(t)$ active requests are being processed by $N(t)$ threads. The server allocates the resource to each thread according to a control policy. A fair policy allocates the same amount

$$r(t) = \frac{R_T}{N(t)} \tag{1}$$

to each thread at time t. Thus, a thread will complete serving request $h \in H$ when

$$s(t) \leq 0,$$

where

$$\dot{s}(t) = -r(t),\ s(t_0) = requires(h), \tag{2}$$

if that request arrived at time t_0. To maintain a minimum quality of service, the web server rejects an incoming request if

$$r \leq R_{min}. \tag{3}$$

In a SHIFT specification, the programmer defines prototypes or classes that describe the behavior of components. A component (entity) denotes a hybrid automaton and its behavior is a computation of the hybrid automaton . A simulation starts with an initial set of components that evolve concurrently.

The data model of a class consists of strongly typed variables. The type of a variable is a continuous number, (discrete) number, symbol, the name of a class, or nested sets and arrays of these types. For the web server, the data model of a client is given as (SHIFT keywords are in italics):

```
type Server {
   ...
   state continuous number r;
   ...
}
type Client {
   input Server server;
   output symbol h;

   export httpRequest, httpResponse, httpError;

   setup
     do {
```

```
      h := $get;
      server := theServer;
    };
}

global Server theServer := create(Server);
global set(Client) theClients :=  create(Client), create(Client) ;
```

The `Server` class will be defined shortly. This model fragment defines a named class called `Client` that has two variables: a symbolic variable called `h` and a link to a Server called `server`. A component of type `Client` assigns values to these variables with the initial value given by the setup clause. In this case, the request `h` of the client will be a GET command (denoted with the symbol `$get`) and its link will be set to a global shared server called `theServer`. The simulation will initially contain three components: the global server and the two global clients.

Interaction amongst components occurs via links. For example, the SHIFT notation `r(theServer)` denotes the variable `r` of the component `theServer`. The data model supports data encapsulation by specifying whether a variable of a class is `input`, `output` or `state`. A class has read-only access to its input variables and read/write access to its output and state variables. Outside the class, output variables are read-only and input variables are read/write. The encapsulation mechanism is complemented by an exported set of event labels that represent the command interface of the class—that is, the names of discrete actions of the hybrid automaton that describe the behavior of the class. These actions are synchronous so issuing an action or responding to it are not distinguished. In the example, the Client performs three actions visible to the outside world—it issues an `httpRequest` and awaits a `httpResponse` or `httpError`.

In the example, the behavior of clients is not part of the web server so we do not model their behavior. (Of course, one could design a component that creates clients periodically or randomly in time.) However, the client's interface—inputs, outputs and exported events—is used in the web server model, and so it is specified.

The behaviour of a class is a computation of its hybrid automaton. While the hybrid automaton is in one of its discrete modes, ordinary differential equations (ODEs) and algebraic equalities describe the time-evolution of its state and output varaiables (continuous numbers). Transitions among the discrete modes of the hybrid automaton are labeled with guards, events and actions. Guards are enabling conditions for the transitions; events provide a synchronous composition interface for different components; and actions describe the discrete behavior of the state and output variables.

We are now ready to describe the SHIFT model of the web server. The model relies on the interface of the `Client` class and the global set of clients defined above. We model the server and its threads as separate entities using two class definitions: the `Server` class and the `Thread` class. A `Server` component controls several `Thread` components. The control authority of the server is expressed in its ability to create a thread component, to create its connection with a client,

and to decline a connection request due to insufficient resources. This gives a 2-layer hierarchical model of the web server with the server at the higher layer controlling the threads at the lower layer (the numbers 50 and 75 below are arbitrary):

```
#define requires(h)   if h=$get then 50
                         else if h=$put then 75;

type Thread {
  input Client client;
        continuous number r;
  state continuous number s;

  discrete
    process { s'=-r; };

  transition
    process -> exit { client:httpReply }
      when s <= 0;

  setup
    do {
      s := requires(h(client));
    };
}

type Server {
   state number R_T := 100;
         number R_min := 1;
         set(Thread) threads := nil;
         number requires;
         Client client;
         continuous number r, N;

   flow computeResource {
              N = size(threads);
              r = R_T / N; };

   discrete
     listen { computeResource; },
     acceptOrDeny { computeResource; };

   transition
     listen -> acceptOrDeny { clients:httpRequest(one:c) }
       do {
          requires := requires(h(client));
```

```
            client := c;
         },
      acceptOrDeny -> listen { client:httpError }
         when r <= R_min,
      acceptOrDeny -> listen { }
         when r > R_min
         define {
            Thread newThread := create(Thread,
                                       client := client);
         }
         do {
           threads := threads + { newThread };
         }
         connect {
            r(newThread) <- r;
         };
}
```

(The notation in the SHIFT program matches that used in the earlier equations.)

A visible interface of a thread is given by the input link to a client, and the input number `r` representing the resources allocated to the thread at any time. The internal interface of the thread is given by the number `s`, which is the remaining amount of service needed for the request. The thread has a single discrete mode, `process`, in which

$$\dot{s} = -r,$$

where the initial value of `s` is given by the `setup` clause (which is executed immediately after a component is created). The initial values of the inputs are set externally—in this case, by the server.

Once the thread has served the request, it issues the `httpReply` command to the client and enters the special mode called `exit` in which it is destroyed.

The server has no visible external interface. Its internal data consist of the parameters `R_T`, `R_min`, `r`, `N` as in (1)-(3) and other auxiliary programming variables. The server sits in the listen mode until some client issues an `httpRequest`. The notation

```
clients:httpRequest(one:c)
```

means, pick `one` client from the set of clients issuing an `httpRequest` and tag that client with the temporary variable `c`. The temporary variable is available only inside the action of the transition, so it is saved in an auxiliary variable called `client` for later use.

In the `acceptOrReject` mode, the server determines if there are sufficient resources to process the incoming request. If not, then the server issues a `httpError` command to the client. Otherwise, the server creates a new thread

(initializing the input variable called `client` to the client). The available resources r of the thread is hardwired to the variable r of the server using the `connect` clause.

The hybrid automata evolve concurrently. To alleviate some of the difficulties associated with this style of programming SHIFT supplies the `define-do-connect` clause as illustrated in the example above. The `define` part allows for a sequential definition of new variables. These variables are then made available for use in the `do` and `connect` parts which are all executed in parallel.

The example illustrates the use of SHIFT's constructs to design a hierarchical system, with subsystems (agents or devices) described by components, and feedback controls described by component interactions. The subtle but simple interactions permit modeling fairly complex control policies. The encapsulation feature, which enforces interfaces, together with inheritance (not illustrated in the example) and typechecking, provide a powerful way to write syntactically correct programs.

4 Semantics of Hierarchical Design

The boxes on the left in figure 4 denote the controllers at the four layers of the survey system. On the right, "Path," "Control," etc. denote the outputs of these controllers. A controller's output is the task assigned to the controller below, except that Vehicle's output is its trajectory. In a SHIFT specification of the dispatcher, its output "Path" might be expressed as a sequence of "targets," issued to the appropriate GTT, one at a time. The GTT takes each target input and selects a sequence of "modes," following its feedback law and depending on the sensor readings. The controller, Discrete, receives the modes from GTT and issues the corresponding fixed control input to the vehicle. Finally, at the lowest

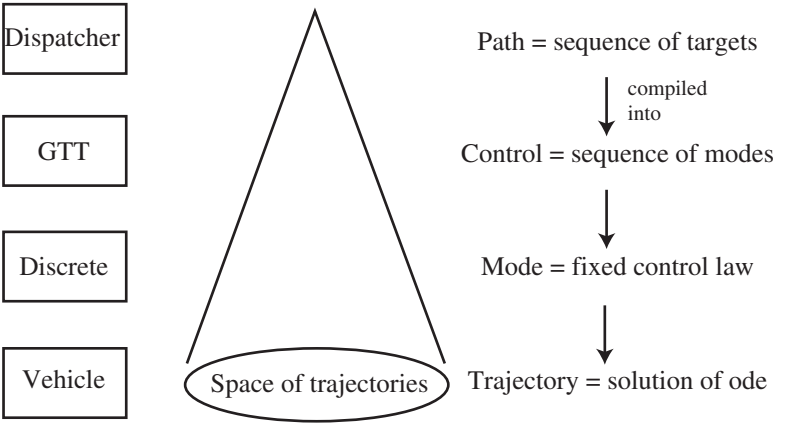


Fig. 4. One-world semantics of surveillance design

level, the differential equation with the appropriate control input is *interpreted* as the trajectory of the vehicle.

In this way, a high-level task or expression like "Path" is compiled (or translated) into a sequence of "targets," each target is compiled into a sequence of "modes," and so on, until, finally, we end up with vehicle trajectories. With this "syntactic flattening" all higher-level expressions (involving terms such as "Path") are ultimately interpreted in the world of vehicle trajectories. We call this one-world semantics.

With one-world semantics, therefore, truth-claims expressed in high-level expressions (e.g. this is a "minimum-length path") are interpreted as claims about trajectories, as is common in verification theory where properties of high-level abstractions can be translated into properties of lower-level behaviors.

However, hierarchical controllers are not designed in this way. For instance, given a region to be surveyed, the dispatcher may run an algorithm which, for a specified $\delta > 0$, yields a minimum-length path whose δ-neighborhood covers every point in the region. The correctness of this algorithm may be proved in the mathematical world of two-dimensional geometry, and not as properties of the vehicle trajectories, whose relation to the path-finding algorithm involves many other system entities. Indeed, the relation between the paths shown on the left in figure 3 is very complex. (In fact, the reader may have noticed that under GTT's feedback law the vehicle may not reach its target in finite time—its position can only be guaranteed to converge to the target. Also, the path produced by the path-planning algorithm is not in any nice mathematical sense an "abstraction" of the trajectories into which it is ultimately compiled.)

Typically, the design of a large system is broken into controllers (like dispather, GTT, etc.). The design of each controller is evaluated in a mathematical world in which alternate controller designs (e.g. different path-planning algorithms) can be compared. This is multi-world semantics. (The mathematical world for one controller makes implicit assumptions about the behavior of lower-layer controllers. For example, the dispatcher assumes that the generated path will be implemented by the GTT.) At the same time, of course, the one-world semantics of the device layer—which interacts with the physical world—is better suited for the study of system *implementation*. There does not seem to be any accepted way of bridging the gap between the multi-world semantics of design and the one-world semantics of implementation. The discussion of "exceptions" below addresses one aspect of this gap.

5 Exception-Handling

In hierarchical system design, upper-layer components are built on the capabilities of lower-layer components and on the assumption that the latter "work as advertised." For example, for a controllable LTI system $\dot{x}(t) = Ax(t) + Bu(t)$, one may design a feedback vector F such that the control law $u(t) = Fx(t)$ stabilizes the system. The assumption here is that the system indeed behaves as advertised by its A and B matrices. It is however unreasonable to expect that all

assumptions made in the design of a large distributed control system are valid. Indeed there are classes of controllers that relax the assumptions. For example, one may replace the feedback law above by a sliding mode controller to ensure stability of the system $\dot{x}(t) = (A+\Delta(t))x(t)+(B+\Gamma(t))u(t)$ where Δ and Γ are small time-varying perturbations. However, even the extended control design is subject to degradation and failure. By exception-handling, we mean an alternate approach to system design that will help the designer identify an error and fix the problem. To simplify the presentation we recall some terminology from [8]. A *situation* is the evaluation of a command issued by an upper-layer component to a lower-layer component. An *exceptional situation* is a situation whose outcome is undefined (or undocumented). This will arise, for example, if the GTT controller is instructed to go to a target that doesn't exist, or the target disappears before the vehicle gets there. If the GTT design had not considered this case then the system exhibits an exceptional situation. In a SHIFT simulation one may end up with a *core dump*; in an implementation, the exceptional situation may cause more damage.

An exception-handling system attempts to provide at least two guarantees, regardless of the particular control structure. The system is always in a well-defined state and there is a means for representing an exceptional situation. Such a system is integrated into λ-SHIFT—a generalized experimental SHIFT [9]. In what follows we explain in SHIFT terminology how the system is realized.

A SHIFT situation is taken as the evaluation of a SHIFT expression—such as a guard, action or flow equation. An exceptional situation is a situation for which the result is undefined. For example, consider the GTT controller,

```
type Target {
      ...
   output continuous number x,y;
}
type Vehicle {
      ...
   export goStop, goLeft, goRight, goFront;
   output continuous number x,y;
}
type GTT {
   input Target target;
   output Vehicle vehicle;
       ...
   transition
      all -> stop  vehicle:goStop
      when distance(x(vehicle),y(vehicle),x(target),
                                    y(target)) <= 1;
}
```

where the SHIFT keyword `all` denotes a transition from all defined discrete modes. It may happen that the target was not initialized (or, since SHIFT com-

ponents are dynamic, the target disappeared for some other reason). Then the evaluation of `x(target)` will result in an exceptional situation.

A SHIFT simulation consists of a closed (self-contained) world constructed by a well-defined syntax and semantics. Thus, it is possible to identify all possible *atomic* causes of exceptions. This is precisely what is done in λ-Shift. The next step is to provide additional syntax to represent and handle exceptional situations. For example, in the GTT controller above, one would write a transition of the form:

```
transition
    ...
  all -> protected {invalidLink, vehicle:goStop}
```

where the event `invalidLink` is delivered by the exception-handling system. The cascaded event `goStop` will command the vehicle to stop. This approach allows the programmer to represent an atomic exceptional situation in the language of SHIFT, provided that for each atomic situation an event such as `invalidLink` is defined and supported by the system.

In simulation, the exception-handling mechanism is used as a debugging and program development tool. The basic mechanism described above is tightly integrated with the component interaction mechanisms—such as direct link access, synchronous composition, and wired connections. If at some time t an exceptional situation occurs in a component, the mechanism provides a focused and (state-) consistent view of that component and all other components with which it is interacting.

When considering a distributed hieararchical system, one needs to consider the following aspects of exception handling. What is the computational cost of deploying such a mechanism? How will state consistency be implemented in a spatially distributed system? How does the debugging process (with its one-world semantics interpretation) relate to the deployment of real-life systems (implicitly designed with multi-world semantics)?

6 Concluding Remarks

Large systems are conveniently designed in a control hierarchy. At the lowest layer are the devices which interact with the physical environment through continuous-time signals. Controllers at higher layers control other controllers and devices through the exchange of messages with richer semantic content. Each controller builds on the capability of those that it controls. SHIFT is a convenient language for hierarchical system design. The modularity implicit in such a structure allows the design of each controller to be carried out independently and evaluated in a suitable mathematical domain. In such a design process, implicit assumptions are made about the behavior of lower-layer controllers. Those assumptions are unlikely to hold, and an exception-handling mechanism that helps debug a design, and makes its implementation more safe. Although exception-handling is considered in computer science, its development in control system design would be a new and important area of study.

References

1. A.M. Geoffrion, *Elements of large-scale mathematical programming*, Santa Monica, CA: Rand Corporation. Memorandum RM-5644-PR, 1968.
2. J. Marschak and R. Radner, *Economic Theory of Teams*, New Haven, CT: Yale University Press, 1972.
3. D. Fudenberg and J. Tirole, *Game Theory*, Cambridge, MA: MIT Press, 1991.
4. A. Deshpande, A. Gollu, and L. Semenzato, *The shift programming language and run-time system for dynamic networks of hybrid automata*, Technical Report UCB-ITS-PRR-97-7, California PATH, 1997.
5. A. Gollu and P. Varaiya, "Smartahs: a simulation framework for automated vehicles and highway systems," *Mathematical and Computer Modelling*, 27(9-11):103–28, 1998.
6. The SmartAHS Team, The SmartAHS manual. `http://www.path.berkeley.edu/smart-ahs`, 1998
7. J. Borges de Sousa, A. Girard, and N. Kourjanskaia, "The mob shift simulation framework," *Proceedings of Third International Workshop on Very Large Floating Structures*, pages 474–482, 1999.
8. K. Pitman, "Exceptional Situations in Lisp," *Proceedings for the First European Conference on the Practical Application of Lisp (EUROPAL'90)*, Churchill College, Cambridge, UK, March 27-29, 1990
9. `http://www.gigascale.org/shift`

Formalizing Software Architectures for Embedded Systems

Pam Binns and Steve Vestal*

Honeywell Laboratories
3660 Technology Drive
Minneapolis, MN 55418
{pam_binns,steve_vestal}@htc.honeywell.com

Abstract. This paper outlines an approach to embedded computer system development that is based on integrated use of multiple domain-specific languages; on increased use of mathematical analysis methods; and on increased integration between domain-specific specification and mathematical modeling and code generation. We first outline some general principles of this approach. We then present a bit more detail about the emerging SAE standard Avionics Architecture Description Language and our supporting MetaH toolset. We conclude with a summary of some research challenge problems, technical approaches, and preliminary results uncovered during our work.

1 Introduction

The use of domain-specific languages (4GLs) and tools for embedded applications is wide-spread and will increase. For example, a number of COTS tools are already in wide use for the development of feed-back control and display applications. In many cases the use of domain-specific technologies in preference to general-purpose software development technologies can result in cost savings and improvements in quality factors that justify the additional development and acquisition costs of the domain-specific tools. Available meta-tool technologies have been used to lower the development cost of specialized domain-specific tools[5].

Three main elements of a domain-specific language and toolset are illustrated in Figure 1. There is the domain-specific language and editor, which allows rigorous, concise, very high level specification of the structure and semantics relevant to a particular engineering discipline. There are modeling and analysis methods and tools to support design during the early phases of development and verification during the later phases of development. There is a code generation or synthesis method to produce an implementation from a design specification. We

* This work has been supported by DARPA, ONR and AMCOM under contracts N00014-91-C-0195, DAAH01-97-C-0195 and DAAH01-00-C-R226; by AFOSR under contract F469620-97-C-0008; and by Honeywell. An earlier version of this paper appeared in the Monterey Workshop 2001.

