Springer
Berlin
Heidelberg
New York
Hong Kong
London
Milan
Paris
Tokyo

Embedded Software

Third International Conference, EMSOFT 2003
Philadelphia, PA, USA, October 13-15, 2003
Proceedings

Springer

Volume Editors

Rajeev Alur
Insup Lee
University of Pennsylvania
Department of Computer and Information Science
3330 Walnut Street, Philadelphia
PA 19104-6389, USA
E-mail: {alur,lee}@cis.upenn.edu

Cataloging-in-Publication Data applied for

A catalog record for this book is available from the Library of Congress.

Bibliographic information published by Die Deutsche Bibliothek
Die Deutsche Bibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data is available in the Internet at <http://dnb.ddb.de>.

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

ISSN 0302-9743
ISBN 3-540-20223-4 Springer-Verlag Berlin Heidelberg New York

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

http://www.springer.de

© Springer-Verlag Berlin Heidelberg 2003
Printed in Germany

Typesetting: Camera-ready by author, data conversion by PTP Berlin GmbH
Printed on acid-free paper SPIN: 10960394 06/3142 5 4 3 2 1 0

Preface

The purpose of the EMSOFT series is to provide an annual forum to researchers, developers, and students from academia, industry, and government to promote the exchange of state-of-the-art research, development, and technology in embedded software. The previous meetings in the EMSOFT series were held in Lake Tahoe, California in October 2001, and in Grenoble, France in October 2002. This volume contains the proceedings of the third EMSOFT held in Philadelphia from October 13 to 15, 2003. This year's EMSOFT was closely affiliated with the newly created ACM SIGBED (Special Interest Group on Embedded Systems).

Once the strict realm of assembly programmers, embedded software has become one of the most vital areas of research and development in the field of computer science and engineering in recent years. The program reflected this trend and consisted of selected papers and invited talks covering a wide range of topics in embedded software: formal methods and model-based development, middleware and fault-tolerance, modeling and analysis, programming languages and compilers, real-time scheduling, resource-aware systems, and systems on chips. We can only predict that this trend will continue, and hope that the void in universally accepted theoretical and practical solutions for building reliable embedded systems will be filled with ideas springing from this conference. The program consisted of six invited talks and 20 regular papers selected from 60 regular submissions. Each submission was evaluated by at least four reviewers.

We would like to thank the program committee members and reviewers for their excellent work in evaluating the submissions and participating in the online program committee discussions. Special thanks go to Alan Burns (University of York, UK), Alain Deutsch (Polyspace Technologies, France), Kim G. Larsen (Aalborg University, Denmark), Joseph P. Loyall (BBN Technologies, USA), Keith Moore (Hewlett-Packard Laboratories, USA), and Greg Spirakis (Intel, USA) for their participation as invited speakers. We are also grateful to the Steering Committee for helpful guidance and support.

Many other people worked hard to make EMSOFT 2003 a success, and we thank Stephen Edwards for maintaining the official Web page and handling publicity, Oleg Sokolsky and Kathy Venit for local arrangements, Usa Sammapun for setting up the registration Web page, and Li Tan for putting together the proceedings. Without their efforts, this conference would not have been possible, and we are truly grateful to them.

We would like to express our gratitude to the US National Science Foundation and the University of Pennsylvania for financial support. Their support helped us to reduce the registration fee for graduate students.

July 2003 Rajeev Alur and Insup Lee

Organizing Committee

Program Co-chairs	Rajeev Alur (University of Pennsylvania)
	Insup Lee (University of Pennsylvania)
Local Organization	Oleg Sokolsky (University of Pennsylvania)
Publicity	Stephen Edwards (Columbia University)

Program Committee

Rajeev Alur, Co-chair (University of Pennsylvania, USA)
Albert Benveniste (IRISA/INRIA, France)
Giorgio C. Buttazo (University of Pavia, Italy)
Rolf Ernst (Technical University of Braunschweig, Germany)
Hans Hansson (Malardalen University, Sweden)
Kane Kim (University of California at Irvine, USA)
Hermann Kopetz (Technical University of Vienna, Austria)
Luciano Lavagno (Politecnico di Torino, Italy)
Edward A. Lee (University of California at Berkeley, USA)
Insup Lee, Co-chair (University of Pennsylvania, USA)
Sharad Malik (Princeton University, USA)
Jens Palsberg (Purdue University, USA)
Martin C. Rinard (Massachusetts Institute of Technology, USA)
Heonshik Shin (Seoul National University, Korea)
Kang Shin (University of Michigan, USA)
John Stankovic (University of Virginia, USA)
Janos Sztipanovits (Vanderbilt University, USA)
Wayne Wolf (Princeton University, USA)
Sergio Yovine (Verimag, France)

Steering Committee

Gerard Berry (Estrel Technologies, France)
Thomas A. Henzinger (University of California at Berkeley, USA)
Hermann Kopetz (Technical University of Vienna, Austria)
Edward A. Lee (University of California at Berkeley, USA)
Ragunathan Rajkumar (Carnegie Mellon University, USA)
Alberto L. Sangiovanni-Vincentelli (University of California at Berkeley, USA)
Douglas C. Schmidt (Vanderbilt University, USA)
Joseph Sifakis (Verimag, France)
John Stankovic (University of Virginia, USA)
Reinhard Wilhelm (Universitat des Saarlandes, Germany)
Wayne Wolf (Princeton University, USA)

Sponsors

School of Engineering and Applied Science, University of Pennsylvania
US National Science Foundation
University Research Foundation, University of Pennsylvania

Referees

Evans Aaron
Sherif Abdelwahed
Astrit Ademaj
Luis Almeida
David Arney
Andre Arnold
Ted Bapty
Marius Bozga
Victor Braberman
Benoit Caillaud
Luca Carloni
Paul Caspi
Elaine Cheong
Namik Cho
Chun-Ting Chou
Thao Dang
Luca de Alfaro
Wilfried Elmenreich
Gian Luca Ferraro
Diego Garbervetsky
Alain Girault
Flavius Gruian
Zonghua Gu
Rhan Ha
Wolfgang Haidinger
Seongsoo Hong
Haih Huang
Zhining Huang
Thierry Jeron
Bernhard Josko
Gabor Karsai
Jesung Kim
Moon H. Kim
Raimund Kirner
Sanjeev Kohli
Ben Lee
Jaejin Lee
Xavier Leroy
Xiaojun Liu
Thmoas Losert
Florence Maraninchi
Eleftherios Matsikoudis
Anca Muscholl
Sandeep Neema
Steve Neuendorffer
Ramine Nikoukhah
David Nowak
Ileana Ober
Iulian Ober
Roman Obermaisser
Philipp Peti
Babu Pillai
Sophie Pinchinat
Peter Puschner
Daji Qiao
Wei Qin
Subbu Rajagopalan
Anders Ravn
Vlad Rusu
Usa Sammapun
Insik Shin
Oleg Sokolsky
Wilfried Steiner
Li Tan
Stavros Tripakis
Manish Vachharajani
Marisol Garcia Valls
Igor Walukievicz
Hangsheng Wang
Shaojie Wang
Shige Wang
Jian Wu
Yang Zhao
Haiyang Zheng
Rachel Zhou
Xinping Zhu

Table of Contents

A Probabilistic Framework for Schedulability Analysis

Alan Burns, Guillem Bernat, and Ian Broster

Real-Time Systems Research Group
Department of Computer Science
University of York, UK

Abstract. The limitations of the deterministic formulation of scheduling are outlined and a probabilistic approach is motivated. A number of models are reviewed with one being chosen as a basic framework. Response-time analysis is extended to incorporate a probabilistic characterisation of task arrivals and execution times. Copulas are used to represent dependencies.

1 Introduction

Scheduling work in real-time systems is traditionally dominated by the notion of absolute guarantee. The load on a system is assumed to be bounded and known, worst-case conditions are presumed to be encountered, and static analysis is used to determine that all timing constraints (deadlines) are met in all circumstances.

This deterministic framework has been very successful in providing a solid engineering foundation to the development of real-time systems in a wide range of applications from avionics to consumer electronics. The limitations of this approach are, however, now beginning to pose serious research challenges for those working in scheduling analysis. A move from a deterministic to a probabilistic framework is advocated in this paper where we review a number of approaches that have been proposed. The sources of the limitations are threefold:

1. Fault tolerant systems are inheritly stochastic and cannot be subject to absolute guarantee.
2. Application needs are becoming more flexible and/or adaptive – work-flow does not follow pre-determined patterns, and algorithms with a wide variance in computation times are becoming more commonplace.
3. Modern super-scalar processor architectures with features such as cache, pipelines, branch-prediction, out-of-order execution etc. result in computation times for even straight-line code that exhibits significant variability. Also, execution time analysis techniques are pessimistic and can only provide upper bounds on the execution time of programs.

Note, these characteristics are not isolated to so called 'soft real-time systems' but are equally relevant to the most stringent hard real-time application. Nevertheless, the early work on probabilistic scheduling analysis has been driven by a wish to devise effective QoS control for soft real-time systems [1,17,18,38].

In this paper we consider four interlinked themes:

1. Probabilistic guarantees for fault-tolerant systems
2. Representing non-periodic arrival patterns
3. Representing execution-time
4. Estimating extreme values for execution times.

In the third and fourth themes it will become clear that one of the axioms of the deterministic framework – a well founded notion of worst-case execution time – is not sustainable. The parameterisation of work-flow needs a much richer description than has been needed hitherto.

The above themes are discussed in Sections 3 to 5 of this paper. Before that we give a short review of standard schedulability analysis using a fixed priority scheme as the underlying dispatching policy (see Burns and Wellings [11] for a detailed discussion of this analysis). We restrict our consideration to the scheduling of single resources – processors or networks. In Section 6 we bring the discussion together and draw some conclusions.

2 Standard Scheduling Analysis

For the traditional fixed priority approach, it is assumed that there is a finite number (N) of tasks (τ_1 .. τ_N). Each task has the attributes of minimum inter arrival time, T, worst-case execution time, C, deadline, D and priority P. Each task undertakes a potentially unbounded number of invocations; each of which must be finished by the deadline (which is measured relative to the task's invocation/release time). All tasks are deemed to share a *critical instance* in which they are all released together; this is often taken to occur at time 0. It is important to emphasise that the standard analysis assumes that the two limits on load (minimum T and maximum C) are actually observed at run time. No compensation for average or observed T or C is accommodated.

We assume a single processor platform and restrict the model to tasks with $D \leq T$. For this restriction, an optimal set of priorities can be derived such that $D_i < D_j \Rightarrow P_i > P_j$ for all tasks τ_i, τ_j [26]. Tasks may be periodic or sporadic (as long as two consecutive releases are separated by at least T). Once released, a task is not suspended other than by the possible action of a concurrency control protocol surrounding the use of shared data. A task, however, may be preempted at any time by a higher priority task. System overheads such as context switches and kernel manipulations of delay queues etc. can easily be incorporated into the model [21][10] but are ignored here.

The worst-case response time (completion time) R_i for each task (τ_i) is obtained from the following [20][2]:

$$R_i \; = \; C_i \, + \, B_i \, + \sum_{j \in \mathbf{hp}(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \qquad (1)$$

where $\mathbf{hp}(i)$ is the set of higher priority tasks (than τ_i), and B_i is the maximum blocking time caused by a concurrency control protocol protecting shared data.

Table 1. Example Task Set

Task	P	T	C	D	B	R	Schedulable
τ_1	1	100	30	100	0	30	TRUE
τ_2	2	175	35	175	0	65	TRUE
τ_3	3	200	25	200	0	90	TRUE
τ_4	4	300	30	300	0	150	TRUE

To solve equation (1) a recurrence relation is produced:

$$r_i^{n+1} = C_i + B_i + \sum_{j \in \mathbf{hp}(i)} \left\lceil \frac{r_i^n}{T_j} \right\rceil C_j \tag{2}$$

where r_i^0 is given an initial value of 0. The value r^n can be considered to be a computational window into which an amount of computation C_i is attempting to be placed. It is a monotonically non-decreasing function of n. When r_i^{n+1} becomes equal to r_i^n then this value is the worst-case response time, R_i [10]. However if r_i^n becomes greater than D_i then the task cannot be guaranteed to meet its deadline, and the full task set is thus unschedulable. It is important to note that a fixed set of time points are considered in the analysis: 0, r_i^1, r_i^2, ..., r_i^n.

Table 1 describes a simple 4 task system, together with the worst-case response times that are calculated by equation (2). Priorities are ordered from 1, with 4 being the lowest value, and blocking times have been set to zero for simplicity. Scheduling analysis is independent of time units and hence simple integer values are used (they can be interpreted as milliseconds).

All tasks are released at time 0. For the purpose of schedulability analysis, we can assume that their behaviour is repeated every LCM, where LCM is the least common multiple of the task periods. When faults are introduced it will be necessary to know for how long the system will be executing. Let L be the lifetime of the system. For convenience we assume L is an integer multiple of the LCM. This value may however be very large (for example LCM could be 200ms, and L fifteen years!).

3 Probabilistic Guarantees for Fault-Tolerant Systems

In this review we restrict our consideration to transient faults. Castillo *at al* [13] in their study of several systems indicate that the occurrences of transient faults are 10 to 50 times more frequent than permanent faults. In some applications this frequency can be quite large; one experiment on a satellite system observed 35 transient faults in a 15 minute interval due to cosmic ray ions [12].

Hou and Shin [19] have studied the probability of meeting deadlines when tasks are replicated in a hardware-redundant system. However, they only consider permanent faults without repair or recovery. A similar problem was studied by Shin et al [36]. Kim et al [22] consider another related problem: the probability of a real-time controller meeting a deadline when subject to permanent faults with repair.

To tolerate transient faults at the task level will require extra computation. This could be the result of restoration and re-execution of some routine, the execution of an exception handler or a recovery block. Various algorithms have been published which attempt to maximise the available resources for this extra computation [37,35,3]. Here we consider the nature of the guarantee that these algorithms provide. Most approaches make the common *homogeneous Poisson process* (HPP) assumptions that the fault arrival rate is constant and that the distribution of the fault-count for any fixed time interval can be approximated using a Poisson probability distribution. This is an appropriate model for a random process where the probability of an event does not change with time and the occurrence of one fault event does not affect the probability of another such event. A HPP process depends only on one parameter, viz., the expected number of events, λ, in unit time; here events are transient faults with $\lambda = 1/MTBF$, where $MTBF$ is the Mean Time Between transient Faults[1]. Per the definition of a Poisson Distribution,

$$\mathbf{Pr}_n(t) = \frac{e^{-\lambda t}(\lambda t)^n}{n!} \tag{3}$$

gives the probability of n events during an interval of duration t. If we take an event to be an occurrence of a transient fault and Y to be the random variable representing the number of faults in the lifetime of the system (L), then the probability of zero faults is given by

$$\mathbf{Pr}(Y = 0) = e^{-\lambda L}$$

and the probability of at least one fault

$$\mathbf{Pr}(Y > 0) = 1 - e^{-\lambda L}$$

Modelling faults as stochastic events means that an absolute guarantee cannot be given. There is a finite probability of any number of faults occurring within the deadline of a task. It follows that the guarantee must have a confidence level assigned to it and this is most naturally expressed as a probability. One way of doing this is to calculate the worst case fault behaviour that can (just) be tolerated by the system, and then use the system fault model to assign a probability to that behaviour. Two ways of doing this have been studied in detail.

- Calculate the maximum fault arrival rate that can be tolerated [9] – represented by T_F, the minimum fault arrival interval.
- Calculate the maximum number of faults each task can tolerate before its deadline [31].

The first approach is more straightforward (there is only a single parameter) and is reviewed in the following section. The basic form of the analysis is to obtain T_F from the task set, and then to derive a probabilistic guarantee from T_F. An alternative formulation

[1] MTBF usually stands for mean time between failures, but as the systems of interest are fault tolerant many faults will not cause system failure. Hence we use the term MTBF to model the arrival of transient faults.

is to start with a required guarantee (for example, probability of fault per task release of 10^{-6}) and to then test for schedulability. This is the approach of Broster et al [6] and is outlined in Section 3.2.

3.1 Probabilistic Guarantee for T_F

Let F_k be the extra computation time needed by τ_k if an error is detected during its execution. This could represent the re-execution of the task, the execution of an exception handler or recovery block, or the partial re-execution of a task with checkpoints. In the scheduling analysis the execution of task τ_i will be affected by a fault in τ_i or any higher priority task. We assume that any extra computation for a task will be executed at the task's (fixed) priority[2].

Hence if there is just a single fault, equation (1) will become [33][7][3]:

$$R_i \;=\; C_i \,+\, B_i \,+\, \sum_{j \in \mathbf{hp}(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \,+\, \max_{k \in \mathbf{hep}(i)} (F_k) \tag{4}$$

where $\mathbf{hep}(i)$ is the set of tasks with priority equal or higher than τ_i, that is $\mathbf{hep}(i) = \mathbf{hp}(i) \cup \{\tau_i\}$.

This equation can again be solved for R_i by forming a recurrence relation. If all R_i values are still less than the corresponding D_i values then a deterministic guarantee is furnished.

Given that a fault tolerant system has been built it can be assumed (although this would need to be verified) that it will be able to tolerate a single isolated fault. And hence the more realistic problem is that of multiple faults; at some point all systems will become unschedulable when faced with an arbitrary number of fault events.

To consider maximum arrival rates, first assume that T_f is a known minimum arrival interval for fault events. Also assume the error latency is zero (this restriction is easily removed [9]). Equation (4) becomes [33,7]:

$$R_i \;=\; C_i \,+\, B_i \,+\, \sum_{j \in \mathbf{hp}(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \,+\, \left\lceil \frac{R_i}{T_f} \right\rceil \max_{k \in \mathbf{hep}(i)} (F_k) \tag{5}$$

Thus in interval $(0\ R_i]$ there can be at most $\left\lceil \frac{R_i}{T_f} \right\rceil$ fault events, each of which can induce F_k amount of extra computation. The validity of this equation comes from noting that fault events behave identically to sporadic tasks, and they are represented in the scheduling analysis in this way [2].

Table 2 gives an example of applying equation (5). Here full re-execution is required following a fault (ie. $F_k = C_k$). Two different fault arrival intervals are considered. For one the system remains schedulable, but for the shorter interval the final task cannot be guaranteed. In this simple example, blocking and error latency are assumed to be zero. Note that for the first three tasks, the new response times are less than the shorter T_f value, and hence will remain constant for all T_f values greater than 200.

[2] Recent results had improved the following analysis by allowing the recovery actions to be executed at a higher priority [27].

[3] We assume that in the absence of faults, the task set is schedulable.

Table 2. Example Task Set - T_f = 300 and 200

Task	P	T	C	D	F	R	R
						$T_f = 300$	$T_f = 200$
τ_1	1	100	30	100	30	60	60
τ_2	2	175	35	175	35	100	100
τ_3	3	200	25	200	25	155	155
τ_4	4	300	30	300	30	275	UNSCH

Table 3. Example Task Set - T_F set at 275

Task	P	T	C	D	R
					$T_F = 275$
τ_1	1	100	30	100	60
τ_2	2	175	35	175	100
τ_3	3	200	25	200	155
τ_4	4	300	30	300	275

The above analysis has assumed that the task deadlines remain in effect even during a fault handling situation. Some systems allow a relaxed deadline when faults occur (as long as faults are rare). This is easily accommodated into the analysis.

Limits to Schedulability

Having formed the relation between schedulability and T_f, it is possible to apply sensitivity analysis to equation (5) to find the minimum value of T_f that leads to the system being just schedulable. As indicated earlier, let this value be denoted as T_F (it is the threshold fault interval).

Sensitivity analysis [39,24,23,34] is used with fixed priority systems to investigate the relationship between values of key task parameters and schedulability. For an unschedulable system it can easily generate (using simple branch and bound techniques) factors such as the percentage by which all Cs must be reduced for the system to become schedulable.

Similarly for schedulable systems, sensitivity analysis can be used to investigate the amount by which the load can be increased without jeopardising the deadline guarantees. Here we apply sensitivity analysis to T_f to obtain T_F.

When the above task set is subject to sensitivity analysis it yields a value of T_F of 275. The behaviour of the system with this threshold fault interval is shown in Table 3. A value of 274 would cause τ_4 to miss its deadline.

In the paper cited earlier for this work, formulae are derived for the probability that during the lifetime of the system, L, no two faults will be closer than T_F. This is denoted by $Pr(W < T_F)$; where W denotes the actual (unknown) minimum inter-fault gap. Of course, $Pr(W < T_F)$ is equivalent to $1 - Pr(W \geq T_F)$. The exact formulation is

$$\begin{aligned}
Pr(W \geq T_F) &= \sum_{n=0}^{\infty} P_{n,\,(T_F/L)}\; e^{-\lambda L}\frac{(\lambda L)^n}{n!} \\
&= e^{-\lambda L}\left\{1 + \lambda L + \sum_{n=2}^{\left\lceil \frac{L}{T_F}\right\rceil} \left(1 - (n{-}1)\left(\frac{T_F}{L}\right)\right)^n \frac{(\lambda L)^n}{n!}\right\} \\
&= e^{-\lambda L}\left\{1 + \lambda L + \sum_{n=2}^{\left\lceil \frac{L}{T_F}\right\rceil} \frac{\lambda^n}{n!}\,(L - (n{-}1)T_F)^n\right\}
\end{aligned} \tag{6}$$

this leads to

$$Pr(W \geq T_F) = e^{-\lambda L}\left\{1 + \lambda L + \sum_{n=2}^{\infty} \frac{(\lambda L - (n-1)\lambda T_F)_+^n}{n!}\right\} \tag{7}$$

Fortunately upper and lower bands can also be derived.

Theorem 1. *If $L/(2T_F)$ is a positive integer then*

$$Pr(W{<}T_F) < 1 + \left[e^{-\lambda T_F}\,(1 + \lambda T_F)\right]^{\frac{L}{T_F}-1} - 2\left[e^{-2\lambda T_F}\,(1 + 2\lambda T_F)\right]^{\frac{L}{2T_F}}$$

Theorem 2. *If $L/(2T_F)$ is a positive integer then*

$$Pr(W{<}T_F) > 1 - \left[e^{-\lambda T_F}\,(1 + \lambda T_F)\right]^{\frac{L}{T_F}}$$

which gives rise to the approximations

Corollary 1. *An approximation for the upper bound on $Pr(W{<}T_F)$ given by Theorem 1 is $\frac{3}{2}\lambda^2 LT_F$, provided that λT_F, $\lambda^2 LT_F$ are small, and $L{\gg}T_F$.*

Corollary 2. *An approximation for the lower bound on $Pr(W{<}T_F)$ given by Theorem 2 is $\frac{1}{2}\lambda^2 LT_F$, provided only that λT_F, $\lambda^2 LT_F$ are small.*

The important upper bound approximation of Corollary 1 can be written in the form $\frac{3}{2}(\lambda L)(\lambda T_F)$. It will often be the case that $\lambda T_F < 10^{-2}$; indeed this constraint allowed the approximations to deliver useful values. But λL can vary quite considerably from 10^{-2} or less in friendly environments to 10^3 or more in long-life, hostile domains.

The example introduced in earlier had a T_F value of 275ms. Table 4 gives the upper bound on the probability guarantee for various values of λ and L (in seconds).

A typical outcome of this analysis is that in a system that has a non-stop run-time (L) of 10 hours with a mean time between transient faults of 1000 hours and a tolerance of faults that do not appear closer than 1/100 of an hour, the probability of missing a deadline is upper bounded by $1.5{\times}10^{-7}$. A lower bound is also derived (Corollary 4) and this yields a value of $0.5{\times}10^{-7}$. For these parameters the exact analysis produces a value very close to $1.0{\times}10^{-7}$.

When $\lambda L{<}10^{-2}$, λL approximates the probability of any fault happening during the mission of duration L. So, $\frac{2}{3}(\lambda T_F)^{-1}$ represents the gain in reliability that is achieved by the use of fault tolerance, under the other assumptions stated. For example, in Table 4, when $\lambda = 10^{-2}$ and $L = 1$ the gain is approximately 10^6.

Table 4. Upper bound on Non-Schedulability due to Faults

	λ		
L	1	10^{-2}	10^{-4}
1	1.1×10^{-4}	1.1×10^{-8}	1.1×10^{-12}
10^1	1.1×10^{-3}	1.1×10^{-7}	1.1×10^{-11}
10^2	1.1×10^{-2}	1.1×10^{-6}	1.1×10^{-10}
10^4	1	1.1×10^{-4}	1.1×10^{-8}

3.2 Scheduling Analysis for Probabilistic Events

In contrast to the above approach Broster et al [6] produce a response time profile that is a direct result of the probabilisitic fault model. They do not assume a single T_F value. Their analysis was originally derived for CAN scheduling but is generalised here. Rather than consider the lifetime of the system, L, the analysis is formulated in terms of the likelihood of failure per execution of the task. There is an obvious straightforward relationship between these two formulations.

It was noted earlier that standard response analysis for fixed priority scheduling looks at a series of time points, 0, r_i^1, r_i^2, ..., r_i^n. (We shall ignore the task subscript in the following description). This approach assumes no faults. It is possible to generalise the scheme by replacing the single point r^1 by a series of points that are obtained by assuming one fault, two faults etc. That is $r(0)^1$, $r(1)^1$, $r(2)^1$, ..., $r(m)^1$ (with $r(0)^1 = r^1$). The fault model (eg. Poisson) is able to provide a value for the probability of s faults in $r(s)^1$ – see equation(3).

Once all probabilities are obtained that are greater then the required guarantee (eg. 10^{-6}) the sequence is terminated and for each value of $r(s)^1$ a set of secondary points are obtained $r(s,q)^2$ with the second parameter, q, being the number of faults in the interval $[r(s)^1, r(s,q)^2)$, $q = 0, 1, 2, \ldots$. This process is repeated and at each interaction, events (faults) with a likelihood below the threshold of interest are ignored. Although the overall search space is potentially large, the trimming that occurs due to dismissing rare events, leads to a scheme has been shown to be tractable for real sized problems. Once all sequences terminate, deadlines are checked and a probability distribution for response times is obtained.

Scheduling Analysis for Required Probabilistic Guarantee. An alternative way of formulating the question of a probabilistic guarantee of schedulability is to calculate the worst-case response time when fault occurrences, below the threshold of interest, are ignored. Note this is not the same question as that addressed by Broster. To answer this formulation of the question is much easier as only a single iteration of the scheduling equation is needed. First equation (4) is solved assuming zero faults (let this value be represented by $R(0)$ – we again ignore the task subscript). Then the fault model, the threshold and this value $R(0)$ are used to estimate the number of faults (of interest) in the interval. Let this value be S_1. From equation (4) we can now calculate a new value for response time, $R(S_1)$:

$$R_i(S_1) \; = \; C_i \; + \; B_i \; + \sum_{j \in \mathbf{hp}(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \; + \; S_1 \max_{k \in \mathbf{hep}(i)} (F_k) \tag{8}$$

The number of faults of interest in $R(S_1)$ is then calculated. If this new value, S_2, is equal to S_1 then the formulation is stable and $R(S_1)$ is the worst case response time. Alternatively equation (8) is solved for S_2 and the process continues until either stability is obtained or a response time greater than deadline is calculated and unschedulability is proclaimed.

To illustrate the approach consider the small example given earlier for the other approach. If we set the threshold value to 10^{-6} and assume λ is obtained from a mean time between errors of 0.1 seconds then the same response time give in Table 3 are observed (eg. 60, 100, 155 and 275).

Note that the above analysis is relatively straightforward when a Poisson derived fault model is assumed. Nevertheless, the framework can still be used if other arrival distributions are more appropriate.

3.3 Summary

The schemes reviewed in this section all have a common theme. Scheduling approaches are used to maximise the effective resources that can be made available, when required, for fault tolerance. Then limits to schedulability are derived in conjuction with the probability of those limits being observed during execution. This furnishes the probabilistic guarantee. Alternatively, a standard yes/no guarantee is obtained while faults below a threshold of likelihood are ignored.

4 Probabilistic Guarantees with Non-periodic Work

Initially scheduling analysis assumed a purely periodic work flow [28]. The sporadic jobs were incorporated by assuming a minimum arrival time, that in the worst case was exhibited by the system. In effect a sporadic job behaved exactly the same as a periodic one. Response time analysis, as outlined in Section 2 can actually deal with a much more general model of non-periodic work. Let $A_k(t)$ be defined to be a function that delivers the maximum number of arrivals of task k in any interval [0,t). Then equation (1) becomes

$$R_i \; = \; C_i \; + \; B_i \; + \sum_{j \in \mathbf{hpp}(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \; + \sum_{k \in \mathbf{hpn}(i)} A_k(R_i) C_k \tag{9}$$

where $\mathbf{hpp}(i)$ is now the set of higher priority periodic tasks, and $\mathbf{hpn}(i)$ the set of higher priority non-periodic tasks.

Although a useful generalisation, equation (9) is still a deterministic one. It assumes that the worst-case number of arrivals of all sporadics tasks will occur with probability one. To deal with non-periodic tasks that follow a stochastic model a different framework is needed. First, some form of probabilistic density function will be needed for all sources of sporadic work. If nothing is known about the arrival pattern of work then clearly no guarantee, not even a probabilistic one, can be given. One method of incorporating this stochastic work load is to use the same approach as for fault tolerance. After all, faults handling routines are, from a scheduling point of view, just one form of non-periodic work. The approach outlined in Section 3.2 can then be applied.

A probability threshold for the system must be defined. This is the value below which events are sufficiently rare to be ignored. Let ρ be this threshold value. We redefined the function A given earlier in this section as follows: $A_k(\rho, t)$ is the number of arrival events in any interval of length t with a probability of more than ρ. So $A_k(10^{-6}, 30)$, for example, would give the result 2 if the probability of 3 or more arrivals in 30 time units is less than 10^{-6} (and the probability of 2 is more than this value). Equation (9) then becomes:

$$R_i \;=\; C_i \,+\, B_i \,+ \sum_{j \in \mathbf{hpp}(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \,+ \sum_{k \in \mathbf{hpn}(i)} A_k(\rho, R_i) C_k \tag{10}$$

This approach contains some explicit assumptions that would need to be clarified. For example, it assumes each source of arrivals is independent of each other; also that the computation time of the job is independent of the arrival behaviour. The existence of correlations would complicate the analysis – but pessimistic assumptions may be relatively straightforward to incorporate (see later discussion on the use of Copulas).

Care must of course be taken with choosing the probability threshold for the system. If an application is, by its specification, meant to deal with rare events then the threshold must be chosen so that such events (at least one in any small interval) are always incorporated into the run-time behaviour that is being analysed.

5 Representing Execution Time

The above discussions have generalised the notion of work flow by allowing the arrival of work to be described stochastically. However the worst-case resource requirement of each job is still represented by a single parameter C. This represents the maximum processor (resource) time needed by the job on each and every arrival. In developing a general framework for scheduling analysis, where application code and processor behaviour combined to produce a rich execution profile, it is not surprising that this single parameter approach is becoming limited in its application. Even a two (average and worst-case) or three (add minimum) parameter scheme is far from adequate.

In other works [8,16,4] we have argued that it is now inadequate to use analysis alone to obtain a single worst-case execution time (WCET) value. Rather a combination of analysis and measurement must be used to obtain a probabilistic representation of the entire execution profile of the task. Moreover, this probability density function must extend beyond observed data to predict the likelihood of experiencing, during the real execution of the system, extreme (large) values for execution time. Data obtained from measurement of relatively straightforward code illustrates two general characteristics of execution time profiles (let O be the maximum observed value during meansurement).

– Large observed values for computation time may be sufficiently rare that for non-hard systems it would be inappropriate for any schedulability test to assume this value for every task's execution (ie. O is too large to use).
– Large observed values for computation time may not represent the worst-case that will be experienced during real execution, and extrapolations beyond observed values will be needed for some hard real-time systems (ie. O is too small to use).

The alternative to simple parameterisation is to model execution time as a random variable following some probability distribution. These distributions (execution time profiles) being derived from measurement. But the granularity of measurement remains an open issue. Three levels are possible:

1. The basic block - a sequence of instructions.
2. The task - which consists of a number of basic blocks.
3. The system - which consists of a number of tasks.

If measurement is used at the task level then knowledge about the structure of the task is being ignored, however uncertainties arising from the interactions of basic blocks are being sampled. If analysis is used at the task level (with measurements only being done for basic blocks) then the rules for combining the execution time profiles need to be articulated. A similar trade-off exists at the system level.

In the work we have undertaken, we have used measurement only at the basic block level and hence we must address the issue of how to combine probability distributions. If independence could be assumed then standard statistical methods could be applied. Unfortunately there seems to be ample evidence that this assumption would be overtly optimistic. A series of basic blocks may be strongly correlated. Moreover a series of task executions within a schedule may also be dependent upon one another. Indeed the execution of the same task, one or more times, within the response time of a lower priority task may exhibit a strong correlation. These may be positive (a long execution time is more likely to be followed by another large one) or negative (long will induce a short one next time).