T.A. Henzinger and C.M. Kirsch (Eds.): EMSOFT 2001, LNCS 2211, pp. 451–468, 2001.
© Springer-Verlag Berlin Heidelberg 2001

believe these elements should be integrated and automated as much as is practical. Models and code should be generated from a common specification where possible, eliminating the hand-development of separate model specifications in different modeling languages. The mapping between specification, models and code should be structure-preserving, intuitive, and easily verified. It should be possible to easily trace in any direction between design specification, models, model analysis results, and implementation.

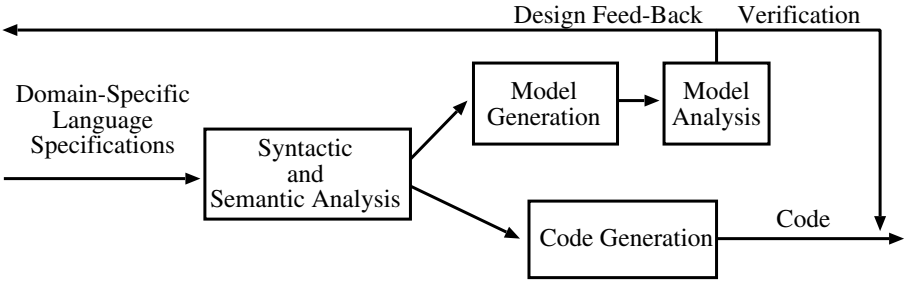


Fig. 1. Notional Domain-Specific Toolset

The construction of complex embedded computer systems is an inherently multi-disciplinary effort and requires integrated use of multiple domain-specific languages and tools. The mix of specification languages and tools needed in a particular development environment will depend on the mix of embedded functionality needed in a particular product line.

Our work has focused on applying the above principles in the development of an embedded computer system architecture specification language and toolset. This language and toolset are designed for use by embedded computer system architects, among whose tasks is the integration of various hardware components and software applications developed by other engineering groups using other domain-specific languages and tools. Figure 2 illustrates how the outputs of multiple domain-specific tools feed into the architecture specification toolset, which supports computer system integration and modeling and analysis. The output of the toolset, in addition to models and analysis results, is software that integrates the pieces of the system together.

In addition to supporting embedded system developers, the language and toolset also provide a context and enabler for research activities. We are using the language and toolset as an object of study and a prototyping testbed in research activities intended to enable large, dynamically reconfigurable, safety-critical distributed systems that efficiently host both time-triggered and event-triggered workloads. We will conclude this paper with a survey of some challenge problems, technical approaches, and preliminary results in these areas.

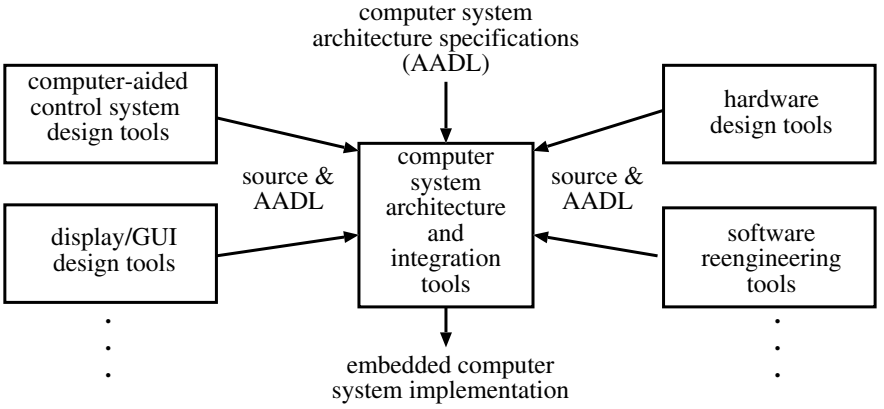


Fig. 2. Domain-Specific and AADL Toolsets

2 AADL and MetaH

The emerging SAE standard Avionics Architecture Description Language (AADL) is a language for specifying software and hardware architectures for real-time, safety-critical, scalable, embedded multi-processor systems. The AADL allows developers to specify how a system is composed from software components like processes and packages and hardware components like processors and memories. Our MetaH AADL toolset performs syntactic and semantic checks, compliance checks between specification and source code, schedulability analysis, reliability analysis, partition isolation analysis, and generates/configures a system executive (middleware) layer that can be subjected to formal analysis using linear hybrid automata models. Figure 3 illustrates the current toolset.

The AADL includes constructs to describe source components written in a traditional programming language like C/C++ or Ada. The source components themselves come from domain-specific tools, or are hand-written, or are re-engineered from existing code. Subprogram and package specifications describe important attributes of source modules such as the file containing the source code, nominal and maximum compute times on various kinds of processors, stack and heap requirements, mutual exclusion protocol to be used for shared packages, etc. Event names, message types, and buffer variables used to hold message values, can be declared in source modules and must be cited in the AADL specifications. The current MetaH toolset accepts source in C/C++ or Ada, but only Ada code is parsed and checked for compliance against the AADL interface descriptions.

The higher-level software constructs of the AADL are processes, modes and systems. Processes group together source modules that are to be scheduled as either periodic (time-triggered) or aperiodic (event-triggered) processes. A process is also the basic unit of security and fault containment, and memory protection and compute time enforcement may be provided on some targets. Systems and

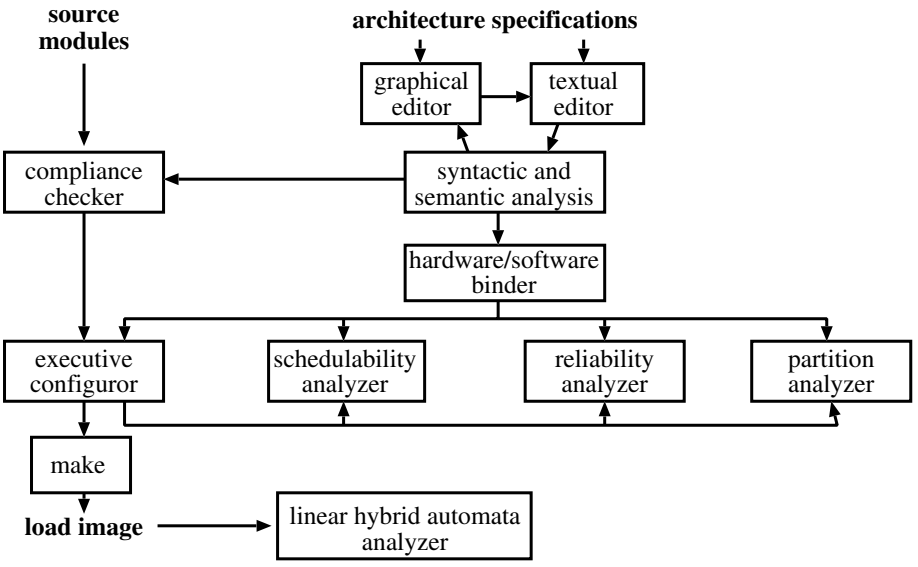


Fig. 3. MetaH/AADL Toolset

modes group processes, define connections between data and event ports, and define bindings between objects that are to be accessable between processes. The difference is that systems are concurrent with each other, while modes are mutually exclusive with respect to other modes in the same implementation diagram. Event connections between modes are used to define hierarchical mode transition diagrams, where mode changes at run-time can stop or start processes or change connections.

The AADL also allows hardware architectures to be specified using memory, processor, bus, and device components grouped into systems. Hardware objects may have data and event ports and packages in their interfaces. Software and hardware data and event ports can be connected to software data and event ports, and software components can access hardware packages (which provide hardware-specific APIs). Hardware descriptions identify (among other things) hardware-dependent source code modules for device drivers, and code to provide a portable interface between automatically composed applications and the underlying RTOS and hardware.

Both graphical and textual specification is supported. The two can be mixed (part of a specification can be maintained textually and part graphically), and the toolset can translate graphical to textual and vice versa. This is convenient in our assumed multi-domain, multi-toolset development context. Portions of a specification may be automatically generated by different domain-specific tools (text is often preferred for this), while higher-level integration and design review is usually performed by hand (graphics is often preferred for this).

A simple software/hardware binding tool assigns to hardware those software objects in a specification that are not explicitly assigned, possibly subject to user-specified constraints.

An executive configuration tool automatically produces the "glue" code needed to compose the various source modules to form the overall application. The resulting tailored system executive (middleware) is responsible for process dispatching, event and message passing, mode changing, etc. There is a makeh tool that performs all the comples and links needed to produce a loadable image for each processor specified in the system.

The design schema for the configured executive is based on preemptive fixed priority scheduling theory[7]. Using AADL specifications of process period, preperiod deadline, criticality, and precedence constraints, the executive generator derives priority, period transformation, dispatch rate, and time slice information used in data tables and dispatching code[12]. Data connection and process timing specifications are used to schedule and generate code to move data between processes' data buffer variables, including message scheduling and synchronization for certain classes of multi-processor systems. Code to vector events to dispatch aperiodics or to trigger mode changes, and code to manage mode changes, is also generated.

Using information contained in the AADL specification and produced by the executive generator, the schedulability modeler generates a detailed preemptive fixed priority schedulability model of the application. The model includes scheduling and communication overheads as well as application workloads. In support of traceability, a human-readable form of the model is written as well as the results of analyzing the model. The schedulability analysis algorithm we currently use is an extension of the exact characterization algorithm that can perform certain kinds of parametric analysis[24].

The executive code generated from a MetaH specification may enforce integrated modular avionics partitioning (protected address spaces, process criticalities, enforced compute time limits, capability lists for run-time services). Source objects may be annotated with a safety level determined during system safety hazard analysis[1] (required application code verification activities and hence the degree of assurance depend on the assigned safety level). The tool checks to insure that correct operation of an object cannot be affected by any error in any other object having a lower safety level. For example, an object with a high safety level should not depend on data from an object with a low safety level (unless the connection is explicitly annotated in the specification to allow this). The deadline of a process with high safety level must be guaranteed even if processes with low safety levels exceed their stated compute times.

AADL semantics and the configured executive include features to deal with error detection and handling, but no particular redundancy management approach is built into the language or system executive. Different systems have different requirements and missions that may result in different dominant failure modes. These are best addressed using different redundancy management architectures. For example, manned flight control systems traditionally use triple or

quad redundancy with cross-voting and error masking, while long-life satellite systems traditionally use cold spares and dynamic reconfiguration. Reliability analysis and system safety, including linear hybrid automata verification, will be further discussed in subsequent sections.

3 Research Activities

Our long-term goal is a language and computationally efficient toolset that support the development of embedded systems that are distributed, dynamically reconfigurable, fault-tolerant, support periodic (time-triggered) and aperiodic (event-triggered) task models with complex inter-task interactions, make efficient use of resources, and are verifiable to the highest levels of system safety and design assurance. In the following sections we will cite some challenging problems in a few of these areas and outline some of the approaches we are pursuing to deal with them.

3.1 Decomposition Scheduling

The distributed scheduling problem for systems that host periodic feed-back control applications is different than the multi-media scheduling problem. Tight end-to-end latencies comparable to task periods must be guaranteed. Often no loss of data will be tolerated. Solutions may need to be verified to the highest levels of assurance, which in practice means worst-case schedulability analysis must be available. Our notional set of requirements is

- high achieveable hardware utilization, e.g. over 90% processor and over 50% bus utilizations
- small end-to-end latencies, e.g. one sampling delay (one period)
- high assurance that deadlines will be met, e.g. formal schedulability analysis
- tractability for large systems, e.g. generate schedules for thousands of tasks and messages on hundreds of processors in tens of seconds, incrementally change a schedule in fractions of a second
- adaptable to different COTS bus/network hardware, adaptable to different scheduling disciplines for different resources, adaptable to different redundancy management techniques and interconnect topologies

We have been exploring an approach we call decomposition scheduling, illustrated in Figure 4. The basic idea is to decompose the overall system scheduling problem into a set of individual resource scheduling problems, solve the individual problems, then combine the results of parametric schedulability analysis for the individual resources to obtain a better system decomposition. Each individual resource scheduling problem consists of the tasks or messages allocated to that resource, together with release times and deadlines that are selected by the decomposition algorithm. Once each resource has been scheduled, the results of parametric schedulability analysis (such as available laxity and slack for

the various tasks and messages) are used to pick a new set of release times and deadlines. The new deadlines and release times make the individual scheduling problems easier for previously unschedulable resources at the expense of previously schedulable resources. The approach is iterative and continues until a solution is found or the solution does not change significantly between successive iterations. Different resources can potentially be scheduled using different disciplines, as long as parametric schedulability analysis is availble for each discipline (our experiments used preemptive fixed priority scheduling).

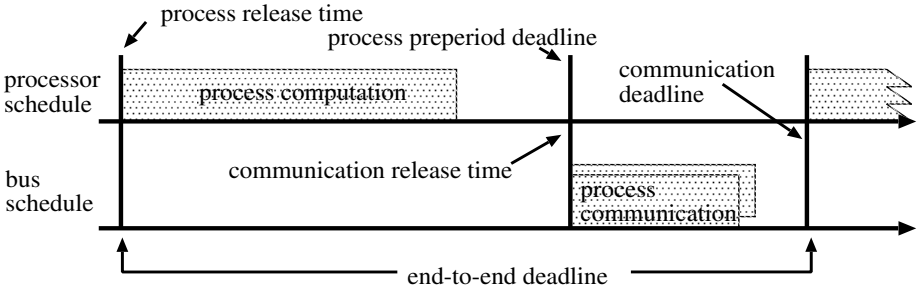


Fig. 4. Notional Decomposition Scheduling Timeline

We performed experiments using a highly abstracted workload from the Boeing 777 aircraft information management system (113 tasks, 6 processors, 50% bus utilization), and an early development workload from the Comanche mission equipment package (281 tasks, 24 processors). We constructed synthetic workloads by "connecting" multiple copies until the bus became unschedulable. Our prototype tool was able to schedule a very abstracted B777 problem in under 1 second, and was able to schedule 6 connected Comanche systems (1686 tasks, 144 processors, 57% bus utilization) in 3 seconds of Sparc Ultra-2 CPU time. For comparison, the University of Maryland required 23 hours of CPU time to produce a schedule for another abstraction of the B777 problem using a simulated annealing approach[15]. The carefully tuned production scheduling tool, which uses constraint programming techniques, required a few hours to produce a schedule for the fully detailed problem.

Our approach and preliminary results are similar to those of García and Harbour[16], although we use a different decomposition algorithm at each iteration. Our prototype also currently only schedules chains of length two (one task and its outgoing messages, with a direct bus available between sender and receiver).

The iterative nature of the approach means it might be adapted to work incrementally, quickly producing a new schedule as a small change to an existing feasible schedule. This might make possible on-line adaptation to changes in large distributed hard real-time workloads.

3.2 Slack Stealing

Traditional control applications use periodic task and communication models, but many applications use event-triggered interactive task models. It is a challenge to mix the two types of workloads in a way that guarantees periodic task deadlines, provides quick response times and high throughputs to the aperiodic tasks, and achieves high processor utilizations. Specific examples of challenge problems are the hosting of a telecommunication application, or a TCP/IP stack, or a Real-Time CORBA ORB, on the same system that also supports vehicle control and flight management applications.

We have been developing slack stealing methods to address this need. Slack stealing, as first proposed in [20], is a preemptive processor scheduling algorithm that delays the execution of high priority periodic tasks to improve the response times of aperiodic tasks while guaranteeing the periodic task deadlines. An online slack server determines at each event arrival (each request for slack CPU time) how big of a time slice can be immediately granted at a particular priority level without causing any periodic deadlines to be missed.

We will illustrate slack stealing by comparing it with background service for a hypothetical set of (random) event arrivals listed in Table 1. Assume there is a single periodic task with (hyper)period $H = 10$ and compute time $C = 6$. Event arrivals are processed by a single aperiodic task. The arrival time of the n^{th} event is denoted by a_n and its corresponding departure (completion) time by d_n. The response time is $r_n = d_n - a_n$.

A background server processes aperiodic tasks only when there are no periodic tasks to be executed. The example background server execution timeline is shown in Figure 5. Suppose an aperiodic task α is in service, having completed x'_j of its execution when periodic task τ_n arrives at time $(n-1)H$. Task α will be preempted until time $(n-1)H+C$ while the periodic task executes, at which time it will resume service with a remaining execution time requirement of $x_j - x'_j$.

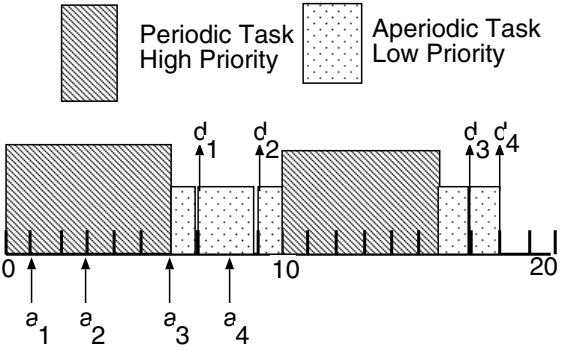


Fig. 5. Background Server Timeline

In contrast, a slack server processes aperiodic tasks at the highest priority as long as a periodic task will not miss its deadline. The example slack server execution timeline is shown in Figure 6. Let $C(t)$ be the amount of compute time an executing periodic task has consumed at time t, $0 \leq C(t) \leq \min(C, t)$. When $C(t) = C$, the value remains at C until $t = H$ and is then reset to 0. $C - C(t)$ is the compute time required by the periodic task to complete. $H - t$ is the time remaining in the current hyperperiod. The slack available in the current hyperperiod at time t is $(H - t) - (C - C(t))$. In other words, an aperiodic task α arriving at time t to an empty aperiodic queue would complete without delay caused by the execution of a periodic task provided $x_j \leq (H - t) - (C - C(t))$. If $x_j > (H - t) - (C - C(t))$ then the aperiodic task would be blocked during the interval $[t, t + H - (C - C(t))]$ while the periodic task executes and completes exactly at its deadline. Note that the periodic task is blocking aperiodic tasks in the time interval $[7, 10]$ otherwise periodic execution occurs only when no aperiodic tasks are in the system.