5.1 Use of Copulas

Copulas are a general mathematical tool to construct multivariate distributions and to investigate dependence structures between random variables [32]. A copula is basically a joint distribution function with uniform marginals. The main feature is that they allow one to separate the marginal distributions from the dependency between the two random variables, therefore given a joint probability distribution it is possible to characterize it uniquely with the marginal distributions and a copula. Similarly, given two marginal distributions and a copula, it is possible to derive the joint distribution and this is unique.

The importance is that the copula captures the *dependence structure* between random variables. So given two joint distributions with different marginal distributions but that capture the same dependency process, they would have the same copula.

There are two additional results of importance for this analysis, the first one is that the set of copulas is a partially ordered set and there exist two special copulas, called the lower and upper Fréchet bounds that characterize the maximum and minimum dependence between random variables.

The problem of timing analysis can be formulated as follows, if X and Y are two random variables that represent the execution time of two blocks of code with respective distribution functions $F_X(t)$ and $F_Y(t)$, we want to determine the distribution of $Z = X + Y$ which is the execution time of X followed by Y, $F_Z(t)$. If X and Y are independent, the probability density function of Z corresponds to the standard convolution of the probability density functions of X and Y. However, if this hypothesis is not correct then we can use the theory of copulas to construct $F_Z(t)$.

If the joint distribution is known, (or its copula) then the distribution of Z is a straightforward generalisation of the convolution but using the joint distribution instead. More importantly, if the dependency is not known, then it is possible to find upper and lower bounds of the distribution function for *any* possible dependency between the marginal distributions [5,29,14]. Some generalisations of these results allow to tighten even more these bounds if partial knowledge of the dependence is known.

5.2 Representing Extreme Execution Times

It was noted earlier that for some hard real-time systems execution time values beyond what have been observed during tests need to be taken into account if very low levels of failure are to be tolerated. One means of addressing this issue is to apply the branch of statistics concerned with extreme values. One of the three extreme value distributions is used to 'fit' the data and then give predictions beyond the observed data range. We have had some success [16] in fitting the Gumble distribution but it is still not clear if this is a general purpose technique. What this approach provides is a probability distribution for the worst-case value for a task's execution. One useful result of this study is that a collection of tasks has a bounded behaviour. Let $C_1..C_N$ be the worst-case times derived from the above approach with probability threshold ρ; then the sequential execution of each task will have a total expected execution time of $C_1 + C_2 + C_3 + .. + C_N$ with probability bound ρ.

6 Other Relevant Work on Probabilistic Analysis

There have been some other approaches using probabilistic methods in real-time systems. The work of Diaz et.al. [15] computes probability distributions of the response times of entirely periodic (fixed release times) task systems with random execution times. The work relies on the independence of the execution times of the different tasks. The work improves on an earlier work by Gardner et.al. [18]. The works of Nissanke [25] and Eles [30] also tackle this problem. However, none of these approaches address the issue of extreme distributions or dependencies between execution times or task arrivals.

7 Conclusion: A Probabilistic Framework

Bringing together the above approaches we are able to postulate one means of constructing a scheduling framework that can deal with stochastic parameterisation of work flow. The following are the main components of such a framework.

1. All tasks have an arrival pattern expressed as $A(\rho, t)$ - the number of instances of the task likely to occur in any interval of length t, where the probability of greater than $A(\rho, t)$ occurring is less than ρ.
2. All tasks have an execution profile derived that extends beyond the data observed during test.
3. A threshold probability is defined for the system. Events (task arrivals or execution times) with a likelihood of occurring less than this threshold are ignored. The threshold could be expressed as a likelihood of failure per execution of any task of interest.
4. A worst-case response time of each task is calculated from the above data as follows:
   - An initial estimate, R_0, is obtained by assuming all tasks arrive once with execution times derived from their profiles and ρ.
   - The number of tasks arriving in R_0 is derived (using $A(\rho, R_0)$ and any dependency relationships).
   - A conservative copula is used to combine the execution profiles of those jobs.
   - A new value for R_0 (i.e. R_1) is obtained by using the threshold probability value on this derived distribution.
   - Repeat until a stable value of R is obtained (or R expands beyond the task's deadline).

It would be at least theoretically possible to vary the probability threshold to derive a relation between response time and this threshold.

In conclusion, we have argued in support of the developed the notion of a probabilistic assessment of schedulability and shown how it can be derived from the stochastic behaviour of the work that the real-time system must accomplish. Many aspects of this framework require significant further study, and we aim to continue with this line of investigation.

References

1. A.K. Atlas and A. Bestavros. Statistical rate monotonic scheduling. In *Proceedings of the 19th IEEE Real-Time Systems Symposium, Madrid, Spain*, pages 123–132. IEEE Computer Society Press, 1998.
2. N. C. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, 1993.
3. G. Bernat and A. Burns. New results on fixed priority aperiodic servers. In *20th IEEE Real-Time Systems Symposium*, Phoenix. USA, December 1999.
4. G. Bernat, A. Colin, and S. M. Petters. WCET analysis of probabilistic hard real–time systems. In *Proceedings of the 23rd Real-Time Systems Symposium RTSS 2002*, pages 279–288, Austin, Texas, USA, 2002.

5. G. Bernat and M. Newby. Probabilistic WCET analysis, an approach using copulas. Technical report, Department of Computer Science University of York, Technical Report, 2003.
6. I. Broster, A. Burns, and G. Rodríguez-Navas. Probabilistic analysis of CAN with faults. In *Proceedings of the 23rd Real-time Systems Symposium*, Dec 2002.
7. A. Burns, R. I. Davis, and S. Punnekkat. Feasibility analysis of fault-tolerant real-time task sets. *Euromicro Real-Time Systems Workshop*, pages 29–33, June 1996.
8. A. Burns and S. Edgar. Predicting computation time for advanced processor architectures. In *Proceedings 12th EUROMICRO conference on Real-time Systems*, 2000.
9. A. Burns, S. Punnekkat, L. Strigini, and D.R. Wright. Probabilistic scheduling guarantees for fault-tolerant real-time systems. In *Proceedings of the 7th International Working Conference on Dependable Computing for Critical Applications. San Jose, California*, pages 339–356, 1999.
10. A. Burns and A. J. Wellings. Engineering a hard real-time system: From theory to practice. *Software-Practice and Experience*, 25(7):705–26, 1995.
11. A. Burns and A. J. Wellings. *Real-Time Systems and Programming Languages*. Addison Wesley Longman, 3rd edition, 2001.
12. A. Campbell, P. McDonald, and K. Ray. Single event upset rates in space. *IEEE Transactions on Nuclear Science*, 39(6):1828–1835, December 1992.
13. X. Castillo, S.P. McConnel, and D.P. Siewiorek. Derivation and Calibration of a Transient Error Reliability Model. *IEEE Transactions on Computers*, 31(7):658–671, July 1982.
14. H. Cossette, M. Denuit, and E. Marceau. Distributional bounds for functions of dependent risks. Technical report, Bulletin suisse des actuaires., 2001.
15. J.L. Diaz, D. F. García, K. Kim, C.-G. Lee, L.L. Bello, J.M. Lopez, S.L. Min, and O. Mirabella. Stochastic analysis of periodic real-time systems. In *22nd IEEE Real-Time Systems Symposium.*, Austin, TX. USA, 2002.
16. S. Edgar and A. Burns. Statistical analysis of WCET for scheduling. In *Proceedings IEEE Real-Time Systems Symposium*, 2001.
17. M. K. Gardner. *Probabilstic Analysis and Scheduling of Critical Soft Real-time Systems*. PhD thesis, University of Illinois, Computer Science, Urbana, Illinois, 1999.
18. M. K. Gardner and J.W. Lui. Analysing stochastic fixed-priority real-time systems. In *Proceedings of the 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 1999.
19. C.-J. Hou and K. G. Shin. Allocation of periodic task modules with precedence and deadline constraints in distributed real-time systems. *IEEE Transactions on Computers*, 46(12):1338–1356, 1997.
20. M. Joseph and P. Pandya. Finding response times in a real-time system. *BCS Computer Journal*, 29(5):390–395, 1986.
21. D.I. Katcher, H. Arakawa, and J.K. Strosnider. Engineering and analysis of fixed priority schedulers. *IEEE Trans. Softw. Eng.*, 19, 1993.
22. H. Kim, A.L.White, and K. G.Shin. Reliability modeling of hard real-time systems. In *Proceedings 28th Int. Symp. on Fault-Tolerant Computing (FTCS-28)*, pages 304–313. IEEE Computer Society Press, 1998.
23. M. H. Klein, T. A. Ralya, B. Pollak, R. Obenza, and M. G. Harbour. *A Practitioner's Handbook for Real-Time Analysis: A Guide to Rate Monotonic Analysis for Real-Time Systems*. Kluwer Academic Publishers, 1993.
24. J.P. Lehoczky, L. Sha, and V. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. Tech report, Department of Statistics, Carnegie-Mellon, 1987.
25. A. Leulseged and N. Nissanke. Stochastic analysis of periodic real-time systems. In *9th Intl. Conf. on Real-Time and Embeded Computing Systems and applications (RTCSA 2003)*, Taiwan, 2003.

26. J.Y.T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation (Netherlands)*, 2(4):237–250, 1982.
27. G. Lima and A. Burns. An optimal fixed-priority assignment algorithm for supporting fault-tolerant hard real-time systems. *IEEE Transactions on Computer Systems (to appear)*, 2003.
28. C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *JACM*, 20(1):46–61, 1973.
29. G.D. Makarov. Estimates for the distribution function of a sum of two random variales when the marginal distributions are fixed. *Theory Probab. Appli.*, 26:803–806, 1981.
30. S. Manolache, P. Eles, and Z. Peng. Memory and time efficient schedulability analysis of task sets with stochastic execution time. In *Proceedings 13th EUROMICRO conference on Real-time Systems*, 2001.
31. N. Navet, Y.-Q.Song, and F. Simonot. Worst-case deadline failure probability in real-time applications distributed over controller area network. *Journal of Systems Architecture*, 46(1):607–617, 2000.
32. R.B. Nelsen. *An introduction to Copulas*. Springer, 1998.
33. S. Punnekkat. *Schedulability Analysis for Fault Tolerant Real-time Systems*. PhD thesis, Dept. Computer Science, University of York, 1997.
34. S. Punnekkat, R. Davis, and A. Burns. Sensitivity analysis of real-time task sets. In *Proceedings of the Conference of Advances in Computing Science - ASIAN '97*, pages 72–82. Springer, 1997.
35. S. Ramos-Thuel and J.P. Lehoczky. On-line scheduling of hard deadline aperiodic tasks in fixed-priority systems. In *Proceedings of 14th IEEE Real-Time Systems Symposium*, pages 160–171, December 1993.
36. K. G. Shin, M. Krishna, and Y. H. Lee. A unified method for evaluating real-time computer controllers its application. *IEEE Transactions on Automatic Control*, 30:357–366, 1985.
37. M. Silly, H. Chetto, and N. Elyounsi. An optimal algorithm for guaranteeing sporadic tasks in hard real-time systems. In *Proceedings 2nd IEEE Symposium on Parallel and Distributed Systems*, pages 578–585, 1990.
38. T.S. Tia, Z. Deng, M. Shankar, M. Storch, J. Sun, L.C. Wu, and J.S. Liu. Probabilisitc performance guenrantee for real-time tasks with varying computation times. In *Proceedings of the Real-Time Technology and Applications Symposium*, pages 164–173, 1995.
39. S. Vestal. Fixed Priority Sensitivity Analysis for Linear Compute Time Models. *IEEE Transactions on Software Engineering*, 20(4):308–317, April 1994.

Resource-Efficient Scheduling for Real Time Systems

Kim G. Larsen

BRICS[*], Aalborg University Denmark
Fredrik Bajers Vej 7, 9220 Aalborg Ø – Denmark
kgl@cs.auc.dk

1 Introduction

For embedded systems efficient utilization of resources is an acute problem arising from the increasing computational demands in all sorts of applications. The consumers constantly demand better functionality and flexibility of embedded products which imply an increase in the resources needed for their realization. In several areas – e.g. portable devices such as PDAs, mobile phones and laptops as well as mission critical systems such as space applications – the ability to design resource efficient solutions is crucial.

Our own work in this area include development and applications of the real-time verification tool UPPAAL to the modeling, analysis and synthesis of resource-efficient and -optimal schedules for real-time systems. Whereas (hard) timeliness and resource-efficiency may be seen as conflicting goals, this approach allows for both goals to be achieved.

2 Verification Using UPPAAL

UPPAAL [20] is an integrated tool environment for modeling, simulating and verification of real-time systems, developed jointly by BRICS at Aalborg University in Denmark and by DoCS at Uppsala University in Sweden. The modeling language of UPPAAL supports model checking safety and (bounded) liveness properties of systems that can be modeled as a collection of timed automata communicating through (broadcast as well as binary) channels or shared variables. Typical application areas include real-time controllers where timely execution of a number of (periodic or sporadic) tasks is controlled by a particular scheduling policy. Given timed automata models of the tasks, the scheduler(s) as well as the real-time environment of the control systems the verification engine of UPPAAL may validate (or refute) the correctness and resouce requirements of the particular scheduling policy applied.

In a number of papers [13,16,11] this approach has been applied to systems controlled by LEGO® MINDSTORM™ bricks. Here the UPPAAL models of task are automatically synthesised from the RCX™ programs together with a model of the (round-robin) scheduling policy applied. The significant difference in the frequency by which changes occur in the scheduler (numerous samples per second) and in the environment (possibly seconds between changes) causes a fragmentation of the symbolic state space of UPPAAL.

[*] Basic Research in Computer Science (www.brics.dk), funded by the Danish National Research Foundation.

As a remedy an exact acceleration technique has been developed [12] and demonstrated efficient on a number of examples.

3 Optimal Scheduling Using UPPAAL

In more recent work UPPAAL has been applied to the *synthesis* of the scheduling policy itself. This work was initiated during the now terminated ESPRIT project VHS [21] and is continuing within the ongoing IST project AMETIST [2]. Modeling the tasks to be scheduled, the constraining, shared resources involved as well as timing assumptions of the environment allows the scheduling problem to be stated as a (time-bounded) reachability question. The (possible) diagnostic trace provided by UPPAAL offers a valid schedule to the problem. Extending UPPAAL with mechanisms for guiding the exploration has proved extremely successful in obtaining feasible solutions to industrial scheduling problems including the synthesis of production schedules for a the Steel Production Plant SIDMAR in Ghent, Belgium [10,14].

Often one want not just a an arbitrary valid schedule but rather a schedule which is *optimal* with respect to some suitable cost measure (e.g. in terms of total elapsed timed or total power consumption). For this purpose an extension of the timed automata model with a notion of *cost* was introduced in [5]: each action transition has an associated price, and likewise, each location has an associate rate giving the increase in cost for delaying on time-unit. In [5], and independently in [1], computabitlity of minimal-cost reachability was demonstrated based on a cost-extension of the classical notion of regions. Later, in [6,17], efficient zone-based algorithms for computing minimum-time respectively minimum-cost reachability has been given and applied to a range of optimal scheduling problems including job shop scheduling problems and aircraft landing problems. The optimization criteria distinguish scheduling algorithms from classical, full state space exploration model checking algorithms. In the UPPAAL implementation [3] they are are used together with, for example, branch-and-bound techniques to prune parts of the search space that are guaranteed not to contain optimal solutions.

Current research considers efficient computation of optimal *infinite* schedules as well as optimal *dynamic* schedules (i.e. optimal under uncertainty or in the presence of uncontrollable behaviour).

4 Applications

Emphasis in the talk will be given to the application of UPPAAL to two industrial scheduling problems focusing on memory utilization and power/energy consumption respectively.

The first case study [4] is provided by Terma A/S who is developing and producing radar sensor equipment. The case study, conducted in the IST AMETIST project, focuses on the memory interface of the video processing board of a radar sensor system used for ground surveillance at airports and for coastal surveillance. The task of the memory interface is to control access to a single memory bus used by 9 different (buffered) data streams. A valid scheduler must guarantee that none of the data streams are ever interrupted, and efficiency is measured in the requirement to buffer sizes. During the first year of AMESTIST several models has been developed including models in UPPAAL [19], where verification has confirmed the validity of the scheduling principle applied by Terma today. Also, the developed models have lead to identification of a memory-optimal, new scheduling principle [22].

The second case study focuses on Dynamic Voltage Scaling, which appears as one of the most promising methods for reducing energy consumption. The principle consists in dynamically adjusting the clock-cycle length as well as the supply voltage depending on the actually task load in the system. However, optimality depends highly on the concrete hardware platform as well as the type of applications. Within the newly formed Danish center for embedded software systems, CISS [8], and in collaboration with Analog Devices various DVS scheduling principles are modeled, simulated and analyzed, with Analog Devices Blackfin DSP processor (ADSP-21535 EZ-KIT Lite) as ultimate target.

References

1. R. Alun, S. La Torre, and G. J. Pappas. Optimal paths in weighted timed automata. To appear in HSCC2001.
2. The AMETIST home page. `http://ametist.cs.utwente.nl`.
3. G. Behrmann. Guiding and cost optimizing uppaal. Web-page, 2002.
4. G. Behrmann, S. Bernicot, T. Hune, K.G. Larsen, S. Lecamp, and A. Skou. Case study 2: A memory interface for radar systems. Deliverable for AMETIST.
5. Gerd Behrmann, Ansgar Fehnker, Thomas Hune, Kim G. Larsen, Paul Pettersson, Judi Romijn, and Frits Vaandrager. Minimum-Cost Reachability for Priced Timed Automata. In Maria Domenica Di Benedetto and Alberto Sangiovanni-Vincentelli, editors, *Proceedings of the 4th International Workshop on Hybrid Systems: Computation and Control*, number 2034 in Lecture Notes in Computer Sciences, pages 147–161. Springer–Verlag, 2001.
6. Gerd Behrmann, Ansgar Fehnker, Thomas Hune, Kim G. Larsen, Paul Pettersson, and Judi Romijn. Efficient Guiding Towards Cost-Optimality in UPPAAL. In T. Margaria and W. Yi, editors, *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, number 2031 in Lecture Notes in Computer Science, pages 174–188. Springer–Verlag, 2001.
7. Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, Yi Wang, and Carsten Weise. New Generation of UPPAAL. In *Int. Workshop on Software Tools for Technology Transfer*, June 1998.
8. The CISS home page. `http://ciss.auc.dk`.
9. H. Dierks, G. Behrmann, and K.G. Larsen. Solving planning problems using real-time model checking (translating pddl3 into timed automata). 2003.
10. A. Fehnker. *Citius, Vilius, Melius - Guiding and Cost-Optimality in Model Checking of Timed and Hybrid Systems*. PhD thesis, KUN Nijmegen, 2002.
11. M. Hendriks. Translating uppaal to not quite c. Technical Report 8, Nijmegen University, 2001.
12. M. Hendriks and K. G. Larsen. Exact acceleration of real-time model checking. In E. Asarin, O. Maler, and S. Yovine, editors, *Electronic Notes in Theoretical Computer Science*, volume 65. Elsevier Science Publishers, April 2002.
13. Thomas Hune. Modelling a Real-Time Language. *In proceedings of 4th Workshop on Formal Methods for Industrial Critical Systems, FMICS'99.*, 1999.

14. Thomas Hune, Kim G. Larsen, and Paul Pettersson. Guided Synthesis of Control Programs Using UPPAAL. In Ten H. Lai, editor, *Proc. of the IEEE ICDCS International Workshop on Distributed Systems Verification and Validation*, pages E15–E22. IEEE Computer Society Press, April 2000.
15. Thomas Hune, Kim G. Larsen, and Paul Pettersson. Guided Synthesis of Control Programs using Uppaal. *Nordic Journal of Computing*, 8(1):43–64, 2001.
16. Torsten K. Iversen, Kåre J. Kristoffersen, Kim G. Larsen, Morten Laursen, Rune G. Madsen, Steffen K. Mortgensen, Paul Pettersson, and Chris B. Thomasen. Model-Checking Real-Time Control Programs. To be published in Proceedings of Euromicro 2000.
17. Kim G. Larsen, Gerd Behrmann, Ed Brinksma, Ansgar Fehnker, Thomas Hune, Paul Pettersson, and Judi Romijn. As cheap as possible: Efficient cost-optimal reachability for priced timed automata. In G. Berry, H. Comon, and A. Finkel, editors, *Proceedings of CAV 2001*, number 2102 in Lecture Notes in Computer Science, pages 493–505. Springer–Verlag, 2001.
18. Kim G. Larsen, Gerd Behrmann, Ed Brinksma, Ansgar Fehnker, Thomas Hune, Paul Pettersson, and Judi Romijn. As cheap as possible: Efficient cost-optimal reachability for priced timed automat. In G. Berry, H. Comon, and A. Finkel, editors, *Proceedings of CAV 2001*, number 2102 in Lecture Notes in Computer Science, pages 493–505. Springer–Verlag, 2001.
19. E. Seshauskaire and M. Mikucionis. Memory interface analysis using the real-time model checker uppaal. Deliverable for AMETIST.
20. The UPPAAL home page. `http://www.uppaal.com`.
21. The VHS home page. `http://www-verimag.imag.fr//VHS/main.html`.
22. Gera Weiss. Optimal Scheduler for a Memory Card. Research report, Weizmann, 2002.

Emerging Trends in Adaptive Middleware and Its Application to Distributed Real-Time Embedded Systems

Joseph P. Loyall

BBN Technologies
Cambridge, MA
jloyall@bbn.com

Abstract. Embedded systems have become prevalent in today's computing world and more and more of these embedded systems are highly distributed and network centric. This adds increasing degrees of resource contention, unpredictability, and dynamism to software that has traditionally been designed with resources being provisioned statically and for the worst case. This paper describes the research that we've been doing in the development of middleware for QoS adaptive systems – an extension to standard off-the-shelf distributed object middleware – and its application to two military distributed real-time embedded systems. These real-world evaluations of the technology then motivate a discussion of the next directions in which we are taking this research.

1 Introduction

Over 99% of all microprocessors are now used for embedded systems [2] that control physical, chemical, biological, or defense processes and devices in real-time. Increasingly, these embedded systems are part of larger *distributed* embedded systems, such as military combat or command and control systems, manufacturing plant process systems, emergency response systems, and telecommunications. A recent OMG workshop on real-time embedded and distributed object computing had commercial representatives describing distributed, real-time embedded (DRE) systems in the following domains [17]:

- Avionics
- Submarine combat control
- Satellite flight control
- Signal analysis
- Software defined radio
- Industrial production
- Automated assembly

Middleware, such as CORBA, is being applied to these types of applications because of its ability to abstract issues of distribution, heterogeneity, and programming language from the design of systems. CORBA, specifically, has spearheaded this trend because of its development of standards supporting the needs of DRE systems, such as RTCORBA [18], FT-CORBA [15], and Minimum CORBA [16].

More and more of these systems are networked *systems of systems*, with heterogeneous platforms and networks. Because of this, predictable real-time behavior in these DRE systems relies on the ability to do the following:

- *Manage resources end-to-end.* Real-time performance might be dominated by a single, most constrained or loaded resource, but it will rely on all the resources end-to-end, processing at each node and the network link between each node along each source-to-sink data path.
- *Adapting and reconfiguring* to changing conditions. In DRE systems of systems, participants might come and go, failures will happen, and mission modes will change. The system must be able to compensate for additional and changing demands, changing mission requirements and priorities, and failures.

Another characteristic of these DRE systems is that their heterogeneous resources will often have varying degrees of available control. For example, a military combat and command/control system that includes unmanned aerial vehicles (UAVs), fighter aircraft, and ground and air command and control (C2) nodes will include a variety of processing nodes, from embedded VME boards to workstations, and a variety of network links, from tactical radio links to ground IP-based network links. Tactical links currently utilize statically allocated and preconfigured time slots for dividing up the available bandwidth. Once time slots are allocated to a source-sink link, they cannot be easily changed. In contrast, dynamic reservation-based and priority-based network management is becoming available for wired, IP networks.

This motivates the need for middleware that extends the current distribution, abstraction, and limited QoS support provided in CORBA. DRE systems need *QoS adaptive middleware*, which supports the following:

- *Specification* of QoS and mission requirements, mission modes and system states, priorities, and trade offs.
- *Measurement* of system conditions, delivered QoS, and resource availability and constraints.
- *Control* of resource mechanisms, managers, and services, and of application level behavior (such as data manipulation).
- *Adaptation and reconfiguration* to compensate for changing system conditions, mission modes, and failures.

In this paper, we describe the Quality Objects (QuO) QoS adaptive middleware, which we have applied to the development of several DRE applications. We will first describe QuO. Then we will describe a couple of DRE applications which we have developed using QuO. Finally, we will describe the directions in which the research is proceeding.

2 Overview of the Adaptive QuO Middleware

Quality Objects (QuO) is a distributed object computing (DOC) framework designed to develop distributed applications that can specify (1) their QoS requirements, (2) the system elements that must be monitored and controlled to measure and provide QoS, and (3) the behavior for controlling and providing QoS and for adapting to QoS variations that occur at run-time. By providing these features, QuO separates the role of functional application development from the role of developing the QoS behavior of the system.

Figure 1 illustrates a client-to-object logical method call in distributed object computing as instantiated by the CORBA standard. In a traditional DOC application, a client makes a logical method call to a remote object. A local ORB proxy (i.e., a stub) marshals the argument data, which the local ORB then transmits across the network. The ORB on the server side receives the message call, and a remote proxy (i.e., a skeleton) then unmarshals the data and delivers it to the remote servant. Upon method return, the process is reversed.

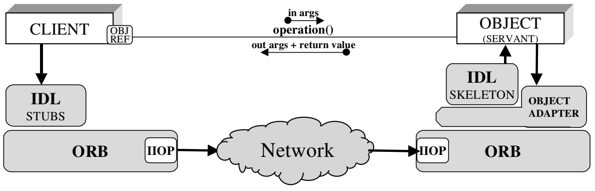


Fig. 1. The CORBA computing model

A method call in the QuO framework is a superset of a traditional DOC call, and includes the following components, illustrated in Figure 2:

- *Contracts* negotiate the QoS an application needs, the QoS available in the system, and the policies for mediating, controlling, and adapting to changes.
- *Delegates* act as local proxies for remote objects. Each delegate provides an interface similar to that of the remote object stub, but adds locally adaptive behavior based upon the current state of QoS in the system, as measured by the contract.

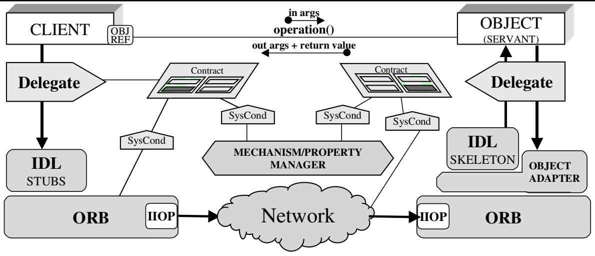


Fig. 2. QuO adds components to control, measure, and adapt to QoS aspects of an application

- *System condition objects* provide interfaces to resources, mechanisms, objects, and ORBs in the system that need to be measured and controlled by QuO contracts.
- *Callback objects* provide notification interfaces to clients or objects in the application.

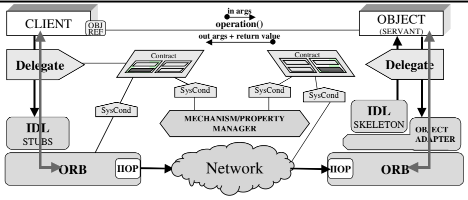
QuO applications can use specialized QoS mechanisms, ORBs, services and managers, such as RSVP network reservation [27], the TAO real-time ORB [23], RTCORBA [18], and the AQuA dependability manager [4], respectively.

Besides the traditional application developers (who develop the client and object implementations) and mechanism developers (who develop the ORBs, property managers, and other distributed resource control infrastructure), QuO middleware supports another development role, namely QoS developers or Qoskateers. Qoskateers are responsible for defining contracts, system condition objects, callback mechanisms, and object delegate behavior. To support the added role of Qoskateer, we have developed an open-source QuO toolkit [12, 13, 25], consisting of the following:

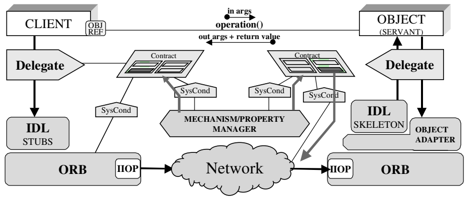
- A set of aspect languages, collectively called *Quality Description Languages (QDL)* for describing QoS contracts and the adaptive behavior of objects and object interactions [12, 13].
- *Code generators* that weave measurement, control, and adaptation code into application programs [12].
- A library of reusable *system condition objects* that provide interfaces to system resources, mechanisms, services, and managers; and guidelines for building custom system condition objects.
- A *runtime kernel*, which coordinates evaluation of contracts and monitoring of system condition objects [25].
- *Object gateways* for inserting measurement, control, and adaptation beneath the ORB layer [19].
- An encapsulation model, called *Qoskets*, for localizing the code associated with a logically connected QoS adaptive behavior so that it can be referred to, and reused, as a single component, even though the behavior might be woven into various, distributed locations in the executable code [20].

2.1 In-Band and Out-of-Band Adaptation

The QuO architecture supports two means for triggering adaptation at many levels throughout the system, e.g., (property) manager-level, middleware-level, and application-level, as illustrated in Figure 3. QuO delegates trigger *in-band* adaptation by making choices upon method calls and returns. A delegate intercepts all method calls to sets of remote objects and adapts its interactions to the remote objects based on the current QoS region given by the contract. Contracts trigger *out-of-band* adaptation when changes in observed system condition objects cause region transitions. In this way, the contract monitors and controls the system's QoS, but operates orthogonal to and outside the in-band



(a) QuO's delegates provide in-band adaptation on message call and return



(b) Out-of-band adaptation occurs when system condition objects and contracts recognize system changes

Fig. 3. QuO supports adaptation upon method call/return and in response to changes in the system

functional behavior of the system. For example, Section 3 describes QuO's use in an avionics dynamic mission replanning system. In this system, in-band QuO adaptation is used to measure the progress of and change the size of data (using tiling and quality controls) being exchanged between aircraft across a constrained network link. Meanwhile, out-of-band QuO adaptation is used to monitor the available CPU and aid in scheduling of the hard and soft real-time avionics tasks.

2.2 The Qosket Encapsulation Model

One goal of QuO is to separate the role of QoS (or systemic) programmer from that of application programmer. A complementary goal of this separation of programming roles is that systemic behaviors can be encapsulated into reusable units that are not only developed separately from the applications that use them, but that can be reused by selecting, customizing, and binding them to an application program. To support this goal, we have defined *Qoskets* as a unit of encapsulation and reuse in QuO applications. Qoskets are used to bundle in one place all of the specifications and objects for controlling systemic behavior, as illustrated in Figure 4, independent of the application in which the behavior might end up being used, and whether or not the behavior will be used in-band or out-of-band.

Qoskets encapsulate, as reusable components, the following systemic aspects:

- *Adaptation policies* – As expressed in QuO contracts
- *Measurement and control* – As defined by system condition objects and callback objects
- *Adaptive behaviors* – Some of which are partially specified until they are specialized to a functional interface.
- *QoS implementation* – Defined by qosket methods.

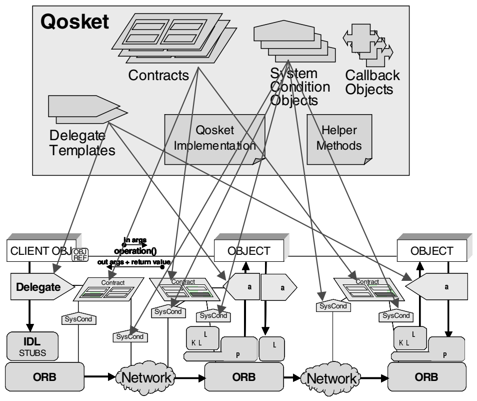


Fig. 4. Qoskets encapsulate QuO objects into reusable behaviors.

For example, the UAV image dissemination application described in Section 5 includes qoskets that help maintain the QoS of image delivery in the face of limited resource availability, e.g., CPU and network. It includes qoskets that encapsulate

- *adaptation policies* to maintain appropriate timeliness and fidelity tradeoffs for the imagery;
- *measurement* of video throughput and network and CPU load and *control* of CPU and network resources;

- *adaptive behaviors* that shape the imagery, reserve and prioritize resources, and migrate functionality;
- *QoS implementation* that provides setup of the qosket objects and helper methods for shaping specific image formats and fragmenting packets to facilitate network management.

3 DRE Application of QoS Adaptive Middleware – Weapons System Open Architecture

The Weapons System Open Architecture (WSOA) program prototyped a dynamic collaborative mission planning capability that could be used in-flight between a command and control (C2) aircraft and a fighter aircraft. As illustrated in Figure 5, the fighter aircraft and the C2 aircraft establish a collaboration to exchange virtual target folders (VTFs), consisting of images and plain text data to update the fighter's mission [3, 14].

WSOA is built upon a layered architecture of domain-specific, QoS adaptive, and distribution middleware; as well as QoS mechanisms, services, and managers. Middleware on the fighter node consists of

- The Bold Stroke avionics middleware [24]
- QuO QoS adaptive middleware
- TAO distribution middleware

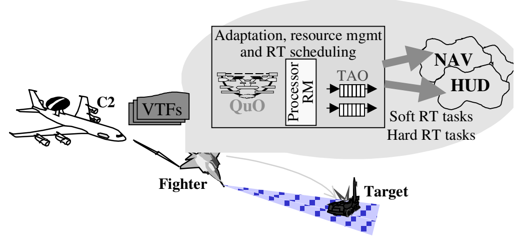


Fig. 5. The C2 aircraft sends VTF data to the fighter aircraft for dynamic mission planning

The C2 node utilizes ORBexpress, a CORBA ORB supporting the C2 legacy software applications written in Ada, and Visibroker. QoS mechanisms, services, and managers in WSOA consist of

- RT-ARM [7], a CPU admission control service, on the fighter node
- Kokyu [6], a real-time scheduling and dispatching service, on the fighter node
- Tactical communication links, using legacy Link 16 incorporated on the C2 and fighter nodes using portable protocols. Link 16 utilizes statically allocated time slots for network allocation.

3.1 QoS Management in WSOA

Processing on the fighter node involves hard and soft real-time tasks. Hard real-time tasks are statically scheduled to assign rates and priorities. WSOA uses QuO, RT-ARM, and Kokyu to schedule soft real-time tasks in the extra CPU availability available for the worst case, but unused in the usual case, and to adapt processing and task rates dynamically to maintain the operation of hard and soft real-time tasks on the fighter's mission computer.

The nature of the Link 16 tactical network means that bandwidth is statically allocated, so dynamic management of the network has to be on the edges, i.e., adjusting the amount and nature of the data being transported. WSOA uses a QuO contract, illustrated in Figure 6, to download imagery as a sequence of tiles of varying quality, starting with a point of interest in the image. The QuO contract monitors download progress and adjusts image tile compression reactively to meet the required image transmission deadline. Image tiles near the point of interest – which are the first to be transmitted – are sent at as low compression as possible to improve image quality, but surrounding images can still be useful at lower resolution. In addition, the QuO contract interfaces with RT-ARM to adjust the rates of execution for the decompression operation.

The QuO contract in the WSOA application performs reactive QoS management of one node only to attempt to achieve the equivalent of end-to-end QoS management. QuO works with RT-ARM and Kokyu to manage the scheduling of CPU resources on the fighter node. However, the static preallocated nature of the Link 16 network and the legacy nature of the C2 software limits the QoS contract to calling data management functionality on the C2 (tiling and compression) and adjusting the data to fit the network characteristics. The next section describes an application of QuO in which we address end-to-end QoS management.

4 DRE Application of QoS Adaptive Middleware – End-to-End QoS in a Distributed UAV Simulation

In another application of QuO middleware, we have developed a prototype unmanned aerial vehicle (UAV) simulation for the US Navy, DARPA, and the US Air Force. As illustrated in Figure 7, this is a DRE system centered on the concept of disseminating sensor information from a set of UAVs to command and control centers for use in time-critical target detection and identification. In this application, we are concerned with developing QuO contracts that manage the end-to-end requirements for (a) the delivery of imagery from UAVs to a command and control (C2)

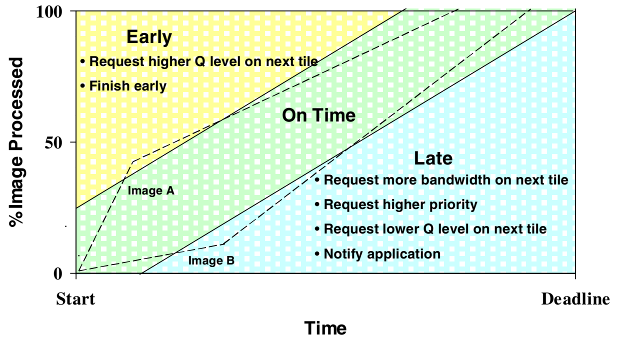


Fig. 6. A QuO contract monitors the progress of image download and adapts to keep it within the "On Time" range

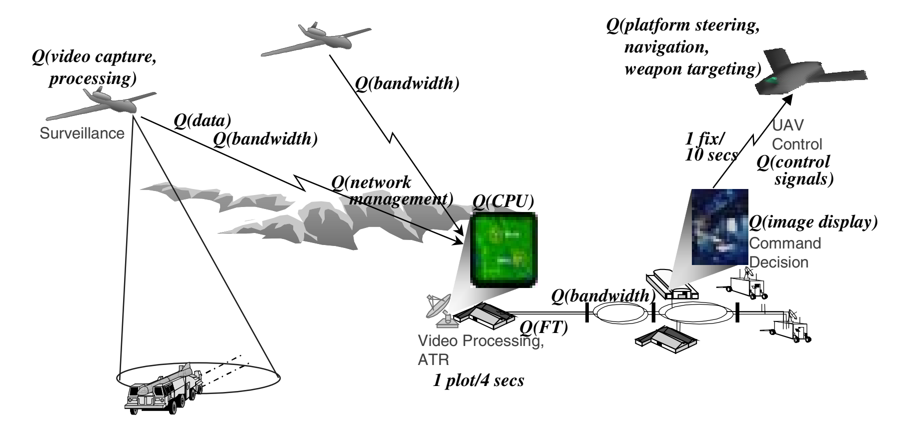


Fig. 7. The prototype UAV image dissemination system requires end-to-end QoS management

platform (e.g., a ground station, air C2 node, or shipboard environment) and (b) the delivery of control signals from the C2 platform back to the UAVs. QoS management includes trading off image quality and timeliness and managing resources from end-to-end, with heterogeneous resources (e.g., tactical radio and wired IP networks), changing mission requirements, and dynamic environmental conditions.

Figure 8 illustrates the architecture of the prototype. It is a three-stage pipeline, with simulated UAVs or live UAV surrogates (such as airships with mounted cameras) sending imagery to processes (*distributors*), which distribute the imagery to the simulated C2 stations, which display, collect, or process the imagery (e.g., with an automated target recognition process). Various versions of the UAV image dissemination prototype have been used for evaluation within the US Navy's HiPER-D platform, as a basis for US Army and US Air Force prototype systems, and as an open experimental platform for DARPA's Program Composition of Embedded Systems (PCES) program [5].

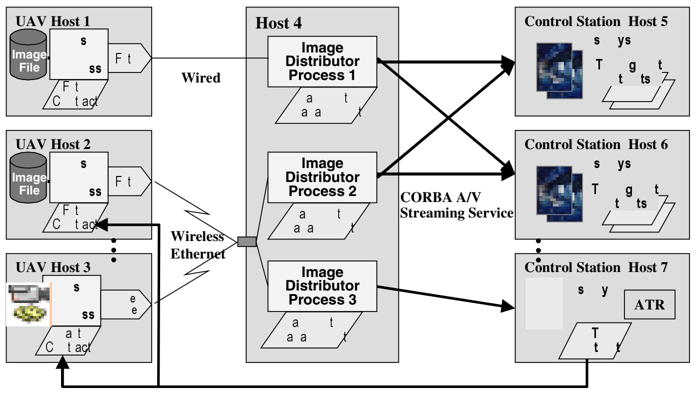


Fig. 8. Architecture of the UAV Prototype

The current UAV prototype uses the QuO middleware to integrate several QoS management strategies, services, and mechanisms:

- End-to-end priority-based resource management, using RTCORBA and scheduling services for CPU resource management and Differentiated Services (DiffServ) [8] for network priority management.
- End-to-end reservation-based resource management, using the CPU reservation capabilities of Timesys Linux and RSVP network reservation [27].
- Application-level management, including data management strategies (e.g., filtering, tiling, compression, scaling, and rate shaping) and process migration.

The UAV image dissemination prototype employs QoS management all along the pipeline, with CPU management at each node, network management at each link, and in-band and out-of-band application QoS management at several locations. Preliminary results of experiments for end-to-end QoS management capabilities are described in [21]. This work is ongoing as part of the PCES program; more detail is available in [1, 9, 10, 21].

5 Next Steps in Research of QoS Adaptive Middleware for DRE Systems

We have made great strides in the development of QoS adaptive middleware technology and applying it to the problems of DRE systems. However, there is still much research to be done. Three areas in which we are currently building upon our QuO research to tackle larger, more ambitious problems of adaptive DRE systems are the following:

- Higher level abstraction and specification of large-scale, system-wide adaptation strategies with synthesis of middleware code
- Control-based adaptation strategies to replace and complement reactive based strategies
- Component-based middleware and applications as a basis for constructing DRE systems by composing reusable pieces.

The following sections provide a preview of each of these.

5.1 Modeling of QoS Adaptation Strategies

QoS aspects define a space, illustrated in Figure 9, in which an application has unacceptable quality of service attributes if the measure of any dimension of interest falls below the minimal operating threshold. Above a maximum operating threshold, improvements in QoS make no difference in application operation. Between the minimum and maximum thresholds is a space in which tradeoffs and adaptations can be made in order to maintain as high a level as feasible of acceptable quality of service. While Figure 9 illustrates three dimensions – corresponding to the dimensions of amount of data, fidelity, and latency – the space can consist of any number of dimensions, 1, 2, … N, corresponding to the QoS dimensions relevant to the application.

While the minimum and maximum thresholds define an acceptable operating space, it is usually not the case that every point in the acceptable space is equivalent. Nor is it true that it is always possible to move smoothly from one point in the space to another. The underlying system, mechanisms, and resource managers provide *knobs* to control the level of QoS (e.g., through resource allocation, rate or priority adjustment, or data shaping). The granularity at which QoS can be adapted depends on these knobs. Furthermore, adaptation of one QoS dimension will frequently affect other QoS dimensions (e.g., increasing the amount of data will use more bandwidth). Finally, the application's requirements will often mean that some tradeoffs and adaptations are preferred over others.

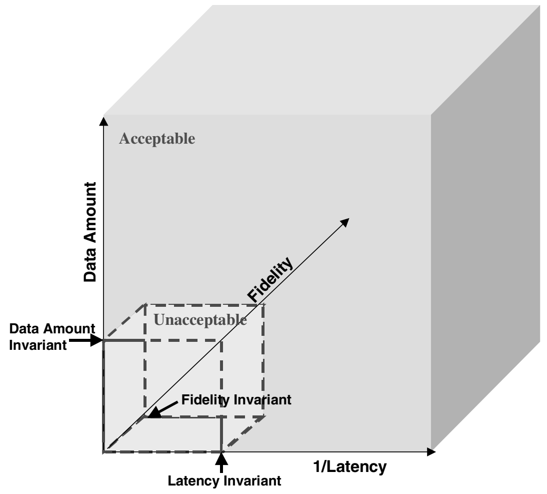


Fig. 9. QoS aspects define an operating space. When QoS is too low performance of the application is unacceptable. Adaptation

Under the auspices of the DARPA MoBIES program, we are applying model integrated computing (MIC) to the problems of designing, customizing, parameterizing, and managing the adaptive runtime characteristics of DRE applications. We are developing a semantically rich, domain-specific modeling language using the Generic Modeling Environment (GME) [11] supporting the high-level design of QoS adaptation strategies and designing and implementing mappings and code generators to derive runtime interfaces and middleware constructs from the high-level representation.

The main goal of our research is modeling *Adaptation Strategy Preferences*, which indicate how the adaptation moves the application through the acceptable operating space as system, functional, and mission conditions change. These preferences specify which adaptation behaviors and mechanisms are employed, the manner and order in which they are employed, the tradeoffs to be made, and the conditions that trigger adaptation. To this end, our modeling language supports the following elements of a QoS adaptive runtime system:

- *The application's structure* – Primarily the functional structure of the application, including data and control flow and points for inserting adaptive decisions.
- *Mission requirements* – The functional and QoS goals that must be met by the application. These help determine the relative merit of possible adaptations and points in the adaptation space.
- *Minimum and maximum acceptable ranges of operation* – This includes the lower bound on the level of acceptable QoS below which the system is unacceptable for a

given mission and the upper bound above which additional resources lead to no improvement in system QoS.

- *Observable parameters* – The system conditions that need to be monitored at runtime in order to determine the system's QoS state and drive adaptation. Examples include latency, throughput, and bandwidth between a pair of hosts, as well as reflective information about application execution, such as the nature of data content or the speed of operation. These parameters determine the application's current position in the adaptation space.
- *Controllable parameters and adaptation behaviors* – The knobs available to the application for QoS control and adaptation. These can be in the form of interfaces to mechanisms, managers or resources, or in the form of packaged adaptations, such as those provided by QuO's Qoskets.
- *The system dynamics* – The interactions between observable and controllable parameters and adaptation behaviors. These help define the set of possible trajectories that an application can take through the N-dimensional QoS parameter space.
- *Adaptation strategies* – The adaptation strategy specifies the adaptations employed and the tradeoffs made in response to dynamic system conditions in order to maintain an acceptable mission posture.

5.2 Control-Based Adaptation Strategies

Also as part of our MoBIES research, we are investigating the use of control theory based strategies for designing the adaptation strategies. Many of our existing applications of QuO in DRE systems have been reactive, gracefully degrading in the presence of limited resource availability or overload situations. Others have been more proactive, such as reserving resources for critical operations or information delivery, but have often been based upon local information. Control theory offers a promising avenue to build systems that are predictable and proactive. The adaptation strategy would be modeled as a controller and instantiated as a set of deterministic contracts that lead the system through a controlled set of adaptive behaviors.

We expect that large DRE systems can benefit from a hierarchy of controllers, in which the outermost tries to maximize a measure of *utility* based upon the overall mission requirements and system dynamics. We have identified and defined two types of controllers that we expect to be useful for modeling controlled behavior and generating predictable QoS adaptive middleware software: *supervisory controllers* and *classical compensators*. Supervisory controllers are represented by finite state machines, readily modeled in GME and then translated into QuO contracts. We will model classical compensators as difference/differential equations, from which we will generate functions, contained in qoskets and referenced by contracts.

This research will ease the programming of controlled behavior, a crucial piece of traditional deterministic embedded systems. It will not be intertwined with the functional code, but will be developed separately in middleware, facilitating reuse, maintenance, and changing control policies.

5.3 Towards QoS Adaptive Component Architectures

Component middleware is becoming prevalent in the development of large-scale distributed applications, especially enterprise applications. However, conventional component middleware is not yet well suited for DRE systems because it is still lacking much of the type of QoS management support that we have discussed in this paper. In joint activities with Washington University, St. Louis, and Vanderbilt University, we are enhancing the standard CORBA Component Model (CCM) with static QoS provisioning (a static QoS implementation of CCM in TAO called CIAO) and dynamic adaptation behaviors based on QuO.

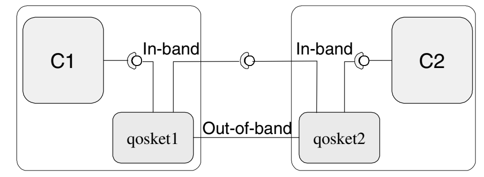


Fig. 10. Qosket design-time code will be distributed as qosket runtime components that operate in-band or out-of-band

In the object form of QuO described in Section 2, design-time Qosket elements (such as contracts, system condition objects, and delegates) are distributed among the objects in the system. In the component version of QuO that we are currently developing, design-time Qosket elements will be encapsulated into distributed run-time qosket components, as illustrated in Figure 10. As in the object version of QuO, qosket components can operate *in-band* between functional component interactions or *out-of-band* between the functional components and the containers, platform, and resources that comprise the environment.

Developing dynamic QoS support in CCM, and thereby component support in QuO, offers several advantages over object-based architectures. Whereas the Qosket model already provides encapsulation of reusable design-time code representing a single logical behavior, moving to CCM also encapsulates the collocated runtime elements comprising behaviors. This makes the runtime elements more reusable, while other CCM elements, such as *containers* and *homes*, support placement and creation, respectively, of the runtime elements.

The CCM life-cycle definition shows promise for supporting construction by composition and late binding of QoS concerns. CCM's *assembly* stage promises tool-based support for hooking up components, both functional and QoS – something that is currently written by hand in the Qosket code. CCM's *deployment* stage offers a time at which target platform information will be known, so that flexible systems can be constructed, with many resource, host, and network specific details left unspecified until deployment time. QuO adds the *runtime* stage to the CCM lifecycle, by providing the capabilities to defer decisions and information binding until runtime.

Dynamic QoS-enabled component middleware, utilizing QuO and CIAO, shows great promise in moving toward constructing robust DRE systems with predictable, controlled behavior by composition of reusable components. More detail about this work is available in [26].

6 Conclusions

Embedded applications are becoming increasingly networked and distributed. These DRE applications are becoming more and more infeasible to develop in the custom-programmed, stove-piped manner in which embedded systems have historically been developed. This has led to a trend of developing DRE systems using layers of middleware – a trend already adopted elsewhere, such as in Internet and enterprise applications, for similar reasons. However, DRE systems have stricter QoS concerns that are not completely supported by existing domain-specific and distribution middleware. Thus, an important layer for DRE systems is a layer that can support QoS control, which cross-cuts multiple layers in applications, requires end-to-end enforcement, and involves trading off competing QoS requirements among distributed components. Furthermore, because of the dynamic, unpredictable, and hostile nature of many of the environments in which DRE systems are deployed, and the presence of changing mission modes, requirements, and priorities, static QoS policies and allocation are not sufficient. DRE systems need support for dynamic QoS control and adaptation.

BBN's Quality Objects middleware supports the separate programming of QoS aspects and supports the development of systems that are less brittle and more adaptive in the face of constrained, dynamic conditions. The QuO approach to dynamic QoS management and adaptation has been evaluated and validated in several DRE military applications. As we focus the QuO research on the development of larger-scale DRE systems, it is important to focus on higher-level abstractions and modeling of system-wide behaviors; control based adaptive behaviors; component abstractions to support reuse, encapsulation, and construction by composition; and developing tools to support a more standard methodology for effectively fielding systems with these attributes.

QuO software is available in open-source format at quo.bbn.com.

Acknowledgements. The author would like to acknowledge the contributions of the other members of the Distributed Systems Technology group at BBN Technologies for their contributions to the research described in this paper: Rick Schantz, John Zinky, Partha Pal, Rich Shapiro, Michael Atighetchi, Gary Duzan, Craig Rodrigues, Paul Rubel, Chris Jones, Praveen Sharma, and Prakash Manghwani. The author would also like to acknowledge the collaborations with Vanderbilt University, Boeing, Washington University, St. Louis, and many other researchers.

This work was supported by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory under contracts F30602-98-C-0187, F33615-00-C-1694, and F33615-02-C-4037.

References

1. BBN Technologies, http://www.dist-systems.bbn.com/projects/AIRES/UAV
2. A. Burns, A. Wellings. *Real-Time Systems and Programming Languages, 3rd Edition.* Addison Wesley Longmain, 2001.

3. D. Corman,, “WSOA–Weapon Systems Open Architecture Demonstration–Using Emerging Open System Architecture Standards to Enable Innovative Techniques for Time Critical Target (TCT) Prosecution,” *20th Digital Avionics Systems Conference (DASC)*, Daytona Beach, Florida, October 2001.
4. M. Cukier, J. Ren, C. Sabnis, D. Henke, J. Pistole, W. Sanders, D. Bakken, M. Berman, D. Karr, R. Schantz, “AQuA: An Adaptive Architecture that Provides Dependable Distributed Objects ”, *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems*, pages 245–253, October 1998.
5. DARPA, http://dtsn.darpa.mil/ixo/programdetail.asp?progid=37
6. C. Gill, D. Levine, and D. Schmidt, "The Design and Performance of a Real-Time CORBA Scheduling Service", *Real-Time Systems, The International Journal of Time-Critical Computing Systems, special issue on Real-Time Middleware*, Kluwer, 2001.
7. J. Huang, R. Jha, W. Heimerdinger, M. Muhammad, S. Lauzac, B. Kannikeswaran, K. Schwan, W. Zhao, R. Bettati, “RT-ARM: A Real-Time Adaptive Resource Management System for Distributed Mission-Critical Applications,” *Workshop on Middleware for Distributed Real-Time Systems, RTSS–97*, San Francisco, California, 1997.
8. IETF, An Architecture for Differentiated Services, http://www.ietf.org/rfc/rfc2475.txt
9. D. Karr, C. Rodrigues, J. Loyall, R. Schantz, Y. Krishnamurthy, I. Pyarali, D. Schmidt, “Application of the QuO Quality-of-Service Framework to a Distributed Video Application,” *Proceedings of the International Symposium on Distributed Objects and Applications*, Rome, Italy, 18–20 September 2001.
10. D. Karr, C. Rodrigues, J. Loyall, R. Schantz, “Controlling Quality-of-Service in a Distributed Video Application by an Adaptive Middleware Framework,” *Proceedings of ACM Multimedia 2001*, Ottawa, Ontario, Canada, 30 September – 5 October 2001.
11. A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason, G. Nordstrom, J. Sprinkle, P. Volgyesi. “The Generic Modeling Environment,” WISP'2001, May 2001, Budapest, Hungary.
12. J. Loyall, D. Bakken, R. Schantz, J. Zinky, D. Karr, R. Vanegas, K. Anderson, “QoS Aspect Languages and Their Runtime Integration,” *Lecture Notes in Computer Science*, 1511, Springer-Verlag. *Proceedings of the Fourth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers (LCR98)*, Pittsburgh, Pennsylvania, 28–30 May 1998.
13. J. Loyall, R. Schantz, J. Zinky, D. Bakken, “Specifying and Measuring Quality of Service in Distributed Object Systems”, *Proceedings of The 1st IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC 98)*, 1998.
14. J. Loyall, J. Gossett, C. Gill, R. Schantz, J. Zinky, P. Pal, R. Shapiro, C. Rodrigues, M. Atighetchi, D. Karr, “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)*, Phoenix, Arizona, 16–19 April 2001.
15. Object Management Group, Fault Tolerant CORBA Specification, OMG Document orbos/99–12–08, December 1999.
16. Object Management Group, Minimum CORBA – Joint Revised Submission, OMG Document orbos/98–08–04, August 1998.
17. OMG Real-Time and Embedded Distributed Object Systems Workshop, July 15–18, 2002, Arlington, Virginia.
18. Object Management Group, Real-Time CORBA 2.0: Dynamic Scheduling Specification, OMG Final Adopted Specification, *September 2001*, http://cgi.omg.org/docs/ptc/01–08–34.pdf.
19. R. Schantz, J. Zinky, D. Karr, D. Bakken, J. Megquier, J. Loyall, “An Object-level Gateway Supporting Integrated-Property Quality of Service”, *Proceedings of The 2nd IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC 99)*, May 1999.

20. R. Schantz, J. Loyall, M. Atighetchi, Partha Pal, "Packaging Quality of Service Control Behaviors for Reuse, *Proceedings of the 5th IEEE International Symposium on Object-Oriented distributed Computing, (ISORC 02)*, Washington DC, April 29–May1 2002.
21. R. Schantz, J. Loyall, C. Rodrigues, D. Schmidt, Y. Krishnamurthy, I. Pyarali, "Flexible and Adaptive QoS Control for Distributed Real-time and Embedded Middleware," *Proceedings of the ACM/IFIP/USENIX International Middleware Conference*, Rio de Janeiro, Brazil, June 2003.
22. D. Schmidt, "ACE: an Object-Oriented Framework for Developing Distributed Applications," *Proceedings of the 6th USENIX C++ Technical Conference*, April 1994.
23. D. Schmidt, D.Levine, and S. Mungee, "The Design and Performance of Real-Time Object Request Brokers," Computer Communications, April 1998.
24. D. Sharp, "Reducing Avionics Software Cost Through Component Based Product Line Development," *Proceedings of the 10th Annual Software Technology Conference*, 1998.
25. R. Vanegas, J. Zinky, J. Loyall, D. Karr, R. Schantz, D. Bakken, "QuO's Runtime Support for Quality of Service in Distributed Objects," *Proceedings of Middleware 98, the IFIP International Conference on Distributed Systems Platform and Open Distributed Processing*, September 1998.
26. N. Wang, C. Gill, D. Schmidt, A. Gokhale, B. Natarajan, J. Loyall, R. Schantz, C. Rodrigues, "QoS-Enabled Middleware," *Middleware for Communications*, Qusay H. Mahmoud ed., John Wiley & Sons, Ltd, 2003.
27. L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala, "RSVP: A New Resource ReSerVation Protocol," *IEEE Network*, September 1993.

Heterogeneous Reactive Systems Modeling and Correct-by-Construction Deployment*

Albert Benveniste[1], Luca P. Carloni[3], Paul Caspi[2], and Alberto L. Sangiovanni-Vincentelli[3]

[1] Irisa/Inria, Campus de Beaulieu, 35042 Rennes cedex, France,
Albert.Benveniste@irisa.fr
http://www.irisa.fr/sigma2/benveniste/

[2] Verimag, Centre Equation, 2, rue de Vignate, F-38610 Gieres,
Paul.Caspi@imag.fr
http://www.imag.fr/VERIMAG/PEOPLE/Paul.Caspi

[3] U.C. Berkeley, Berkeley, CA 94720,
{lcarloni,alberto}@eecs.berkeley.edu
http://www-cad.eecs.berkeley.edu/HomePages/{lcarloni,alberto}

Abstract. We propose a mathematical framework to deal with the composition of heterogeneous reactive systems. Our theory allows to establish theorems, from which design techniques can be derived. We illustrate this by two cases: the deployment of synchronous designs over GALS architectures, and the deployment of synchronous designs over the so-called Loosely Time-Triggered Architectures.

1 Introduction

The *notion of time* has been a crucial aspect of electronic system design for years. Dealing with concurrency, time and causality has become increasingly difficult as the complexity of the design grows. The *synchronous programming model* has had major successes at the specification level because it provides a simpler way to access the power of concurrency in functional specification. Synchronous Languages like ESTEREL [8], LUSTRE [14], and SIGNAL [19], the STATECHARTS modeling methodology [15], and design environments like SIMULINK/STATEFLOW [23] all benefit from the simplicity of the *synchronous assumption*, i.e.: (1) the system evolves through an infinite sequence of successive atomic reactions indexed by a *global logical clock*, (2) during a reaction each component computes new events for all its output signals based on the presence/absence of events computed in the previous reaction and, (3) the communication of events among components occur *instantaneously* between two successive reactions.

However, if the synchronous assumption simplifies system specification, the problem of deriving a correct physical implementation from it does remain. In

* This research was supported in part by the European Commission under the projects COLUMBUS, IST-2002-38314, and ARTIST, IST-2001-34820, by the NSF under the project ITR (CCR-0225610), and by the GSRC.

particular, difficulties arise when the target architecture for the embedded system has a distributed nature that badly matches the synchronous assumption because of large variance in computation and communication times and because of the difficulty of maintaining a global notion of time. This is increasingly the case for many important classes of embedded applications in avionics, industrial plants, and the automotive industry. Here, multiple processing elements operating at different clock frequencies are distributed on an extended area and connected via communication media such as busses (e.g., CAN for automotive applications, ARINC for avionics, and Ethernet for industrial automation) or serial links. Busses and serial links can, however, be carefully designed to comply with a notion of global synchronization as the family of Time-Triggered Architectures (TTA), introduced and promoted by H. Kopetz [17], testifies. A synchronous implementations must be conservative, forcing the clock to run as slow as the slowest computation/communication process (*worst-case approach*). The overhead implied by time-triggered architectures and synchronous implementations is often enough to convince designers to use asynchronous communication architectures such as the ones implemented by the busses mentioned above.

We argue that imposing an "homogeneous" design policy, such as the fully synchronous approach, on complex designs will be increasingly difficult. Heterogeneity will manifest itself at the component level where different models of computation may be used to represent component operation and, more frequently, at different levels of abstraction, where, for example, a synchronous-language specification of the design may be refined into a globally asynchronous locally synchronous (GALS) architecture.

In this paper, we provide a mathematical framework for the heterogeneous modeling of reactive and real-time systems to allow freedom of choice between different synchronization policies at different stages of the design. The focus of our framework is handling communication and coordination among heterogeneous processes in a mathematically sound way. Interesting work along similar lines has been the Ptolemy project [13,22], the MoBIES project [1], the Model-Integrated Computing (MIC) framework [16], and *Interface Theories* [12].

Our main contributions are a mathematical model for heterogeneous system built as a variation of the "Tagged-Signal Model" of Lee and Sangiovanni-Vincentelli [21] (in this paper, called the *LSV model*) and a set of theorems that support effective techniques to generate automatically correct-by-construction adaptors between designs formulated using different design policies. We illustrate these concepts with two applications that are of particular relevance for the design of embedded systems: the deployment of a synchronous design over a GALS architecture and over a so-called Loosely Time-Triggered Architecture (LTTA) [7]. The idea followed in these examples is to abstract away from the synchronous specifications the constraints among events of different signals due to the synchronous paradigm and, then, to map the "unconstrained" design into a different architecture characterized by a novel set of constraints among events. In doing so, we must make sure that, when we remap the design, the intended "behaviour" of the system is retained. To do so we introduce a formal notion of *semantic preserving* transformation. The constraints on coordination among processes are captured by using the "tags" in the LSV model and the transformations are handled with morphisms among tag sets. For more details, the reader is referred to the extended version [5] of this paper.

2 Tagged Systems and Heterogeneous Systems

In this section, we build on the Lee and Sangiovanni-Vincentelli (LSV) "Tagged-Signal Model" [21]. For reasons that will be clear in the sequel, we slightly deviate from the original LSV model.

2.1 Tagged Systems

Symbol $\mathbf{N}$ denotes the set of positive integers. $X \mapsto Y$ denotes the set of all maps having X as domain, and whose range is contained in Y. Also, if $(X, \leq_X)$ and $(Y, \leq_Y)$ are partial orders, a map $f \in X \mapsto Y$ is called *nondecreasing* if

$$\forall x, x' \in X : x \leq_X x' \Rightarrow \neg[f(x') <_Y f(x)] \tag{1}$$

Condition (1) expresses that a nondecreasing map cannot invert orders. However, it can add or remove some orders. Thus, nondecreasing is weaker than increasing.

Definitions. We assume an underlying partially ordered set $\mathcal{T}$ of *tags,* we denote by $\leq$ the partial oder on $\mathcal{T}$, and we write $\tau < \tau'$ iff $\tau \leq \tau'$ and $\tau \neq \tau'$. A *clock* is a nondecreasing map $h \in \mathbf{N} \mapsto \mathcal{T}$. Assume an underlying set $\mathcal{V}$ of variables, with domain D. For $V \subset \mathcal{V}$ finite, a *V-behaviour,* or simply behaviour, is an element:

$$\sigma \ \in \ V \mapsto \mathbf{N} \mapsto (\mathcal{T} \times D), \tag{2}$$

meaning that, for each $v \in V$, the n-th occurrence of v in behaviour σ has tag $\tau \in \mathcal{T}$ and value $x \in D$. For v a variable, the map $\sigma(v) \in \mathbf{N} \mapsto (\mathcal{T} \times D)$ is called a *signal.* For σ a behaviour, an *event* of σ is a tuple $(v, n, \tau, x) \in V \times \mathbf{N} \times \mathcal{T} \times D$ such that $\sigma(v)(n) = (\tau, x)$; thus we can regard behaviours as sets of events. We require that, for each $v \in V$, the 1st projection of the map $\sigma(v)$ (it is a map $\mathbf{N} \mapsto \mathcal{T}$) is nondecreasing. Thus it is a clock, we call it the *clock of v in σ.*

A *tagged system* is a triple $P = (V, \mathcal{T}, \Sigma)$, where V is a finite set of variables, $\mathcal{T}$ is a tag set, and Σ a set of V-behaviours. If $\mathcal{T}_1 = \mathcal{T}_2 =_{\text{def}} \mathcal{T}$, the *parallel composition* of systems P_1 and P_2 is by intersection:

$$\begin{array}{r} P_1 \parallel P_2 =_{\text{def}} (V_1 \cup V_2, \mathcal{T}, \Sigma_1 \wedge \Sigma_2), \text{ where} \\ \Sigma_1 \wedge \Sigma_2 =_{\text{def}} \left\{\sigma \,\middle|\, \sigma_{|V_i} \in \Sigma_i, i = 1, 2\right\}, \end{array} \tag{3}$$

and $\sigma_{|W}$ denotes the restriction of σ to a subset W of variables. The set $\mathcal{T}$ of tags can be adjusted to account for different classes of systems.