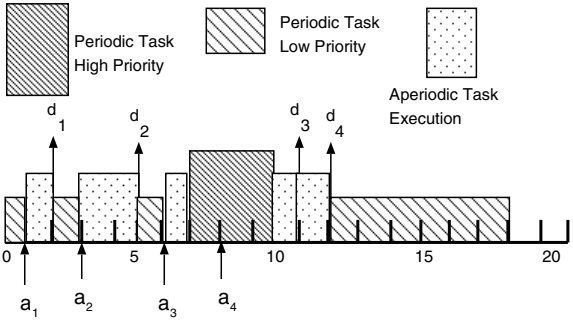


Fig. 6. Slack Server Timeline

Table 1 records the timelines of both cases in numeric form. The subscripts "bg" and "ss" refer to background server and slack server, respectively. Columns $r_{n,\text{bg}}$ and $r_{n,\text{ss}}$ show the slack server provides smaller response times than the background server.

Table 1. Fig 5 and 6 Sample Data ($H = 10, C = 6$)

id	a_n	x_n	$d_{n,\text{bg}}$	$r_{n,\text{bg}}$	$d_{n,\text{ss}}$	$r_{n,\text{ss}}$
1	1	1	7	6	2	1
2	3	2	9	6	5	2
3	6	2	17	11	11	5
4	8	1	18	10	12	4

Other aperiodic task service disciplines have been proposed, such as the deferred and sporadic servers[19,23]. Our interest in slack stealing is motivated in part by its provable optimality under certain assumptions (such as ignoring context swap and slack computation overheads), but also because this method can reclaim unused time from tasks that complete early. When periodic tasks complete in less than their worst case execution time, the unused execution time can be reallocated at the priority at which it would have been executed. This form of slack is known as reclaimed slack (timeline slack is what is shown in Figure 6). Reclaimed slack is particularly important when safety-critical applications are present because very conservative worst-case compute times are normally used to assure safety-critical deadlines. Table 2 is an augmented version of Table 1 for the execution timeline shown in Figure 7, where each execution of C actually completes after 4 units, and 2 units are reclaimed.

Table 2. Fig 7 Sample Data ($H = 10, C = 6, R = 2$)

id	a_n	x_n	$d_{n,\mathrm{ss}}$	$r_{n,\mathrm{ss}}$
1	1	1	2	1
2	3	2	5	2
3	6	2	9	3
4	8	1	10	2
5	13	4	19	6
6	17	1	20	3

Note that we differentiate between aperiodic execution on timeline versus reclaimed slack. In Figure 7, task invocations 3 and 4 are serviced on reclaimed slack (compare with Figure 6). Task invocation 5 begins its service on timeline slack (in interval $[13, 16]$), and is then preempted by the periodic task to ensure the periodic deadline. The periodic task completes early, allowing task invocation 5 to finish its execution on reclaimed slack (in interval $[19, 20]$).

Slack stealing is not a single specific algorithm and method, but rather an approach that can be applied to solve a number of blended scheduling problems. We have developed a variety of slack stealing techniques for use in actual embedded systems. Our first real-time implementation was in MetaH[8] with designed support for slack overhead accounting (e.g. context switches, blocking times, and the slack algorithms themselves), user defined criticalities, implied criticalities derived from preperiod deadlines and message passing models, and noted optimizations for harmonic periodic task sets. We later adapted slack algorithms to efficiently support incremental and design-to-time processing[9]; and then dynamic threads and time partitioning[10] in DEOS (an RTOS contracted for use in six different FAA-certified commercial jets).

We obtained significant performance improvements relative to the deferred server algorithm that was originally used in DEOS. When slack servers are used at one or both ends of a communication link with a handshaking protocol, overall throughput increases are possible because the response time for each reply-

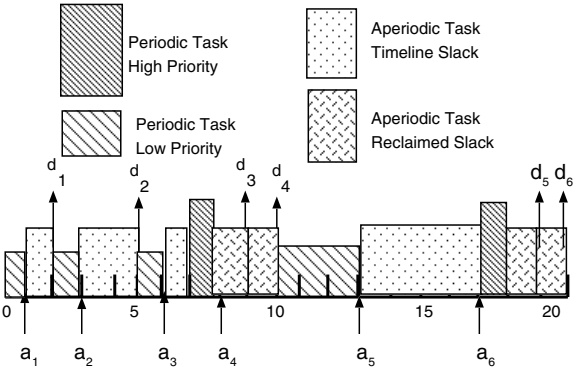


Fig. 7. Recalimed Slack Timeline

response exchange may be greatly reduced. The throughput of an FTP stack hosted on DEOS improved by a factor of 3. Perhaps more importantly, the bandwidth reserved for this application was reduced from about 70% to less than 10%, dramatically increasing the available CPU utilization for periodic applications. Slack stealing has also been used in DEOS to support incremental display tasks, where a minimum tolerable refresh update rate is guaranteed, with slack being used to typically achieve a higher refresh rate. Our algorithms provide the safe co-hosting of Level E (unverified) COTS FTP software with Level A (safety critical) software without compromising real-time performance measurements or achieveable CPU utilization.

We are currently investigating the applicability of, and extensions to, slack stealing for more complex models of task interaction, such as remote procedure calls and queueing networks; and application of some of these concepts to bus/network scheduling in distributed systems.

3.3 Response Time Analysis

In many application areas, such as telecommunications, performance is usually discussed in stochastic rather than deterministic terms. Averages alone are not sufficient, metrics based on knowledge of the response time distribution are desired (e.g. the expected percentage of requests that will be serviced within a stated deadline). A challenge problem is to analytically predict response time distributions for aperiodic tasks when they must share the CPU with periodic tasks. Our goals for analytic modeling of response time distributions in the presence of periodic tasks are

- efficient generation of aperiodic response time distribution approximations with confidence bands
- on-line parameter sensing/estimation for response time model validation, admission control, and dynamic reconfiguration
- analytic models that enable efficient bus/network scheduling for blending periodic feed-back control messages and event-triggered messages

We are investigating models for slack servers that execute at various priority levels. Figures 8 and 9 illustrate the predictions of some different analytic models plotted against simulation data for slack and background servers, respectively[11] ($H = 64$ ms, $C = 0.75H$). In the figures, the execution time X of an aperiodic task is exponentially distributed with rate μ. More compactly, $\Pr[X \leq x] = 1 - e^{-\mu x}$, where $\mu = 1$ms is the mean service rate in this example. Equivalently, the service time distribution of the aperiodic tasks is $\mathcal{E}(\mu)$. Similarly, the interarrival time distribution of the aperiodic tasks is $\mathcal{E}(\lambda)$, where $\lambda = 0.2$ms. Aperiodic traffic utilization is $0.2 = \lambda\mu^{-1}$.

We have developed new models called the long and intermediate hyperperiod models for both the slack and background servers (labeled LHM and IHM in the respective plots). In the slack server figure, we also show an M/M/1 response time plot (labeled MM1). The M/M/1 model is the theoretical response time distribution for the aperiodic traffic in the absense of any periodic traffic. The M/M/1 response time distribution is $\mathcal{E}(\mu - \lambda)$. In the background server figure we have included a heavy traffic (or diffusion) response time model [27] (labeled HTM) for theoretical comparison. In both figures, we have included a degraded server model (labeled DSM). Conceptually, a degraded server (model) simply reduces the server speed by the fraction of the CPU taken by the periodic task. The simulation data points appear as a heavy (wiggly) line.

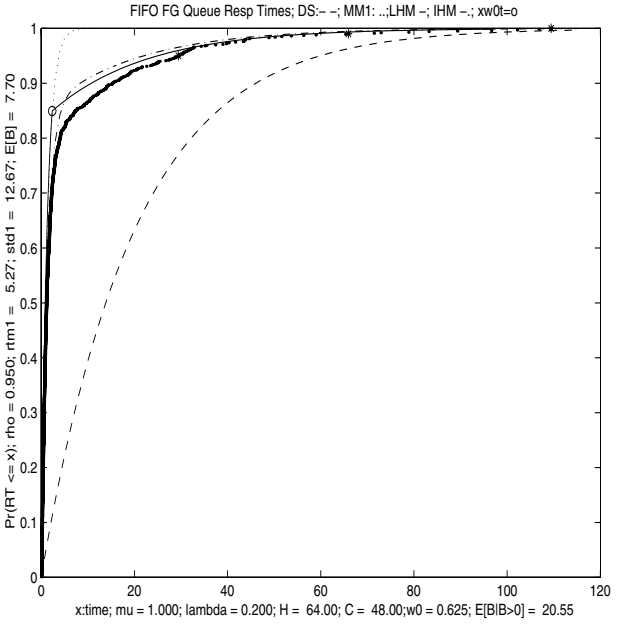


Fig. 8. Slack Server Response Time

The observed aperiodic response time distribution for a slack server in Figure 8 lies completely above the DSM response time prediction. In contrast, the observed aperiodic response time distribution when processed at background priority falls completely below the DSM response time distribution in Figure 9. For the slack server discipline in the configuration shown, the slack server long hyperperiod model gives estimates closer to the simulation data. For the background server discipline, the background server intermediate hyperperiod model gives estimates closest to the simulation data. Different models are better for different system configurations, and we have developed criteria for selecting among the different models.

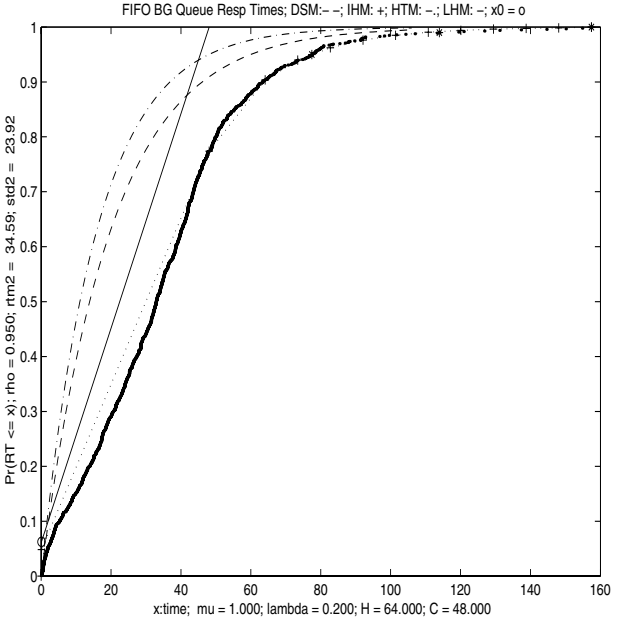


Fig. 9. Background Server Response Time

We are also investigating the impact that periodic traffic patterns have on the delay distributions of the aperiodic traffic. In many commercial bus communication protocols with integrated periodic and event-triggered traffic, bus traffic is slotted and has designated start times for time-driven messages. Event-triggered traffic also has dedicated time slots, which are usually the remaining gaps not allocated to critical periodic messages. There may be no event-triggered messages waiting at the start of a preallocated event message time slot, or many messages might have been queued for a long time. In many regards, the study of these gaps on busses is analogous to the study of event-triggered task response times when run as background tasks on a CPU with predefined scheduling times for critical time-triggered periodic tasks. We have found that the spacing and size of

the aperiodic gaps can have significant impact on the response delivery time distributions, suggesting it is possible to improve bus performance by appropriate scheduling of these gaps.

3.4 Hybrid Automata

Traditional real-time task models cannot easily deal with variability and uncertainty in clock and computation and communication times, synchronizations (rendezvous) between tasks, remote procedure calls, anomalous scheduling in distributed systems, dynamic reconfiguration and reallocation, end-to-end deadlines, and timeouts and other error handling behaviors. One of our goals is to analyze the schedulability of real-time systems that cannot be easily modeled using traditional scheduling theory. For example, we would like to be able to model and analyze a system of tens of tasks on a few processors, where tasks may make remote procedure calls to each other, may have complex internal behaviors (multiple internal states with state transitions dependent on inter-task interactions), and have hard deadlines between specified pairs of state transitions.

At the implementation level, task schedulers and communication protocols are reactive components that respond to events like interrupts, message arrivals, service calls, task completions, error detections, etc. Another of our goals is to model and verify implementations of real-time functions. We would like to model important implementation details such as control logic and data variables. We would like the mapping between model and code to be clear and simple to better assure that the model really does describe the implementation. For example, we would like to be able to model and verify a real-time scheduler or a real-time bus driver.

We have been working with linear hybrid automata models of such systems[25]. Our experience suggests these are powerful and natural models for complex real-time system behaviors. However, computational intractability is currently a much more severe problem for hybrid automata model checking than it is for finite state model checking. We overcame some of these problems by developing our own prototype reachability tool that uses polynomial-time algorithms to compute hybrid state transitions, uses an oracle to concisely encode the scheduling semantics for a particular model, and does on-the-fly identification of reachable discrete states. For example, we were able to solve some problems having 100 times more discrete states than we could with other tools, although our prototype does not currently support rate ranges or provide parametric analysis. We also showed that the reachability problem becomes decideable under restrictions that are very reasonable for this problem domain.

We demonstrated these technologies by formally verifying key behaviors of the core scheduling and time partitioning modules of the MetaH system executive. The standard executive library modules were modified by inserting calls to generate linear hybrid automata models of the code that manages basic scheduling and time partitioning functions (excluding slack stealing and dynamic reconfiguration features). Time-varying variables were used to model hardware timers and accumulated process compute time. Zero-rate variables were used to model some variables in the code. A complete linear hybrid automata model for this

portion of the system executive (about 1800 SLOC, roughly half of the executive) was automatically generated by executing each subprogram independently with test data, i.e. automatically generated as a by-product of unit testing. This complete model was then subjected to a reachability analysis to verify timing and time partitioning requirements and assertions in the code. We analyzed a set of applications that was sufficient to achieve full coverage of the modeled code.

Our work to date suggests the technology has passed the threshold of utility for verifying implementations of certain real-time functions. However, significant improvements in analysis techniques are needed in order to analyze and verify the schedulability of complex real-time workloads of non-trivial size.

3.5 System Safety

Our MetaH toolset supports a construct called an error model, which allows users to specify sets of fault events and error states. An error model includes specifications of transition functions to define how the error states of objects change due to fault, error propagation and recovery events. An individual object within a specification can be annotated to specify the error model and fault arrival and error propagation rates for that object. An AADL specification also contains consensus expressions to describe the error detection protocols implemented in source modules, and to specify the conditions under which system operation is acceptable (e.g. when at least 2 out of 3 subsystems are operational).

We have a prototype reliability modeling tool that generates a stochastic concurrent process reliability model[18,22]. Error propagations between objects are modeled as synchronizations or rendezvous between stochastic concurrent processes. Error propagation synchronizations in the model can be controlled (guarded) using an associated consensus expression, which can conditionally mask propagations depending on the current error states of selected objects. The reliability modeler uses the error model specifications and annotations to generate the object error state machines, and uses the consensus expressions and design structure to generate the propagation synchronizations between these object error state machines. A subset of the reachable state space of this stochastic concurrent process is a Markov chain that can be analyzed using existing tools and techniques[21].

We selected a stochastic concurrent process model because it allowed us to easily generate a hierarchical reliability model whose structure is intuitively traceable back to the original specification and vice versa. The tool generates the stochastic concurrent process model in human-readable form for use during design and implementation review.

We believe this work substantiates the basic concept of generating a reliability model from a design specification, but we have identified a number of shortcomings that must be addressed[13]. Markov model generation and analysis is subject to state space explosion; features in the language to control abstraction in the generated model, and state space optimization methods in the analysis tool, are needed. Analysis results that are easily traceable back to the specification and include parametric sensitivity data are needed.

Markov reliability analysis and partition isolation analysis are only two types of analysis used in a comprehensive system safety program[26]. The language already contains some features for fault tree specification, but features are needed to capture the results of hazard analysis and failure modes and effects analysis and summary. All these different analyses are synergistic and related, and the analysis toolset should perform certain consistency checks between the different models and results. For example, basic events in a fault tree are assumed to be statistically independent, and all common cause analyses (such as partition isolation analysis) should check for the absence of common causes for pairs of fault tree basic events.

System safety program activities must be closely integrated with development activities, and safety data and analyses must be clearly traceable to development work products such as designs and code[3,2]. Design assurance standards require review and analysis in addition to testing, and encourage as high a degree of formal analysis as is practical[1]. The well-integrated and formalized development process and environment that we are pursuing can significantly contribute to meeting these requirements.

3.6 Dynamic Reconfiguration

Our concept of dynamic reconfiguration extends beyond adaptive scheduling. We want to allow changes to be made to the mix of applications currently running, we want to allow the architecture of a running system to change. For systems that host applications having varying design assurance levels up to safety-critical, we are evaluating mixtures of off-line specification and analysis with on-line reconfiguration and enforcement.

The current AADL allows hierarchical specification of alternative system and subsystem configurations, called modes. A mode change can start and stop processes and change the pattern of event and data connections in effect. The current MetaH toolset enumerates and analyzes modes off-line, e.g. the schedulability analysis report includes a section for each processor and each bus in each potential mode of operation. Many generated tables in the system executive have the current mode of operation as an index, and the generated code contains case statements to make operation dependent on the current mode where needed.