Synchronous Systems. To represent *synchronous systems* with our model, take for $\mathcal{T}$ a totally ordered set, and require that all clocks are *strictly increasing.* The tag index set $\mathcal{T}$ organizes behaviours into successive reactions, as explained next. Call *reaction* a maximal set of events of σ with identical τ. Since clocks are strictly increasing, no two events of the same reaction can have the same variable. Regard a behaviour as a sequence of global reactions: $\sigma = \sigma_1, \sigma_2, \ldots$, with tags $\tau_1, \tau_2, \ldots \in \mathcal{T}$. Thus $\mathcal{T}$ provides a global, logical time basis. Particular instances for $\mathcal{T}$ correspond to different views of synchronous systems:

– Taking $\mathcal{T} = \mathbf{N}$ means that we assume some basic logical clock (the identity map, from $\mathbf{N}$ to $\mathbf{N}$), which is global to all considered systems, and all clocks are sub-clocks of this basic one. This is a good model for closed systems.
– Now, take for $\mathcal{T}$ a totally ordered dense set—e.g., $\mathcal{T} = \mathbf{R}$, the set of real numbers, or $\mathbf{Q}$, the set of rational numbers. Then, for any given clock h, there exists another clock k whose range has empty intersection with the range of h. This models the fact that, for any given system, there exists another system that is working while the former is sleeping; this is a suitable model for open systems. In fact, adequate models for open systems are stuttering invariant systems we define next. Call *time change* any bijective and strictly increasing function $\rho : \mathcal{T} \mapsto \mathcal{T}$, and denote by $\mathsf{R}_{\mathcal{T}}$ the set of all time changes over $\mathcal{T}$. Then a synchronous system $P = (V, \mathcal{T}, \Sigma)$ is called *stuttering invariant* iff it is invariant under time change, i.e., for every behaviour $\sigma \in \Sigma$ and every time change $\rho \in \mathsf{R}_{\mathcal{T}}$, then $\sigma^\rho \in \Sigma$ holds, where

$$(v, n, \tau, x) \in \sigma^\rho \;\; \Leftrightarrow_{\text{def}} \;\; \sigma(v, n, \rho(\tau), x) \in \sigma. \tag{4}$$

Examples of stuttering invariant systems are the stallable processes of latency-insensitive design [9], where $\mathcal{T} = \mathbf{N}$.

Timed Synchronous Systems. Timed synchronous systems are synchronous systems in which physical time is available, in addition to the ordering of successive reactions. Note that events belonging to the same reaction may occur at different physical instants. For this case we take $\mathcal{T} = \mathcal{T}_{\text{synch}} \times \mathcal{T}_\varphi$, where $\mathcal{T}_{\text{synch}}$ indexes the reactions, and $\mathcal{T}_\varphi$ is the physical time basis (e.g., $\mathcal{T}_\varphi = \mathbf{R}$ for real-time).

Asynchronous Systems. The notion of asynchronicity is vague. Any system that is not synchronous could be called asynchronous, but we often want to restrict somewhat this notion to capture particular characteristics of a model. In this paper, we take a very liberal interpretation for an asynchronous system. If we interpret a tag set as a constraint on the coordination of different signals of a system and the integer $n \in \mathbf{N}$ as the basic constraint of the sequence of events of the behaviour of a variable, then the most "coordination unconstrained" system, the one with most degrees of freedom in terms of choice of coordination mechanism, could be considered an ideal asynchronous system. Then an asynchronous system corresponds to a model where the tag set does not give any information on the absolute or relative ordering of events. In more formal way, take $\mathcal{T} = \{.\}$, the trivial set consisting of a single element. Then, behaviours identify with elements $\sigma \in V \mapsto \mathbf{N} \mapsto D$.

Running example. This simple example will be used throughout the rest of the paper to illustrate our results and their implications. Let P and Q be two synchronous systems involving the same set of variables: b of type boolean, and x of type integer. Each system possesses only a single behaviour, shown on the right hand side of $P : \ldots$ and $Q : \ldots$, respectively. Each behaviour consists of a sequence of successive reactions, separated by vertical bars. Events sitting in the same reaction can be seen *aligned.* Each reaction consists of an assignment of values to a subset of the variables; a blank indicates the absence of the considered variable in the considered reaction.

$$P : \begin{array}{r|c|c|c|c|c|c|l} \hline b : & t & f & t & f & t & f & \ldots \\ x : & 1 & & 1 & & 1 & & \ldots \\ \hline \end{array} \;, \quad Q : \begin{array}{r|c|c|c|c|c|c|l} \hline b : & t & f & t & f & t & f & \ldots \\ x : & & 1 & & 1 & & 1 & \ldots \\ \hline \end{array}$$

One of the questions addressed in this paper is how to deploy a synchronous design on a less constrained architecture. The general strategy we follow is to eliminate first the constraints introduced by the synchronous assumption and then to map the resulting asynchronous system on a less constrained architecture that may range from asynchronous to relaxed versions of timed-triggered architectures. The *desynchronization* of a synchronous system like P or Q consists in (1) removing the synchronization barriers separating the successive reactions, and, then, (2) compressing the sequence of values for each variable, individually. This yields:

$$P_\alpha = Q_\alpha : \begin{array}{l} \hline b : t \; f \; t \; f \; t \; f \; \ldots \\ \hline x : 1 \; 1 \; 1 \; . \; . \; . \\ \hline \end{array}$$

where the subscript $_\alpha$ refers to asynchrony. The reader may think that events having identical index for different variables are aligned, but this is not the case. In fact, as the absence of vertical bars in the diagram suggests, there is *no alignment* at all between events associated with different variables.

Regarding desynchronization, the following comments are in order. Note that $P \neq Q$ but $P_\alpha = Q_\alpha$. Next, the synchronous system R defined by $R = P \cup Q$, the nondeterministic choice between P and Q, possesses two behaviours. However, its desynchronization R_α equals P_α, and possesses only one behaviour.

Now, we use the proposed framework to derive formal models for P, Q, and P_α. For the synchronous systems P and Q, we take $\mathcal{T} = \mathbf{N}$ to index the successive reactions. P possesses a single behaviour (note the absence of x at tag $2n$):

$$\begin{array}{lll} \sigma(b)(2n-1) = (2n-1, t) & , & \sigma(b)(2n) = (2n, f) \\ \sigma(x)(n) \quad\;\; = (2n-1, 1) & & \end{array} \tag{5}$$

For Q, we have (note the difference):

$$\begin{array}{lll} \sigma(b)(2n-1) = (2n-1, t) & , & \sigma(b)(2n) = (2n, f) \\ & & \sigma(x)(n) \;\; = (2n, 1) \end{array} \tag{6}$$

For the asynchronous systems $P_\alpha = Q_\alpha$, we take $\mathcal{T} = \{.\}$, the trivial set with a single element. The reason is that we do not need any additional time stamping information. Thus, $P_\alpha = Q_\alpha$ possess a single behaviour:

$$\sigma_\alpha(b)(2n-1) = t, \sigma_\alpha(b)(2n) = f, \text{ and } \sigma_\alpha(x)(n) = 1. \tag{7}$$

Discussion. A proactive reader would immediately criticize the definition (2) of behaviours. Our definition uses two indexing mechanisms, namely $\mathbf{N}$, to order events in each individual signal, and $\mathcal{T}$, to order (time stamp) events globally across signal boundaries. Of course, since the events of a signal are totally ordered by their "time stamps", their local index n results redundant. The redundancy of the indexing is particularly visible in (5) and (6). The reason for introducing the redundancy is related to the different semantics of the two orders: one is intrinsic in the very notion of events of a signal, the other is related to the constraints on event ordering due to the synchronous assumption. Decoupling the two orders allows us to represent cleanly the desynchronization operation and the deployment on more general architectures. We refer to [5] for a more detail discussion with respect to the original LSV model.

2.2 Heterogeneous Systems

Assume a functional system specification using a synchronous approach, for subsequent deployment over a distributed asynchronous architecture (synchronous and asynchronous are considered in the sense of subsection 2.1). When we deploy the design on a different architecture, we must make sure that the original intent of the designer is maintained. This step is non trivial because the information on what is considered correct behaviour is captured in the synchronous specifications that we want to relax in the first place. We introduce the notion of semantic-preserving transformation to identify precisely what is a correct deployment. We present the idea with our running example:

Running example, cont'd. The synchronous parallel composition of P and Q, defined by intersection: $P \parallel Q =_{\text{def}} P \cap Q$, is empty. The reason is that P and Q disagree on where to put absences for the variable x. On the other hand, since $P_\alpha = Q_\alpha$, then $P_\alpha \parallel Q_\alpha =_{\text{def}} P_\alpha \cap Q_\alpha = P_\alpha = Q_\alpha \neq \emptyset$. Thus, for the pair (P, Q), desynchronization does not preserve the semantics of parallel composition, in any reasonable sense. $\diamond$

How to model that semantics is preserved when replacing the ideal synchronous broadcast by the actual asynchronous communication? In the case of deployment using the LTTA-protocol, we face the same issues, but with the occurrence of time as an additional component of tags. In fact, an elegant solution was proposed by Le Guernic and Talpin for the former GALS case [20]. We cast their approach in the framework of tagged systems and we generalize it.

Morphisms. For $\mathcal{T}, \mathcal{T}'$ two tag sets, call *morphism* a map $\rho : \mathcal{T} \mapsto \mathcal{T}'$ which is nondecreasing and surjective[1]. For $\rho : \mathcal{T} \mapsto \mathcal{T}'$ a morphism, and $\sigma \in V \mapsto \mathbf{N} \mapsto (\mathcal{T} \times D)$ a behaviour, replacing τ by $\rho(\tau)$ in σ defines a new behaviour having $\mathcal{T}'$ as tag set, we denote it by

$$\sigma_\rho, \text{ or by } \sigma \circ \rho. \tag{8}$$

Performing this for every behaviour of a tag system P yields the tag system

$$P_\rho. \tag{9}$$

For $\mathcal{T}_1 \xrightarrow{\rho_1} \mathcal{T} \xleftarrow{\rho_2} \mathcal{T}_2$ two morphisms, define:

$$\mathcal{T}_1 \; {}_{\rho_1}\!\times_{\rho_2} \; \mathcal{T}_2 =_{\text{def}} \{ (\tau_1, \tau_2) \mid \rho_1(\tau_1) = \rho_2(\tau_2) \} . \tag{10}$$

A case of interest is $\mathcal{T}_i = \mathcal{T}_i' \times \mathcal{T}, i = 1, 2$, and the $\mathcal{T}_i'$ are different. Then $\mathcal{T}_1 \; {}_{\rho_1}\!\times_{\rho_2} \; \mathcal{T}_2$ identifies with the product $\mathcal{T}_1' \times \mathcal{T} \times \mathcal{T}_2'$.

Desynchronization. For example, the *desynchronization* of synchronous systems is captured by the morphism $\alpha : \mathcal{T}_{\text{synch}} \mapsto \{.\}$, which erases all global timing information (see Equations (5,6), and (7)).

Heterogeneous Parallel Composition. In this subsection we define the composition of two tagged systems $P_i = (V_i, \mathcal{T}_i, \Sigma_i), i = 1, 2$, when $\mathcal{T}_1 \neq \mathcal{T}_2$. This definition is provided in two stages. For the first stage, we assume that $\mathcal{T}_1 = \mathcal{T}_2$. For σ_i a behaviour of P_i, say that (σ_1, σ_2) is a *unifiable* pair, written

$$\sigma_1 \bowtie \sigma_2 \text{ iff } \sigma_{1|V_1 \cap V_2} = \sigma_{2|V_1 \cap V_2}.$$

For $\sigma_1 \bowtie \sigma_2$, the *unification* of σ_1 and σ_2 is the behaviour $\sigma_1 \sqcup \sigma_2$ having $V_1 \cup V_2$ as set of variables, and such that:

$$(\sigma_1 \sqcup \sigma_2)_{|V_i} = \sigma_i, i = 1, 2.$$

Now, return to the case $\mathcal{T}_1 \neq \mathcal{T}_2$, and assume two morphisms $\mathcal{T}_1 \xrightarrow{\rho_1} \mathcal{T} \xleftarrow{\rho_2} \mathcal{T}_2$. Write:

$$\sigma_1 \; {}_{\rho_1}\!\bowtie_{\rho_2} \; \sigma_2 \text{ iff } \sigma_1 \circ \rho_1 \bowtie \sigma_2 \circ \rho_2. \tag{11}$$

For (σ_1, σ_2) a pair satisfying (11), define

$$\sigma_1 \; {}_{\rho_1}\!\sqcup_{\rho_2} \; \sigma_2 \tag{12}$$

as being the set of events $(v, n, (\tau_1, \tau_2), x)$ such that $\rho_1(\tau_1) = \rho_2(\tau_2) =_{\text{def}} \tau$ and (v, n, τ, x) is an event of $\sigma_1 \circ \rho_1 \sqcup \sigma_2 \circ \rho_2$. We are now ready to define the *heterogeneous conjunction* of Σ_1 and Σ_2 by:

$$\Sigma_1 \; {}_{\rho_1}\!\wedge_{\rho_2} \; \Sigma_2 =_{\text{def}} \{ \sigma_1 \; {}_{\rho_1}\!\sqcup_{\rho_2} \; \sigma_2 \mid \sigma_1 \; {}_{\rho_1}\!\bowtie_{\rho_2} \; \sigma_2 \} \tag{13}$$

[1] Strictly speaking, these are not morphisms of order structures. We use this word by abuse of terminology.

Finally, the *heterogeneous parallel composition* of P_1 and P_2 is defined by:

$$P_1 \ {}_{(\rho_1}\|_{\rho_2)} \ P_2 = (\, V_1 \cup V_2 \, , \, \mathcal{T}_1 \ {}_{\rho_1}\!\times_{\rho_2} \mathcal{T}_2 \, , \, \Sigma_1 \ {}_{\rho_1}\!\sqcup_{\rho_2} \ \Sigma_2 \,) \, . \tag{14}$$

We simply write ${}_{(\rho_1}\|$ instead of ${}_{(\rho_1}\|_{\rho_2)}$ when ρ_2 is the identity.

GALS and Hybrid Timed/Untimed Systems. To model the interaction of a synchronous system with its asynchronous environment, take the heterogeneous composition $P_{(\alpha}\| \ A$, where $P = (V, \mathcal{T}_{\text{synch}}, \Sigma)$ is a synchronous system, $A = (W, \{.\}, \Sigma')$ is an asynchronous model of the environment, and $\alpha : \mathcal{T}_{\text{synch}} \mapsto \{.\}$ is the trivial morphism, mapping synchrony to asynchrony (hence the special notation).

For GALS, take $\mathcal{T}_1 = \mathcal{T}_2 = \mathcal{T}_{\text{synch}}$, where $\mathcal{T}_{\text{synch}}$ is the tag set of synchronous systems. Then, take $\mathcal{T} = \{.\}$ is the tag set of asynchronous ones. Take $\alpha : \mathcal{T}_{\text{synch}} \mapsto \{.\}$, the trivial morphism. And consider $P_1 \ {}_{(\alpha}\|_{\alpha)} \ P_2$.

For timed/untimed systems, consider $P_{(\rho}\| \ Q$, where $P = (V, \mathcal{T}_{\text{synch}} \times \mathcal{T}_\varphi, \Sigma)$ is a synchronous timed system, $Q = (W, \mathcal{T}_{\text{synch}}, \Sigma')$ is a synchronous but untimed system, and $\rho : \mathcal{T}_{\text{synch}} \times \mathcal{T}_\varphi \mapsto \mathcal{T}_{\text{synch}}$ is the projection morphism.

3 Application to Correct Deployment

In this section we formalize the concept of semantics preserving and present a general result on correct-by-construction deployment.

3.1 Preserving Semantics: Formalization

We are given a pair $P_i = (V_i, \mathcal{T}_i, \Sigma_i), i = 1, 2$, such that $\mathcal{T}_1 = \mathcal{T}_2$, and a pair $\mathcal{T}_1 \xrightarrow{\rho} \mathcal{T} \xleftarrow{\rho} \mathcal{T}_2$ of identical morphisms. We can consider two semantics:

$$\begin{aligned} \textit{The strong semantics} &: \ P_1 \, \| \, P_2 \\ \textit{The weak semantics} &: \ P_1 \ {}_{(\rho}\|_{\rho)} \ P_2 . \end{aligned}$$

We say that ρ is *semantics preserving* with respect to $P_1 \, \| \, P_2$ if

$$P_1 \ {}_{(\rho}\|_{\rho)} \ P_2 \ \equiv \ P_1 \, \| \, P_2 . \tag{15}$$

Running example, cont'd. The reader can check the following as an exercise: $P \, \| \, Q = \emptyset$, and, as we already discussed, $P_\alpha \, \| \, Q_\alpha = P_\alpha$. Now we compute $P \ {}_{(\alpha}\|_{\alpha)} \ Q$. From (12) we get that, using obvious notations, (σ_P, σ_Q) is a pair of behaviours that are unifiable modulo desynchronization, i.e., $\sigma_P \ {}_{\alpha}\!\bowtie_{\alpha} \ \sigma_Q$. Then, unifying these yields the behaviour σ such that:

$$\forall n \in \mathbf{N} : \sigma(b)(n) = ((n, n), v_b) \text{ and } \sigma(x)(n) = ((2n-1, 2n), 1) \tag{16}$$

where $v_b = t$ if n is odd, and $v_b = f$ if n is even. In (16), the expression for $\sigma(b)(n)$ reveals that desynchronizing causes no distortion of logical time for b, since (n, n) attaches the same tag to the two behaviours for unification. On the other hand, the expression for $\sigma(x)(n)$ reveals that desynchronizing actually causes distortion of logical time for x, since $(2n-1, 2n)$ attaches different tags to the two behaviours for unification. Thus $P \,\|\, Q = \emptyset$, but $P \;_{(\alpha}\|_{\alpha)}\; Q$ consists of the single behaviour defined in (16). Hence, $P \;_{(\alpha}\|_{\alpha)}\; Q \not\equiv P \,\|\, Q$ in this case: semantics is not preserved. $\diamond$

3.2 A General Result on Correct Deployment

Here we analyse requirement (15). The following theorem holds (see (9) for the notation P_ρ used in this theorem):

Theorem 1. *The pair (P_1, P_2) satisfies condition (15) if it satisfies the following two conditions:*

$$\forall i \in \{1,2\} : (P_i)_\rho \text{ is in bijection with } P_i \tag{17}$$

$$(P_1 \;\|\; P_2)_\rho \;=\; (P_1)_\rho \;\|\; (P_2)_\rho \tag{18}$$

Comments. The primary application of this general theorem is when P and Q are synchronous systems, and $\rho = \alpha$ is the desynchronization morphism. This formalizes GALS deployment. Thus, Theorem 1 provides sufficient conditions to ensure correct GALS deployment.

Conditions (17) and (18) are not effective because they involve (infinite) behaviours. In [3,4], for GALS deployment, condition (17) was shown equivalent to some condition called *endochrony,* expressed in terms of the transition relation, not in terms of behaviours. Similarly, condition (18) was shown equivalent to some condition called *isochrony,* expressed in terms of the pair of transition relations, not in terms of pairs of sets of behaviours. Endochrony and isochrony are model checkable and synthesizable, at least for synchronous programs involving only finite data types (see [3,4] for a formal definition of these conditions).

In the same references, it was claimed that the two conditions (17) and (18) "mean" the preservation of semantics for a GALS deployment of a synchronous design. Several colleagues pointed to us that they did not see why this claim should be obvious. In the subsequent paper [2], an attempt was provided to fill this gap, with no real formalization, however. Theorem 1 provides the missing formal justification for this claim.

Proof. Inclusion $\supseteq$ in (15) always hold, meaning that every pair of behaviours unifiable in the right hand side of (15) is also unifiable in the left hand side. Thus it remains to show that, if the two conditions of Theorem 1 hold, then inclusion $\subseteq$ in (15) does too. Now, assume (17) and (18). Pick a pair (σ_1, σ_2) of behaviours which are unifiable in $P_1 \;_{(\rho}\|_{\rho)}\; P_2$. Then, by definition of $_{(\rho}\|_{\rho)}$, the pair $((\sigma_1)_\rho, (\sigma_2)_\rho)$ is unifiable in $(P_1)_\rho \,\|\, (P_2)_\rho$. Next, (18) guarantees that $(\sigma_1)_\rho \sqcup (\sigma_2)_\rho$ is a behaviour of $(P_1 \,\|\, P_2)_\rho$. Hence there must exist some pair (σ_1', σ_2') unifiable in $P_1 \;\|\; P_2$, such that $(\sigma_1' \sqcup \sigma_2')_\rho = (\sigma_1)_\rho \sqcup (\sigma_2)_\rho$. Using the same argument as before, we derive that $((\sigma_1')_\rho, (\sigma_2')_\rho)$ is also unifiable with respect to its associated (asynchronous) parallel composition, and $(\sigma_1')_\rho \sqcup (\sigma_2')_\rho =$

$(\sigma_1)_\rho \sqcup (\sigma_2)_\rho$. But $(\sigma'_1)_\rho$ is the restriction of $(\sigma'_1)_\rho \sqcup (\sigma'_2)_\rho$ to its events labeled by variables belonging to V_1, and similarly for $(\sigma'_2)_\rho$. Thus $(\sigma'_i)_\rho = (\sigma_i)_\rho$ for $i = 1, 2$ follows. Finally, using (17), we know that if (σ'_1, σ'_2) is such that, for $i = 1, 2$: $(\sigma'_i)_\rho = (\sigma_i)_\rho$, then: $\sigma'_i = \sigma_i$. Hence (σ_1, σ_2) is unifiable in $P_1 \parallel P_2$. ◇

Corollary 1. *Let P_1 and P_2 be synchronous systems whose behaviors are equipped with some equivalence relation $\sim$, and assume that P_1 and P_2 are closed with respect to $\sim$. Then, the pair (P_1, P_2) satisfies condition (15) if it satisfies the following two conditions:*

$$\forall i \in \{1, 2\} : (P_i)_\rho \text{ is in bijection with } P_i/\!\sim \tag{19}$$

$$\left(P_1 \parallel P_2\right)_\rho = (P_1)_\rho \parallel (P_2)_\rho \tag{20}$$

where $P_i/\!\sim$ is the quotient of P_i modulo $\sim$.

Proof. Identical to the proof of Theorem 1 until the paragraph starting with "Finally". Finally, using (19), we know that if (σ'_1, σ'_2) is such that, for $i = 1, 2$: $(\sigma'_i)_\rho = (\sigma_i)_\rho$, then: $\sigma'_i \sim \sigma_i$. Hence (σ_1, σ_2) is unifiable in $P_1 \parallel P_2$, since all synchronous systems we consider are closed under $\sim$. ◇

This result if of particular interest when $\sim$ is the equivalence modulo stuttering, defined by (4).

Running example, cont'd. Since P and Q possess a single behaviour, they clearly satisfy condition (17). However, the alternative condition (18) is violated: the left hand side is empty, while the right hand side is not. This explains why semantics is not preserved by desynchronization, for this example. In fact, it can be shown that the pair (P, Q) is not isochronous in the sense of [3,4]. More examples and counter-examples can be found in [5].

Discussion. In [10] the following result was proved. For P and Q two synchronous systems such that both P, Q, and $P \parallel Q$ are functional, clock-consistent, and with loop-free combinational part, then $P \parallel Q$ can be seen as a Kahn network—for our purpose, just interpret Kahn networks as functional asynchronous systems. This result applies to functional systems with inputs and outputs, it gives no help for partial designs or abstractions. Our conditions of endochrony and isochrony allows us to deal even with partial designs, not only with executable programs. Hence, they do represent effective techniques that can be used as part of the formal foundation for a successive-refinement design methodology.

As said before, this paper extends the ideas of [20] on desynchronization. A more naive "token-based" argument to explain GALS deployment is also found in [6], Sect. V.B. This argument is closely related to the use of Marked Graphs in [11] to justify GALS desynchronization in hardware.

Another example can be found in theory of latency-insensitive design [9]: here, if $P \parallel Q$ is a specification of a synchronous system and P and Q are *stallable* processes, then it is always possible to automatically derive two corresponding

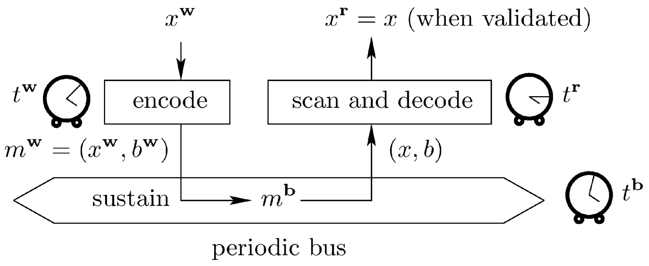


Fig. 1. The LTTA-protocol

patient processes P_p and Q_p that seamlessly compose to give a system implementation $P_p \,\|\, Q_p$ that preserves semantics while being also robust to arbitrary, but discrete, latency variations between P and Q. Again, $P_p \,\|\, Q_p$ is a correct deterministic executable system made of endochronous sub-systems. In fact, as the notion of stallable system and patient system correspond respectively to the notion of stuttering-invariant system and endochronous system, the extension to Theorem 1 subsumes the result presented in [9] on the compositionality of latency-insensitivity among patient processes.

4 Deploying Timed Synchronous Specifications over LTTA

Loosely Time-Triggered Architectures (LTTA) were introduced in [7] as a weakening of H. Kopetz' TTA. We revisit the results of [7] and complete them, by using the results from the previous section.

4.1 The LTTA-Protocol and Its Properties

See Figure 1 for an illustration of this protocol (the three watches shown indicate a different time, *they are not synchronized*). We consider three devices, the *writer,* the *bus,* and the *reader,* indicated by the superscripts $(.)^{\mathbf{w}}, (.)^{\mathbf{b}}$, and $(.)^{\mathbf{r}}$, respectively. Each device is activated by its own, approximately periodic, clock. The different clocks are *not* synchronized. In the following specification, the different sequences written, fetched, or read, are indexed by the set $\mathbf{N} = \{1, 2, 3, \ldots, n, \ldots\}$ of natural integers, and we reserve the index 0 for the initial conditions, whenever needed. Set $\mathbf{N}$ will serve to index the successive events of each individual signal, exactly as in our model of Section 2.1. Thus our informal description below is in agreement with our formal model. On the other hand, we believe that this informal description is quite natural.

The writer: At the time $t^{\mathbf{w}}(n)$ of the n-th tick of his clock, the writer generates a new value $x^{\mathbf{w}}(n)$ and a new alternating flag $b^{\mathbf{w}}(n)$ with:

$$b^{\mathbf{w}}(n) = \begin{cases} \textit{false} & \text{if } n = 0 \\ \textit{not } b^{\mathbf{w}}(n-1) & \text{otherwise} \end{cases}$$

and stores both in its private output buffer. Thus at any time t, the writer's output buffer content $m^{\mathbf{w}}$ is the last value that was written into it, that is the one with the largest index whose tick occurred before t:

$$m^{\mathbf{w}}(t) = (x^{\mathbf{w}}(n), b^{\mathbf{w}}(n)) \,, \text{ where } n = \sup\{n' \mid t^{\mathbf{w}}(n') < t\} \tag{21}$$

The bus: At the time $t^{\mathbf{b}}(n)$ of its n-th clock tick, the bus fetches the value in the writer's output buffer and stores it, immediately after, in the reader's input buffer. Thus, at any time t, the reader's input buffer content offered by the bus, denote it by $m^{\mathbf{b}}$, is the last value that was written into it, i.e., the one written at the latest bus clock tick preceding t:

$$m^{\mathbf{b}}(t) = m^{\mathbf{w}}(t^{\mathbf{b}}(n)) \,, \text{ where } n = \sup\{n' \mid t^{\mathbf{b}}(n') < t\} \tag{22}$$

The reader: At the time $t^{\mathbf{r}}(n)$ of its n-th clock tick, the reader copies the value of its input buffer into auxiliary variables $x(n)$ and $b(n)$:

$$(x(n), b(n)) = m^{\mathbf{b}}(t^{\mathbf{r}}(n))$$

Then, the reader extracts from the x sequence only the values corresponding to the indices for which b has alternated. This can be modeled thanks to the counter c, which counts the number of alternations that have taken place up to the current cycle. Hence, the value of the extracted sequence at index k is the value read at the earliest cycle when the counter c exceeded k:

$$\begin{aligned} c(n) &= \begin{cases} 0 & \text{if } n = 0 \\ c(n-1)+1 & \text{if } b(n) \neq b(n-1) \\ c(n-1) & \text{otherwise} \end{cases} \\ x^{\mathbf{r}}(k) &= x(n) \,, \text{ where } k = c(n) \end{aligned} \tag{23}$$

Call LTTA-protocol the protocol defined by the formulas (21,22,23).

Theorem 2 ([7]). *Let the writing/bus/reading be systems with physically periodic clocks of respective periods $w/b/r$. Then, the* LTTA*-protocol satisfies the following property whatever the written input sequence is:*

$$\forall k : x^{\mathbf{r}}(k) = x^{\mathbf{w}}(k), \tag{24}$$

iff the following conditions hold:

$$w \geq b \text{ , and } \left\lfloor \frac{w}{b} \right\rfloor \geq \frac{r}{b} \,, \tag{25}$$

where, for x a real, $\lfloor x \rfloor$ denotes the largest integer $\leq x$.

Condition (24) means that the bus provides a coherent system of logical clocks. Note that, since $w \geq b$, then $w/2b < \lfloor w/b \rfloor$ follows. On the other hand, $\lfloor w/b \rfloor \leq w/b$, and $\lfloor w/b \rfloor \sim w/b$ for w/b large. Hence, for a fast bus, i.e. $b \sim 0$, the conditions (25) of Theorem 2 reduce to $w \gg b, w > r$. In [7], Theorem 2 is extended to clocks subject to time-varying jitter, assuming some bounds for this jitter, and the case of multiple-users is briefly discussed.

Let us now consider a more involved situation, where several units communicate, and where there are data dependencies between communicated values. A simple example would be that the reader, upon receiving $x(n)$ has to send back $y(n) = f(x(n))$ to the writer, by using the same protocol. Thus, the reader has also to maintain its own $b^{\mathbf{r}}$ and to update an output buffer $m^{\mathbf{r}}$. It now computes, at the time $t^{\mathbf{r}}(n)$ of its n-th clock tick:

$$\begin{aligned}
(x(n), b(n)) &= m^{\mathbf{b}}(t^{\mathbf{r}}(n)) \\
c(n) &= \begin{cases} 0 & \text{if } n = 0 \\ c(n-1)+1 & \text{if } b(n) \neq b(n-1) \\ c(n-1) & \text{otherwise} \end{cases} \\
x^{\mathbf{r}}(k) &= x(n) \\
y^{\mathbf{r}}(k) &= f(x^{\mathbf{r}}(k)) \\
b^{\mathbf{r}}(k) &= \begin{cases} \mathit{false} & \text{if } k = 0 \\ \mathit{not}\ b^{\mathbf{r}}(k-1) & \text{otherwise} \end{cases} \\
m^{\mathbf{r}}(t^{\mathbf{r}}(n)) &= (y^{\mathbf{r}}(k), b^{\mathbf{r}}(k)), \text{ where } k = c(n)
\end{aligned}$$

This means that the computations indexed by k are only performed when the counter is increased, i.e., when the reader sees a bit alternation coming from the writer.

Timing issues: Theorem 2 says that, for the sake of protocol correctness, the reader should be faster than the writer. But now, the reader is also a writer and conversely. Hence, we need to distinguish between the period r_i at which any of these actors (reader or writer) i reads, and the period w_i at which the same actor i writes (updates its output buffer). This yields the following condition:

$$\forall i, j, \ \left\lfloor \frac{w_i}{b} \right\rfloor \ \geq \ \frac{r_j}{b}$$

Comparison with Lamport's logical clocks [18]: When several communicating actors are involved, the LTTA protocol is very reminiscent of Lamport's logical clock synchronization: in Lamport's protocol, each actor maintains a logical clock and when sending a message, time-stamps it with this clock. When receiving a message, the receiver compares its own clock with the time-stamp of the message. If its own clock is late with respect to the received time-stamps, it updates it. We could have stated the LTTA protocol similarly, by time-stamping the messages by the values of the counter c. Yet, the problem with Lamport's time-stamps is that they are ever increasing and, thus, subject to overflow. Here, the knowledge of periods and the properties of the LTTA architecture allow us to abstract these time-stamps into boolean ones:

$$b(n) = \begin{cases} \mathit{false} & \text{if } c(n) = 0 \text{ modulo } 2 \\ \mathit{true} & \text{otherwise} \end{cases}$$

4.2 Correct LTTA Deployment

Here, our work is slightly more involved. The reason is that the systems considered also involve physical time. And they involve physical time in two forms: absolute physical time and approximate physical time as provided by the imperfect watches. More precisely, in analyzing the deployment of synchronous designs over LTTA, the following notions need to be considered:

- Synchronous logical time to index the successive reactions of the synchronous specification.
- Physical global time as provided by an ideal and perfect clock. Corresponding dates can be used as part of the system specification.
- Actual local physical time as provided by each quasi-periodic writer/reader/bus clock (recall these are only loosely synchronized).