It should be possible in principle to shift mode enumeration and analysis from off-line to on-line in some cases. Part of the workload of the system would be a process to enumerate, pre-analyze and cache modes reachable from the current mode. A run-time request for a mode change would be subject to on-line admission control, it would be permitted only if the requested mode had been pre-approved. Some of the challenging problems here are insuring that a mode change request be dealt with acceptably (e.g. it can either be rejected or the request modified to a pre-approved mode) and assuring the correctness of complex on-line mode enumeration and analysis and executive configuration functionality.

It should be possible in principle to allow system managers to modify and re-verify the architectural specification of a system off-line and then upgrade the running system. One approach to this is to adopt a change process that always

results in a common mode of operation between old and new specifications. The running system is then placed in this common mode, the new executive configuration tables and code downloaded while the system operates, and finally an atomic change from old to new executive tables and code performed. Some of the challenging problems here are developing a suitable change and verification process, and the complex details of performing high-assurance downloads and upgrades in a distributed system.

The architecture of a large and dynamic system of systems cannot always be statically specified, but one can statically specify classes of components. For example, a system might consist of an unspecified number of aircraft, but with a finite number of known types of aircraft. The avionics architecture of each type of aircraft would have an AADL specification. From the computer systems perspective, we might wish to show properties such as the schedulability of each individual avionics system given conditions on the number of, and potential interactions between, aircraft. Theorem proving methods, or model checking methods that symbolically encode certain patterns of infinite states, will probably be required.

References

1. *Software Considerations in Airborne Systems and Equipment Certification*, RTCA/DO-178B, RTCA, Inc., Washington D.C., December 1992.
2. *Software System Safety Handbook*, Joint Software System Safety Committee, December 1999, www.nswc.navy.mil/safety/handbook.pdf
3. *Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment*, SAE/ARP 4761, December 1996.
4. *MetaH User's Guide*, Honeywell Laboratories, 3660 Technology Drive, Minneapolis, MN, www.htc.honeywell.com/metah.
5. *Domain Modeling Environment*, Honeywell Laboratories, 3660 Technology Drive, Minneapolis, MN, www.htc.honeywell.com/dome.
6. Rajeev Alur, Thomas A. Henzinger, and Pei-Hsin Ho, "Automatic Symbolic Verification of Embedded Systems," *IEEE Transactions on Software Engineering*, vol. 22, no. 3, March 1996, pp 181-201.
7. Neil C. Audsley, Alan Burns, Robert I. Davis, Ken W. Tindell and Andy J. Wellings, "Fixed Priority Pre-emptive Scheduling: An Historical Perspective," *Journal of Real-Time Systems*, 8, pp 173-198.
8. Pam Binns, "Scheduling Slack in MetaH," *Real-Time Systems Symposium*, work-in-progress session, December 1996.
9. Pam Binns, "Incremental Rate Monotonic Scheduling for Improved Control System Performance," *Real-Time Applications Symposium*, 1997.
10. Pam Binns, "A Robust High-Performance Time Partitioning Algorithm; The Approach Taken in DEOS," to appear *20^{th} Digital Avionics Systems Conference*, November 2001
11. Pam Binns, *Aperiodic Response Time Distributions in Queues with Deadline Guarantees for Periodic Tasks*, Ph.D. Thesis, Department of Statistics, University of Minnesota, October 2000.
12. Pam Binns and Steve Vestal, "Message Passing in MetaH using Precedence-Constrained Multi-Criticality Preemptive Fixed Priority Scheduling," *Life Cycle Software Engineering Conference*, Redstone Arsenal, AL, August 2000.

13. Pam Binns, Steve Vestal, William Sanders, Jay Doyle and Dan Deavours, "MetaH/Möbius Integration Report," prepared by Honeywell Laboratories and University of Illinois Coordinated Science Laboratory, prepared for U.S. Army AMCOM Software Engineering Directorate, April 2000.
14. S. Campos, E. Clarke, W. Marrero, M. Minea and H. Hiraishi, "Computing Quantitative Characteristics of Finite-State Real-Time Systems," *Proceedings IEEE Real-Time Systems Symposium*, December 1994.
15. Shent-Tzong Cheng and Ashok K. Agrawala, "Allocation and Scheduling of Real-Time Periodic Tasks with Relative Timing Constraints," University of Maryland Department of Computer Science Technical Report, 1993.
16. José Javier Gutiérrez García and Michael González Harbour, "Optimized Priority Assignment for Tasks and Messages in Distributed Hard Real-Time Systems," *Third Workshop on Parallel and Distributed Real-Time Systems*, April 1995.
17. Bruce Lewis, "Software Portability Gains Realized with MetaH, an Avionics Architecture Description Language," 18th *Digital Avionics Systems Conference*, St. Louis, MO, October 24-29, 1999.
18. Holger Hermanns, Ulrich Herzog and Vassilis Mertsiotakis, "Stochastic Process Algebras as a Tool for Performance and Dependability Modeling," *Proceedings of the IEEE International Computer Performance and Dependability Symposium* (IPDS'95), April 24-26, 1995, Erlangen, Germany.
19. J. P. Lehoczky, L. Sha and J. K. Strosnider, "Enhanced Aperiodic Responsiveness in Hard Real-Time Environments," *Proceedings IEEE Real-Time Systems Symposium*, 1987, pp 261-270.
20. J. P. Lehoczky and S. Ramos-Thuel, "An Optimal Algorithm for Scheduling Aperiodic Tasks in Fixed-Priority Preemptive Systems," *Proceedings IEEE Real-Time Systems Symposium*, December 1992.
21. W. H. Sanders, W. D. Obal, M. A. Quershi and F. K. Widjanarko, "The UltraSAN Modeling Environment," *Performance Evaluation Journal*, vol. 25 no. 1, 1995.
22. Frederick T. Sheldon, Krishna M. Kavi and Farhad A. Kamangar, "Reliability Analysis of CSP Specifications: A New Method Using Petri Nets," *Proceedings of AIAA Computing In Aerospace*, San Antonio, TX, March 28-30, 1995.
23. B. Sprunt, L. Sha and J. P. Lehoczky, "Aperiodic Task Scheduling for Hard Real-Time Systems," *Journal of Real-Time Systems*, 8, 1998, pp 27-60.
24. Steve Vestal, "Fixed Priority Sensitivity Analysis for Linear Compute Time Models," *IEEE Transactions on Software Engineering*, April 1994.
25. Steve Vestal, "Modeling and Verification of Real-Time Software Using Extended Linear Hybrid Automata," *NASA Langley Formal Methods Workshop*, June 2000, shemesh.larc.nasa.gov/fm/Lfm2000/Proc/
26. Steve Vestal, "MetaH Avionics Architecture Description Language Software and System Safety and Certification Study," prepared by Honeywell Laboratories, prepared for U.S. Army AMCOM Software Engineering Directorate, March 2001.
27. Ward Whitt, "Weak Convergence Theorems for Priority Queues: Preemptive Resume Discipline," *Journal of Applied Probability*, Volume 8, pp. 74-94, 1971

Reliable and Precise WCET Determination for a Real-Life Processor

Christian Ferdinand[1], Reinhold Heckmann[1],
Marc Langenbach[2], Florian Martin[2], Michael Schmidt[2],
Henrik Theiling[2], Stephan Thesing[2], and Reinhard Wilhelm[2]

[1] AbsInt Angewandte Informatik GmbH, Saarbrücken[***]
[2] Fachrichtung Informatik, Universität des Saarlandes, Saarbrücken[†]

Abstract. The USES-group at the Universität des Saarlandes follows an approach to compute reliable run-time guarantees which is both well-based on theoretical foundations and practical from a software engineering and an efficiency point of view. Several aspects are essential to the USES approach: the resulting system is modular by structuring the task into a sequence of subtasks, which are tackled with appropriate methods. Generic and generative methods are used whenever possible. These principles lead to an understandable, maintainable, efficient, and provably correct system. This paper gives an overview of the methods used in the USES approach to WCET determination. A fully functional prototype system for the Motorola ColdFire MCF 5307 processor is presented, the implications of processor design on the predictability of behavior described, and experiences with analyzing applications running on this processor reported.

1 Introduction

Real-time systems are subject to stringent timing constraints which are dictated by the surrounding physical environment. Failure of a safety-critical real-time system can lead to considerable damage and even loss of lives. Therefore, a schedulability analysis has to be performed in order to guarantee that all timing constraints will be met ("timing validation"). All existing techniques for schedulability analysis require the worst-case execution time (WCET) of each task in the system to be known prior to its execution. Since this is not computable in general, estimates of the WCET have to be calculated. These estimates have to be safe, i. e., they must never underestimate the real execution time. Furthermore, they should be tight, i. e., the overestimate should be as small as possible. For processors with fixed execution times for each instruction there are established

[***] This work is supported by the RTD project IST-1999-20527 DAEDALUS of the European FP5 programme.

[†] Work of the USES group (**U**niversity of the **S**aarland **E**mbedded **S**ystems group) is partially supported by TFB 14 of the Deutsche Forschungsgemeinschaft and by the RTD project IST-1999-20527 DAEDALUS of the European FP5 programme.

T.A. Henzinger and C.M. Kirsch (Eds.): EMSOFT 2001, LNCS 2211, pp. 469–485, 2001.
© Springer-Verlag Berlin Heidelberg 2001

methods to compute sharp WCET bounds [PK95,PS91]. However, in modern microprocessor architectures caches and pipelines are key features for improving performance. Caches are used to bridge the gap between processor speed and the access time of main memory. Pipelines enable acceleration by overlapping the executions of different instructions. The consequence is that the execution behavior of the instructions cannot be analysed separately since this depends on the execution history. Therefore, the classical approaches to worst-case execution time prediction are not directly applicable or lead to results exceeding the real execution time by orders of magnitude. This may influence the degree of success of timing validations or may lead to a waste of hardware resources. For many mass products, e. g., in the automotiv or telecommunications industries this waste may have a tremendous effect on system costs.

2 The USES Approach to WCET Computation

2.1 Rationale and Essentials

Several aspects are essential to the USES approach:

Modularity: As far as possible for a given processor architecture, the task of determining the WCET is structured into a sequence of subtasks that can be solved in isolation. Interference between processor components may prevent this modularity.

Adequacy of methods: Each sub-task is tackled by appropriate methods. This guarantees efficiency of computation and precision of results.

Genericity: Generic and generative methods are used whenever possible. This allows a large number of different processor architectures and the evolution of processor families to be properly dealt with.

Separation into phases. The determination of the WCET of a program is composed of several different tasks: computation of address ranges for instructions accessing memory by a *value analysis*, classification of memory references as cache misses or hits, called *cache analysis*, predicting the behavior of the program on the processor pipeline, called *pipeline analysis*, and the determination of the worst-case execution path of the program, called *path analysis*. Many of these tasks are quite complex for modern microprocessors and DSPs. Combining them into a single analysis adds complexity.

An arrangement into a sequence of phases would limit complexity. The sequence of the value, cache, pipeline and path analyses was chosen in the first experiments using the SPARC processor [Fer97]. The results of the value analysis were used by the cache analysis to predict the behavior of the (data) cache. The results of the cache analysis were fed into the pipeline analysis allowing the prediction of pipeline stalls due to cache misses. The combined results of the cache and pipeline analyses were used to compute the execution times of program paths. The separation of WCET determination into several phases has the

additional effect that different methods tailored to the subtasks can be used. In our case, the value analysis, the cache analysis, and the pipeline analysis were done by *abstract interpretation*, a semantics based method for static program analysis. Path analysis is done by integer linear programming. The precision of the results and the efficiency of the WCET computation were convincing.

This serialization of subtasks works if there are no cyclic dependencies of processor components on one another which lead to cyclic dependencies of the phases on one another. However, some architectures create dependencies between these tasks which do not allow a straightforward serialization. This is another instance of a *phase ordering problem* encountered frequently in compiler design [WM95].

There are also cases when serialization may lead to a loss of precision which is partly caused by the use of static program analysis, but is unavoidable with any static approach. Cache analysis assumes that all program paths are executable. Cache analysis may collect cache information from non-executable paths thereby corrupting precision, albeit not correctness. Path analysis could model the control flow in such a fine-grained way so as to exclude some of these paths. Hence, there is a slight chance of losing precision by separating cache analysis from path analysis.

Architectural features may ruin the unidirectional dependence between cache analysis and pipeline analysis: for example Motorola ColdFire has two pipelines (a *fetch* and an *execute pipeline*) coupled by an instruction buffer. The order of memory references and, hence, the effects on the cache depend on the pipeline behavior, which in turn depends on the number of prefetched instructions in the buffer.

Generic and generative methods. Several problems can be solved by generic or generative methods. A *generic* method is realized with some formal parameters unspecified. Cache analysis for most processors can be implemented generically by abstracting away from the parameters *capacity, associativity, line size,* and *replacement strategy.* An approach is *generative* if the implementation is generated from some specification by precomputation. For example, the generic cache analysis is generated from a (generic) specification of the domain of *abstract cache states* and a specification of *abstract semantic functions.* The latter was made possible by using abstract interpretation, a well-established method offering a host of available theoretical results. The theory underlying abstract interpretation makes this generative support possible. As context-free parsers are generated using `yacc` or `bison`, analyses can be generated from appropriate specifications by generators. Much of the machinery of each individual analysis is reusable. In our case, the *Program Analysis Generator* `PAG` [Mar99,Mar98] is used.

The generative approach has the advantage of separating specification and implementation. Specifications are more easily understood than hand-coded analyses. Modifying and extending of an analysis by changing the specification is easier than changing hand-written implementations.

Frontends reading assembly or executable code are also generated from descriptions. They translate into a common intermediate language, CRL, described in [Lan98]. Analyses and transformations of machine programs can then be performed on CRL representations.

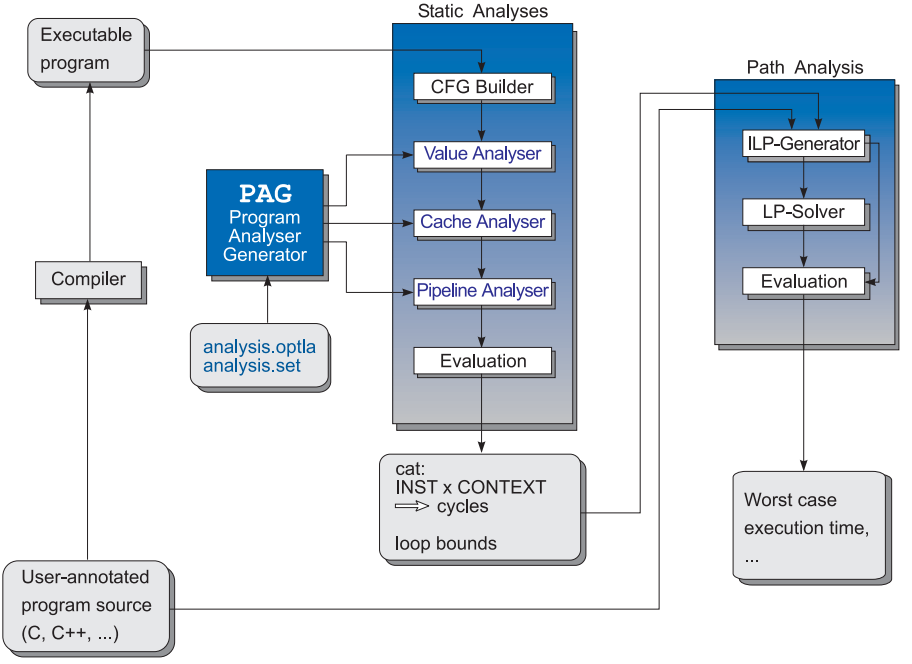


Fig. 1. The structure of the analysis

2.2 Underlying Framework

Composition of the Framework. The starting point of our analysis framework (*see* Figure 1) is a binary program and additional user-provided information about numbers of loop iterations, upper bounds for recursion, etc. Plans have been made to derive more information from the program source if available so that the user has to provide only a minimum of annotations.

In the first step a parser reads the compiler output and reconstructs the control flow [The00,The01]. This requires some knowledge about the underlying hardware, e. g., which instructions represent branches or calls. The reconstructed control flow is annotated with the information needed by subsequent analyses

and then translated into `CRL` (Control Flow Representation Language)[1]. This annotated control flow graph serves as the input for microarchitecture analyses:

The value analysis (based on [Sic97]) determines ranges for values in registers and by this it can resolve indirect accesses to memory. The cache analysis classifies the accesses to main memory, i. e., whether or not the needed data resides in the cache. The pipeline analysis models the pipeline behavior to determine execution times for a sequential flow of instructions. The result is an execution time for each instruction in each distinguished execution context.

Following from the results of the microarchitecture analyses, the path analysis determines a safe estimate of the WCET. First an ILP generator models the program's control-flow using an integer linear program and a mapping from variable names to basic blocks which will be needed later on. The resulting integer linear program is solved by `lp_solve` [2]. The value of the objective function in the solution is the predicted worst-case execution time for the input program. The ILP solution is then evaluated using the variable mapping resulting in execution and traversal counts for every basic block and edge. The path analysis is generic in such a way that it abstracts from the underlying target machine.

3 Program Analysis to Predict Run-Time Behavior

Program analysis is a widely-used technique to determine the runtime properties of a given program without actually executing it. Such information is used, for example, in optimizing compilers [ASU86,WM95] to detect the applicability of efficiency-improving program transformations. A program analyzer takes a program as input and attempts to determine properties of interest. It computes an approximation to an often undecidable or very hard to compute program property. There is a well-developed theory of program analysis, *abstract interpretation* [CC77,NNH99]. This theory states criteria for correctness and termination of a program analysis. A program analysis is considered an abstraction of the standard semantics of the programming language.

The standard (operational) semantics of a language is given by a *concrete domain* of data and a set of functions describing how the statements of the language transform concrete data. The *abstract semantics* then consists of a (simpler) abstract domain and a set of abstract semantic functions, so called transfer functions, for the program statements computing over the abstract domain.

The designer of a program analysis must define the *abstract domain*. It is obtained from the concrete domain by abstracting from all aspects up to those which are the subject of the analysis to be designed.

Both domains are usually complete partially ordered lattices of values. The partial order in the abstract domain corresponds to precision, i.e. quality of

[1] `CRL` is a human-readable intermediate format designed to simplify analyses and optimizations at the executable/assembly level.

[2] `lp_solve` solves LP, ILP, and MILP problems. The latest version is to be found on ftp://ftp.ics.ele.tue.nl/pub/lp_solve.

information. By agreement, elements higher up in the order are considered to contain less information[3], i.e. to be less precise.

The partial order determines the *least upper bound* operation, $\sqcup$, on the lattice, which is used to combine information stemming from different sources, e.g. from several possible control flows into one program point.

The designer must also define the *transfer functions*. They describe how the statements transform abstract data. They must be monotonic in order to guarantee correctness and termination.

3.1 Value Analysis

The *value analysis* computes for each processor register an interval of possible values as approximations to the values occurring during runtime. To do this, abstract versions of all processor instructions have to be modeled which are based on interval values as operands. This includes not only simple arithmetic operations like *add* or *mul*, but also complex addressing modes like *register indirect with scaled index* to approximate the effective addresses of memory references.

The $\sqcup$ operator for merging two abstract register or memory cell values is a simple union of intervals:

$$\sqcup : D_{abs} \times D_{abs} \longrightarrow D_{abs}, [l_1, u_1] \sqcup [l_2, u_2] := [min(l_1, l_2), max(u_1, u_2)]$$

Since registers and memory cells have a finite precision, the detection of (possible) overflows requires special attention to compute a *correct* approximation. For example, the *add* instruction is implemented as follows:

$$add : D_{abs} \times D_{abs} \longrightarrow D_{abs},$$
$$[l_1, u_1] + [l_2, u_2] := \begin{cases} [l_1 + l_2, u_1 + u_2] & \text{if no overflow is possible,} \\ \text{unknown} & \text{otherwise} \end{cases}$$

In some cases the value analysis enables the detection of infeasible paths, e.g., for a conditional branch instruction, where the approximated values indicate, that a branch condition always (or never) holds. The information about infeasible paths is forwarded to the cache and pipeline analyses to improve analysis quality by reducing the number of combine operations.

3.2 Cache Analysis

Two analyses have been designed and implemented in different variations corresponding to different cache architectures. A fully associative cache with an LRU replacement strategy is assumend in the following explanations. More complete descriptions that explicitly describe direct mapped and A-way set associative

[3] Things would work equally well the other way round due to the duality principle of lattice theory.

caches are to be found in [Fer97,FMW97,FMWA98]. Section 4 describes a cache architecture with quite a different replacement strategy wich exerts a bad influence on prediction quality.

The *must analysis* determines a set of memory blocks that are in the cache at a given program point whenever execution reaches this point. The *may analysis* determines all memory blocks that may be in the cache at a given program point. The complement of the information derived by this analysis is used to determine the absence of a memory block in the cache.

These analyses are used to compute a categorization for each memory reference describing its cache behavior. The categories are described in Table 1.

Table 1. Categorizations of memory references and memory blocks.

Category	Abb.	Meaning
always hit	ah	The memory reference will always result in a cache hit.
always miss	am	The memory reference will always result in a cache miss.
not classified	nc	The memory reference could neither be classified as ah nor am.

Must Analysis. The must analysis determines a set of memory blocks that are in the cache at a given program point whenever execution reaches this point. "Good" information is the knowledge that a memory block is in this set. The bigger the set, the better the exploitable information. As we will see, additional information will even tell how long a memory block will remain in the cache at minimum. This is linked to the "age" of a memory block. Therefore, the partial order on the *must*–domain is as follows. Above some abstract cache state in the domain, i.e., less precise, are states where memory blocks in this state are either missing or are older than here. Therefore, applying the $\sqcup$ operator to two abstract cache states will produce a state containing only those memory blocks contained in both and will give them the maximum of their ages in the operand states (see Figure 2). Thus the positions of the memory blocks in the abstract cache state are the upper bounds of the *ages* of the memory blocks in the concrete caches occurring in the collecting cache semantics.

May Analysis. In order to determine whether a memory block will never be in the cache, the complementary information is computed, i.e., the sets of memory blocks that *may* be in the cache. "Good" information is that a memory block is not in this set, because this memory block can be classified as definitely not being in the cache whenever execution reaches the given program point. Thus, the smaller the sets are, the better. Additionally, the older blocks will reach the desirable situation of being removed from the cache faster than the younger ones. Therefore, the partial order in this domain is as follows: above some abstract cache state in the domain, i.e., less precise, are those states which contain additional memory blocks or where memory blocks in these state are younger than

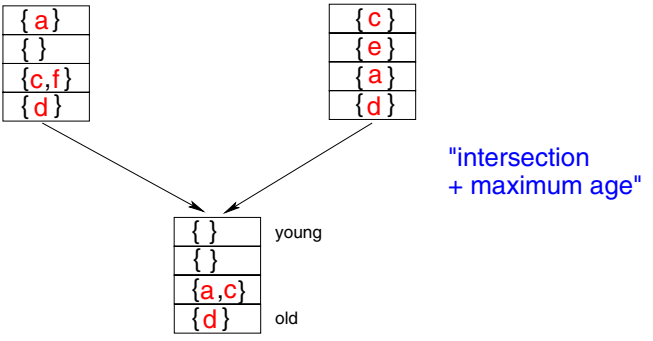


Fig. 2. Upper Bound in the Must Analysis

in the given state. Therefore, the $\sqcup$ operator applied to two abstract cache states will produce a state containing those memory blocks contained in at least one of the operand states and will give them the minimum of their ages (see Figure 3).

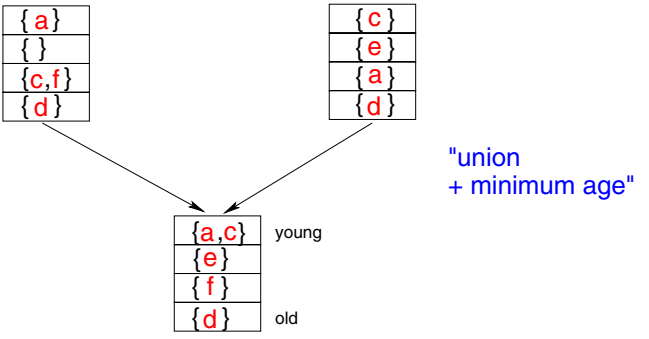


Fig. 3. Upper Bound in the May Analysis

Thus the positions of the memory blocks in the abstract cache state are the lower bounds of the *ages* of the memory blocks in the concrete caches occurring in the collecting cache semantics.

Analysis of Loops and Recursive Procedures. Loops and recursive procedures are of special interest, since programs spend most of their runtime there. Treating them naïvely when analyzing programs for their cache and pipeline behavior will result in a high loss of precision.

Frequently, the following observation can be made: the first execution of the loop body usually loads the cache, and subsequent executions find most of their referenced memory blocks in the cache. Hence, the first iteration of the loop

often encounters cache contents quite different from those of later iterations. This has to be taken into account when analyzing the behavior of a loop on the cache. A naïve analysis would combine the abstract cache states from the entry to the loop and from the return from the loop body, thereby losing most of the contents. Therefore, it is useful to distinguish the first iteration of loops from the others.

In the `USES` approach, a method has been designed and implemented in the program analyzer generator `PAG`, which virtually unrolls loops once, the so-called VIVU approach [MAWF98]. Memory references are now considered in different execution contexts, essentially nestings of first and non-first iterations of loops. Experiments have shown VIVU approach to have enormously increased precision [Fer97].

The program analysis techniques thus analyze instruction/context pairs of an input program. A context represents the execution stack, i.e., the function calls and loops along the corresponding path in the control-flow graph to the instruction. It is represented as a sequence of first and recursive function calls ($call_f_f, call_f_r$) and first and other executions of loops ($loop_l_f, loop_l_o$) for the functions f and (virtually) transformed loops l of a program. $INST$ is the set of all instructions *inst* in a program. $CONTEXT$ is the set of all execution contexts *context* of a program. IC is the set of all instruction/context pairs *ic*.

$$CONTEXT = \{call_f_f, call_f_r, loop_l_f, loop_l_o\}^*$$
$$IC = INST \times CONTEXT$$
$$cat : IC \rightarrow \{\mathsf{ah}, \mathsf{am}, \mathsf{nc}\}$$

3.3 Pipeline Analysis

[SF99] describes the foundations of pipeline analysis and an experiment done for the superscalar pipeline of the SuperSPARC I. Again abstract interpretation was used. The way from a concrete pipeline to an abstract pipeline consists of several steps. Most complex pipelines are badly documented by the manufacturers. To be of the safe side, the initial model of the pipeline already reflects pessimistic assumptions about undocumented features. A further pessimizing step may be necessary to simplify the representation of pipeline states. The abstract state is finally a superset of this type of pessimisms of "concrete" pipeline states which could actually occur at this program point.

The abstract pipeline update function reflects what happens when the pipeline is advanced by one step. It takes into account the current set of pipeline states, in particular the resource occupancies, the contents of the prefetch queue, the grouping of instructions, and the classification of memory references as cache hits or misses.

At control-flow merging points, the two abstract pipeline states are combined by set union. This leads to supersets of concrete pipeline states being computed. Section 4 will describe a very complex pipeline model that is the basis for the ColdFire analyzer.

4 Analyzing the ColdFire 5307

The ColdFire family of microcontrollers is the successor of the well known M68k architecture of Motorola. The ColdFire 5307 is an implementation of the Version 3 ColdFire architecture, see [Col97]. It contains an on-chip 4K SRAM, a unified 8K data/instruction cache, and a number of integrated peripherals. It implements a subset of the M68k opcodes, restricting opcodes to two, four or six bytes, thereby simplifying the decoding hardware. The CPU core and the external memory bus can be clocked with different speeds (e.g. 20 MHz bus clock and 60MHz internal core clock).

In the DAEDALUS project, we developed and implemented a complete WCET analysis for the MCF 5307, consisting of a value analysis, an integrated cache and pipeline analysis and a path analysis. In addition to the raw information about the WCET, several parts can be visualized by the **aiSee** tool [aiS] to view detailed information delivered by the analysis.

4.1 The ColdFire Cache

The MCF 5307 has a unified data/instruction cache of 8K. The cache is organized into 128 sets with 4 lines of 16 bytes in each set. Each line contains an additional tag, containing the upper address bits of the block stored in that line and status bits denoting whether that line contains a block or whether the data in that line has been modified.

Caching behavior can be configured for up to three memory areas. An area can be configured as uncacheable or cacheable with write-back or write-through caching.

Data and instruction accesses are mapped to one of the 128 sets by taking bits 4 ... 10 of the address as a 7-bit index. All four lines in the set are then searched for the desired data on a read access by comparing the tags with the upper address bits of the access. If a read misses, i.e., if the desired data is not in the cache, then the data is loaded from main memory and placed into the cache. The requested word in the cacheline is read first, so that the desired data can be delivered quickly back to the CPU core[4]. If a new line has to be loaded on a cache miss, the MCF 5307 selects the line receiving the new data by the following algorithm:

- If a line of the set does not contain valid data then the new data is placed into that line. If several such lines exist, the one with the lowest number is taken.
- If all lines contain valid data, then a *global replacement counter* of two bits is used to give the number of the line to place the data into. The counter is afterwards incremented by one.

Accesses that hit the cache do not influence the replacement counter.

[4] The remaining three words of the line are loaded from main memory in the background.

Example: Assume a program accesses the memory blocks 0, 1, 2, 3, ... , and block i is put in set i mod 128. Such a scenario corresponds to a linear program without memory access (all data are in registers or in the non-cacheable SRAM area). Assume further the program starts with an empty cache. Then blocks 0–127 are put into the first line of each set, blocks 128–255 into the second line, 256–383 into the third line, and 384–511 into the fourth line. The resulting cache state is as follows:

Line 0	0	1	2	3	4	5	···	127
Line 1	128	129	130	131	132	133	···	255
Line 2	256	257	258	259	260	261	···	383
Line 3	384	385	386	387	388	389	···	511

where the columns represent sets. The next memory block 512 is put into set 0. The counter has not been used so far, and still has value of 0, hence block 512 is put into line 0 and replaces block 0. The counter is now set to 1, and so, block 513 is put into line 1 of set 1, replacing block 129. Continuing like this until block 639, the resulting cache state is

Line 0	**512**	1	2	3	**516**	5	···	127
Line 1	128	**513**	130	131	132	**517**	···	255
Line 2	256	257	**514**	259	260	261	···	383
Line 3	384	385	386	**515**	388	389	···	**639**

where the recently added blocks are shown in boldface. Block 640 then replaces 512, 641 replaces 513, etc.

This example shows, that some blocks (like 1 and 128) may stay in the cache forever although they are never referenced again, while other blocks (like 512 and 513) are removed from the cache when their set is referenced for the next time. Although these remarks in their full strength only hold in this especial example, they show that in general, one must take into account that some blocks may survive many cache updates, while others are thrown out immediately.

For our static cache analysis this "pseudo round robin" replacement strategy has serious consequences. Since we cannot define abstract cache states as unions of concrete cache states for space reasons, we have to merge several concrete states into one abstract state, which represents all of them. One might be tempted to abstract the replacement counter in the following manner: if a control-flow join with two incoming states with different replacement counters is encountered during analysis, both counters have to be taken into the resulting state, e.g. counters of '2' and '3' may exist in the incoming states, so that the resulting state would have the set $\{2, 3\}$ to denote both possibilities. Unfourtunately, this set of possible counter values never decreases, i.e. we can never reduce the number of possible counter values again. So after three control-flow joins the set may have all four possible elements and will stay so for the rest of the analysis. Apart from control flow joins, not classifiable cache accesses cause the same phenomen: if an

access cannot be precisely classified as a cache hit or a cache miss, then we have to consider both possibilities. On a miss, the counter would increase by one; on a hit it would stay the same as before. So if the counter was '2' before, this results in the set $\{2, 3\}$ afterwards. After another three non-classifiable accesses, this may lead to total loss of counter information. Yet if we do not know anything about the counter, we cannot say precisely how long a cacheline may survive in the cache. As the above example shows, a cacheline may be thrown out again by the next access to the set of that line.

This leads to the consequence that all that can be said about the ColdFire cache is that a cacheline will survive until the next access to the set it is in. This is essentially the behavior of a direct mapped cache with 128 sets (and lines). On the other hand, nothing can be said about which lines may be in the cache. This is because it can never be ensured that a line that may be in the cache has definitely been removed and is guaranteed to no longer be in the cache.

Following these observations, we implemented the cache analysis for the MCF 5307 as a must analysis for a 2K direct mapped cache. Naturally, this reduces the precision of our cache behavior prediction, since we were looking at a cache four times smaller than the actual hardware. On the other hand, the size of the analysis data became much smaller and we were able to integrate the cache and pipeline analysis without the danger of excessive memory consumption.

4.2 The ColdFire Pipeline

The MCF 5307 has two coupled pipelines, cf. Figure 4: a *fetch pipeline* fetches instructions from memory, partly decodes them, performs branch prediction and places the instructions into an instruction buffer, consisting of a FIFO with eight entries (complete instructions). The *execution pipeline* consists of two stages that fetch complete instructions from the instruction buffer, decode them, followed by executing them.

The instruction fetch pipeline performs branch prediction by scanning the fetched instructions at the IED stage and redirecting the fetching if a backward branch or an unconditional branch is detected. Therefore, memory access behavior is heavily dependent on pipeline behavior, since a mispredicted (and thus fetched) branch may cause different memory accesses depending on the time that the branch is actually performed at the execution pipeline's AGEX stage. Luckily, we can precisely model this behavior and its consequences by using a unified pipeline and cache analysis.

Our abstract pipeline state consists of a set of concrete pipeline states, giving a safe approximation of the set of all possible pipeline states that may occur at any given program point. From this we can compute an upper bound of the number of cycles the instruction at that points needs to execute in the worst case.

We now have to model the effects of performing one cycle of actual pipeline execution. The state and interactions of the different pipeline stages have to be considered. In our model the different stages of the pipeline have an inner state and communicate with one another through two types of *signals*: immediate signals take effect in the same cycle, e.g. model the cancellation of certain stages

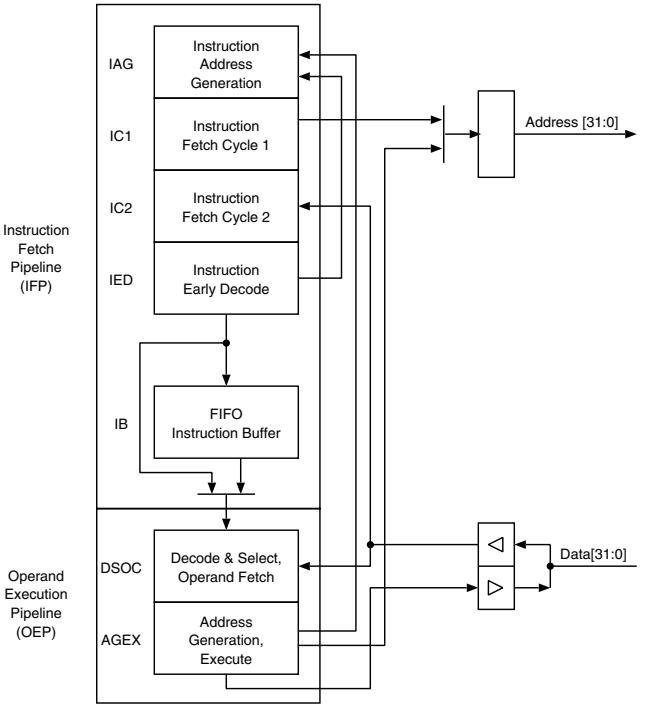


Fig. 4. The MCF 5307 Pipeline

by others. Delayed signals take effect only in the *next* cycle and are used among others to tell the IED stage that an instruction word has been fetched by the IC2 stage.