These are different "times", for combination and use at different stages of the design. As expected, heterogeneous systems will solve the problem.

The Ideal Design. Our reference is the so-called *ideal design.* It consists of a pair of timed synchronous specifications (P_1, P_2) for deployment over an ideal timed synchronous channel. The ideal channel is denoted by *Id*, it implements the *logically* instantaneous broadcast between P_1 and P_2, with some constant *physical* delay δ regarding dates. We adopt the following notational convention: if x_1 is output by P_1 toward P_2, the corresponding input for P_2 is denoted by x_2, and vice versa. Then, *Id* implements $x_2 := x_1$ with physical delay δ. Since we consider timed synchronous systems, we take as tag set: $\mathcal{T}_{\text{synch}} \times \mathcal{T}_{\varphi}$.

The Actual Deployment. It is modeled by $P_1 \;_{(\alpha}\| \; Ltta \; \|_{\alpha)} \; P_2$, where α is the canonical *date-preserving* desynchronization morphism:

$$\alpha \;:\; \mathcal{T}_{\text{synch}} \times \mathcal{T}_{\varphi} \;\longmapsto\; \{.\} \times \mathcal{T}_{\varphi},$$

and *Ltta*, the model of the LTTA medium, is a timed and asynchronous system with tag set $\{.\} \times \mathcal{T}_{\varphi}$. By Theorem 2, *Ltta* preserves the sequence of values of individual signals (this is the *flow invariance* condition of [20]). Regarding timing, if we assume bounded delay for the bus and bounded jitter due to asynchrony in sampling, then *Ltta* communication occurs with a nondeterministic delay within the bounds $[\delta - \varepsilon, \delta + \eta]$, where δ is the delay of the ideal medium. Semantics preserving, from ideal design to actual implementation, is captured as follows:

(a) The preservation of the functional semantics is modeled by the following request, which the reader should compare to relation (15):

$$P_1 \;_{((\alpha,t)}\|_{t)} \; Ltta \;_{(t}\|_{(t,\alpha))} \; P_2 \;\equiv\; P_1 \;_{(t}\|_{t)} \; Id \;_{(t}\|_{t)} \; P_2. \tag{26}$$

In (26), t denotes the morphism consisting of the removal of physical time from the tag, α is our desynchronization morphism introduced before, and the pair (α, t) denotes the morphism consisting in jointly removing physical time and desynchronizing. Thus the right hand side of (26) is the reference untimed semantics of our design, whereas the left hand side is the actual untimed deployment semantics. Thus (26) is indeed the requirement of preserving the functional semantics.

(b) Regarding timing aspects, we consider separately the bounds $[\delta - \varepsilon, \delta + \eta]$, for the timing behaviour of *Ltta* for transmitting each individual message, asynchronously.

The following theorem holds. It refines Theorem 1 for the case of mixed synchronous/timed and asynchronous/timed systems. It relies on the property

$$Ltta_t = Id_{(\alpha,t)},$$

which is a reformulation of property (24) in Theorem 2. Its proof is omitted because it is a mild variation of the proof of Theorem 1.

Theorem 3. *The pair (P_1, P_2) satisfies condition (26) if it satisfies the following two conditions:*

$$\forall i \in \{1,2\} : (P_i)_{(\alpha,t)} \text{ is in bijection with } (P_i)_t \tag{27}$$

$$\big(P_1 \ {}_{(t}\|_{t)} \ Id \ {}_{(t}\|_{t)} \ P_2\big)_\alpha \equiv (P_1)_{(\alpha,t)} \ \| \ Id_{(\alpha,t)} \ \| \ (P_2)_{(\alpha,t)} \tag{28}$$

Discussion. Theorem 3 gives sufficient conditions for semantics preserving, that are *independent* from the precise form of *Ltta*, because only the ideal channel *Id* is involved. The conditions that guarantee semantics preserving are (almost) identical to those of Th. 1, thus endo/isochrony apply as well to this case.

5 Concluding Remarks

In this paper, we proposed a novel mathematical framework (an extension to the LSV tagged signal model) for handling heterogeneous reactive systems. The interest of this theory rests on the theorems it can provide. These theorems support effective techniques to generate automatically correct-by-construction adaptors, between two designs supported by different coordination paradigms.

References

1. R. Alur, T. Dang, J. Esposito, Y. Hur, F. Ivancic, V. Kumar, I. Lee, P. Mishra, G. J. Pappas and O. Sokolsky. Hierarchical Modeling and Analysis of Embedded Systems. *Proc. of the IEEE,* 91(1), 11–28, Jan. 2003.
2. A. Benveniste. Some synchronization issues when designing embedded systems from components. In *Proc. of 1st Int. Workshop on Embedded Software, EMSOFT'01,* T.A. Henzinger and C.M. Kirsch Eds., LNCS 2211, 32–49, 2001.
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 Intl. Conference*, LNCS 1664, pages 162–177. Springer, Aug. 1999.

4. A. Benveniste, B. Caillaud, and P. Le Guernic. Compositionality in dataflow synchronous languages: specification & distributed code generation. *Information and Computation,* 163, 125–171 (2000).
5. A. Benveniste, L. P. Carloni, P. Caspi, and A. L. Sangiovanni-Vincentelli. Heterogeneous reactive systems modeling and correct-by-construction deployment. Technical Report UCB/ERL M03/23, Electronics Research Lab, University of California, Berkeley, CA 94720, June 2003.
6. A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The Synchronous Language Twelve Years Later. *Proc. of the IEEE*, 91(1):64–83, January 2003.
7. A. Benveniste, P. Caspi, P. Le Guernic, H. Marchand, J-P. Talpin and S. Tripakis. A Protocol for Loosely Time-Triggered Architectures. In *Embedded Software. Proc. of the 2nd Intl. Workshop, EMSOFT 2002*, Grenoble, France, Oct. 2002.
8. G. Berry. The Foundations of Esterel. MIT Press, 2000.
9. L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli. Theory of Latency-Insensitive Design. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 20(9):1059–1076, September 2001.
10. P. Caspi, "Clocks in Dataflow languages", *Theoretical Computer Science,* vol. 94:125–140, 1992.
11. J. Cortadella, A. Kondratyev, L. Lavagno, and C. Sotiriou. A concurrent model for de-synchronization. In *Proc. Intl. Workshop on Logic Synthesis*, May 2003.
12. L. de Alfaro and T.A. Henzinger. Interface Theories for Component-Based Design. In *Proc. of 1st Int. Workshop on Embedded Software, EMSOFT'01,* T.A. Henzinger and C.M. Kirsch Eds., LNCS 2211, 32–49, Springer Verlag, 2001.
13. J. Eker, J.W. Janneck, E.A. Lee, J. Liu, J. Ludwig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity–The Ptolemy approach. *Proc. of the IEEE,* 91(1), 127–144, Jan. 2003.
14. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The Synchronous Data Flow Programming Language LUSTRE. *Proc. of the IEEE*, 79(9):1305–1320, Sep. 1991.
15. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.
16. G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty. Model-Integrated Development of Embedded Software. *Proc. of the IEEE,* 91(1), 127–144, Jan. 2003.
17. H. Kopetz, Real-Time Systems: Design Principles for Distributed Embedded Applications. Kluwer Academic Publishers. 1997. ISBN 0-7923-9894-7.
18. L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communication of the ACM*, 21:558–565, 1978.
19. P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire. Programming real-time applications with SIGNAL. *Proc. of the IEEE*, 79(9):1326–1333, Sep. 1991.
20. P. Le Guernic, J.-P. Talpin, J.-C. Le Lann, Polychrony for system design. *Journal for Circuits, Systems and Computers.* World Scientific, April 2003.
21. E.A. Lee and A. Sangiovanni-Vincentelli. A Framework for Comparing Models of Computation. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems,* 17(12), 1217–1229, Dec. 1998.
22. E.A. Lee and Y. Xiong. System-Level Types for Component-Based Design. In *Proc. of 1st Int. Workshop on Embedded Software, EMSOFT'01,* T.A. Henzinger and C.M. Kirsch Eds., LNCS 2211, 32–49, Springer Verlag, 2001.
23. M. Mokhtari and M. Marie. Engineering Applications of MATLAB 5.3 and SIMULINK 3. Springer Verlag, 2000.

HOKES/POKES: Light-Weight Resource Sharing

Herbert Bos and Bart Samwel

LIACS, Leiden University,
Niels Bohrweg 1, 2333 CA, The Netherlands
{herbertb,bsamwel}@liacs.nl

Abstract. In this paper, we explain mechanisms for providing embedded network processors and other low-level programming environments with light-weight support for safe resource sharing. The solution consists of a host part, known as HOKES, and a network processor part, known as POKES. As common operating system concepts are considered to be too heavy-weight for this environment, we developed a system that pushes resource control all the way to the compiler. The HOKES/POKES architecture is described in detail and its implementation evaluated.

1 Introduction

In this paper, we explain how to provide light-weight support for resource sharing in network processors (NPs) and operating system (OS) kernels. In many embedded systems, the programming environment offers little support for resource sharing and safety. Often, the software consists of a single, monolithic application that is special-purpose and well-tested, and not supported by hardware-assisted memory isolation, privileged instructions, or the like. This is true not only for special-purpose signal-processors, but also for reconfigurable computing, such as FPGAs. Increasingly, however, embedded hardware allows for more flexible application domains. The network processor (NP) is an example of this trend [Coo02,Mic99]. It may contain on-chip a general-purpose control processor and a set of independent processing engines all running in parallel (an example is shown in Figure 2). The engines, commonly known as microengines (MEs), run their own micro-applications and share the resources on the board (for instance, buses, memories, data streams, etc.).

In practice, however, the code running on NPs still consists of a single monolithic application (albeit with various parallel components). The reason that NPs are not really shared by multiple programs is not that it would not be beneficial to applications. On the contrary, there are systems that could exploit such functionality to their profit, if only it existed. For example, plugged into a router, an NP board could forward packets on most of its MEs, while using the remaining ones to run user code (e.g. a monitoring application, or a filter). As a more concrete example, the EU SCAMPI project develops a monitoring platform for high-speed links that aims to allow multiple monitoring applications to be active simultaneously on the same network card [SCA01]. The real reason why such architectures are not shared is that there are simply no appropriate mechanisms for doing so. For example, the time-sharing techniques developed for general-purpose operating systems such as UNIX are not appropriate, as they are too heavy-weight.

In this paper we describe the details of a solution known as OKE that pushes the burden of resource sharing in environments without explicit hardware support for this purpose all the way to the compiler. The focus is on NPs and OS kernels and a C-like language. In our terminology, the OKE part that is running on host-like systems is known as HOKES (host OKE system), while POKES (peripheral OKE system) is the part running on the MEs of the NP. Together, they are able to provide a measure of safety to the system by restricting code in its usage of processing time, memory and APIs. A complete HOKES implementation for the Linux kernel is available under the GNU Public License from `www.liacs.nl/~herbertb/projects/oke`. We have also started the POKES implementation and will evaluate the overhead in the results section. Furthermore, we maintain that the principles described in this paper may apply to other environments also. A high-level overview of an early version of HOKES was given in [BS02]. This paper describes the implementation of the low-level mechanisms, compiler support and NP-specific issues.

In Section 2, we describe the HOKES/POKES architecture. Section 3 explains how isolation and resource safety are subject to higher-level policies. In Section 4, we discuss the various mechanisms that we employ to provide resource safety. In Section 5, we evaluate our work. Related work is discussed in Section 6, and conclusions are drawn in Section 7.

2 Architecture

2.1 Processing Hierarchy

As shown in Figure 2, the architecture that we wish to program consists of a five-level (L_0,L_1,...,L_4) processing hierarchy. In our case, it comprises one or more host processors running Linux (with the usual kernel/userspace domains), and one or more Intel IXP network processors, each containing a StrongARM control processor running embedded Linux (also with kernel and userspace code) and a set of microengines that run no OS whatsoever. Observe that, besides the PCI distance and amount of available resources, there is very little to distinguish between the combinations of levels L_1/L_2 and L_3/L_4. Indeed, although they probably will be used for different purposes, in our implementation they are treated the same. The application granularity of L_0 is assumed to be the ME, so application *foo* runs on (at least) one ME, dedicated to *foo*.

In the implementation we have used a P4 1.8 GHz host processor and an Intel IXP1200 NP running at 200 Mhz, with a StrongARM control processor, and 6 MEs with 4 hardware contexts each. The NP is mounted on an ENP2506 board fitted with 2 1Gbps Ethernet ports and 256MB SDRAM and 8MB SRAM.

CL	= code loader
NP	= network processor
ME	= microengine
CREC	= compilation record
OKE	= open kernel environment
HOKES	= host-like OKE system
POKES	= peripheral OKE system
ESC	= environment setup code
FEC	= forward error correction
GC	= garbage collector
SC	= stack checking
DM	= dynamic memory management
PC	= pointer checks
MT	= multithreading protection
TP	= processing time protection

Fig. 1. Glossary

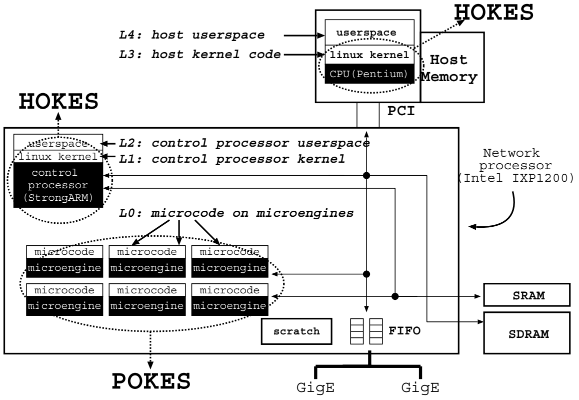


Fig. 2. Processing hierarchy and HOKES/POKES domains

2.2 Software

The problem of efficient resource sharing in this hierarchy is difficult, especially for levels L_0, L_1 and L_3 which concern environments without hardware support for isolation. Worse, we cannot apply traditional OS approaches, as these are too heavy-weight (in particular for the MEs). Instead, we have developed new abstractions for code isolation that push, to a large extent, resource control all the way to the compiler. In our software model, given the appropriate (and explicit) privileges, application code is allowed to run anywhere in the processing hierarchy. However, what such code is allowed to do is limited by static and dynamic checks that may control any resource, including memory and bus access, processing time, stack space, etc. The generic solution for any of the levels in the hierarchy is known as the *Open Kernel Environment* (OKE). The implementation of OKE on a host-like system, i.e. a system running a general-purpose operating system, such as the Pentium and StrongARM in Figure 2, is known as HOKES. The microengines on the NP do not run any OS whatsoever and hence require a somewhat different approach to achieve the same goal. This is known as a peripheral OKE system, or POKES. For the evaluation of HOKES we will use the implementation on the actual host processor (Pentium), as it is more convenient for performing measurements and data collection than the embedded StrongARM. For POKES, however, we have no choice other than using the microengines directly.

In order to make programming safe, efficient, and familiar to programmers, we extended and modified the *Cyclone* programming language, which itself is a dialect of C (retaining the C 'look and feel') [JMG$^+$02]. The result is known as *OKE-Cyclone*. The trust management used in the compilation and loading process is based on *KeyNote* [BFIK99] and described in detail in [BS02].

The idea is that a code loader explictly grants users the right to load code of a specific *type*, where 'type' means: 'subject to a set of restrictions on resource usage'. For this purpose, they request a compiler to compile their code according

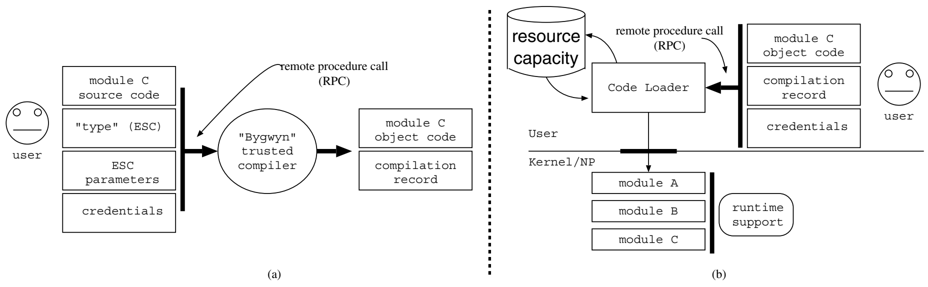


Fig. 3. User (a) compiles and (b) loads code using RPC from a remote site

to this type. A type is defined by so-called 'environment-set-up code', or ESC, that is associated with a user's program (and which instruments the code and checks whether specific conditions on resource usage hold) and is instantiated by the user by parameterisation. For example, the ESC may specify that the maximum amount of heap that can be used is 1 MB, while the parameterisation may indicate that the real maximum to be used for this code is only 0.5 MB. ESC will not normally be written by the users themselves, but defined once and for all by the authority that wants to restrict a specific group of users in a specific way. For example, there could be an ESC for 'students in a course of embedded software'. The compiler compiles the code with the ESC specified and then generates and signs a proof stating that this code was compiled with these restrictions by this compiler. The exact steps are the following(see Figure 3):

– To a (possibly remote) compile-server we submit (1) our source, and (2) the 'type', or ESC, to which the object code should comply.
– This trusted compile-server, known as *bygwyn*, first creates a *combined translation unit* that incorporates the ESC and our source code.
– *Bygwyn* then calls the OKE-Cyclone compiler to generate object code using custom language constructs to enforce safety (and isolation).
– *Bygwyn* returns the object code (an OKE *module*) and a signed *compilation record* (CREC) that vouches for the code's compliance to the ESC.
– Users may then submit the object code, the CREC and ESC to an *OKE code loader* (CL) at a site where the code should be executed.
– The CL uses the CREC to check whether the (1) object code matches the given ESC, (2) the compiler that signed the CREC is trusted, and (3) there is sufficient resource capacity to accommodate the request. It also checks the users' credentials to see if they are *allowed* to load code corresponding to this ESC. If none of the checks fail, the code is loaded.

3 Safety Policy (ESC)

In essence, an ESC is policy code that is *prepended* to, and sets up a compile-time environment for, the user-supplied code. For example, it defines run-time support code and explicitly declares the permissible API, restricting the code to just this API, by taking away the ability to (1) declare/import new APIs, (2) access the “private” parts of the ESC, and (3) perform privileged (unsafe) operations. ESCs are parameterised by the user to include, limit or remove certain capabilities. For example, users may specify that they have no need for dynamic memory, so that the support code that would otherwise be needed can be removed (yielding more efficient code). An ESC also has a component specifying *parameter limitations*. For default parameter values that may be overwritten by users with the appropriate credentials, this includes the range of valid values. It may also include a `FINALLY` keyword to indicate that this parameter may not be overwritten by users. This permits one to specify for instance that some users might be allowed to specify that they need up to 1 MB of dynamic memory, while others might be allowed to specify only 0.5 MB (or may not even be allowed to use heap memory at all). Apart from the ESC, these limitations are expressed in the users’ credentials.

When user code is compiled with a given policy (ESC), the trusted compile-server creates a translation unit that consists of (1) parameterisation of the ESC (in the form of macro definitions), (2) ESC, and (3) user code. This is processed by the OKE-Cyclone compiler to generate compliant object code. The OKE assumes a closed world, allowing for whole program analysis to optimise away dynamic checks that are only needed in an open world. This does not mean that modules consisting of multiple components cannot be composed. As long as the APIs are declared in the ESC, users can build larger programs by explicitly clicking together a score of modules. Even so, the OKE is especially useful for applications in restricted areas (e.g. kernels and embedded systems) and, therefore, the modules are often fairly small (i.e. contained in a single unit).

4 Language Features for Safety

Given the trust mechanism, we still need to ensure that code complies with a given safety policy at compile time. The following issues must be addressed if we want to enable users to run applications in any environment in a safe manner:

1. memory protection in the spatial domain (bounds checking);
2. memory protection in the temporal domain (references to freed memory);
3. stack overrun protection;
4. processing time restriction;
5. API restrictions;
6. hiding of sensitive data;
7. removing/disabling misbehaving code.

4.1 HOKES/POKES Spatial Pointer Safety

Cyclone is strongly typed and provides pointer safety in the following ways. Firstly, it provides *bounded pointers* that are statically tagged with a ‘valid length' or number of elements that must exist in the memory the pointer points to. These pointers are either NULL or point to at least the number of elements indicated by their 'valid length'. Conversions between these pointers are checked statically. Secondly, Cyclone provides *non-nullable* pointers, that can be statically proven to be non-NULL. Unlike 'normal' C pointers, non-nullable pointers do not need dynamic checks when they are dereferenced. Converting pointers to non-nullable pointers before using them in a loop, allows programmers to remove dynamic NULL-checks from inner loops. Thirdly, Cyclone also supports 'normal' C pointers, e.g. pointers with bounds that are only known at run-time and which incur run-time checks on all accesses.

4.2 HOKES/POKES Temporal Pointer Safety

Bounds checking solves the problem of *spatial* pointer safety, but not that of *temporal* pointer safety, such as returning the address of local variable. Cyclone solves this by including *region* information in pointer types [GMJ$^+$02]. The mechanism statically tags every pointer type with an identifier of the region of the memory it points to. A pointer that is tagged to point to a region *foo* is only capable of pointing to memory in region *foo* or in region *bar* that *outlives foo*, i.e., a region whose memory is guaranteed to exist when *foo* goes out of scope. In addition, *foo*'s memory can store only pointers to memory that outlives *foo* (because otherwise a "shorter-lived" region might disappear during the life of *foo*, leaving a dangling pointer in *foo*). When the address of a local variable of function *foo* is taken, the result will be a pointer into the *region* of *foo's* local variables. Any attempt to return it, or store it globally, will lead to region errors at compile time. Region tags can be explicitly specified by the programmer to express certain exceptional situations, but in C-like code the default region tags usually suffice (i.e., the code looks and feels like C, but has invisible safety enforcement). The region system is not used within the dynamic memory heap, which is normally garbage collected.

The Cyclone regions work well for Cyclone-only programs, but there are safety issues when interoperating with explicitly memory-managed languages such as C, as is the case, for instance, in HOKES. When a module holds a pointer to a kernel memory block that is explicitly memory managed, the memory block may be released, leaving the pointer dangling. For HOKES, we have therefore implemented Linux support for *delayed freeing*, ensuring that any memory released by the kernel is not immediately reused, but instead placed on a *kill list*, a list of blocks that still need to be "physically released".

In addition, we have written a HOKES garbage collector (GC) from scratch. We have placed the GC under user control, so that users may explicitly request a GC round (e.g. to clean up their own heap memory), or postpone it indefinitely. However, just prior to becoming active, every HOKES module which may have pointers to kernel memory *always* initiates a GC round (this is known at compile time). GC takes O(n), where n is the number of memory blocks allocated by the module. During garbage collection, all memory on the module's heap that can no longer be reached will be released.

During a module's garbage collection phase the HOKES detects whether or not a module holds pointers to 'freed' memory from the explicitly memory-managed region and acts appropriately by nullifying the pointers or by terminating the module (if the pointers are non-nullable). Once we have verified that no module holds pointers to a 'freed' memory block, the memory block is also physically freed. Currently, the released memory blocks are physically released even earlier, immediately after all modules have *deactivated*. This puts some restrictions on the programming model, because modules cannot safely stay active for very long times without exhausting the kernel's memory resources. We don't think this limitation is inherent in the approach, it is only caused by the simplicity of the current implementation. A more important limitation is that kernel memory cannot currently contain pointers back to memory controlled by HOKES's GC, because the GC cannot detect the kernel's references. All memory shared by HOKES and the kernel must be explicitly managed by the kernel.

We stress that delayed freeing is only needed for those modules that can actually hold pointers to kernel memory. This is a property that is checked at compile time: programs that do not contain any pointers tagged "kernel region" cannot point to explicitly memory-managed kernel memory. In other words, when considering a `free` for a chunk of kernel memory, there is no need to wait for modules to become inactive if they are guaranteed not to have pointers to this memory anyway.

Our current GC implementation uses a mark-and-sweep GC algorithm and is precise (it assumes strong typing and does not need to assume that all memory potentially contains pointers). The GC is supported by automatically generated code based on whole program analysis, which is done in the front end of the OKE-Cyclone compiler. The compiler detects which types of memory can be allocated by a module (by enumerating the types of all `new` and `malloc` expressions in the program) and generates *marking functions* for every type it encounters. A marking function for a type defines how a memory block of the type can be scanned for pointers, it contains a call to a function of the GC for every pointer in the type. The memory allocation calls in the module are then modified to pass to the memory subsystem pointers to the marking functions corresponding to the allocated type. Whenever the memory subsystem wants to find out which other memory blocks are referenced by a given memory block, it calls the marking function associated with the block and the marking function will call the GC with the addresses of all the other blocks referenced in this block.

POKES has no GC and currently does not even support truly dynamic memory. It is assumed that all memory accessible to MEs is statically declared. Region-based protection is still used, but in a more restricted version (e.g. to guard against common programming errors). As we shall see in 4.8, the memory may well be shared between MEs.

4.3 HOKES/POKES Language Restrictions

Although regions and pointer tags provide pointer safety, they provide no access restrictions for APIs and unsafe language features. To be able to restrict explicitly access to unsafe language features and APIs, we added the construct `forbid` to the language. Using this construct, it is possible to restrict access to features like `extern "C"` (which would allow users to import C APIs they don't have access to), `extern "C include"` (a Cyclone construct that allows for the inclusion of unsafe C code). In addition to that, `forbid namespace` enables us to forbid access to a complete namespace to all the code following this declaration. This allows an ESC to declare private helper functions and to import APIs in a private namespace, and to remove this namespace from the user code's view afterwards, leaving only the permissible APIs. In HOKES, unauthorised interaction with other modules is prevented by opening a *random* namespace for the user code. Because HOKES uses Cyclone's exception mechanism to interrupt misbehaving code in a safe manner, it is not safe for HOKES modules to catch certain types of exceptions. To prevent this we created the construct `forbid catch`, that allows exception access control at exception type granularity.

4.4 HOKES/POKES Wrapping: APIs and Entry Points

Upon entry from the environment into code that is part of a HOKES or POKES module, some entry/exit code needs to be executed, e.g. to perform HOKES garbage collection or catch any exceptions thrown from the module. We realised that it would be very useful indeed to be able to pass *function pointers* to external code so that such code may call these functions directly. However, to do so safely, we need to be able to wrap each of these functions. For this purpose, we created a new construct, `wrap extern`, that *automatically* wraps every exported (`extern`) function with wrapping code specified by the `wrap extern` declaration. At the same time, we prevent taking the address of functions that were not declared `extern`, so function pointers to non-extern, unwrapped functions cannot exist. In restrospect, our decision to bind the concept of "potential entry point" to the keyword "extern" has turned out to be inconvenient, because it forces the addition of entry/exit code even to functions that are never used outside of the module, but are called through function pointers.

The structure of the OKE also does not allow an ESC to make an external API directly accessible to an application. The reason is related to timeouts, a subject we will discuss in Section 4.7. Moreover, when an API is potentially unsafe, e.g. if it returns explicitly memory-managed memory directly or if it has weak parameter checking, extra work is needed to make Cyclone and the external API interoperate safely. In these situations, the APIs are *wrapped* by the ESC.

4.5 HOKES/POKES Protection of Sensitive Data

When data is shared between programs, we often want only certain fields of a structure to be shared, while others should not be exposed. In network monitoring, for instance, it may not be permissible for applications to find out which websites a user visits. To prevent this, the IP address of a packet is often scrambled (anonymised). However, scrambling takes time, even if the application is

```
int D() {
  return 0;
}

int C() {
  return D();
}

int B() {
  if (cond)
     return C();
  else return A();
}

int A() {
  return B();
}
```

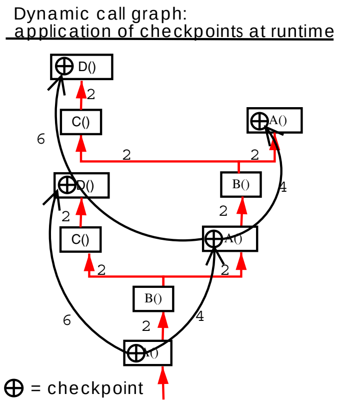


Fig. 4. Dynamic call graph: all functions take 2 units of stack (granularity = 6)

perfectly safe and does not even attempt to access the sensitive field. To address this, we created a new type modifier `locked` in OKE-Cyclone. The value of a `locked` variable cannot be used in calculations and cannot be converted to any other type, and is therefore rendered unusable at zero run-time overhead. An ESC can declare certain fields of structures `locked` to prevent user code from accessing them. The fact that a field is `locked` does *not* imply that the field cannot be written! The Cyclone language already has a construct to express this, `const`. In fact, `locked` non-`const` fields provide the interesting possibility to enforce a certain 'dataflow', that is, if a module is supposed to create a certain data structure to achieve it's task and one of the fields of this data structure is declared `locked`, then it can only fill this field using a source of `locked` data that the ESC provides: it is not possible to cast an existing non-`locked` value to a `locked` value.

4.6 HOKES Stack Overrun Protection

Unfortunately, HOKES stack overruns are hard to prevent without dynamic checking (in POKES this is not a problem as the MEs do not support a stack). However, we do have the advantage of having whole program analysis at our disposal, and use this to reduce the amount of checks. Stack overrun protection is configured by two variables, the *bound* and the *granularity*. The bound defines the amount of stack space that a program is allowed to use, while the granularity defines the maximum distance (measured in stack space) between checks in the program.

The process is illustrated in Figure 4 (it is assumed that for each function call the required stack space is 2 units). The OKE-Cyclone compiler analyses the entry points and the code's call graph to determine good locations for the *checkpoints* that satisfy the granularity constraint. For example, consider the set of (partly recursive) function calls of Figure 4. Suppose checkpoints were placed at the entry points of `A()` and the entry point of `D()` only (and not in the other two functions). Now, whenever `A()` is called, it is possible to check whether enough stack space exists to reach each of the following checkpoints. In the figure, bent arrows denote the distance between checkpoints, while straight arrows represent the call graph. A check is needed whether there is enough stack to store all of `A`'s local variables (in the general case: in the example there are no local variables), make a function call to `B`, store all local variables in `B()` on the two (different) paths to the next function calls (either `A()` or `C()`) and finally to make these calls. In case of a call to `C()`, we repeat this process until we have reached a checkpoint, i.e. we check whether there is enough stack to call `D()`. In other words, at the entrypoint of `A()` we need a total of at least 6 units of stack space. Even though no checkpoints were added in `B()` or `C()` at all, we are ensured that whenever either of these functions is called, there will be enough stack space to proceed to the next checkpoint.

Discussion. In the current implementation it is not possible for the programmer to directly control the locations of the checkpoints, but as this can be added fairly easily, it is expected to be in the next HOKES release. Right now, checkpointing is done by the compiler which tries to balance *bound* and *granularity* (minimising the number of checkpoints, while not placing them too far away from each other). For safety, we assume that all calls to imported functions may result in calls to all entry points into the code, while this is usually not the case. Performance would benefit if the ESC could specify whether an imported function can result in a callback or not.

4.7 HOKES Processing Time Protection

On many host systems that HOKES targets, processor time is a resource that must be protected. To prevent user code from getting stuck in an endless loop or to use up all processor time for too long, we had a choice between two solutions. The first involves the addition of dynamic checks in backward jumps, which incurs considerable run-time overhead and was therefore not used in HOKES. The second involves a limited amount of OS and hardware dependency, exploiting the timer interrupt in Linux. The implication is that for every new OKE environment, either a new method should be invented or dynamic checks should be used. For example, in the next section we will show that for POKES we adopted a very different solution to limit processing time.

HOKES uses the platform's *timer interrupt* to check for timeouts. The HOKES timers are checked on returning from a timer interrupt, i.e. after all truly time-critical tasks have been handled, but *before* returning control to the interrupted code. When a timeout flag is detected, HOKES jumps to a callback function that was registered by the ESC. When appropriate, the callback throws an exception to terminate the computation. The timeout mechanism takes into account whether a modules is executing HOKES or OS code when the timeout occurs. When executing system code, written in a different language, we cannot throw a Cyclone exception and rewind the stack at that point in time. Rather, a flag is set to indicate to the ESC's wrapper functions that they should not return control to the user code but throw a timeout exception instead.

4.8 POKES Processing Time Protection

As each application runs on a separate processor, POKES processor time is not considered to be a shared resource. Hence, there is no use for the timeout mechanisms discussed in the previous section. However, as throughput is essential, packets have to be processed in time and memory is shared between the applications. So we rephrased the processing time constraints in terms of cycle budgets, memory sharing and locking issues. In POKES, MEs are assumed to share a single data stream and the memory in which the stream is stored is a shared resource. If an application is not able to keep up with the arrival rate of the data, the backlog of data-to-process grows and the throughput drops. This is particularly bad in NPs where the goal is precisely to keep up with line speed and which have only a small cycle budget per packet (or set of packets).