The model is depicted in Figure 5. Note that we modeled the execution pipeline in a way different to the actual hardware. Since ColdFire instructions can overlap by 1 cycle at most and the DSOC stage only performs decoding and register fetching for the first cycle, we don't have to model both execution stages. We added another stage, SST, to model a stall that occurs in the pipeline if two write operations follow immediately (the second is stalled for up to two cycles).

Not shown in detail in Figure 5 is the bus unit that is responsible for modeling the fetching and writing of data. It is rather complex since many stall conditions and interactions on the pipelined ColdFire bus have to be taken into account. This stage also incorporates the cache analysis, updating the cache state according to the accesses made.

The evolution of the pipeline is performed at cycle granularity, i.e. one cycle of execution is modeled. Each stage evolves in that cycle according to certain update rules, taking into account the immediate and delayed signals. The internal state of the stage is updated, and signals may be generated. The update is performed in the following order: SST, EX, IB, IED, IC2, IC1, bus unit, and IAG. This way, all the dependencies of immediate signals are obeyed.

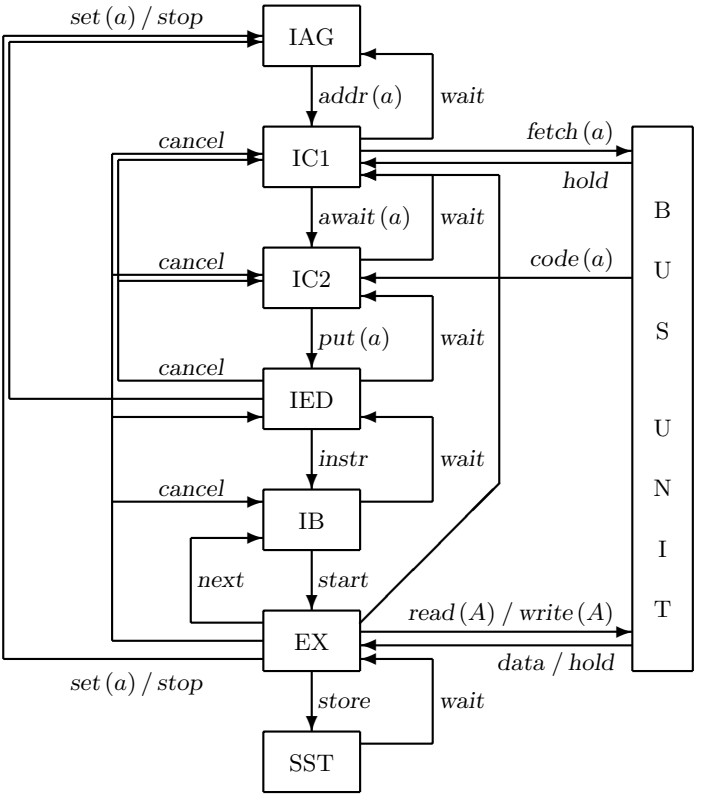


Fig. 5. Map of formal pipeline model

The implementation very closely follows the formal model, with state and updates of the different stages.

The output of the pipeline analysis is the number of cycles a basic block takes to execute, for each context. These results are then fed into the path analysis to obtain the WCET for the whole program.

4.3 Experiments

The prototype system for the Motorola ColdFire MCF 5307 processor was tested with a specific hard real-time benchmark which has been developed and provided by AIRBUS. This benchmark is designed in a way to resemble "real" avionics software. It basically consists of an infinite loop that reacts onto external events by triggering one of 12 independent tasks. Each single task has to satisfy restrictive timing constraints.

Since the tasks are independent from each other, each task was extracted from the control software and analyzed separately for our experiments. The original binary in *.elf* format is about 100kB in size, with the extracted tasks being nearly equal in size.

Value Analysis: Table 2 shows a quality statistics for the value analysis of each task. Memory references are distinguished as read and write accesses. The table shows that the major portion of memory accesses can be resolved *exactly*. References are classified as *good* if the interval for the start address of the access can be restricted to an area of 16 or less cachelines, which is a range of 256 bytes for the MCF 5307. Nothing can be said about the start address of references classified as *unknown*[5]. The analysis time for a single task is a maximum of 61 seconds with a memory usage of about 25MB (computed on a 1.2GHz Athlon system running RedHat Linux).

Table 2. Results of the value analysis.

Task	read accesses [%]			write accesses [%]			unreachable [%]	time [seconds]
	exact	good	unknown	exact	good	unknown		
1	93.32	3.69	2.99	94.51	4.11	1.38	9.1	61
2	93.42	4.01	2.57	93.39	4.62	1.99	8.5	24
3	92.58	4.20	3.22	93.25	4.59	2.16	9.3	31
4	90.24	5.61	4.15	91.71	6.35	1.94	13.5	21
5	93.79	3.33	2.88	94.61	3.83	1.56	7.0	59
6	93.23	4.21	2.56	93.02	4.82	2.16	10.2	26
7	92.06	4.63	3.31	93.13	5.01	1.86	11.0	35
8	90.87	5.03	4.10	91.97	5.88	2.15	11.4	18
9	93.97	3.22	2.81	94.74	3.72	1.54	6.8	56
10	92.40	4.72	2.88	92.79	5.32	1.89	11.0	26
11	92.16	4.47	3.37	92.98	4.90	2.12	9.4	31
12	90.51	5.28	4.21	91.57	6.16	2.27	11.5	24

Cache and Pipeline Analysis: The combined Cache and Pipeline analysis of the EADS benchmark was analyzed in two memory configurations: the *original* configuration partitions the memory in cacheable and non-cacheable areas as well as different memory types (DRAM and On-Chip-SRAM). The *all-cacheable* configuration assumes a simple cacheable DRAM scheme for the whole address space.

Table 3 shows the analysis times for all tasks (computed on a 1.0GHz Athlon running RedHat Linux). The analyzer consumed at most 110MB of memory for every task. The output of the Cache and Pipeline analysis contains the number of cycles for each basic block the number to execute (for each *context*).

[5] Note that the percentage values do not refer to pure instructions, but to instruction/context pairs. This increases the weight of instructions in deeply nested loops or recursive functions.

Table 3. Analysis times of the Cache and Pipeline analysis

Task	memory configuration	
	original [m:s]	all-cacheable [m:s]
1	5:25	8:10
2	4:32	6:47
3	6:03	8:26
4	7:37	10:32
5	4:55	7:32
6	4:34	6:56
7	4:49	8:14
8	7:35	10:31
9	4:54	7:25
10	4:32	6:54
11	5:48	8:09
12	7:35	10:26

5 Conclusion

We have given an overview to the methods for cache, pipeline, and path analysis used in the `USES` approach. We have presented the design of a fully functional prototype to analyze the timing behavior of the Motorola ColdFire 5307 processor and reported our experiences with this tool. The WCET system for the Motorola ColdFire MCF5307 processor has been installed in AIRBUS Toulouse plant. The initial assessment AIRBUS software verification specialists carried out was positive. AIRBUS is currently in the process to decide to use the WCET technology for the timing validation of avionics software (coming soon and future aircraft programs).

References

[aiS] http://www.aisee.com. *aiSee Home Page.*

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison Wesley, 1986.

[CC77] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages*, 1977.

[Col97] Motorola. *Coldfire Microprocessor Family Programmer's Reference Manual*, 1997.

[Fer97] Christian Ferdinand. Cache Behavior Prediction for Real-Time Systems. PhD Thesis, Universität des Saarlandes, 1997.

[FMW97] Christian Ferdinand, Florian Martin, and Reinhard Wilhelm. Applying Compiler Techniques to Cache Behavior Prediction. In *Proceedings of the ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-Time Systems*, 1997.

[FMWA98] Christian Ferdinand, Florian Martin, Reinhard Wilhelm, and Martin Alt. Cache Behavior Prediction by Abstract Interpretation. *Science of Computer Programming, Elsevier*, 1998.

[Lan98] Marc Langenbach. CRL – A Uniform Representation for Control Flow. Technical report, Universität d. Saarlandes, 1998.

[Mar98] Florian Martin. PAG—an efficient program analyzer generator. *International Journal on Software Tools for Technology Transfer*, 2(1), 1998.

[Mar99] Florian Martin. *Generation of Program Analyzers*. PhD thesis, Universität d. Saarlandes, 1999.

[MAWF98] Florian Martin, Martin Alt, Reinhard Wilhelm, and Christian Ferdinand. Analysis of Loops. In Kai Koskimies, editor, *Proceedings of the 7th International Conference on Compiler Construction*, volume 1383 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[NNH99] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[PK95] P. Puschner and C. Koza. Computing Maximum Task Execution Times with Linear Programming Techniques. Technical Report, Technische Universität Wien, Institut für Technische Informatik, Vienna, 1995.

[PS91] Chang Yun Park and Alan C. Shaw. Experiments with a Program Timing Tool Based on Source-Level Timing Schema. *IEEE Computer*, 24(5), 1991.

[SF99] Jörn Schneider and Christian Ferdinand. Pipeline Behavior Prediction for Superscalar Processors. Technical Report A/02/99, Universität des Saarlandes, 1999.

[Sic97] Martin Sicks. Adreßbestimmung zur Vorhersage des Verhaltens von Daten-Caches. Diploma Thesis, Universität d. Saarlandes, 1997.

[The00] Henrik Theiling. Extracting Safe and Precise Control Flow from Binaries. In *Proceedings of the 7th Conference on Real-Time Computing Systems and Applications*, Cheju-do, South Korea, December 2000.

[The01] Henrik Theiling. Generating Decision Trees for Decoding Binaries. In *Proceedings of the ACM SIGPLAN Workshop on Language, Compiler and Tools for Embedded Systems*, Snowbird, Utah, USA, June 2001.

[WM95] Reinhard Wilhelm and Dieter Maurer. *Compiler Design*. International Computer Science Series. Addison–Wesley, 1995. Second Printing.

Embedded Systems and Real-Time Programming

Niklaus Wirth

wirth@inf.ethz.ch

Abstract. Although computers have been employed for decades to control machinery and entire laboratories, the term *embedded system* has received renewed attention and is about to establish itself as a discipline of its own. We therefore try to identify its characteristics and peculiarities. The most outstanding single item is the role of time. The addition of timing conditions to all other specifications of a system causes *real-time programming* to appear as a particularly difficult and challenging subject. We ask how the rapidly growing demands can be met, and emphasize that reliability must be guaranteed in view of the potentially disastrous consequences of failures upon the controlled equipment and its environment.

1. Introduction

For decades we have been told that writing a program is easy, and everybody could do it. That may be so. But certainly getting a program to "run" is already somewhat harder. Providing a complete specification of requirements, and then writing a program that meets these specifications is very much harder, not to speak of analytically verifying its correctness. Designing a program that one is willing to publish in good conscience (for others to study, not just "open source") is even an order of magnitude harder. And writing a program that someone else is willing to adopt and use is almost impossibly hard.

I have used this scale of difficulty for over a decade, and one can readily extend it in many directions. For example, programming concurrent processes presents its own class of new difficulties, particularly if it involves several real processors instead of concurrency simulated by threads. What about real-time programming for embedded systems? Is it also a class of its own? Before we talk about the technical term of *embedded system,* we need to identify the characteristics that distinguish it from conventional programming and perhaps influence the challenges and difficulties encountered in design.

2. Challenges and Difficulties

As the term suggests, an embedded system is part of a larger whole that typically consists of many components, not just computer modules, but also sensors and

T.A. Henzinger and C.M. Kirsch (Eds.): EMSOFT 2001, LNCS 2211, pp. 486-492, 2001.
© Springer-Verlag Berlin Heidelberg 2001

actuators. This implies that many activities occur concurrently, and are to be controlled by the embedded computer system part. This results in a variety of challenges and problems that have no counterpart in conventional programs, neither in commercial data processing nor in scientific computing. Without claim for completeness, let me list a few:

The various ongoing activities imply multiple, *concurrent computing processes*, and with them the problem of their synchronization for harmonious cooperation in their exchange of data, be it via messages or a common store.

The activities in the system run at a given speed, and thereby impose constraints on the delays with which a computing process must generate reactions and responses. These *real-time constraints* are requirements unknown in conventional programs, where the only limitation of delay is typically the impatience of a human user. It is indeed one of the great achievements of digital computing that we can abstract from time. The only condition that matters is that the programmed operations are strictly performed in the prescribed sequence, one after the other.

In mechanical systems, *economy* is more important than it is usually considered by computer programmers. Economy of computing power and of memory are essential, although in a declining degree. The ever growing speed of microprocessors and capacity of memory chips have the negative consequence that careful and economizing design becomes less respected.

An embedded computer system receives its inputs from a variety of *sensors* of voltage, temperature, pressure, light, acceleration, rotation (gyros), radiation, chemical gases. It delivers its outputs to *actuators*, lights, signals, current sources, motors, steppers, relays, valves. The challenge for the programmer is to manage various devices outside the digital world through interfaces, that often are subject to stringent real-time constraints, and at the same time inadequately specified.

Recently, a criterion has entered considerations that has hardly played a role in the past: Electrical *power consumption*. It has been brought into the limelight by battery-powered computers; and many embedded applications belong into this category. To an increasing degree, it is the ratio of computing power over power consumption that is decisive in the selection of a processor. Low power may render cooling equipment unnecessary, thereby reducing overall consumption even further.

The addition of a flexible computer to control mechanical equipment makes it possible to operate this equipment without human intervention. The resulting systems are called *intelligent robots*. This opens new possibilities to operate them in places where humans would not dare to enter, in *hostile environments*. This application calls for special properties of hardware components, such as radiation resistance, but also imposes additional requirements on programs, such as a fail-safe strategy.

Finally, this brings us to the topic of *reliability*, which plays a much heavier role than in pure computing applications. A failure no longer results in a wrong number or a black screen, but in a crash, the loss of very expensive equipment, or even of lives. Programming becomes a serious activity, where personal responsibility cannot be abdicated. Reliability must be a permanent property and have overriding importance.

The consequences of a mistake may affect adversely many people, as the following story will show. It happened in Basel on June 15. Basel is a large railway station connecting systems of Switzerland, Germany and France:

Traffic in Basel's main railway station collapsed totally on Friday between 9:30 and 11:15. 50 trains and thousands of passengers were afflicted. The reason for this standstill was the failure of a computer system in the main switch controlling station.

All trains were blocked; they could neither enter nor leave the station. The failure occurred in a new computer system that was put in operation only recently. It controls all the switches and signals in the entire railway station.

Four days later, the computers failed for the second time paralyzing traffic completely:

The cause for this disaster remains unclear. Further such break downs therefore cannot be excluded.

Two weeks later, reports said the error had been found. The most illuminating comment was that the two computers had failed to understand each other, also causing the backup computer to clonk out. Reliable sources later said that it was still unclear whether the mistake had actually been identified.

3. Ways to Overcome the Complexity of Real-Time Programming

Indeed, the design of embedded systems appears to be of an overwhelming difficulty. One must wonder how so many successful systems had been designed in the past, considering the often flaky techniques and tools at the engineers' disposal. Their achievements are truly remarkable. Large embedded systems control gigawatt power stations, huge water dams, atomic reactors, they guide high-speed trains, aircraft and missiles. "Large" here means megabytes of code and millions of lines of program text. One wonders how it was at all possible to build systems of such a staggering complexity, when it seems utterly impossible to guarantee their reliability and one is content with a reasonably low probability of failure. This state of affairs is well-known in other fields of engineering, where, however, this rest of uncertainty originates in mechanical and physical properties of materials. In software, dealing with abstract artifacts that do not wear out, we should strive to do better.

To master complexity is evidently the name of the game. Therefore, a large effort has recently gone into studying the foundations of programming and into establishing a theory. With a solid theory about the subject, so the theory goes, we will be able to develop programs within a framework and with tools that control the process to the effect that mistakes will become, if not eliminated, then at least rare. An axiomatic theory and a discipline of programming have been created, automated tools, themselves of a staggering complexity have been built, and text books have been written about program verification and programming with inherent correctness proofs. These efforts deserve praise, but they have hardly begun to take the added problems of real-time programming into account. If one considers the very limited degree in which these theories are being practiced, severe doubts appear as to their effectiveness in the world of computing at large. Surely ambitions and expectations have been scaled down.

How do most people learn to program? By learning rules of a language and then writing. This is in contrast to learning how to compose prose. There we first read and read again before testing our own writing talent. Our compositions are scrutinized by the teacher, correcting our spelling mistakes and gradually improving our style by making suggestions. In programming, our compositions are rarely scrutinized by a teacher; our teacher is the computer telling us whether or not "it runs". And sometimes it runs without being correct, and sometimes it fails in spite of appearing correct. Inspecting programs is an unappealing business, and the more intricate and

inscrutable a writing is, the less is it in danger of being inspected. As a teacher I have seen some orderly and much disorderly code. But every time I looked at a larger "piece of code" (notice the term *code*, implying some secrecy), I discovered that much of its intricacy did not stem from the problem to be solved, but from poor mastery of the art of programming. In other words, much of the complexity is home-made. Every time it was possible to achieve the same goal in a simpler, often much simpler way, by factors of 2 to 10. This is staggering!

The truth remains that for good programming talent is required. Complicated theories and overly complex tools are not enough. In fact, sometimes they even foster bad style and add complexity. They seduce the programmer into the belief that he can easily abdicate responsibility to the tools, and the tools will guarantee proper form and style. This is no crusade against sound theoretical foundations and honest tools, but the best, in my experience, is the rule to *combat complexity like the devil.*

4. An Example of Reducing Complexity

Let me tell you about one of my recent encounters with that ubiquitous devil. I am interested since my youth in flying objects. Six years ago, I heard about a project with the aim of constructing a model helicopter for autonomous flight, controlled by an on-board computer [1]. As a layman, I was surprised that this was not standard practice in every helicopter. But actually this is not so, and in fact controlling a model helicopter is harder than a big craft, because the latter's big rotor has a much larger moment of inertia, leaving more time for corrections. Nevertheless, occasionally helicopters crash due to pilot errors in difficult situations. Hence, a computer-based stabilization system might be handy.

Our model, with a rotor diameter of 180 cm, was equipped with two commercial computer boards, each with an Intel 486 processor and an 8 Mbytes store. The software was based on a commercial real-time operating system, chosen because of the many concurrent processes to be managed. The resulting machinery with computer boards, inertial guidance system with gyros, servos, compass, telemetry and, last but not least, batteries weighed some 20 kg, and consumed some 20W. It did not take me long to believe that a simpler system could be designed, the software being only part of the simplification.

The first decision was to build an entirely new system on a single board. As processor I chose a Strong ARM (DEC 1035), a RISC architecture delivering the computing power of (at least) the two Intel 486s with a power consumption of little more than 1W. The next decision was to eliminate the RT-OS, as it seemed possible to do essentially without concurrent processes in the form of threads. The third decision was to program the entire software in Oberon [2], which is very suitable for "programming close to the hardware". This implied, however, to first build an Oberon compiler for the Strong ARM. This turned out to be easy (1 man month) because much could be taken over from existing compilers [3], and provided the invaluable advantage to add SA-specific inline procedures for handling processor initialization and device interfaces. In order to slightly simplify the task of compiler construction, a few features of Oberon were omitted, and a few new ones were added (see Appendix).

The task of the helicopter controller is essentially to sample sensors (accelerometers and gyros), compute new values for power, pitch, roll, and yaw, and output the computed values to the respective servos. This can be done in a single loop which is triggered in equal time intervals, in this case every 20ms. Because for every new group of outputs 4 preceding groups of inputs are required as arguments, the inputs are buffered. Input and output interfaces were implemented by two PLDs, converting the PWM signals to binary (8-bit) values and vice-versa. They could be considerably simplified by converting the (up to) 8 signals sequentially. As a consequence, the interrupt time slice was chosen as 2ms, and a counter is incremented to identify the signals being input and output.

As a result, the time critical operations are kept in a single interrupt handler, triggered by a clock generated in one of the PLDs. This routine is, and must be, very short. It reads a single byte from the input PLD and deposits it in the input buffer, and it picks a single byte from the output buffer and feeds it to the output PLD. The "main program" is a single loop (after initialization), at one point waiting for the next change of the counter. In this loop, four value sets are taken from the input buffer, and a new output set is computed. All that had to be done concerning real-time constraint was to measure the time needed for this computation. Fortunately it turned out to be less than half of the 20ms time slice.

5. Some Conclusions

Certainly, this example may be called academic, because it is simple, or rather turned out to be simple. I am fully aware that different criteria apply in other cases, and that "real-world" systems "out there" are much more intricate. Yet, this example does show that drastic simplifications are possible – even in simple cases! With the addition of further input sources such as a compass and a GPS, for example, affairs become more complicated, because those inputs cannot simply be sampled, but arrive at unpredictable intervals. Hence, the temptation to introduce threads (and a system handling them) arises, and even may turn out to be justified.

However, one must be aware of the consequences. Task switching becomes hidden, and it becomes impossible to guarantee real-time constraints. It may all be very well, as long as processing power is available in excess, i.e. if the processor idles most of the time. But "throwing in resources in abundance" is not sound engineering practice, even if it appears to be cheap.

If we are to design reliable real-time systems, we must demand precise specifications, and then be able to guarantee that the computing time required between specified events is shorter than their interval of occurrence. This implies the availability of tools (compilers) that provide data about the time required to execute certain sequences of statements. Such tools are hardly available to my knowledge. Instead, "break-points" are inserted in the program, and a tool is used to measure the time consumed. But this is like program testing vs. analytic verification: The result applies only to the specific test case, but not in general The practical idea is that the test case is sufficiently "representative", and a safety factor is thrown in, in case of doubt.

The dubiousness of this practice is compounded by two circumstances:

If interrupts are involved, the interrupt handler steals time from the interrupted process (thread). If neither place nor duration of the interrupt are known, the effect on time bounds of the interrupted process may be considerable, and the worst case condition taken for analysis may be so much bigger than the "normal' case that the temptation to ignore the (very rare) worst case may be large.

Modern processors commonly use on-chip cache memories and pipelining, resulting in rather unpredictable performance variations. The time for an instruction sequence cannot be taken as the sum of the time of the individual instructions. Yet, the real-time program must reliably handle the worst case, which may be 10 or more times slower than the average case. Temptations to ignore that ugly worst case loom large.

Not even sophisticated analytical tools help to overcome these intrinsic difficulties with machinery displaying a stochastic behavior. As a consequence, we recommend the following two rules of thumb for the real-time programmer:

Refrain from using interrupts. If interrupts cannot be excluded, make sure that the time consumed by interrupt handlers is orders of magnitude shorter than any specified real-time condition. Handlers must be free of loops (with an unknown number of repetitions).

Be skeptical about sophisticated processors with caches and pipelines. Test your programs with caches turned off. (I am afraid this implies that the caches may just as well be left out anyway). Digital signal processors (DSP) typically do not feature caches, and are therefore to be preferred.

What renders the seemingly hopeless situation less desperate is the fact that in most cases the real-time constraints are weak; they are specified in seconds (or hundreds of milliseconds), while the processor executes millions of instructions per second, implying that real-time constraint can almost be ignored. However, in the case where several interrupt sources must be handled, it might be wise to employ an individual processor for each of them. After all, silicon is now cheap, and system reliability is dear. We should remember that interrupts were originally introduced to spare the use of separate, expensive processors for events that rarely occur and then take a negligible amount of time for a very simple, straight-forward action.

We might note that in the case of embedded systems the requirements concerned with time are in addition to all other correctness specifications. Could it therefore be possible to separate the two categories? Separation of concerns has always been a useful design principles. Hence, the appearance of a formalism (language) allowing real-time constraints to be considered and checked separately, perhaps by separate software tools, would mark a desirable step ahead.

Here we are concerned with systems containing computers embedded as crucial parts of a larger whole. The consequence is that the designer of the computer section must understand the whole, or at least be able to rely on complete and precise specifications of the interfaces between the rest and the computer. Often these interfaces are themselves unnecessarily complex (particularly if they comply with established standards!), giving rise to misunderstandings and mistakes. A second corollary of embededness is that failures of the whole may be due to other than computer malfunction, requiring much wider horizons in the search for errors. The designer of embedded systems should be a mechanical, electrical, and software engineer all in one. In our helicopter example, mechanical oscillations and resonances, loosening contacts, over sensitivity of sensors to vibrations, electrical

noise of the combustion engine, or mechanical fatigue of the exhaust muffler caused failures in spite of correctly functioning software, where crashes could be avoided only by an immediate fall back to the remote pilot, acting as a human reset button. In many applications no such easy solution exists. The Pathfinder vehicle landing on Mars comes to mind.

The major difference between “regular” and “embedded” systems programming, however, lies not so much in the compounded difficulties of the latter, but rather in the more stringent reliability expectations. If a system crashes, it may be fatal. It is as if every mistake may force you to restart construction from scratch, to step back to square zero. This increased awareness of consequences of malfunction and of our responsibility should make us doubly anxious to look for simple (not simplicistic!) solutions, and at the same time triply skeptical towards black boxes that we do not fully understand but nevertheless have to assume responsibility. I also warn against an exaggerated reliance on sophisticated development tools. Although they may be helpful, they cannot be a substitute for detailed understanding. I have witnessed cases where the mastery of the toolbox took a greater effort than the solution of the problem at hand, thereby diverting attention from the essence.

In our programming plight, we rightly expect to receive support from programming languages and their compilers. But even at run-time we are used to rely on checks detecting mistakes such as indices out of array bounds, arithmetic overflow, and other rare cases. When relying on such implicit checks, we are in a state of sin, because they detect mistakes that should have been avoided a priori by proper design. In the case of embedded systems, this becomes manifest by our uncertainty of the actions to be taken in the case of failure. For instance, in the case of the model helicopter, what should be done in the case of an array bound violation, what in the case of an arithmetic overflow? There is no reset button!

In summary, I do not only plead for avoiding complexity in program design, but also in the tools used. The more difficult the problem at hand, the more we must strive for reasonably simple solutions. They must never exceed our mental abilities to understand them fully. And the simpler the theories and tools employed, the more perspicuous are the engineer’s designs, and the more realistically can he take responsibility for a design. Past achievements of clever engineers have been wonderful; think only of the automated flights of satellites around the earth, to the moon and past planets. But the fight against mistakes and the overestimation of our infallibility will last for ever. It must not keep us from trying hard to do better.

References

1. J. Chapuis, C. Eck, M. Kottmann, M. Sanvido, O. Tanner, "Control of Helicopters," in K. J. Åström, P. Albertos, M. Blanke, A. Isidori, W. Schaufelberger, R. Sanz (Eds): Control of Complex Systems, Springer Verlag.
2. N. Wirth. The Programming Language Oberon. *Software – Practice and Experience, 18,* 7, (July 1988), 661-670.
3. N. Wirth. *Compiler Construction*. Addison-Wesley, 1996, ISBN 0-201-40353-6.

Embedded Software for Video

Wayne Wolf

MediaWorks Technology and Princeton University
wolf@princeton.edu

Abstract. Systems-on-chips (SoCs) up the stakes in embedded software in several dimensions. Not only do they allow us to integrate more software on a single chip, they also allow us to implement new types of systems, such as multiprocessor algorithms and networked systems-on-chips. The software for these SoCs must not only meet the traditional requirements of software but must also meet more hardware-like properties such as timing, power consumption, and size. Video provides an excellent example of advanced embedded software for SoCs. This paper uses a smart camera being designed at Princeton University to illustrate how a hierarchy of abstractions can be used in embedded software system design.

1 Introduction

Moore's Law developments in VLSI manufacturing technology have brought us to an inflection point in VLSI system design: **systems-on-chips** (**SoCs**). A system-on-chip includes all the components necessary to make a complete system: processing, I/O, and memory. SoCs vastly increase the stakes for embedded software design in several ways. First, SoCs allow us to put much more software on a single chip. Second, many SoCs will be networked, changing the embedded programmer's job from designing an island of isolated software to building a system that can function within a complex network.

In order to solve embedded software design problems for SoCs, we need to take an expansive view of software. Software traditionally tries to provide a simple, uniform model of computation to the programmer. In designing an embedded system from algorithms through silicon, we need to work through a series of computational models, each closer to the implementation. As we add detail, we get more accurate information by applying more sophisticated an expensive analysis methods. We use this information to optimize the embedded software for the performance, power, and size goals that are so important in embedded system design.

This paper uses a video system being designed at Princeton University to illustrate some techniques for embedded video software design in particular and embedded software design in general. Video is an aggressive application that shows off some important aspects of embedded software: real-time operation, streaming data, and complex algorithms.

The next section outlines trends in consumer electronics that influence the nature of embedded software. Section 3 describes a smart camera system we are designing at Princeton University, which we will use as an example to illustrate the design of

T.A. Henzinger and C.M. Kirsch (Eds.): EMSOFT 2001, LNCS 2211, pp. 493-502, 2001.
© Springer-Verlag Berlin Heidelberg 2001

embedded software for video systems-on-chips. Section 4 looks at why we need multiple levels of abstraction in embedded software design and what sorts of information those abstractions should provide. Section 5 goes into more detail about the levels of abstractions we are using to design our smart camera system.

2 Trends in Consumer Electronics

Systems-on-chips enable new types of products for two major reasons. First, increasing levels of integration drive down product costs as boards are shrunk onto chips. Because the non-recurring engineering costs for SoCs are relatively high ($250,000 for masks today and increasing), these chips must be used in high volume to be viable. Second, intra-chip communication requires much less power than does chip-to-chip communication. As a result, SoCs enable a wide variety of low-power, battery-operated devices.

We need to rethink both hardware and software architectures for systems-on-chips. The high levels of integration encourage us to use multiprocessors to implement sophisticated algorithms on a single chip. Because integrated circuits present very different design constraints than do boards, architectures that are directly mapped from boards may not be as effective as architectures designed specifically for SoCs.

An obvious result of increasing levels of integration is more sophisticated applications. Products are becoming much more complex along all axes: more complex algorithms, fancier user interfaces, networking. The growing sophistication of applications shipped on chips in consumer devices is one of the principal factors that will influence the development of embedded software.

Streaming data is a defining characteristic of next-generation consumer products that affects both software and hardware architectures. Streaming data arrives and must be processed at regular intervals. Telephony, audio, and video systems all process streaming data. Streaming data imposes hard real-time requirements at very low levels of abstraction in the embedded software design. As a result, streaming data must be taken into account throughout the software architecture.

An increasing number of consumer products include **file systems**. MP3 players and digital video recorders both use file systems to organize content for recording and playback. Personal MP3 players tend to use flash memory and customized file systems. Digital video recorders such as Tivo and Replay compliant boxes generally use magnetic disks and rely on Linux to implement the more general-purpose aspects of their systems. Media players are similar to cell phones in that they must both implement streaming and non-streaming functions.

Future consumer products will also increasingly rely on **networking**. The 802.11 wireless network standard has enabled a new generation of easy local area networking for PCs. These standards will soon find their way into other products, such as cameras. Networked systems built from multiple SoCs plays to the VLSI economic model. When a single system requires many identical chips, the non-recurring cost burden for the SoC can be spread across many chips but fewer systems.

Adding networking functionality and local storage will push more and more consumer appliances to operate in **distributed** mode. Despite having local storage for cost and access time reasons, consumers will want to be able to get their data from wherever they happen to be. Just as computers moved from attached storage to local area storage to wide-area access to storage, consumer appliances will generally move toward broader, seamless access to data.

3 Application: Smart Cameras

We are developing at Princeton University a **smart camera** system [1,2] that illustrates many of the trends described in the last section. Our smart camera system leverages advances in VLSI technology to build a relatively inexpensive system that relies on sophisticated software to provide complex functionality.

A smart camera processes image information in order to provide the user with a more abstract view of the scene and to allow advanced services. Several groups have developed smart camera systems [3,4,5,6]; our system is designed to leverage the advantages of multiprocessor SoCs and complex embedded software. The smart camera blurs the distinction between image and video processing, since some image-oriented results may be derived from several frames[7]. For example, a smart camera can process multiple frames to determine when to take a picture that results in a single image. Early vision is especially well-suited to two types of tasks: **focus-of-attention** to select subjects for further study; and **early classification** using multiple views to select the best still images to submit to detailed image understanding.

Our development platform for real-time smart camera algorithms is PC-hosted. We use two Trimedia evaluation boards that can be used either as a single multiprocessor or as separate processors for two cameras. The Trimedia processor is a five-issue VLIW processor optimized for video applications.

Our first stage system, developed by Ozer and Wolf, classifies **human activity** from a single camera. Human activity classification analyzes the video stream to determine when a person is present and the pose—walking, running, sitting, etc.—of that person in each frame. The methodology, developed by Ozer, Wolf, and Akansu [8,9], is summarized in Figure 1. We use information from both compressed and uncompressed frames. Full compression is not strictly necessary, but MPEG-style compression generates two results, discrete cosine transform (DCT) analysis of blocks of pixels and motion vectors, that are useful in video analysis. The results of all these steps are used to first build a graph that describes regions in the components and then to match that graph against a library of graphs that represent known **poses**.

Figure 2 shows the results of several of the early steps in this process. The initial frame is first classified by color to form rough regions. Edges are extracted from these regions using contour following algorithms. The edges are then fitted to superellipses that describe the size and shape of the regions. The figure does not illustrate the final graph-matching stage of the algorithm that determines the pose of the human figure in each frame. The superellipses that describe the various regions in a frame. A labeled graph describes the relationships between the superellipses. A library of graphs describes possible poses. A graph matching algorithm compares the graph structure and labels to determine the pose that best fits the current configuration. (The same

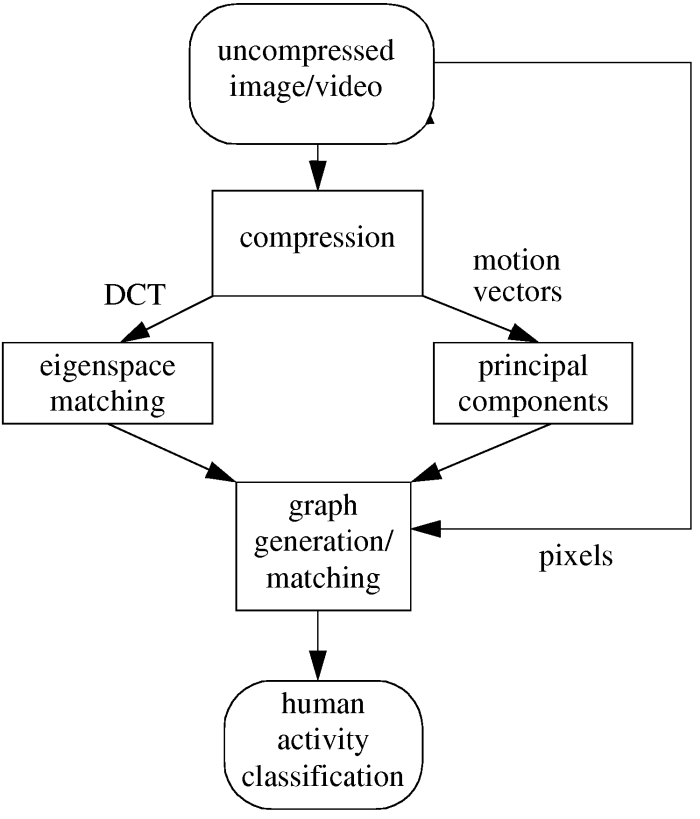


Fig. 1. Major steps in our human activity classification methodology.

methodology can be applied to non-human activity detection by changing the graph library and other modest changes to low-level classification algorithms.)