Processing time protection in the current POKES is closely tied to the application framework, which reserves a single ME for receiving packets from the network and storing them in memory. All other MEs are available to the applications. The ESC for this framework provides for each ME a 'main' loop which prefilters packets and makes them available to the ME hardware contexts. Given the appropriate credentials, programmers may 'plug in' application code in this loop (with POKES restrictions) and load the complete program on the ME.

The packet fetch ME dedicates a single thread (known as 'mop-up') to dealing with tardy applications. Whenever an ME is falling behind too much (it has not processed enough packets in a given amount of time), the mop-up terminates it. We consider this to be the NP equivalent of timeouts. The ME only posts a kill request, the actual termination is done by the StrongARM. The packet fetch that we implemented places packets in a circular buffer spread over SRAM and SDRAM. The actual packets are stored in SDRAM, while buffer control structures are kept in SRAM. For example, a bit field per microengine per buffered packet is used to indicate whether a ME is 'done' with this packet. Applications may choose to process every packet or only a percentage of the packets. However, for all packets in the buffer they have to set their 'done' flag in time. In the same structure we also keep bit fields to implement readers/writers locks (with readers' preference) and fields indicating whether the packet has been fully received.

Packets are written in 64 byte chunks and applications do not have to wait for the packets to be received in their entirety: as soon as the first chunk of bytes has arrived, they may start processing. The point is that they need to process the packets within the cycle budget. This is enforced by the mop-up. At some distance from the writing position the mop-up thread explicitly *removes* packets from the circular buffer. If the mop-up finds that, for any ME, the 'done' bit is not set, it means the corresponding application has not completed the processing of this packet. In other words, the application is too slow and will be terminated.

By varying the distance between mop-up and receive, we may limit or extend the cycle budget for sets of packes available to applications.

Finally, we extended the default Intel device driver to provide memory mapping of all the IXP's SDRAM all the way to the host applications running in userspace. MEs may pass a reference to a packet to a queue destined for host applications, which prompts these applications to process the packet further on the host (accessing the required data from across the PCI bus).

Discussion: A POKES Compiler. Although there is currently no full compiler support for POKES, all mechanisms have been implemented and evaluated in isolation by handcrafting the code exactly like it will be generated by the compiler. Furthermore, we are in the process of implementing a compiler for POKES that will be rather simple in that it generates microengine C, an Intel-proprietary dialect of C with ME programming extensions. The current compiler is capable of producing ANSI C, so the translation to microengine C is fairly straightforward, requiring mainly 2 things: (1) as the storage class for all variables can be explicitly specified in microengine C, the compiler needs to be extended with storage class specifiers, and (2) we need to deal with the presence of many intrinsic functions in microengine C (but as these intrinsics have the same appearance as function calls, they are easy to support).

5 Evaluation

5.1 HOKES: Full Kernel-Based Monitoring and Transcoding

HOKES has been evaluated in various kernel experiments in network transcoding. The most interesting example applications are (1) transcoding with increasing packet size, adding forward error correction (FEC) to packet payloads, and (2) transcoding with decreasing packet size (resampling audio packets to contain only 50% of the original data).They were implemented directly in a single module that attaches itself to a Linux netfilter hook. The ESC is responsible for removing the appropriate IP packets from the netfilter framework, casting the packet structure to the corresponding types in OKE-Cyclone and passing it to the module (ensuring that the appropriate fields are made `const` and/or `locked`). It is also responsible for transmitting packets again.

For both cases, Table 2 shows the results for both the HOKES kernel code and the exact same implementation in C (also running in the kernel). In brackets the relative overhead is mentioned. Because the overhead varies with the packet size (more data needs to be processed), we listed both the overhead for minimum sized and maximum sized packets. A lot of the overhead, especially in the audio resampling transcoder, consists of initial costs. After that, in the loop of the transcoder, there is hardly any overhead. For this reason the difference between optimal C code and the HOKES solution is smaller, percentage-wise, for larger packets.

Table 1. Overhead of various aspects of OKE mechanism

Test	Aspect measured	Result
SC$_1$	init time at entry point	≈ 1 cycle
SC$_2$	additional time to process checkpoint code	5-10 cycles
SC$_3$	code size increase	16 instr. at entrypoint 15 instr. at other checkpoints
PC$_1$	overhead per normal (C-style) pointer dereference	≈ 1 cycle
PC$_2$	code size increase	3 instr. (commonly only 2 executed)
MT$_1$	overhead at entry point	35 cycles
MT$_2$	code size increase at entry point	15 instr.
TP$_1$	overhead at entry point	68 cycles
TP$_2$	extra overhead for calling kernel from HOKES module	10 cycles
TP$_3$	code size increase per entry point	24 instr.
GC$_4$	overhead for turning GC on, but not using it	10 cycles (for check)
GC$_5$	overhead for user requesting GC round with nothing to do	98 cycles
GC$_6$	overhead for GC round for checking 1 non-collectible pointer	73 cycles
GC$_7$	overhead for GC round for checking 1 collectible pointer	440 cycles
GC$_8$	time to sweep a block	10-41 cycles (plus cost of `kfree`)

5.2 HOKES Micro-Measurements

Given the appropriate privileges, an ESC may be parameterised to 'turn off' specific OKE mechanisms (e.g. a module may be compiled without the GC, and/or without stack checks, etc). If such parameters are not explicitly declared as `FINAL`, they may be overwritten by users with the appropriate credentials, allowing them to determine what runtime support is included in the compilation. This conveniently allows us to test the overhead of many of our mechanisms in isolation. In this section we measure the runtime overhead of various aspects of: stack checking (**SC**), garbage collection (**GC**), dynamic memory management (**DM**), pointer checks (**PC**), overhead incurred by spinlocks to prevent multiple threads from accessing the same module at the same time (**MT**), and processing time protection (**TP**). For each of these items we measure various aspects, as summarised in Table 1.

All speed measurements are in cycles and were conducted on an Intel P4 running a Linux 2.4.20 kernel and are fast-path values (i.e. without initial cache misses and inaccurate branch predictions and without any misbehaviour in the code that would cause it to be terminated). After startup, in the applications of Section 5.1, the fast-paths are always taken as the system was used solely used for packet transcoding. All code was compiled to object code using `gcc-2.9.5`. We suspect that using `gcc-3.2` performance will be better as the abstraction penalty (putting all components in function calls, etc.) in `gcc-3.2` is less severe (e.g. because its support for inlining has improved). We should stress than in POKES the overheads (and indeed the mechanisms) for GC, TP and SC do not exist.

Table 2. Total HOKES overhead

Program	**min.pkt** (μs)	**max.pkt** (μs)
FEC$_C$	1.3	18
FEC$_{HOKES}$	1.5 (15%)	19 (6%)
resample$_C$	1.3	8
resample$_{HOKES}$	1.8 (38%)	8 (≈0%)

Table 3. POKES bounds check overhead

	Experiment	**Unchecked code** (cycles)	**POKES** (cycles)
1	TCP_SYN	80	85 (6%)
2	UDP_port	60	70 (17%)
3	String_scan	430	520 (20%)
4	DiffServ	65	70 (8%)

5.3 POKES: Monitoring and Transcoding in a Network Processor

POKES was evaluated using the application framework of Section 4.8. As memory is allocated statically and processors are dedicated, the only true overhead incurred by POKES is caused by memory bounds checks. We tested four applications, all running on their own MEs:.

1. *TCP SYN detection*: denial of service attacks are often caused by sending many TCP SYN packets, so we count the number SYNs per time unit.
2. *UDP port detection*: this application detects any packets destined for UDP ports belonging either to Id Software (Doom), or to SunRPC.
3. *Scanning for a string*: for intrusion detection it is often needed to scan a packet for potentially harmful content (e.g. `/bin/perl`. In our example, we scan the first 16 bytes of the payload for the string."`/bin`".
4. *Set DiffServ field*: in this application we *write* a byte in the DiffServ field.

Although simple, all tasks derive from real-world applications. The throughput that can be achieved without any application running was close to 700 Mbps, on a 1 Gbps card, and we were able to sustain this load also with all applications running.

The average overhead caused by the additional memory bounds checking for the various applications is summarised in Table 3. The large overhead for application 3 is caused by its frequent memory accesses during the string search. Each of these is checked, leading to the highest absolute and relative overhead: 20%. At maximum rate (700 Mbps), the cycle budget per packet is significantly less than 520 cycles: roughly 200. The only reason why we were able to keep up was that we hide latency by using multiple hardware threads (a new thread starts scanning a new packet, whenever the current thread is waiting for a memory access to complete).

6 Related Work

There are several OSs that specifically target embedded systems and we do not intend to cover all of them in detail. Most relevant to our work are embedded Linux and VxWorks AE [Win01]. Embedded Linux is just like ordinary Linux but tailored to embedded processors and it is what we currently run on the NP's general purpose control processor. VxWorks (which can also be used on this processor) is a real-time OS that is widely adopted in the embedded industry to control hardware. Moreover, it provides more than just CPU protection. For example, it is able to use the MMUs of modern microprocessors to provide partitioning and protection of memory in a flexible manner ("dial-in protection"). It differs from our work in that we attempt to provide isolation to systems without having to rely on such hardware assists.

In research labs, we have also seen a number of general-purpose OSs that can be safely extended (e.g.Nemesis [LMB+96], ExoKernels, [EKO94], and SPIN [BCE+95]). Of these SPIN, which makes use of language-specific safety properties to ensure that applications do not interfere with each other, is most similar to our work. Unfortunately, all of these systems are research-oriented and not widely used, certainly not in embedded systems. Our aim is to provide isolation mechanisms that can be applied in a popular OS (notably Linux), and even in systems without OS support whatsoever (notably the MEs of an NP).

Previous attempts at providing software-based isolation include interpreted solutions(e.g. BPF [MJ93]), and native code solutions like Software Fault Isolation (SFI) [WLAG93] and Proof Carrying Code (PCC) [NL96]. We do not consider interpreted solutions suitable for high-throughput systems. Instead, it is our explicit goal to open up lower levels of the processing hierarchy to fully optimised code. SFI uses run-time checks to enforce safety. Not only are such checks costly , they also only take into account memory isolation (e.g. bounds checking), and not control isolation (e.g. branch checking). PCC provides code safety by supplying a *proof of correctness*, so that the loading site may check the correctness of code prior to loading it. The problem here is that generating proofs is a complex task which, to date, cannot be fully automated. Other approaches that allow programmers to load code in the Linux kernel are SILK [SPB+02] and FLAME [AIM+02]. SILK differs from our work in that it does not provide safety, and FLAME in that it is limited to monitoring functionality. RBClick described in [PL03] is similar to OKE, but differs in that all checks are completely static.

7 Conclusions

In this paper we have described the low-level mechanisms and compiler support for the Open Kernel Environment (OKE) which provides light-weight support for resource sharing and isolation in a processing hierarchy consisting of host processors and network processors. We have discussed both host side (HOKES) and network processor side (POKES). Policies are used to restrict a user's code in terms of access to resources. Whether or not users are granted access to resources is determined by their credentials. At the same time, the OKE targets performance by focusing on fully optimised code. Experimental results in the fields of monitoring and transcoding show that in Linux the overhead ranges from roughly 5 to 40 percent, and experimental results on Intel IXP1200 network processors show overheads of up to 20 percent. The OKE is currently a component in the operating system support for the EU SCAMPI project.

Acknowledgements. This work was supported by the EU SCAMPI project (IST-2001-32404). We would like to thank Lennert Buytenhek and Mihai Cristea for their help.

References

[AIM$^+$02] K. G. Anagnostakis, S. Ioannidis, S. Miltchev, J. Ioannidis, Michael B. Greenwald, and J. M. Smith. Efficient packet monitoring for network management. In *Proc. of NOMS'2002*, April 2002.

[BCE$^+$95] B. Bershad, C. Chambers, S. Eggers, C. Maeda, D. McNamee, P. Pardyak, S. Savage, and E. Gun Sirer. Protection is a software issue. In *HotOS-V*, May 1995.

[BFIK99] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. Keromytis. The KeyNote trust-management system version 2. *NWG, RFC 2704*, September 1999.

[BS02] H. Bos and B. Samwel. Safe kernel programming in the OKE. In *Proceedings of OPENARCH'02*, New York, USA, June 2002.

[Coo02] Intel Coorp. Internet exchange architecture: Programmable network processors for today's modular networks. White Paper, 2002.

[EKO94] D. Engler, F. Kaashoek, and J. O'Toole Jr. The exokernel approach to extensibility. In *Proc. of USENIX OSDI*, Monterey, Cal., November 1994.

[GMJ$^+$02] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *Proc. of PLDI'02*, Berlin, Germany, June 2002.

[JMG$^+$02] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *Proc. of USENIX'2002*, June 2002.

[LMB$^+$96] I. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. *JSAC*, 14(7), September 1996.

[Mic99] IBM Microelectronics. The network processor enabling technology for high-performance networking. White Paper, 1999.

[MJ93] S. McCanne and V. Jacobson. The BSD Packet Filter: A new architecture for user-level packet capture. In *Proc. of USENIX*, San Diego, Jan. 1993.

[NL96] G. Necula and P. Lee. Safe kernel extensions without run-time checking. In *Proceedings of OSDI'96*, Seattle, Washington, October 1996.

[PL03] P. Patel and J. Lepreau. Hybrid resource control of active extensions. In *Proc. of OPENARCH'03*, San Francisco, CA, April 2003.

[SCA01] SCAMPI Consortium. SCAMPI - A SCAlable Monitoring Platform for the Internet. Technical report, EU IST Research Project, Proposal Number: IST-2001-32404, Action Line IST-2001-IV.2.2, April 2001.

[SPB$^+$02] N. Shalaby, L. Peterson, A. Bavier, Y. Gottlieb abd S. Karlin, A. Nakao, X. Qie, T. Spalink, and M. Wawrzoniak. Extensible routers for active networks. In *DARPA AN Conference and Exposition*, June 2002.

[Win01] Wind River Systems MCL-DS-VAE-0111. VxWorks AE Datasheet. `http://www.windriver.com/products/vxworks5/`, 2001.

[WLAG93] R. Wahbe, S. Lucco, T.E. Anderson, and S.L. Graham. Efficient software-based fault-isolation. In *SOSP'03*, pages 203–216, December 1993.

Rate Monotonic vs. EDF: Judgment Day

G.C. Buttazzo

University of Pavia, Italy
buttazzo@unipv.it

Abstract. Since the first results published in 1973 by Liu and Layland on the Rate Monotonic (RM) and Earliest Deadline First (EDF) algorithms, a lot of progress has been made in the schedulability analysis of periodic task sets. Unfortunaltey, many misconceptions still exist about the properties of these two scheduling methods, which usually tend to favor RM more than EDF. Typical wrong statements often heard in technical conferences and even in research papers claim that RM is easier to analyze than EDF, it introduces less runtime overhead, it is more predictable in transient overload conditions, and causes less jitter in task execution. Since the above statements are either wrong, or not precise, it is time to clarify these issues in a systematic fashion, because the use of EDF allows a better exploitation of the available resources and significantly improves system's performance. This paper compares RM against EDF under several aspects, using existing theoretical results or simple counterexamples to show that many common beliefs are either false or only restricted to specific situations.

1 Introduction

Before a comprehensive theory was available, the most critical control applications were developed using an off-line table-driven approach (timeline scheduling), according to which the time line is divided into slots of fixed length (*minor cycle*) and tasks are statically allocated in each slot based on their rates and execution requirements. Although very predictable, such a technique is fragile during overload conditions and it is not flexible enough for supporting dynamic systems [24].

Such problems can be solved by using a priority-based approach, according to which each task is assigned a priority (which can be fixed or dynamic) and the schedule is generated on line based on the current priority value. In 1973, Liu and Layland [23] analyzed the properties of two basic priority assignment rules: the Rate Monotonic (RM) algorithm and the Earliest Deadline First (EDF) algorithm. According to RM, tasks are assigned fixed priorities that are proportional to their rate, so the task with the smallest period receives the highest priority. According to EDF, priorities are assigned dynamically and are inversely proportional to the absolute deadlines of the active jobs.

Assuming that each task is characterized by a worst-case execution time C_i and a period T_i, Liu and Layland showed that the schedulability condition of a task set can be derived by computing the *processor utilization factor* $U_p = \sum_{i=1}^{n} C_i/T_i$. Clearly, if $U_p > 1$ no feasible schedule exists for the task set with any algorithm. If $U_p \leq 1$, the feasibility of the schedule depends on the task set parameters and on the scheduling algorithm used in the system.

The Rate Monotonic algorithm is the most used priority assignment in real-time applications, because it is very easy to implement on top of commercial kernels that do not support explicit timing constraints. On the other hand, implementing a dynamic scheme, like EDF, on top of a priority-based kernel would require to keep track of all absolute deadlines and perform a dynamic mapping between absolute deadlines and priorities. Such an additional implementation complexity and runtime overhead often prevents EDF to be implemented on top of commercial real-time kernels, even though it would increase the total processor utilization.

At present, only a few research kernels support EDF as a native scheduling scheme. Examples of such kernels are Hartik [8], Shark [15], Erika [14], Spring [33], and Yartos [17]. A new trend in some recent operating system is to provide support for the development of a user level scheduler. This is the approach followed in MarteOS [27].

In addition to resource exploitation and implementation complexity, there are other evaluation criteria that should be taken into account when selecting a scheduling algorithm for a real-time application. Moreover, there are a lot of misconceptions about the properties of these two scheduling algorithms, that for a number of reasons penalize EDF more than it should be. The typical motivations that are usually given in favor of RM state that RM is easier to implement, it introduces less runtime overhead, it is easier to analyze, it is more predictable in transient overload conditions, and causes less jitter in task execution.

In the following sections we show that most of the claims stated above are either false or do not hold in the general case. Although some discussion of the relative costs of RM and EDF appear in [35], in this paper the behavior of these two algorithms is analyzed under several perspectives, including implementation complexity, runtime overhead, schedulability analysis, robustness during transient overloads, and response time jitter. Moreover, the two algorithms are also compared with respect to other issues, including resource sharing, aperiodic task handling, and QoS management.

2 Implementation Complexity

When talking about the implementation complexity of a scheduling algorithm, we have to distinguish the case in which the algorithm is developed on top of a generic priority based operating system, from the case in which the algorithm is implemented from scratch, as a basic scheduling mechanism in the kernel.

When considering the development of the scheduling algorithm on top of a kernel based on a set of fixed priority levels, it is indeed true that the EDF implementation is not easy, nor efficient. In fact, even though the kernel allows task priorities to be changed at runtime, mapping dynamic deadlines to priorities cannot always be straightforward, especially when, as common in most commercial kernels, the number of priority levels is small. If the algorithm is developed from scratch in the kernel using a list for the ready queue, then the only difference between the two approaches is that, while in RM the ready queue is ordered by decreasing fixed priority levels, under EDF it has to be ordered by increasing absolute deadlines. Thus, once the absolute deadline is available in the task control block, the basic kernel operations (e.g., insertion, extraction, getfirst, dispatch) have the same complexity, both under RM and EDF.

An advantage of RM with respect to EDF is that, if the number of priority levels is not high, the RM algorithm can be implemented more efficiently by splitting the ready queue into several FIFO queues, one for each priority level. In this case, the insertion of a task in the ready queue can be performed in $O(1)$. Unfortunately, the same solution cannot be adopted for EDF, because the number of queues would be too large (e.g., equal to 2^{32} if system time is represented by four byte variables).

Another disadvantage of EDF is that absolute deadlines change from a job to the other and need to be computed at each job activation. Such a runtime overhead is not present under RM, since periods are typically fixed. However, the problem of evaluating the runtime overhead introduced by the two algorithms is more articulated, as discussed in the next section.

3 Runtime Overhead

It is commonly believed that EDF introduces a larger runtime overhead than RM, because in EDF absolute deadlines need to be updated from a job to the other, so slightly increasing the time needed to execute the job activation primitive. However, when context switches are taken into account, EDF introduces less runtime overhead than RM, because it reduces the number of preemptions that typically occur under RM.

The example illustrated in Figure 1 shows that, under RM, to respect the priority order given by periods, the high priority task τ_1 must preempt every instance of τ_2, whereas under EDF preemption occurs only once in the entire hyperperiod[1]. For larger task sets, the number of preemptions caused by RM increases, thus the overhead due to the context switch time is higher under RM than EDF.

To evaluate the behavior of the two algorithms with respect to preemptions, a number of simulation experiments have been performed using synthetic task sets with random parameters.

Figure 2 shows the average number of preemptions introduced by RM and EDF as a function of the number of tasks. For each point in the graph, the average was computed over 1000 independent simulations, each running for 1000 units of time. In each simulation, periods were generated as random variables

[1] The hyperperiod is defined as the smallest interval of time after which the schedule repeats itself and it is equal to the least common multiple of the task periods.

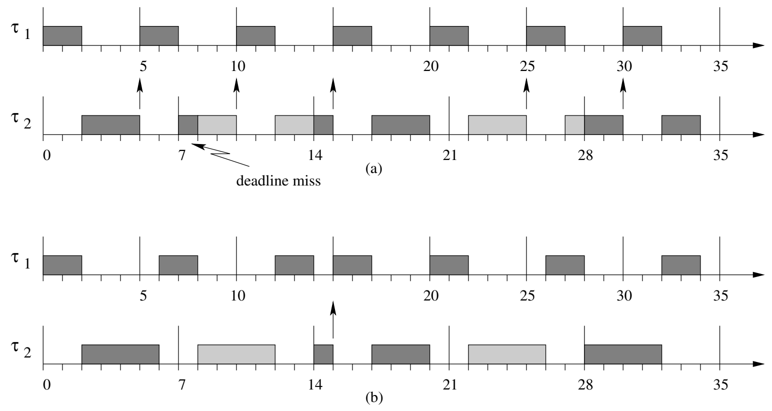


Fig. 1. Preemptions introduced by RM (a) and EDF (b) on a set of three periodic tasks. Adjacent jobs of τ_2 are depicted with different colours to better distinguish them.

with uniform distribution in the range of 10 to 100 units of time, whereas execution times were computed to create a total processor utilization of 0.9.

As shown in the plot, each curve has two phases: the number of preemptions occurring in both schedules increases for small task sets and decreases for larger task sets. This can be explained as follows. For small task sets, the number of preemptions increases because the chances for a task to be preempted increase with the number of tasks in the system. As the number of tasks gets higher, however, task execution times get smaller in the average, to keep the total processor utilization constant, hence the chances for a task to be preempted reduce. As evident from the graph, such a reduction is much more significant under EDF.

In another experiment, we tested the behavior of RM and EDF as a function of the processor load, for a fixed number of tasks. Figure 3 shows the average number of preemptions as a function of the load for a set of 10 periodic tasks. Periods and computation times were generated with the same criterion used in the previous experiment, but to create an average load ranging from 0.5 to 0.95.

It is interesting to observe the different behavior of RM and EDF for high processor loads. Under RM, the number of preemptions constantly increases with the load, because tasks with longer execution times have more chances to be preempted by tasks with higher priorities. Under EDF, however, increasing task execution times does not always imply a higher number of preemptions, because a task with a long period could have an absolute deadline shorter than that of a task with smaller period. In certain situations, an increased execution time can also cause a lower number of preemptions.

This phenomenon is illustrated in Figure 4, which shows what happens when the execution time of τ_3 is increased from 4 to 8 units of time. When $C_3 = 4$, the second instance of τ_2 is preempted by τ_1, that has a shorter absolute

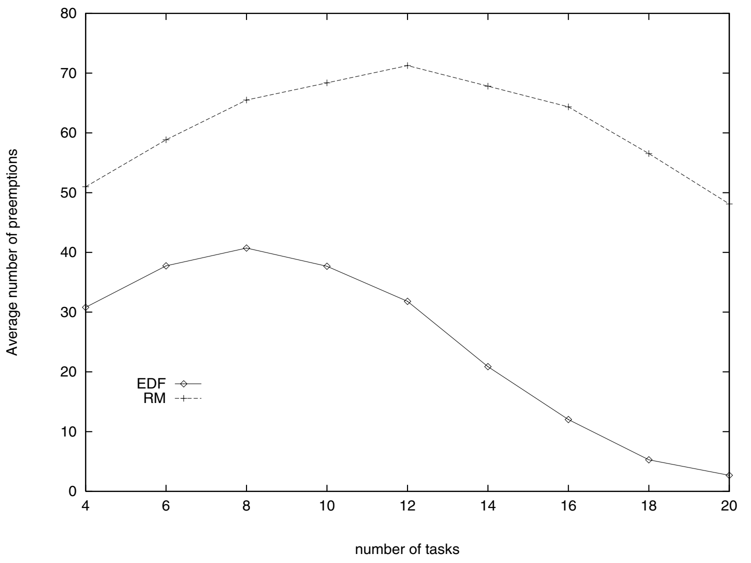


Fig. 2. Preemptions introduced by RM and EDF as a function of the number of tasks.

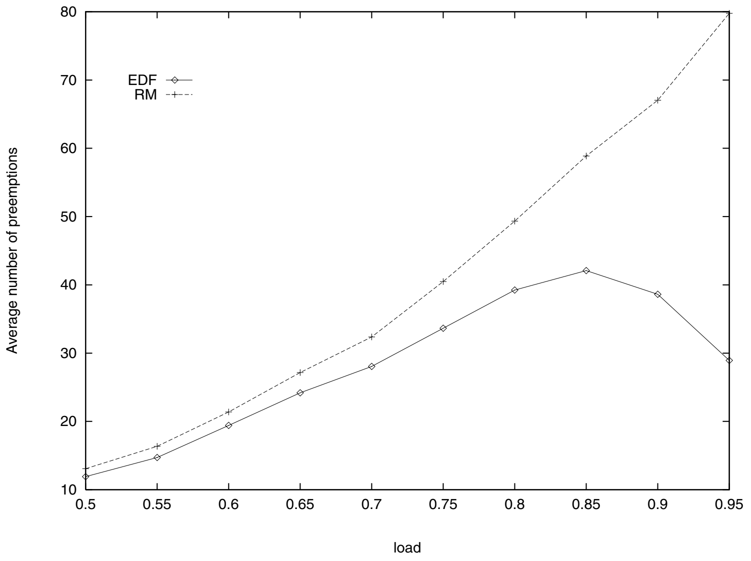


Fig. 3. Preemptions introduced by RM and EDF on a set of 10 periodic tasks as a function of the load.

deadline. If $C_3 = 8$, however, the longer execution of τ_3 (which has the earliest deadline among the active tasks) pushes τ_2 after the arrival of τ_1, so avoiding its preemption. Clearly, for a higher number of tasks, this situation occurs more frequently, offering more advantage to EDF. Such a phenomenon does not occur under RM, because tasks with small period always preempt tasks with longer period, independently of the absolute deadlines.

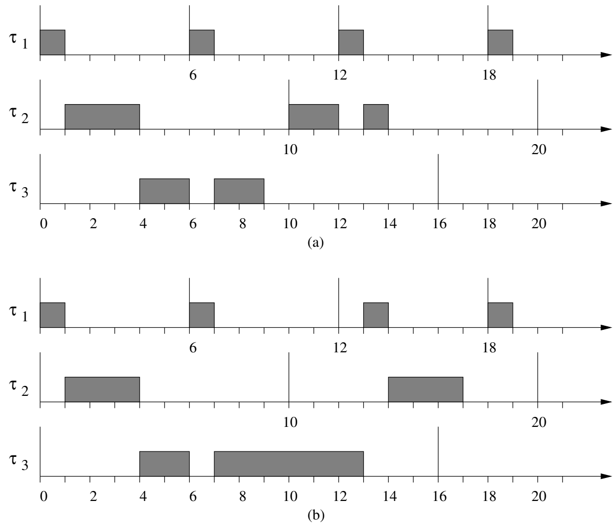


Fig. 4. Under EDF, the number of preemptions may decrease when execution times increase: in case (a), where C_3 is small, τ_2 is preempted by τ_1, but this does not occur in case (b), where τ_3 has a higher execution time.

4 Schedulability Analysis

The basic schedulability conditions for RM and EDF proposed by Liu and Layland in [23] were derived for a set Γ of n periodic tasks under the assumptions that all tasks start simultaneously at time $t = 0$, relative deadlines are equal to periods, and tasks are independent (that is, they do not have resource constraints, nor precedence relations). Under such assumptions, a set of n periodic tasks is schedulable by the RM algorithm if

$$\sum_{i=1}^{n} U_i \leq n\,(2^{1/n} - 1). \tag{1}$$

The schedulability bound of RM is a function of the number of tasks, and it decreases with n. We recall that for large n the bounds tends to $\ln 2 \simeq 0.69$. Under EDF, a task set is schedulable if and only if $\sum_{i=1}^{n} U_i \leq 1$.

In [21], Lehoczky, Sha, and Ding performed a statistical study and showed that for task sets with randomly generated parameters the RM algorithm is able to feasibly schedule task sets with a processor utilization up to about 88%.

However, this is only a statistical result and cannot be taken as an absolute bound for performing a precise guarantee test.

A more efficient schedulability test, known as the Hyperbolic Bound (HB), was proposed by Bini et al. in [6]. This test has the same complexity as the Liu and Layland one, but improves the acceptance ratio up to a limit of $\sqrt{2}$ for large n. According to this method, a set of periodic tasks is schedulable by RM if

$$\prod_{i=1}^{n}(U_i + 1) \leq 2. \tag{2}$$

For RM, the schedulability bound improves when periods have harmonic relations. A common misconception, however, is to believe that the schedulability bound becomes 1.00 when the periods are multiple of the smallest period. This is not true, as can be seen from the example reported in Figure 5.

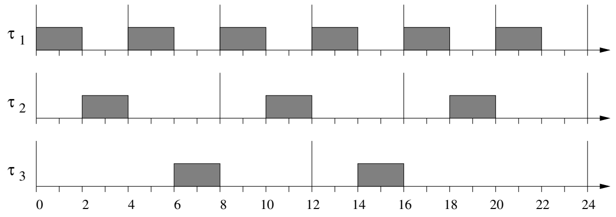


Fig. 5. A task set in which all periods are multiple of the shortest period is not sufficient to guarantee a schedulability bound equal to one.

Here, tasks τ_2 and τ_3 have periods $T_2 = 8$ and $T_3 = 12$, which are multiple of $T_1 = 4$. Since all tasks have a computation time equal to two, the total processor utilization is

$$U = \frac{1}{2} + \frac{1}{4} + \frac{1}{6} = \frac{11}{12} \simeq 0.917$$

and, as we can see from the figure, the schedule produced by RM is feasible. However, it easy to see that increasing the computation time of any task by a small amount τ_3 will miss its deadline. This means that, for this particular task set, the utilization factor cannot be higher than 11/12.

The correct result is that the schedulability bound becomes 1.00 only when *any* pair of periods is in harmonic relation. As an example, Figure 6 shows that, if $T_3 = 16$, then not only T_2 and T_3 are multiple of T_1, but also T_3 is multiple of T_2. In this case, the RM generates a feasible schedule even when $C_3 = 4$, that is when the total processor utilization is equal to 1.00.

In the general case, exact schedulability tests for RM yielding to necessary and sufficient conditions have been independently derived in [18,21,2]. Using the Response Time Analysis (RTA) proposed in [2], a periodic task set (with deadlines less than or equal to periods) is schedulable with the RM algorithm if and only if the worst-case response time of each task is less than or equal to its

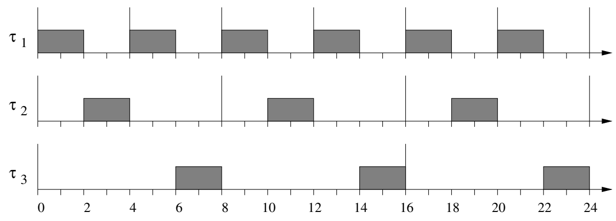


Fig. 6. When *any* pair of periods is in harmonic relation the schedulability bound of RM is equal to one.

deadline. The worst-case response time R_i of a task can be computed using the following iterative formula:

$$\begin{cases} R_i^{(0)} = C_i \\ R_i^{(k)} \ = \ C_i + \displaystyle\sum_{j:D_j<D_i} \left\lceil \frac{R_i^{(k-1)}}{T_j} \right\rceil C_j. \end{cases} \tag{3}$$

where the worst-case response time of task τ_i is given by the smallest value of $R_i^{(k)}$ such that $R_i^{(k)} \ = \ R_i^{(k-1)}$. It is worth noting, however, that the complexity of the exact test is pseudo-polynomial, thus it is not suited to be used for online admission control in applications with large task sets. To solve this problem, an approximate fesibility test with a tunable complexity has been proposed by Bini and Buttazzo in [7].

Under EDF, the schedulability analysis of periodic tasks with relative deadlines less than periods can be performed using the Processor Demand Criterion proposed by Baruah, Howell, and Rosier [5]. According to this method, a set of tasks is schedulable by EDF if and only if

$$\forall L > 0, \quad \sum_{i=1}^{n} \left\lfloor \frac{L + T_i - D_i}{T_i} \right\rfloor C_i \leq L \tag{4}$$

As the response time analysis, this test has also a pseudo-polynomial complexity. It can be shown that the number of points in which the test has to be performend can be significantly restricted to those L equal to deadlines less than a certain value L^*, that is:

$$\forall L \in \mathcal{D}, \quad \mathcal{D} = \{d_k : d_k < min(L^*, H)\}$$

where $H = lcm(T_1, \ldots, T_n)$ is the hyperperiod and

$$L^* = \frac{\sum_{i=1}^{n} U_i(T_i - D_i)}{1 - U}.$$

In conclusion, if relative deadlines are equal to periods, exact schedulabilty analysis can be performed in $O(n)$ under EDF, whereas is pseudo-polynomial under RM. When relative deadlines are less than periods, the analysis is pseudo-polynomial for both scheduling algorithms.

5 Robustness During Overloads

Another common misconception about RM is to believe that, in the presence of transient overload conditions, deadlines are missed predictably, that is, the first tasks that fail are those with the longest period. Unfortunately, this property does not hold under RM (neither under EDF), and can easily be confuted by the counterexample shown in Figure 7. In this figure, there are four periodic tasks with computation times $C_1 = 2$, $C_2 = 3$, $C_3 = 1$, $C_4 = 1$, and periods $T_1 = 5$, $T_2 = 9$, $T_3 = 20$, $T_4 = 30$. In normal load conditions, the task set is schedulable by RM. However, if there is a transient overload in the first two instances of task τ_1 (in the example, the jobs have an overrun of 1.5 time units), the task that misses its deadline is not the one with the longest period (i.e., τ_4), but τ_2.

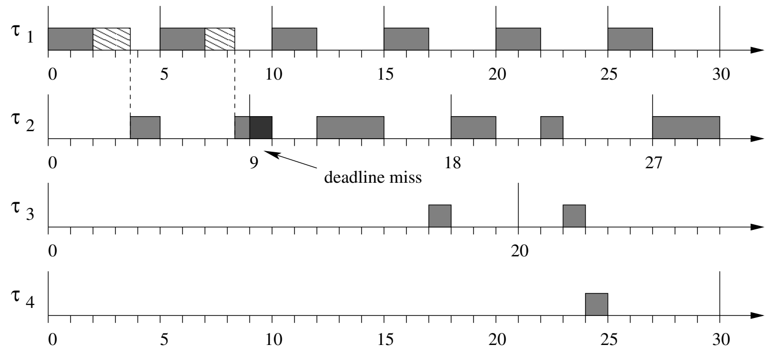


Fig. 7. Under overloads, only the highest priority task is protected under RM, but nothing can be ensured for the other tasks.

So the conclusion is that, under RM, if the system becomes overloaded, any task, except the highest priority task, can miss its deadline, independently of its period. The situation is not better under EDF. The only difference between RM and EDF is that, under RM, an overrun in task τ_i cannot cause tasks with higher priority to miss their deadlines, whereas under EDF any other task could miss its deadline.

The problem caused by execution overruns can be solved by enforcing temporal isolation among tasks through a resource reservation mechanism in the kernel. This issue has been investigated both under RM and EDF and it is discussed in Section 7.

6 Jitter

In a feasible periodic task system, the computation performed by each job must start after its release time and must complete within its deadline. Due to the presence of other concurrent tasks that compete for the processor, however, a task may evolve in different ways from instance to instance; that is, the instructions that compose a job can be executed at different times, relative to the release time, within different jobs. The maximum time variation (relative to the release time) in the occurence of a particular event in two consecutive instances of a task defines the jitter for that event. So, for example, the start time jitter of a task is the maximum time variation between the relative start times of any two consecutive jobs. If $s_{i,k}$ denotes the start time of the k^{th} job of task τ_i, then the start time jitter (STJ_i) of task τ_i is defined as

$$STJ_i = \max_k |s_{i,k+1} - s_{i,k}|. \tag{5}$$

Similarly, the response time jitter is defined as the maximum difference between the response times of any consecutive jobs. If $R_{i,k}$ denotes the response time of the k^{th} job of task τ_i, then the response time jitter (RTJ_i) of task τ_i is defined as

$$RTJ_i = \max_k |R_{i,k+1} - R_{i,k}|. \tag{6}$$

In real-time applications, the jitter can be tolerated when it does not degrade the performance of the system. In many control applications, however, a high jitter can cause instability or a jerky behavior of the controlled system [25], hence it must be kept as low as possible.

Another misconception about RM is to believe that the fixed priority assignment used in RM reduces the jitter during task execution, more than under EDF. This is false, as it is shown by the following example. Consider a set of three periodic tasks with computation times $C_1 = 2$, $C_2 = 3$, $C_3 = 2$, and periods $T_1 = 6$, $T_2 = 8$, $T_3 = 12$. For this example we consider the response time jitter, as defined by equation (6). Figure 8 illustrates that, under RM, the three tasks experience a response time jitter equal to 0, 2, and 8, respectively. Under EDF, the same tasks have a response time jitter equal to 1, 2, and 3, respectively. Hence, under RM, the jitter experienced by task τ_3 is much higher than that given under EDF.

Notice that this example does not prove that EDF always introduces less jitter than RM, but just confutes the common belief that RM outperforms EDF in reducing jitter.

7 Other Issues

7.1 Resource Sharing

If tasks share mutually exclusive resources, particular care must be used for accessing shared data. In fact, if critical sections are accessed through classical semaphores, then tasks may experience long delays due to a phenomenon known as *priority inversion*, where a task may be blocked by a lower priority task for an unbounded amount of time. The problem can be solved by adopting specific concurrency control protocols for accessing critical sections, such as the Priority Inheritance Protocol (PIP) or the Priority Ceiling Protocol (PCP), both

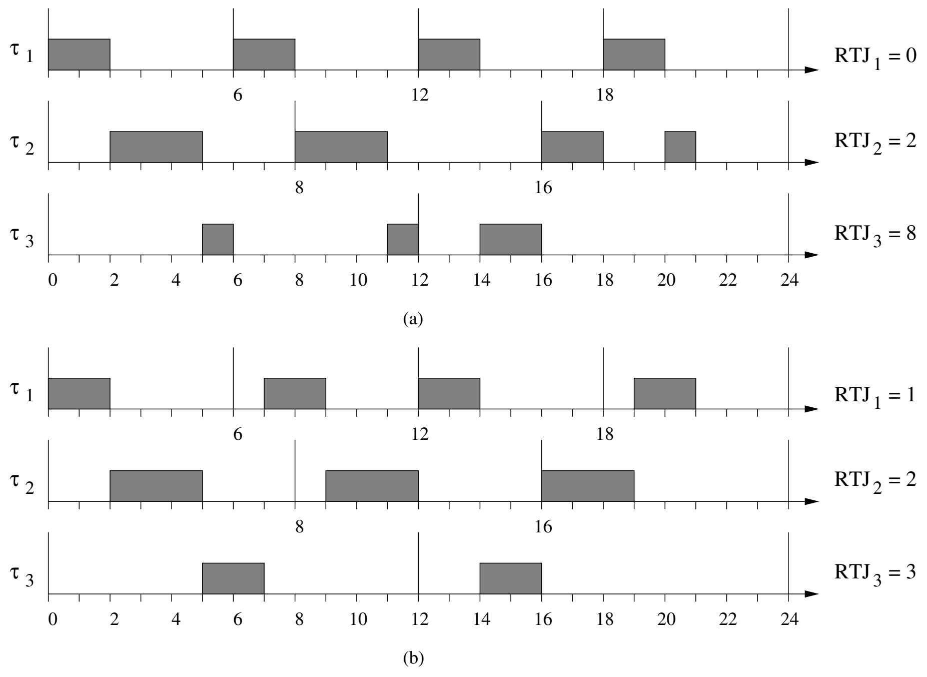


Fig. 8. Response time jitter introduced under RM (a) and under EDF (b).

developed by Sha, Rajkumar and Lehoczky in [28]. For the reason that both PIP and PCP are well known in the real-time literature and were originally deviced for RM, some people believe that only RM can be predictably used and analyzed in the presence of shared resources. However, this is not true, because a number of protocols also exist for accessing shared resources under EDF, such as the Dynamic Priority Ceiling (DPC) [12], the Stack Resource Policy (SRP) [3], and other similar methods specifically developed for deadline-based scheduling algorithms [16,35].

7.2 Aperiodic Task Handling

When real-time systems include soft aperiodic activities to be scheduled together with hard periodic tasks, RM and EDF show a significant difference in achieving good aperiodic responsiveness. In this case, the objective is to reduce the aperiodic response times as much as possible, still guaranteeing that all periodic tasks complete within their deadlines.

Several aperiodic service methods have been proposed under both algorithms. The common approach adopted in such cases is to schedule aperiodic requests through a periodic server, which allocates a certain amount of budget C_s, for aperiodic execution, in every period T_s. Under RM, the most used service mechanisms are the Deferrable Server [20,34], and the Sporadic Server [29], which have good responsiveness and easy implementation complexity. Under EDF, the most efficient service mechanism in terms of performance/cost ratio is the Total Bandwidth Server (TBS), proposed by Spuri and Buttazzo in [30,32].

The superior performance of EDF-based methods in aperiodic task handling comes from the higher processor utilization bound. In fact, the lower schedulability bound of RM limits the maximum utilization ($U_s = C_s/T_s$) that can be assigned to the server for guaranteeing the feasibility of the periodic task set. As a consequence, the spare processor utilization that cannot be assigned to the server is wasted as a background execution. This problem does not occur under EDF, where, if U_p is the processor utilization of the periodic tasks, the full remaining fraction $1 - U_p$ can always be allocated to the server for aperiodic execution.

A different approach used under RM to improve aperiodic responsiveness is the one adopted in the Slack Stealing algorithm, originally proposed by Lehoczky and Thuel in [22]. The main idea behind this method is that there is no benefit in completing periodic tasks much before their deadlines. Hence, when an aperiodic request arrives, the Slack Stealer steals all the available slack from periodic tasks (pushing them as much as possible towards their deadlines) and uses the slack to execute aperiodic requests as soon as possible. Although the Slack Stealer performs much better than the Deferrable Server and the Sporadic Server, it is not optimal, in the sense that it cannot minimize the response times of the aperiodic requests.

Tia, Liu, and Shankar [36] proved that, if periodic tasks are scheduled using a fixed-priority assignment, no algorithm can minimize the response time of every aperiodic request and still guarantee the schedulability of the periodic tasks. In particular, the following theorems were proved in [36]:

Theorem 1 (Tia-Liu-Shankar). *For any set of periodic tasks ordered on a given fixed-priority scheme and aperiodic requests ordered according to a given aperiodic queueing discipline, there does not exist any valid algorithm that minimizes the response time of every soft aperiodic request.*

Theorem 2 (Tia-Liu-Shankar). *For any set of periodic tasks ordered on a given fixed-priority scheme and aperiodic requests ordered according to a given aperiodic queueing discipline, there does not exist any on-line valid algorithm that minimizes the average response time of the soft aperiodic requests.*

Notice that Theorem 1 applies both to clairvoyant and on-line algorithms, whereas Theorem 2 only applies to on-line algorithms. These results are not true for EDF, where otpimal algorithms have been found for minimizing aperiodic response times. In particular, the Improved Total Bandwith server (ITB) [10] assigns an initial deadline to an aperiodic request according to a TBS with a bandwidth $U_s = 1 - U_p$. Then, the algorithm tries to shorten this deadline as much as possible to enhance aperiodic responsiveness, still maintaining the periodic tasks schedulable. When the deadline cannot be furtherly shortened, it can be used to schedule the task with EDF, thus minimizing its response time. The algorithm that stops the deadline shortening process after N steps is denoted as TB(N). Thus, TB(0) denotes the standard TBS and TB* denotes the optimal algorithm which continues to shorten the deadline until its minimum value.

Figure 9 shows the performance of the Slack Stealer against TB(0), TB(1), TB(3), and TB* as a function of the aperiodic load, when the periodic utilization is $U_p = 0.85$. The periodic task set has been chosen to be schedulable both under RM and EDF. As can be seen from the plots, the Slack Stealer performs better than the standard TBS. However, just one iteration on the TBS deadline assignment, TB(1), dominates the Slack Stealer.

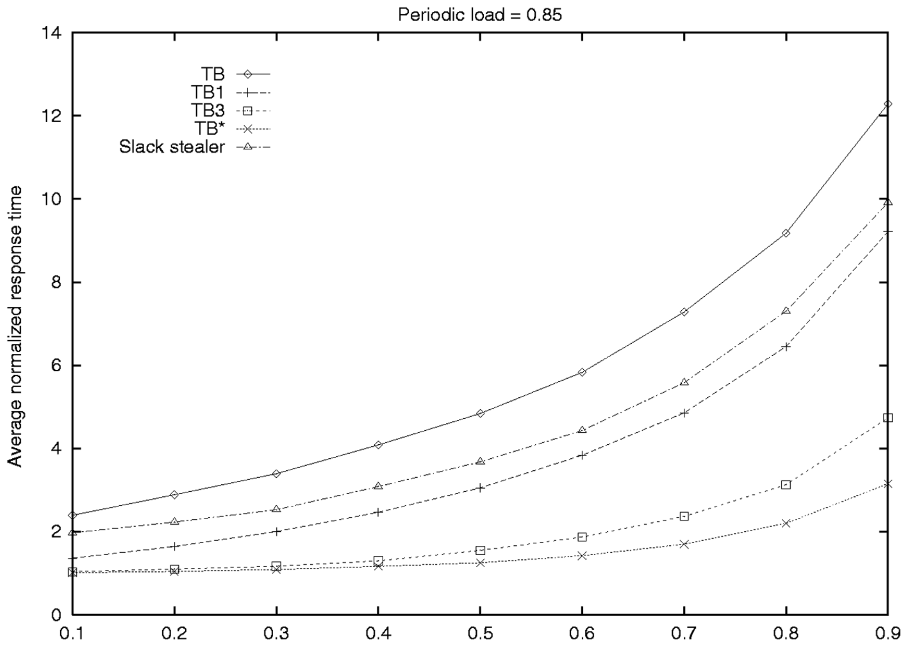


Fig. 9. Performance of the TB(N) against the Slack Stealer.

7.3 Resource Reservation

The problem caused by execution overruns can be solved by enforcing temporal isolation among tasks through a resource reservation mechanism in the kernel. This issue has been investigated both under RM and EDF and several approaches have been proposed in the literature. Under RM, the problem has been investigated by Mercer, Savage and Tokuda [26], who proposed a capacity reserve mechanism that assigns each task a given budget in every period and downgrades the task to a background level when the reserved capacity is exhausted.

Under EDF, a similar mechanism has been proposed by Abeni and Buttazzo, through the Constant Bandwidth Server [1]. This method also uses a budget to reserve a desired processor bandwidth for each task, but it is more efficient in that the task is not executed in background when the budget is exhausted. Instead a deadline postponement mechanism guarantees that the used bandwidth never exceeds the reserved value.

Resource reservation mechanisms are essential for preventing task interference during execution overruns, and this is particularly important in application tasks that have highly variable computation times (e.g., multimedia activities). In addition, isolating the effects of overruns within individual tasks allows relaxing worst case assumptions, increasing efficiency and performing much better quality of service control.

8 Conclusions

In this paper we compared the behavior of the two most famous policies used today for developing real-time applications: the Rate Monotonic (RM) and the Earliest Deadline First (EDF) algorithm. Although widely used, in fact, there are still many misconceptions about the properties of these two scheduling methods, mainly concerning their implementation complexity, the runtime overhead they introduce, their behavior during transient overloads, the resulting jitter, and their efficiency in handling aperiodic activities. For each of these issues we tried to confute some typical misconception and tried to clarify the properties of the algorithms by illustrating simple examples or reporting formal results from the existing real-time literature. In some cases, specific simulation experiments have also been performed to verify the overhead introduced by context switches and the effectiveness in improving aperiodic responsiveness.

In conclusion, the real advantage of RM with respect to EDF is its simpler implementation in commercial kernels that do not provide explicit support for timing constraints, such as periods and deadlines. Other properties typically claimed for RM, such as predictability during overload conditions, or better jitter control, only apply for the highest priority task, and do not hold in general. On the other hand, EDF allows a full processor utilization, which implies a more efficient exploitation of computational resources and a much better responsiveness of aperiodic activities. These properties become very important for embedded systems working with limited computational resources, and for multimedia systems, where quality of service is controlled through resource reservation mechanisms that are much more efficient under EDF. In fact, most resource reservation algorithms are implemented using service mechanisms similar to aperiodic servers, which have better performance under EDF.

Finally, both RM and EDF are not very well suited to work in overload conditions and to achieve jitter control. To cope with overloads, specific extensions have been proposed in the literature, both for aperiodic [9] and periodic [19,11] load. Also a method for jitter control under EDF has been addressed in [4] and can be adopted whenever needed.

References

1. L. Abeni and G. Buttazzo, "Integrating Multimedia Applications in Hard Real-Time Systems", *Proc. of the IEEE Real-Time Systems Symposium*, Madrid, Spain, December 1998.
2. N.C. Audsley, A. Burns, M. Richardson, K. Tindell and A. Wellings, "Applying New Scheduling Theory to Static Priority Preemptive Scheduling", *Software Engineering Journal* 8(5), pp. 284–292, September 1993.
3. T.P. Baker, Stack-Based Scheduling of Real-Time Processes, The Journal of Real-Time Systems 3(1), pp. 76–100, (1991).
4. S. Baruah, G. Buttazzo, S. Gorinsky, and G. Lipari, "Scheduling Periodic Task Systems to Minimize Output Jitter," Proceedings of the 6th IEEE International Conference on Real-Time Computing Systems and Applications, Hong Kong, December 1999.
5. S. K. Baruah, R. R. Howell, and L. E. Rosier, "Algorithms and Complexity Concerning the Preemptive Scheduling of Periodic Real-Time Tasks on One Processor," *Real-Time Systems*, 2, 1990.
6. E. Bini, G. C. Buttazzo and G. M. Buttazzo, "A Hyperbolic Bound for the Rate Monotonic Algorithm", *Proceedings of the* 13th *Euromicro Conference on Real-Time Systems*, Delft, The Netherlands, pp. 59–66, June 2001.
7. E. Bini and G. C. Buttazzo, "The Space of Rate Monotonic Schedulability", *Proceedings of the* 23rd *IEEE Real-Time Systems Symposium*, Austin, Texas, December 2002.
8. G. C. Buttazzo, "HARTIK: A Real-Time Kernel for Robotics Applications", Proceedings of the 14th IEEE Real-Time Systems Symposium, Raleigh-Durham, December 1993.
9. G. Buttazzo and J. Stankovic, "Adding Robustness in Dynamic Preemptive Scheduling", in Responsive Computer Systems: Steps Toward Fault-Tolerant Real-Time Systems, Edited by D. S. Fussell and M. Malek, Kluwer Academic Publishers, Boston, 1995.
10. G. Buttazzo and F. Sensini, "Optimal Deadline Assignment for Scheduling Soft Aperiodic Tasks in Hard Real-Time Environments," IEEE Transactions on Computers, Vol. 48, No. 10, October 1999.
11. G. Buttazzo, G. Lipari, M. Caccamo, and L. Abeni, "Elastic Scheduling for Flexible Workload Management", *IEEE Transactions on Computers*, Vol. 51, No. 3, pp. 289–302, March 2002.
12. M.I. Chen, and J.K. Lin, Dynamic Priority Ceilings: A Concurrency Control Protocol for Real-Time Systems, Journal of Real-Time Systems 2, (1990).
13. M.L. Dertouzos, "Control Robotics: the Procedural Control of Physical Processes," *Information Processing*, 74, North-Holland, Publishing Company, 1974.
14. P. Gai, G. Lipari, M. Di Natale, "A Flexible and Configurable Real-Time Kernel for Time Predictability and Minimal Ram Requirements," Technical Report, Scuola Superiore S. Anna, Pisa, RETIS TR2001–02, March 2001.

15. P. Gai, L. Abeni, M. Giorgi, G. Buttazzo, "A New Kernel Approach for Modular Real-TIme Systems Development", Proceedings of the 13th Euromicro Conference on Real-Time Systems, Delft (NL), June 2001.
16. K. Jeffay, "Scheduling Sporadic Tasks with Shared Resources in Hard-Real-Time Systems," Proceedings of IEEE Real-Time System Symposium, pp. 89–99, December 1992.
17. K. Jeffay, D.L. Stone, and D. Poirier, "YARTOS: Kernel support for efficient, predictable real-time systems," Real-Time Systems Newsletter, Vol. 7, No. 4, pp. 8–13, Fall 1991. Republished in Real-Time Programming, W. Halang and K. Ramamritham, eds. Pergamon Press, Oxford, UK, 1992.
18. M. Joseph and P. Pandya, "Finding Response Times in a Real-Time System," *The Computer Journal*, 29(5), pp. 390–395, 1986.
19. G. Koren and D. Shasha, "Skip-Over: Algorithms and Complexity for Overloaded Systems that Allow Skips," *Proc. of the IEEE Real-Time Systems Symposium*, 1995.
20. J. P. Lehoczky, L. Sha, and J. K. Strosnider, "Enhanced Aperiodic Responsiveness in Hard Real-Time Environments," *Proceedings of the IEEE Real-Time Systems Symposium*, pp. 261–270, 1987.
21. J. P. Lehoczky, L. Sha, and Y. Ding, "The Rate-Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behaviour", *Proceedings of the IEEE Real-Time Systems Symposium*, pp. 166–171, 1989.
22. J. P. Lehoczky and S. Ramos-Thuel, "An Optimal Algorithm for Scheduling Soft-Aperiodic Tasks in Fixed-Priority Preemptive Systems," Proceedings of the IEEE Real-Time Systems Symposium, 1992.
23. C.L. Liu and J.W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard real-Time Environment", *Journal of the ACM* 20(1), 1973, pp. 40–61.
24. J. Locke, "Designing Real-Time Systems", Invited talk, *IEEE International Conference of Real-Time Computing Systems and Applications* (RTCSA '97), Taiwan, December 1997.
25. P. Marti, G. Fohler, K. Ramamritham, and J.M. Fuertes, "Control performance of flexible timing constraints for Quality-of-Control Scheduling," Proc. of the 23rd IEEE Real-Time System Symposium, Austin, TX, USA, December 2002.
26. C.W. Mercer, S. Savage, and H. Tokuda, "Processor Capacity Reserves for Multimedia Operating Systems," Technical Report, Carnegie Mellon University, Pittsburg (PA), CMU-CS-93-157, May 1993.
27. M. Aldea Rivas and M. González Harbour, "POSIX-Compatible Application-Defined Scheduling in MaRTE OS," Euromicro Conference on Real-Time Systems (WiP), Delft, The Netherlands, June 2001.
28. L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," *IEEE Transactions on Computers*, 39(9), pp. 1175–1185, 1990.
29. B. Sprunt, L. Sha, and J. Lehoczky, "Aperiodic Task Scheduling for Hard Real-Time System," *Journal of Real-Time Systems*, 1, pp. 27–60, June 1989.
30. M. Spuri, and G.C. Buttazzo, "Efficient Aperiodic Service under Earliest Deadline Scheduling", *Proceedings of IEEE Real-Time System Symposium*, San Juan, Portorico, December 1994.
31. M. Spuri, G.C. Buttazzo, and F. Sensini, "Robust Aperiodic Scheduling under Dynamic Priority Systems", *Proc. of the IEEE Real-Time Systems Symposium*, Pisa, Italy, December 1995.
32. M. Spuri and G.C. Buttazzo, "Scheduling Aperiodic Tasks in Dynamic Priority Systems," *Real-Time Systems*, 10(2), 1996.

33. J. Stankovic and K. Ramamritham, “The Design of the Spring Kernel,” *Proceedings of the IEEE Real-Time Systems Symposium*, December 1987.
34. J.K. Strosnider, J.P. Lehoczky, and L. Sha, “The Deferrable Server Algorithm for Enhanced Aperiodic Responsiveness in Hard Real-Time Environments”, *IEEE Transactions on Computers*, 44(1), January 1995.
35. J. Stankovic, K. Ramamritham, M. Spuri, and G. Buttazzo, Deadline Scheduling for Real-Time Systems, Kluwer Academic Publishers, Boston-Dordrecht-London, 1998.
36. T.-S. Tia, J. W.-S. Liu, and M. Shankar, “Algorithms and Optimality of Scheduling Aperiodic Requests in Fixed-Priority Preemptive Systems,” Journal of Real-Time Systems, 10(1), pp. 23–43, 1996.

Translating Discrete-Time Simulink to Lustre*

Paul Caspi, Adrian Curic, Aude Maignan, Christos Sofronis, and Stavros Tripakis

VERIMAG
Centre Equation, 2, avenue de Vignate, 38610 Gières, France.
twww-verimag.imag.fr

Abstract. We present a method of translating discrete-time Simulink models to Lustre programs. Our method consists of three steps: type inference, clock inference and hierarchical bottom-up translation. In the process, we formalise typing and timing mechanisms of Simulink. The method has been implemented in a prototype tool called S2L. The tool has been used to translate part of an industrial automotive controller provided by Audi.

1 Introduction

Model-based design has been established as an important paradigm in the development of embedded systems today. The main principle of the paradigm is to use models all along the development cycle, from design to implementation. Using models, rather than, say, building prototypes, is essential for keeping the development costs manageable. However, models alone are not enough. They need to be accompanied by powerful reasoning tools: simulation, verification, synthesis, code generation, and so on. Automation here is the key: high system complexity and short time-to-market make model reasoning a hopeless task, unless it is largely automatised.

In previous work [4] we have proposed a design approach in three layers (Simulink, Lustre, TTA) and sketched how passing from one layer to another can be automatised. In this paper, we focus on the first two layers, Simulink and Lustre. We describe in detail a technique for translating discrete-time Simulink models into Lustre programs. In the process, we formalise important parts of the semantics of Simulink, such as typing and timing.[1]

The motivation for using Simulink as a high-level design model comes from the fact that this language has become a de-facto standard in many application domains, such as automotive control. We assume that the designer uses Simulink blocks to design a controller. The designer can benefit from Simulink's simulator and graphical capabilities

* Matlab, Simulink and Stateflow are Registered Trademarks of MathWorks, Inc. SCADE and Simulink Gateway are Registered Trademarks of Esterel Technologies, SA. This work has been supported in part by European IST projects "NEXT TTA" under project No IST-2001-32111 and "RISE" under project No IST-2001-38117.

[1] It can be claimed that "Simulink has no semantics". We take a different view, namely, that Simulink has many semantics (depending on user-configurable options), they are not formal, but can be defined as "what is given by the simulator" (e.g., what is observed in terms of inputs/outputs).

to check the correctness of the controller and measure its performance, in closed-loop with a plant, also modelled in Simulink.

As the first implementation step, we propose the translation of the Simulink model into the synchronous language Lustre [9]. There are a number of reasons motivating this choice. Lustre has been conceived primarily as a programming language, in particular for critical applications, and has several features desirable in such applications, such as formal semantics, strong typing and modularity. A number of formal validation tools exist for Lustre, such as the model-checker Lesar [17], the tester Lurette [18], plus a number of other tools for simulation, static-analysis and controller synthesis. Regarding implementation, a number of techniques and tools exist for generating C code from Lustre programs. In particular, SCADE, the graphical version of Lustre commercialised by Esterel Technologies, is endowed with a DO178B-level-A code generator which allows it to be used in highest criticality applications. SCADE has been used in important European avionic projects (Airbus A340-600, A380, Eurocopter) and is also becoming a de-facto standard in this field.

Simulink, on the other hand, started purely as a simulation environment and lacks many desirable features of programming languages. Simulink has a multitude of semantics (depending on user-configurable options), informally and sometimes partially documented. Although commercial code generators exist for Simulink (Real-time workshop from Mathworks, TargetLink from dSpace) these present major restrictions. For example, TargetLink does not generate code for blocks of the "Discrete" library of Simulink, but only for blocks of the dSpace library, and currently handles mono-periodic systems. Another issue not addressed by these tools is the preservation of semantics. Indeed, the relation between the behaviours of the generated code and those of the simulated model is unclear. Often, speed and memory optimisation is given more attention than semantic consistency.

In the rest of the paper, we present our translation method. Section 2 contains a more technical discussion of the main differences between Simulink and Lustre and states the main goals and limitations of our translation. In Section 3 we give a short description of Lustre. We assume that the reader is more or less familiar with Simulink (see `http://www.mathworks.com`www.mathworks.com). Important Simulink features are discussed in the corresponding sections of the paper. Sections 4, 5 and 6 present the translation method, which consists of three steps: type inference, clock inference and hierarchical translation. The algorithms have been implemented in a prototype tool. The tool has been used to translate an industrial Simulink model, provided by Audi. For reasons of space, this is not presented here and can be found in [3].

Related Work. [22] report on an approach to co-simulate discrete controllers modelled in the synchronous language Signal [8] along with continuous plants modelled in Simulink. [6] present tools for co-simulation of process dynamics, control task execution and network communication in a distributed real-time control system. [20] use a model-checker to verify a Simulink/Stateflow model from the automotive domain, however, they translate their model manually to the input language of the model-checker.

A number of approaches are based in extending Simulink with libraries of predefined blocks and then using Simulink as a front-end or simulator. The hybrid-system model-checker CheckMate uses such an approach [19,7]. [13] extend Simulink with the capability of expressing designs in the time-triggered language Giotto. [1] translate hybrid automata into Simulink and use its advanced numerical integration algorithms and user interface to perform simulation.

[10] report on translating Simulink to the *SPI model*, a model of concurrent processes communicating with FIFO queues or registers. The focus seems to be the preservation of value over-writing which can occur in multi-rate systems when a “slower” node receives input from a “fast” one.

[11] report on MAGICA, a type-inference engine for Matlab. The focus is on deriving information such as whether variables have real or imaginary values, array sizes for non-scalars, and so on.

[21] gives a formal semantics to Stateflow models using communicating pushdown automata. [16] presents the data model of Simulink and Stateflow as a UML class diagram.

2 The Goals of the Translation

Simulink and Lustre are models manipulating *signals* and *systems*. Signals are functions of time. Systems take as input signals and produce as output other signals. In Simulink, which has a graphical language, the signals are the “wires” connecting the various blocks. In Lustre, the signals are the program variables, called *flows*. In Simulink, the systems are the built-in blocks (e.g., adders, gains, transfer functions) as well as composite blocks, called *subsystems*. In Lustre, the systems are the various built-in operators as well as user-defined operators called *nodes*.

In the sequel, we use the following terminology. We use the term *block* for a basic Simulink block (e.g., adder, gain, transfer function) and the term *subsystem* for a composite Simulink block. We will use the term *system* for the root subsystem. We use the term *operator* for a basic Lustre operator and the term *node* for a Lustre node.

	Simulink	Lustre
Signals	“wires”	flows
Systems	adder, gain, unit delay, ..., subsystems	$+$, `pre`, `when`, `current`, ..., nodes

2.1 Differences of Simulink and Lustre

Both Simulink and Lustre are dataflow-like languages.[2] They both allow the representation of *multi-periodic sampled systems*. However, despite their similarities, they also differ in many ways:

- Lustre has a discrete-time semantics, whereas Simulink has a continuous-time semantics. It is important to note that even the “*Discrete library*” Simulink blocks produce piecewise-constant continuous-time signals.[3]

[2] The foundations of data-flow models were laid by Kahn [12]. Various such models are studied in [14,15,5].

[3] Thus, in general, it is possible to feed the output of a continuous-time block into the input of a discrete-time block and vice-versa.

- Lustre has a unique, precise semantics. The semantics of Simulink depends on the choice of a simulation method. For instance, some models are accepted if one chooses variable-step integration solver and rejected with a fixed-step solver.
- Lustre is a strongly-typed language with explicit types for each flow. In Simulink, explicit types are not mandatory, although they can be set (e.g., using the *data type converter* block, or an expression such as *single(1.2)* for the constant 1.2 of type *single*).
- Lustre is modular in certain aspects, while Simulink is not, in the sense that a Simulink model may contain implicit inputs (cf. 5.2).
- Hierarchy in Simulink is present both at the definition and at the execution levels. In Lustre, the execution graph is hierarchical (nodes calling other nodes), whereas the definition of nodes is not.

2.2 Translation Goals and Limitations

The ultimate objective of the translation is to automatise the implementation of embedded controllers as much as possible. We envisage a tool chain where controllers are designed in Simulink, translated to Lustre, and implemented on a given platform using the Lustre C code generator and a C compiler for this platform.

We only translate the discrete-time part of a Simulink model. Concretely, this means blocks of the "*Discrete*" library (such as "Unit-delay", "Zero-order hold", "Discrete filter" and "Discrete transfer function"), generic mathematical operators such as sum, gain, logical and relational operators, other useful operators such as switches, and, finally, subsystems or triggered subsystems. Discrete-time Simulink encompasses also blocks, like the "Read" and "Write" blocks, which exhibit side effects: their behaviour then depends on complex characteristics of the Simulink simulation algorithm and look poorly deterministic. As their use can be considered quite unsafe, they are not translated.