Our next stage, currently under development, fuses data from several cameras. Data fusion is helpful in several ways. First, it can be used to determine which camera has the best view of the subject for further analysis. Rather than use mechanical zooming and panning to provide an adequate view, we can deploy more cameras and use algorithms to select. Second, we can build a more complete model of the scene that avoids occlusions that will inevitably occur from any single camera view. Third, we can synthesize an abstract model of the scene much more easily and completely than is possible from a single-camera view.

The multiple camera system clearly relies on networking. Since we want to use the same smart camera chip for several different applications running several different algorithms, we need flexibility in both the node software architecture and in the networking architecture. For example, some algorithms may require distributed processing while others may require client/server processing. The system is cheapest and most powerful when the same software and hardware can be used in both modes.

We are also starting work on a VLSI architecture for the smart camera. In order to create a useful, implementable system, we must architect both the software and hardware of the smart camera chip simultaneously.

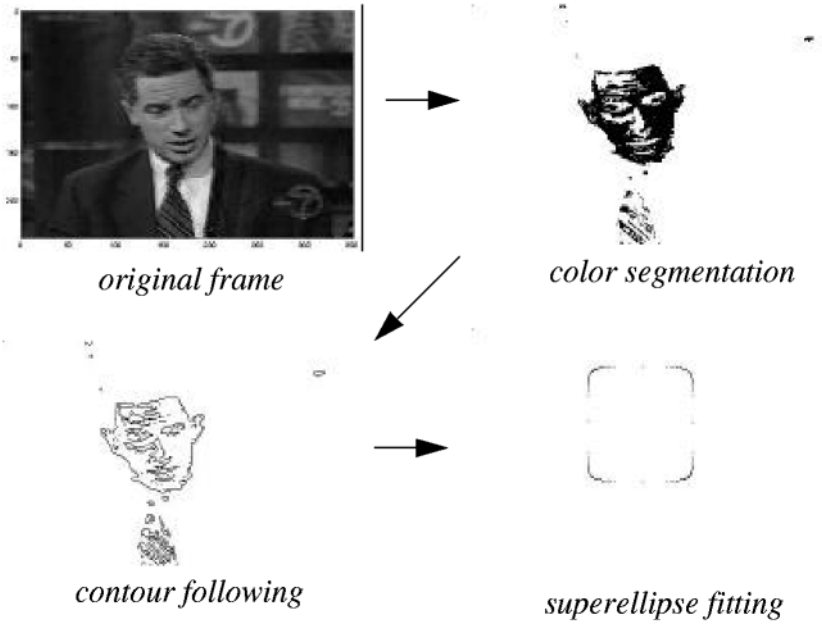


Fig. 2. Steps in human activity classification

4 Levels of Abstraction and Models of Computation

Traditionally, software engineering theory and practice has been plagued by a relatively flat space of abstractions that are put in use to build systems. Compilers have allowed software designers to very successfully abstract away the CPU's instructions but fewer useful abstractions are available at higher levels. In building embedded systems in general and video systems in particular, we must take advantage of a richer range of abstractions as we refine the design. Abstractions are necessary to allow us to meet deadlines and to control power consumption; they are also necessary to take advantage of the heterogeneous architectures often found in embedded systems.

Abstractions that we use must be able to provide sufficient accuracy for the level of detail required at that level. They must also be able to provide those estimates with a reasonable amount of effort. At early stages in the design, accurate estimates of performance, power, etc. are generally not possible because we do not have enough information about the implementation. However, even modestly accurate estimates allow us to make basic architectural decisions. As we refine the design, we expect more accurate values for our design metrics. We are also willing and able to expend

more effort to obtain those estimates. Estimation also becomes more tractable as we add implementation and reduce the search space.

Embedded software design resembles a hybrid between traditional software and hardware [10]. On the one hand, embedded software must be designed to meet criteria that are standard in hardware but rare in the mainframe software world: timing [11], power consumption [12,13], memory space. On the other hand, embedded software must also, in many cases, be designed to be as maintainable as traditional software since it will be used across many product generations. It must also meet concerns such as safety [14] that are most thoroughly addressed by software engineering techniques. As a result, embedded software tools increasingly resemble a cross of traditional software and hardware tools. Embedded software tool users are more willing to wait for expensive algorithms to terminate, so long as those algorithms provide satisfactory results. Embedded software tools also must take more specific directives. Software compilers generally use optimizations that provide a reasonable result on a typical spectrum of cases. Hardware CAD tools, in contrast, are able to accept and operate on specific goals, such as the timing on a given path.

5 Design Methodology

The design methodology we are following in creating the smart camera system illustrates the role of abstractions in embedded software design.

Our algorithms were developed using Matlab running on workstations. This provided a powerful, flexible, and also non-real-time development environment. Matlab is a powerful development environment for signal processing precisely because it presents a simple programming model that does not reflect the architecture on which the software will ultimately be written. Matlab provides a library-rich environment that is sequential at the function level but may hide parallelism at lower levels. The uniform memory space provides easy access to all data.

In retrospect, given our experience with a real-time video processing platform, we believe that real-time results are very important to the rapid development of video processing systems. Although current real-time video processors must be programmed in C, providing a somewhat lower level of abstraction than Matlab, the ability to see your results in real time allows one to judge algorithms much more quickly.

After developing the algorithms, we moved them directly onto the Trimedia platform. We first implemented all the algorithms, including graph matching, directly on a single Trimedia processor. The system was functionally pipeline, with each function in the Matlab code implemented by an equivalent C function. Translating the system to C was relatively straightforward using evaluation system software as an example. Our first implementation of the algorithms were straightforward and did not take into account Trimedia-specific library calls or instructions. We used this as a starting point for a more complex architecture that might include multiple Trimedias and/or use of the PC host.

At this point, we could start to profile the programs to determine where cycles were consumed. This was the first step in adapting the algorithms to the underlying architecture and determining the details of that architecture. But at this point in the design, the most important use of profiling is to make microscopic optimizations,

such as tuning code to run on the processor. It also provides valuable information for macroscopic information, such as the number of processors and the amount of data transferred. Figure 3 shows measured CPU times as a function of frame for two steps, skin detection and contour following. Contour following runs about three times slower than skin detection. Both steps show significant frame-to-frame variations in computation time. This information, even though it is derived from unoptimized code, provides valuable help in determining an initial architecture for the system.

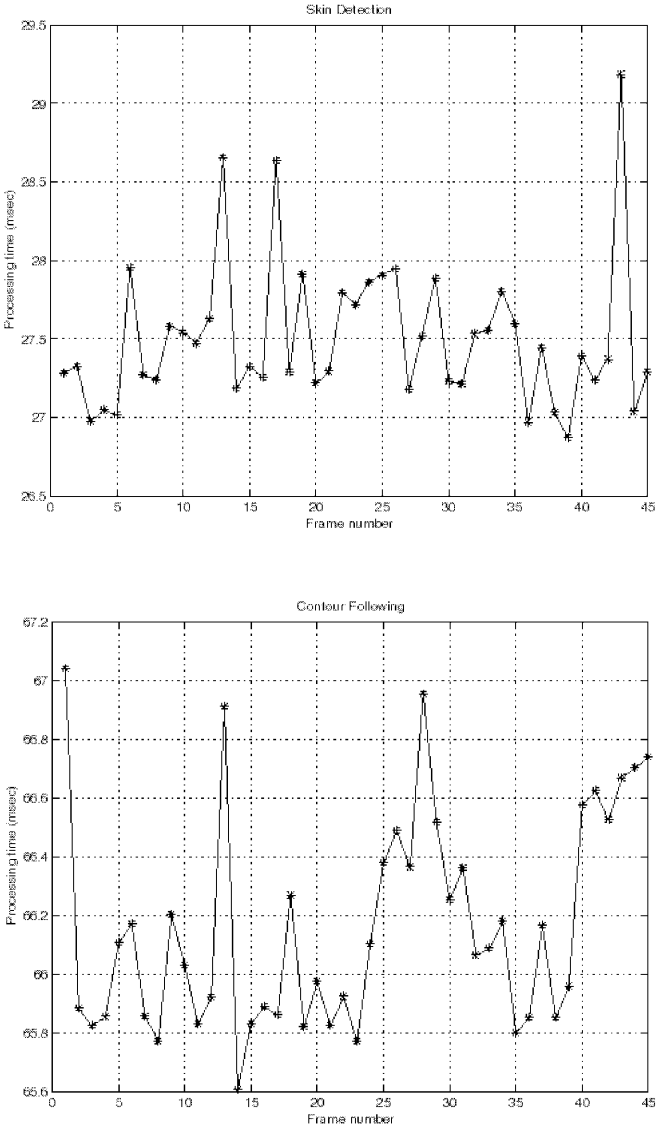


Fig. 3. CPU times for skin detection and contour following

As we continue to design the VLSI architecture of the smart camera, we expect to make use of, among others, several abstractions for embedded software design developed in our group over the past few years.

Our initial architectural experiments show that the system will have multiple processes running on a CPU, controlled by a real-time operating system (RTOS) due to the run-time variations within and between processes. As a result, we need to analyze the effect of the cache on program performance in the multiprogramming system. Li and Wolf [15] developed a process-level model for caches. The model treats each process as an atomic unit or as a small collection of units, not as a set of cache lines. During construction of a system schedule or a simulation of that schedule, algorithms can compute the occupancy of the cache at each step. A schedule synthesis algorithm uses one of two execution time values for each process: a longer worst-case time if it is not in the cache at the start of its next occupied time quantum or a shorter average-case time if it is in the cache. (A best-case ideal cache value is also used to bound the set of feasible schedules.) This process-level cache model allows us to much more accurately predict the execution times for cached systems at a very reasonable computational cost.

At later stages in the design, we must verify our assumptions about the processes. Process-level design typically assumes that context switching and interrupts introduce no overhead. As the system becomes more and more heavily loaded, this assumption becomes less and less tenable. Rhodes and Wolf [16] developed a simulation-based design method that accurately models preemption and interrupt effects. This model is more computationally expensive than the process-level cache model, but it also provides more accurate values. Experimental results show that high-utilization systems require this level of modeling to ensure that they do not fail in the field.

After developing a process structure, we need to refine the programs themselves. Streaming data is prevalent in video and other media processing applications. Loop transformations are very useful in optimizing streaming data systems. Loop transformations provide information about the order and partitioning of data accesses without requiring instruction-level modeling. Lin and Wolf [17] used loop transformations to develop an algorithm for designing custom interleaved memory systems for embedded computing systems. This algorithm first transforms the program's loops to provide the amount of parallelism required to meet the system's performance requirements; it then synthesizes a parallel memory system to match the software design.

We also need to refine the programs at the instruction level. This requires detailed profiling of the program, experimentation with compiler options, and possible hand assembly coding. Fritts and Wolf [18] profiled a series of media processing programs. Programming in a high-level language is much more productive and maintainable than assembly-language programming, but is possible only if the compiler generates satisfactory code. Embedded system programmers spend a lot of time selecting compilers and compiler options to get the compiler to produce the code they want in critical sections of the program. Because today's compilers provide a one-size-fits-all optimization strategy, it is often necessary to restructure the code around optimization units— sections of the program that must be optimized differently must be put in different compilation modules.

6 Conclusions

Embedded software design methodologies are a hybrid of traditional software and hardware design methodologies. Embedded software must be designed to meet hardware-like performance, power, and size constraints as well as software-like maintainability standards. One implication of the nature of embedded software design is that it requires more levels of abstraction than are typically used in mainframe software design. The levels of abstraction are used to manage the design, check assumptions, and incrementally make design decisions in a top-down fashion. Video provides a good example of the use of such techniques because of its high performance requirements and complex algorithms. We believe that more embedded software abstractions and evaluation methods, such as network-oriented models, are necessary to meet the challenges introduced by embedded software for systems-on-chips.

Acknowledgments. Work on the smart video system was funded by the New Jersey Commission on Science and Technology through the New Jersey Center for Pervasive Information Technology. My views on the effects of SoCs on consumer products and their embedded software has been informed by my experience at MediaWorks Technology.

References

1. Burak Ozer and Wayne Wolf, "Smart cameras for video analysis," in *Proceedings, SiSP '01*, IEEE, 2001.
2. Wayne Wolf, "VLSI architectures for smart cameras," in *Proceedings of the SPIE*, vol. 4313, SPIE, 2001.
3. J. Watlington and V. M. Bove, Jr., "A system for parallel media processing," Parallel Computing, 12(12), December 1997.
4. Jonathan Foote and Don Kimber, "FlyCam: practical panoramic video and automatic camera control," in *Proceedings, 2000 International Conference on Multimedia and Expo*, IEEE, 2000.
5. Mircea Nicolesceu and Gerard Medioni, "Electronic pan-tilt-zoom: a solution for intelligent room systems," in *Proceedings, 2000 International Conference on Multimedia and Expo*, IEEE, 2000.
6. S. M. Chai, A. Gentile, W. E. Lugo-Beauchamp, J. Fonesca, J. L. Cruz-Rivera, and D. S. WIlls, "Focal plane processing architectures for real-time hyperspectral image processing," *Applied Optics*, Special Issue on Optics in Computing, 39(5), February 2000, pp. 835-849.
7. Wayne Wolf, "Key frame selection by motion analysis," in *Proceedings, ICASSP '96*, IEEE, 1996, pp. 1240-1243.
8. Burak Ozer, Wayne Wolf, and Ali Akansu, "A graph-based object description for information retrieval in digital image and video libraries," in *Proceedings, CBAIVL*, IEEE, 1999.
9. Burak Ozer, Wayne Wolf, and Ali Akansu, "Human activity detection in MPEG sequences," in *Proceedings, Workshop on Human Motion 2000*, IEEE, 2000.
10. Wayne Wolf, *Computers as Components: Principles of Embedded Computing System Design*, Morgan Kaufman, 2000.
11. Yau-Tsun Steven Li and Sharad Malik, "Performance analysis of embedded software using implicit path enumeration," *IEEE Transactions on CAD/ICAS*, 16(12), December 1997, pp. 1477-1487.

12. Yanbing Li and Joerg Henkel, "A framework for estimating and minimizing energy dissipation of embedded HW/SW systems," in Proceedings, DAC 98, ACM Press, 1998, pp. 188,193.
13. L. Benini, A. Bogliolo, G. A. Paleologo, and G. De Micheli, "Policy optimizatoin for dynamic power management," *IEEE Transactions on CAD/ICAS*, 18(6), June 1999, pp. 742-760.
14. Nancy G. Leveson, "Software safety: why, what, and how," *Computing Surveys*, 18(2), June 1986, pp. 125-163.
15. Yanbing Li and Wayne Wolf, "Hardware/software cosynthesis of memory systems," *IEEE Transactions on CAD/ICAS*, 18(10), October 1999, pp. 1405-1417.
16. David Rhodes and Wayne Wolf, "Co-synthesis of heterogeneous multiprocessors using arbitrated communication," in *Proceedings, ICCAD '99*, ACM Press, 1999, pp. 339-342.
17. Hua Lin and Wayne Wolf, "Co-design of interleaved memory systems," in *Proceedings, CODES 2000*, IEEE Computer Society Press, 2000.
18. Jason Fritts and Wayne Wolf, "Multi-level cache hierarchy evaluation for programmable media processors," in *Proceedings, SiPS 2000*, IEEE, 2000.

Author Index

Alexander, P. 1
de Alfaro, L. 148
Alur, R. 14

Beebee, W.S. 289
Benveniste, A. 32
Berry, G. 50
Binns, P. 451
Bostic, D. 66
Broy, M. 51
Buonadonna, P. 114
Burch, J.R. 324
Butts, K. 66

Caspi, P. 80
Chakraborty, S. 416
Chutinan, A. 66
Cook, J. 66
Cousot, P. 97
Cousot, R. 97
Culler, D.E. 114
Cytron, R.K. 131

Dang, T. 14
Deters, M. 131
Donahue, S.M. 131

Esposito, J. 14

Ferdinand, C. 469
Fierro, R. 14

Goddard, S. 204
Greutert, J. 416
Gries, M. 416

Hampton, M.P. 131
Heckmann, R. 469
Henzinger, T.A. 148, 166
Hill, J. 114
Horowitz, B. 166
Hudak, P. 185
Hur, Y. 14

Ivančić, F. 14

Jeffay, K. 204

Karsai, G. 403
Kavi, K.M. 131
Kirsch, C.M. 166
Kong, C. 1
Koo, T.J. 344
Kopetz, H. 223
Kumar, V. 14

Langenbach, M. 469
Lee, E.A. 237
Lee, I. 14
Liebman, J. 344

Ma, C. 344
Malik, S. 254
Martin, F. 469
Maxiaguine, A. 416
Milam, B. 66
Mishra, P. 14

Nye, J.M. 131

Palem, K.V. 257
Pappas, G. 14
Pasetti, A. 274
Passerone, R. 324
Pree, W. 274

Rajkumar, R. 287
Rinard, M. 289
Rushby, J. 306

Sangiovanni-Vincentelli, A.L. 324
Sastry, S.S. 344
Schmidt, D.C. 361
Schmidt, M. 469
Sifakis, J. 373
Simsek, T. 435
Slotosch, O. 51
Sokolsky, O. 14
Stankovic, J.A. 390
Szewczyk, R. 114
Sztipanovits, J. 403

Taha, W. 185

Talla, S. 257
Theiling, H. 469
Thesing, S. 469
Thiele, L. 416

Varaiya, P. 435
Vestal, S. 451

Wan, Z. 185
Wang, Y. 66
Wilhelm, R. 469
Wirth, N. 486
Wolf, W. 493
Wong, W.-F. 257
Woo, A. 114

Xiong, Y. 237