Of course, controllers can be modelled in continuous-time as well. This is typically done in control theory, so that analytic results for the closed-loop system can be provided (e.g., regarding its stability). Analytical results can also be provided using the sampled-data control theory.

In any case, the implemented controller must be discrete-time. How to obtain this controller is a control problem which is clearly beyond the scope of this paper.[4]

Other goals and limitations of our translation are the following.

(1) We aim at a translation method that preserves the semantics of Simulink. This means that the original Simulink model and the generated Lustre program should have the same observable output behaviour, given same inputs, modulo precisely defined conditions. Since Simulink semantics depends on the simulation method, we restrain ourselves only to one method, namely, "*solver: fixed-step, discrete*" and "*mode: auto*". We also assume that the Lustre program is run at the time period the Simulink model was simulated. Thus, an outcome of the translation must be the period at which the Lustre program *shall* be run (see also Section 5).

[4] According to classical text-books [2], there are two main ways of performing this task: either design a continuous-time controller and sample it or sample the environment and design a sampled controller.

(2) We do not translate *S-functions* or *Matlab functions*. Such functions are often helpful. On the other hand, they can also create side-effects, which is something to be avoided and contrary to the "functional programming" spirit of Lustre.

(3) As the Simulink models to be translated are in principle controllers embedded in larger models containing both discrete and continuous parts, we assume that for every input of the model to be translated (i.e., every input of the controller) the sampling time is explicitly specified. This also helps the user to see the boundaries of the discrete and the continuous parts in the original model.

(4) In accordance with the first goal, we want the Lustre program produced by the translator to type check if and only if the original Simulink model "type checks" (i.e., is not rejected by Simulink because of type errors). However, the behaviour of the type checking mechanism of Simulink depends on the simulation method and the "*Boolean logic signals*" flag (BLS). Thus, apart from the simulation method which must be set as mentioned in (1), we also assume that BLS is on. When set, BLS imposes that inputs and outputs of logical blocks (and, or, not) be of type *boolean*. Not only this is good modelling and programming practice, it also makes type inference more precise (see also Section 4) and simplifies the verification of the translated Simulink models using Lustre-based model-checking tools. We also set the "*algebraic loop*" detection mechanism of Simulink to the strictest degree, which rejects models with such loops. These loops correspond to cyclic data dependencies in the same instant in Lustre. The Lustre compiler rejects such programs.

(5) For reasons of traceability, the translation must preserve the hierarchy of the Simulink model as much as possible. Since Lustre has a textual language and nodes are declared sequentially rather than nested one inside another, the way to preserve hierarchy is by suitable naming.

(6) Current limitations of our tool are that it only handles scalar Simulink signals and does not translate "*virtual blocks*" such as "*Mux*".

It should also be noted that Simulink is a product evolving in time. This evolution has an impact on the semantics of the tool. For instance, earlier versions of Simulink had weaker type-checking rules. We have developed and tested our translation method and tool with Simulink 4.1 (Matlab 6 release 12). All examples given in this report refer to this version as well.

3 A Short Description of Lustre

A Lustre program essentially defines a set of equations:

$$\begin{aligned} x_1 &= f_1(x_1, ..., x_n, u_1, ..., u_m) \\ &\vdots \\ x_n &= f_n(x_1, ..., x_n, u_1, ..., u_m) \end{aligned}$$

where x_i are *internal* or *output* variables, and u_i are *input* variables. The variables in Lustre are also called *flows*. A flow is a partial function $x : N \to V_x$, where N is the set of natural numbers and V_x is the domain of x. N models *logical time*, counted in *instants*, $i = 0, 1, 2, ...$

The functions f_i are made up of usual arithmetic operations, control-flow operators (e.g., `if then else`), plus a few more operators, namely, `pre`, `->`, `when` and `current`.

`pre` is used to give *memory* (*state*) to a program. More precisely, $\texttt{pre}(x)$ defines a flow y, such that the value of y at instant i is equal to the value of x at instant $i - 1$ (at $i = 0$, the value of y is undefined).

`->` initialises a flow. If $z = x\texttt{->}y$, then the value of z is equal to the value of x for $i = 0$ and equal to the value of y for $i \geq 1$. The operator `->` is typically used to initialise a flow obtained using `pre`. For example, a counter is defined by the equation $x = 0\texttt{->}(\texttt{pre}(x) + 1)$.

`when` is used to *sample* a flow: x `when` b, where x is a flow and b is a boolean flow defined at the same instants as x, defines a flow y which is defined only at instants i where $b(i) = \texttt{t}$ (where t is the boolean value "true" and f is the boolean value "false"). For these instants, $y(i) = x(i)$.

`current` is used to extend the instants where a sampled flow y is defined. If $y = x$ `when` b and $z = \texttt{current}(y)$ then z is defined at all instants where x is defined. At all instants i where y is defined, $z(i) = y(i)$. At instants where y is undefined, z keeps its previous value.

Structure is given to a Lustre program by declaring and calling Lustre nodes, in much the same way as, say, C functions are declared and called. Here is an example:

```
node A(b: bool; i: int; x: real) returns (y: real);
  var j: int; z: real;
  let j = if b then 0 else i;
      z = B(j, x);
      y = if b then pre(z) else C(z);
  tel.
```

A is a node taking as inputs a boolean flow b, an integer flow i and a real flow x and returning a real flow y. A uses internal flows j and z (with usual scope rules). The body of A is declared between the `let` and `tel` keywords. A calls node B to compute z and node C to compute y (conditionally). Nodes B and C are declared elsewhere.

4 Type Inference

4.1 Types in Lustre

Lustre is a strongly typed language, meaning that every variable has a declared type and operations have precise type signatures. For instance, we cannot add an integer with a boolean or even an integer with a real.[5] Each flow has a type. Basic types in Lustre are *bool*, *int* and *real*.

4.2 Types in Simulink

In Simulink, types need not be explicitly declared. However, Simulink does have typing rules: some models are rejected because of type errors. The objective of the type inference

Table 1. Types of some Simulink blocks.

Constant$_\alpha$	$: \alpha, \alpha \in SimNum$
Adder	$: \alpha \times \cdots \times \alpha \rightarrow \alpha, \alpha \in SimNum$
Gain	$: \alpha \rightarrow \alpha, \alpha \in SimNum$
Relation	$: \alpha \times \alpha \rightarrow$ *boolean*$, \alpha \in SimNum$
Switch	$: \alpha \times \beta \times \alpha \rightarrow \alpha, \alpha, \beta \in SimNum$
Logical Operator	: *boolean* $\times \cdots \times$ *boolean* $\rightarrow$ *boolean*
Discrete Transfer Function	: *double* $\rightarrow$ *double*
Zero-Order Hold, Unit Delay	$: \alpha \rightarrow \alpha, \alpha \in SimT$
Data Type Converter$_\alpha$	$: \beta \rightarrow \alpha, \alpha, \beta \in SimT$
InPort, OutPort	$: \alpha \rightarrow \alpha, \alpha \in SimT$

step is to find the type of each Simulink signal, which will then be used as the type of the corresponding Lustre flow.

Basic "*data types*" in Simulink are: *boolean*, *double*, *single*, *int8*, *uint8*, *int16*, *uint16*, *int32*, *uint32*. Informally, the type system of Simulink can be described as follows. By default, all signals are *double*, except when: (1) the user explicitly sets the type of a signal to another type (e.g., by a *Data Type Converter* block or by an expression such as *single(23.4)*); or (2) a signal is used in an operation which demands another type (e.g., all inputs and outputs of *Logical Operator* blocks are *boolean*. A type error occurs when incompatible types are used in an operation (e.g., trying to add a *boolean* or trying to input an explicitly defined *int8* into a *Discrete Transfer Function* block).

We can formalise the above type system as follows. First, denote by $SimT$ the set of all Simulink types and let $SimNum = SimT - \{$*boolean*$\}$. Then, the type system of Simulink can be seen as a standard type system with type *classes* ($SimT$, $SimNum$, $\{$*boolean*$\}$). Every Simulink block has a (polymorphic) type, given in Table 1.

4.3 Type Inference

The type inference algorithm is a standard fix-point computation on the lattice shown in Figure 1. $\bot$ is the undefined type and $error$ means type error (model rejected).

The fix-point is computed on a set of equations derived according to Table 2. In the table, $x_1, x_2, ..., y$ are variables of the Simulink model and $x_1^T, x_2^T, ..., y^T$ are their corresponding types (which must be inferred by the fix-point computation). For each Simulink equation, the table gives the corresponding type equations. The order $\leq$ and the sup operator are for the lattice of Figure 1. For example, $double \leq single$ and $\sup(double, boolean) = error$. In the "Switch" block, x_2 is the input deciding (based on a threshold) whether the output y will be set to x_1 or x_3. Thus, the type of x_2 does not influence the types of x_1, x_3, y.

[5] Predefined *casting* operators such as *int2real* can be used in the latter case.

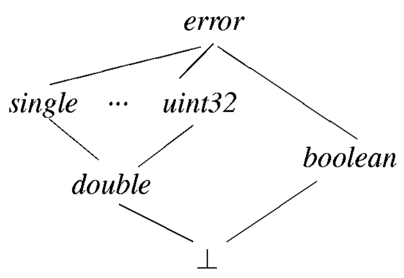


Fig. 1. The type lattice

Table 2. Type inference equations.

Simulink equation	Type equations
$y = \mathit{Adder}(x_1, ..., x_k)$	$y^T = x_1^T = \cdots = x_k^T = \sup(\mathit{double}, y^T, x_1^T, ..., x_k^T)$
$y = \mathit{Constant}_\alpha$	$y^T =$ if $y^T \leq \alpha$ then α else error
$y = \mathit{Data\ Type\ Converter}_\alpha(x)$	$y^T =$ if $y^T \leq \alpha$ then α else error
$y = \mathit{Unit\ Delay}(x)$	$x^T = y^T$
$y = \mathit{Zero\text{-}Order\ Hold}(x)$	$x^T = y^T$
$y = \mathit{Transfer\ Function}(x)$	$y^T = x^T =$ if $y^T \leq \mathit{double}$ then double else error
$y = \mathit{Relation}(x_1, x_2)$	$x_1^T = x_2^T = \sup(\mathit{double}, x_1^T, x_2^T), y^T = \mathit{boolean}$
$y = \mathit{Logical}(x_1, ..., x_k)$	$y^T = x_1^T = \cdots = x_k^T = \mathit{boolean}$
$y = \mathit{Switch}(x_1, x_2, x_3)$	$x_1^T = x_3^T = y^T = \sup(x_1^T, x_3^T, y^T)$

The right-hand sides of equations in Table 2 define a monotonic function in the type lattice. However, we can notice that the equations are not always oriented from inputs to outputs. Thus, it can be the case that several equations apply to the same variable, one when the variable acts as an output and some other ones when it acts as an input. This is not a conventional fix-point system of equations. However we can easily turn it to a conventional one; if several equations apply to the same variable:

$$\begin{aligned} x^T &= e_1 \\ ... &= ... \\ x^T &= e_n \end{aligned}$$

these equations are replaced by the single one:

$$x^T = \sup(e_1, \ldots, e_n)$$

Thus a least fix-point exists. Computing it takes $O(n^3)$ time, where n is the number of signals in the model. Indeed, there can be at most n equations (one for each basic block, each block having at least one output signal) of n variables. The fix-point is reached in $O(n)$ iterations. In each iteration, at most n equations are examined. Examining each equation takes $O(n)$ time.

Once the inference of Simulink types is performed, these types are mapped to Lustre types as follows: *boolean* is mapped to *bool*; *int8*, *uint8*, *int16*, *uint16*, *int32* and *uint32* are mapped to *int*; *single* and *double* are mapped to *real*; $\bot$ is mapped to *real*. The latter case is consistent with the fact that the default type in Simulink is *double*. Note that the type of some signals may indeed remain undefined ($\bot$) until the end. For example, this is the case of a system consisting of a single "unit-delay" block.

5 Clock Inference

5.1 Time in Lustre

As mentioned in Section 3, Lustre time is logical and is counted in discrete instants. Logical time means there is no a-priori notion of instant duration or time elapsing between two instants. Associated with each Lustre flow x is a boolean flow b_x, called the *clock* of x, and specifying the instants when x is defined: x is defined at instant i iff $b_x(i) = \texttt{t}$. For example, if x is defined at $i = 0, 2, 4, ...$ then $b_x = \texttt{t f t f} \cdots$.

Input variables are by definition defined at every instant: their clock is called the *basic* clock, represented by the boolean flow $\mathit{true} = \texttt{t t} \cdots$. "Slower" clocks are obtained from the basic clock using the `when` operator. For example, if `in` is an input then the flow x defined only at even instants can be generated by the following Lustre code:

```
cl1_2 = true -> not pre(cl1_2) ;
x     = in when cl1_2 ;
```

Clocks can be seen as extra typing information. The compiler ensures that the Lustre program satisfies a set of constraints on clocks, otherwise the program is rejected. For example, in the expression $x + y$, x and y must have the same clock, which is also the clock of the resulting flow. The set of these constraints and how to calculate clocks is called *clock calculus*. A simplified version of this calculus is shown in Table 3.

Table 3. Clock calculus of Lustre.

expression e	`clock`(e)	constraints
input x	*basic*	
$x + y$	`clock`(x)	`clock`(x) = `clock`(y)
`pre`(x)	`clock`(x)	
x `when` b	b	`clock`(x) = `clock`(b), b boolean
`current`(x)	`clock`(`clock`(x))	

It is worth mentioning that the Lustre compiler checks clock correctness in a *syntactic*, not *semantic* manner. Indeed, finding whether two boolean flows are semantically equivalent is an undecidable problem. Therefore, in an expression such as (x `when` b) + (y `when` b'), b and b' must be identical for the expression to clock-check.

5.2 Time in Simulink

Discrete-time Simulink signals are in fact piecewise-constant continuous-time signals. These signals can have associated timing information, called "*sample time*" and consisting of a *period* and an initial *phase*. Sample times may be set in blocks such as input ports, unit-delay, zero-order hold or discrete transfer functions. The sample time of a signal is derived from the block producing the signal and specifies when the signal is updated. A signal x with period π and initial phase θ is updated only at times $k\pi + \theta$, for $k = 0, 1, 2, ...$, that is, it remains constant during the intervals $[k\pi + \theta, (k + 1)\pi + \theta)$. Sample times also serve as an extra type system in Simulink: some models are rejected because of timing errors. An interesting example is shown in Figure 2. The sample times of inputs "In1" and "In2" are set to 2 and 3, respectively.[6] This model is rejected by Simulink. However, if the "Gain" block (a simple multiplication by 1) is removed, then the model is accepted! The explanation is given at the end of this section.[7]

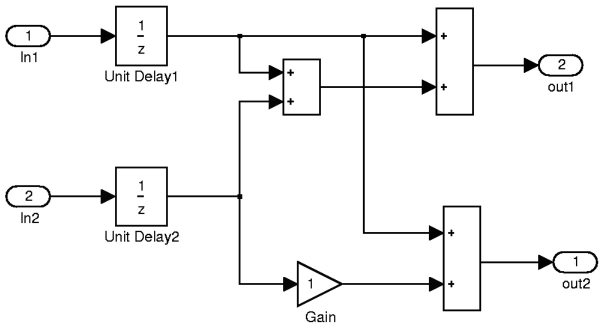


Fig. 2. A Simulink model producing a strange error.

Another timing mechanism of Simulink is by means of "*triggers*". Only subsystems (not basic blocks) can be triggered. A subsystem A can be triggered by a signal x (of any type) in three ways, namely, "*rising, falling*" or "*either*", which specify the time the trigger occurs depending on the direction with which x "crosses" zero. The sample time of blocks inside a triggered subsystem cannot be set by the user: it is "*inherited*" from the sample time T of the triggering signal. The sample times of the input signals must be all equal to T. The sample time of all outputs is T.

Timing in Simulink is not as modular as in Lustre, in the following sense. Lustre nodes do not have their own time: they are only activated at instants where their inputs are active. Consequently, a node B called by a node A cannot be active at instants when A is not active. This is true for triggered subsystems in Simulink as well. However,

[6] Unless otherwise mentioned, we assume that all phases are 0.

[7] Despite some Reviewers' remarks, the same anomaly persists when using Matlab Version 6.5.0.180913a Release 13 and Simulink version 5.0.1 (R13) dated 19-Sep-2002. But it only happens when setting simulation parameters to "fixed step", "auto".

Simulink allows a block inside a (non-triggered) subsystem to have any sample time, possibly smaller than the parent subsystem. Thus, the block can be active while its parent system is inactive. This can be considered a useful mechanism which allows so-called "oversampling". However, when used in conjunction with the "enabling" mechanism, it can give birth to real monsters whose behaviour is hardly understandable. The basic reason is that an extra, implicit input: the "absolute simulation time" — appears to flow through every block (but the triggered ones), even when this block is not enabled. We consider this a non-modular feature of Simulink.[8]

Another difference lies in how changes of timing are performed in the two languages. In Lustre, this can only be done using the `when` and `current` operators. In Simulink, the sample time of a signal can be changed using the "unit-delay" block or the "zero-order hold" block. In order to do this, however, the following rules must be obeyed:[9]

1. *"When transitioning from a slow to fast rate, a Unit Delay running at the slow rate must be inserted between the two blocks. When transitioning from a fast to a slow rate, a Zero Order Hold running at the slow rate must be inserted between the two blocks."*
2. *"Illegal rate transition found involving Unit Delay block ... When using this block to transition rates, the input must be connected to the slow sample time and the output must be connected to the fast sample time. The Unit Delay must have a sample time equal to the slow sample time and the slow sample time must be a multiple of the fast sample time. Also, the sample times of all destination blocks must be the same value."*
3. *"Illegal rate transition found involving Zero Order Hold block ... When using this block to transition rates, the input must be connected to the fast sample time and the output must be connected to the slow sample time. The Zero Order Hold must have a sample time equal to slow sample time and the slow sample time must be a multiple of the fast sample time. Also, the sample times of all source blocks must be the same value."*[10]

The second rule explains why the model of Figure 2 is rejected when the "Gain" block is present and accepted otherwise. Indeed, after computing the sample times according to the method described in the following section, we find that the output of "Unit Delay 2" has sample time 3. This output is "fed" to the "Gain" block, which also has sample time 3, and to the left-most "Adder" block, which has sample time 1 (the GCD of 2 and 3). This violates the second rule above.

[8] Still, we are able to translate such models in Lustre, by calling the parent node with the "faster" clock (on which the child block will run) and passing as parameter the "slower" clock as well.

[9] These rules are quoted from error messages produced by the Simulink tool.

[10] The third rule may seem strange since it implies that a zero-order hold block can have more than one inputs. In fact, this happens when the input to this block comes from a "Mux" block, thus, encoding a vector of signals.

5.3 Clock Inference

The objective of clock inference is to compute the period and phase of each Simulink signal and use this information (1) when creating the corresponding Lustre flows and (2) for defining the period at which the Lustre program must be run.

The clock inference algorithm is based on a fix-point computation on a lattice representing the sample times. The lattice is defined to be $ST = (\mathsf{Q} \times \mathsf{Q}) \cup \{\perp\}$, where Q is the set of non-negative rational numbers (of finite precision). A pair (π, θ) represents the sample time with period π and initial phase θ. $\perp$ represents the undefined sample time. The order in this lattice is defined as follows:

$$(\pi_1, \theta_1) \leq (\pi_2, \theta_2) \equiv \exists \kappa . \pi_1 = \kappa \pi_2 \wedge (\theta_1 = \theta_2 \vee (\theta_2 = 0 \wedge \exists \lambda . \theta_1 = \lambda \pi_2)) \quad (1)$$

The formula is slightly complicated because of the phases. If both phases are zero, then the order reduces to π_1 being a multiple of π_2 (κ and λ are integers). Naturally, we also set $\perp \leq (\pi, \theta)$, for all $(\pi, \theta) \in ST$. For example, $(8, 2) \leq (4, 2) \leq (2, 0) \leq (1, 0)$, but $(4, 1)$ and $(2, 0)$ are incomparable.

The order $\leq$ formalises the term "multiple" in the three Simulink rules quoted in Section 5.2. Thus, sample time s_1 is a "multiple" of sample time s_2 means $s_1 \leq s_2$.

The $\sup$ operator in this lattice corresponds essentially to a *GCD* (greatest common divisor) operation, complicated by the presence of phases. The precise definition is given in the lemma below.

Lemma 1. *Let* $(\pi_i, \theta_i) \in ST$, *for* $i = 1, 2$. *Then* $(\pi, \theta) = \sup((\pi_1, \theta_1), (\pi_2, \theta_2))$ *exists and is equal to:*

$$\pi = \begin{cases} gcd(\pi_1, \pi_2), & \textit{if}\ \theta_1 = \theta_2 \\ gcd(\pi_1, \pi_2, \theta_1, \theta_2), & \textit{otherwise} \end{cases} \qquad \theta = \begin{cases} \theta_1, \ \textit{if}\ \theta_1 = \theta_2 \\ 0, \quad \textit{otherwise} \end{cases}$$

For example,

$$\sup((2, 0), (3, 0)) = (1, 0) \qquad \sup((12, 3), (6, 3)) = (6, 3)$$
$$\sup((12, 6), (12, 0)) = (6, 0) \qquad \sup((12, 3), (12, 4)) = (1, 0)$$

The fix-point is computed on a set of equations derived according to Table 4. In the table, $x_1, x_2, ...$ are variables of the Simulink model and $x_1^C, x_2^C, ...$ are their corresponding sample-times. The equation for "*Triggered*" refers to the out-most triggered subsystem: we do not need to infer sample times inside a triggered subsystem, since they are all of type "inherited". The equation for "*DTL*" refers to blocks of the "Discrete" library. The user can set the sample time of such blocks, denoted st in the table. The default value of st is -1, meaning the sample time is "inherited" from the input.

Monotonicity ensures existence of a least fix-point. Although ST is an infinite lattice, the fix-point computation terminates. This is because there is a finite number of sample times in a given model. If the smallest fractional part among these sample times is 10^{-k} then it can be shown that the sample time $(10^{-k}, 0)$ is above all others according to the order $\leq$ defined above, thus acts as a "top" element for the given model.

Once the sample times are calculated, a number of rules need to be verified. First, we must verify a strange restriction of Simulink, which forbids sample times (π, θ) with

Table 4. Clock inference equations.

Simulink equation	Sample time equations
$y = Adder(x_1, ..., x_k)$	$y^C = \sup(x_1^C, ..., x_k^C)$
$(y_1, ..., y_l) = Triggered(s, x_1, ..., x_k)$	$s^C = y_1^C = \cdots = y_l^C = x_1^C = \cdots = x_k^C$
$y = DTL_{st}(x)$	$y^C =$ if $st = -1$ then x^C else st

$\theta \geq \pi$. If such a sample time is produced during the inference procedure, Simulink seems to apply the following procedure: (1) if $\theta \bmod \pi = 0$ (where mod is the modulo operator) then change θ to $\theta \bmod \pi$ (note that this preserves the $\leq$ order); (2) otherwise, reject the model.

Then, the three Simulink rules given in Section 5.2 are checked. This can be done following the structure of the model. For example, in a block F with input x and output y, if $x^C \leq y^C$ (i.e., x is "slower" than y), we must verify that x is the output of a unit-delay block, according to the first rule. In the opposite case, where $y^C \leq x^C$, F must be a zero-order hold block, according to the same rule.

6 Translation

The type and clock inference steps are independent and can be performed in any order. Once this is done, the translation itself is performed, in a hierarchical manner. The Simulink model is organised as a tree, where the children of a subsystem (or system) are the subsystems (or blocks) directly appearing in it. The translation is performed following this hierarchy in a bottom-up fashion (i.e., starting from the basic blocks). Since the Lustre node declarations are not hierarchical, in order to preserve the Simulink hierarchy, we name each Lustre node with the corresponding path of names in the Simulink tree. For example, if a subsystem B contained in a subsystem A will be translated to a Lustre node called "`NameofA_B`" where "`NameofA`" is the name of the Lustre node for A.

Simple Simulink blocks (e.g., adders, multipliers, the $\frac{1}{z}$ transfer function) are translated into basic Lustre operators. For example, an adder is simply translated into $+$ and $\frac{x}{z}$ is translated to the Lustre expression `init ->pre x` where `init` is the initial value specified in the dialog box of the unit delay.

More complex Simulink blocks (e.g., discrete filters) are translated into Lustre nodes. For example, the discrete transfer function $\frac{z+2}{z^2+3z+1}$ is translated into the Lustre node:[11]

```
node Transfer_Function_3(E: real) returns(S: real);
var Em_1, Em_2, Sm_1, Sm_2: real;
let S = 1.0*Em_1+2.0*Em_2-3.0*Sm_1-1.0*Sm_2 ;
    Em_1 = 0.0 -> pre(E) ;
    Em_2 = 0.0 -> pre(Em_1) ;
    Sm_1 = 0.0 -> pre(S) ;
    Sm_2 = 0.0 -> pre(Sm_1) ;
```

[11] This translation comes very directly from the translation of the $\frac{x}{z}$ expression, by usual algebraic manipulations.

```
tel.
```

A Simulink subsystem is translated into a Lustre node, possibly containing calls to other nodes. The Lustre node has the same inputs and outputs as the Simulink subsystem, plus, sometimes, the clock of some of its inputs (this is done for modularity). Here is an example of such a translation. Consider the Simulink model shown in Figure 3, with subsystems A and B. The Lustre code generated for this example is as shown below:

```
node A(A_in1, A_in2, A_in3 : real) returns (A_out1, A_out2 : real);
let A_out1 = B(A_in1 , A_in2);
    A_out2 = ...
tel.

node B(B_in1, B_in2 : real) returns (B_out : real);
...
```

An example where it is necessary to pass clock information as input to the Lustre node is given in [3].

Simulink signals are mapped into Lustre flows. When a block changes the sample time of a signal, the appropriate `when` and `current` operations need to be applied. This must also be done for triggered subsystems, as well as in cases where a subsystem receives input signals with different sample times.

For example, consider a “Zero-Order-Hold” block and assume that the sample time of input x is 1 and the sample time set to the zero-order hold block is 2. Then, the sample time of the output y is also 2 and the generated Lustre code is as follows:

```
cl1_2 = true -> not pre(cl1_2) ;
y     = x when cl1_2 ;
```

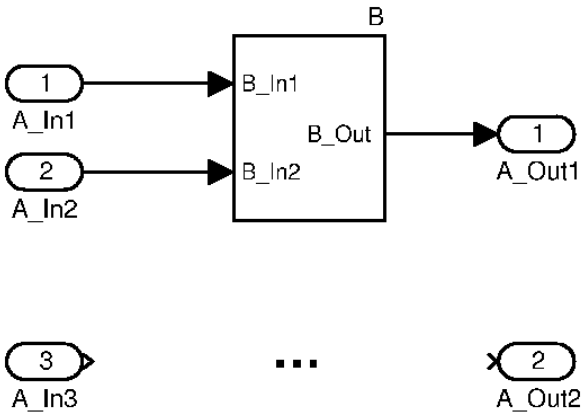


Fig. 3. Simulink system A with subsystem B.

We implemented the algorithms described above in a prototype tool, called S2L. The tool is written in Java. It takes as input a Simulink model (“.mdl” file) and produces a

Lustre program (".lus" file) or an error message in case the input model contains type or timing errors. S2L uses an intermediate representation of the Simulink model, in XML. This is done to facilitate evolution of the tool as the syntax of the ".mdl" file changes.

7 Conclusions

We have presented a method for translating a discrete-time subset of Simulink models into Lustre programs. The translation is done in three steps: type and clock inference, followed by a hierarchical bottom-up translation. We have implemented the method in a tool called S2L and applied it to a controller used in Audi cars. The interest of our tool is that it opens the way to the use of formal and certified verification and implementation tools attached to the Lustre tool chain. Also, in the process of translation, we formalised (for the first time, to our knowledge) the typing and timing mechanisms of Simulink.

Perhaps the most significant drawback of our approach is its dependency on syntax and semantics of Simulink models. New versions of Simulink appear as often as every six months and sometimes major changes are made with respect to previous versions. This situation seems difficult to avoid given the relative "monopoly" of Simulink/Stateflow in the control design landscape. Another weakness of our tool is its incompleteness: several unsafe constructs of Simulink are not translated. Yet this can be seen as the price to pay for having a sound translation.

We are currently enhancing the capabilities of S2L. Features such as "masked" subsystems are already handled. We are now studying the translation of some "virtual" blocks such as "Mux". We are also currently studying the translation of Stateflow. Stateflow presents a number of semantic problems, such as behaviour depending on how the model is graphically drawn, or non-termination of a simulation step. Our first goal is to identify a subset of Stateflow avoiding these problems.

References

1. E. Asarin, T. Dang, and J. Esteban. Simulation of hybrid automata using Matlab/Simulink. Technical report, Verimag, 2003.
2. K.J. Åström and B. Wittenmark. *Computer Controlled Systems*. Prentice-Hall, 1984.
3. P. Caspi, A. Curic, A. Maignan, C. Sofronis and S. Tripakis. Translating Discrete-Time Simulink to Lustre. Verimag Research Report, July 2003.
4. P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis, and P. Niebert. From Simulink to SCADE/Lustre to TTA: a layered approach for distributed embedded applications. In *Languages, Compilers, and Tools for Embedded Systems (LCTES'03)*. ACM, 2003.
5. P. Caspi and M. Pouzet. Synchronous Kahn networks. In *ACM SIGPLAN International Conference on Functional Programming*, pages 226–238, 1996.
6. Anton Cervin, Dan Henriksson, Bo Lincoln, and Karl-Erik Årzén. Jitterbug and Truetime: Analysis tools for real-time control systems. In *Proceedings of the 2nd Workshop on Real-Time Tools*, Copenhagen, Denmark, August 2002.
7. A. Chutinan and B.H. Krogh. Computational techniques for hybrid system verification. *IEEE Trans. Automatic Control*, 48(1), 2003.
8. P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire. Programming real-time applications with signal. *Proc. of the IEEE*, 79(9):1321–1336, 1991.

9. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9), September 1991.
10. M. Jersak, D. Ziegenbein, F. Wolf, K. Richter, R. Ernst, F. Cieslog, J. Teich, K. Strehl, and L. Thiele. Embedded system design using the SPI workbench. In *Proc. of the 3rd International Forum on Design Languages*, 2000.
11. P.G. Joisha and P. Banerjee. The MAGICA type inference engine for MATLAB. In *Compiler Construction (CC)*. LNCS 2622, Springer, 2003.
12. G. Kahn. The semantics of a simple language for parallel programming. In *Proc. of the IFIP Congress*, 1974.
13. C.M. Kirsch, M.A. Sanvido, T.A. Henzinger, and W. Pree. A Giotto-based helicopter control system. In *EMSOFT'02*. LNCS 2491, Springer, 2002.
14. E. Lee and D. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75, 1987.
15. E. Lee and T. Parks. Dataflow process networks. *Proceedings of the IEEE*, pages 773–799, May 1995.
16. S. Neema. Simulink and Stateflow data model. Technical report, ISIS, Vanderbilt University, 2001.
17. C. Ratel, N. Halbwachs, and P. Raymond. Programming and verifying critical systems by means of the synchronous data-flow programming language Lustre. In *ACM-SIGSOFT Conference on Software for Critical Systems*, 1991.
18. P. Raymond, D. Weber, X. Nicollin, and N. Halbwachs. Automatic testing of reactive systems. In *19th IEEE Real-Time Systems Symposium*, Madrid, Spain, December 1998.
19. B.I. Silva, K. Richeson, B.H. Krogh, and A. Chutinan. Modeling and verifying hybrid dynamical systems using CheckMate. In *ADPM*, 2000.
20. S. Sims, K. Butts, R. Cleaveland, and S. Ranville. Automated validation of software models. In *ASE*, 2001.
21. A. Tiwari. Formal semantics and analysis methods for Simulink Stateflow models. Technical report, SRI International, 2002.
22. S. Tudoret, S. Nadjm-Tehrani, A. Benveniste, and J.-E. Stromberg. Co-simulation of hybrid systems: Signal-Simulink. In *FTRTFT*, 2000.

Minimizing Variables' Lifetime in Loop-Intensive Applications

Noureddine Chabini and Wayne Wolf

Computer Engineering, Department of Electrical Engineering, Princeton University
Mail address: Engineering Quadrangle, Olden Street, Princeton, NJ 08544, USA.
{nchabini,wolf}@ee.princeton.edu

Abstract. In this paper, we address a set of research problems regarding loop-intensive applications. The first problem consists of minimizing variables' lifetime under timing constraints. Any variable that is alive for more than one iteration must be saved by for instance storing it into a register. The second problem is derived from the first one, and consists of balancing variables' lifetime for a target total number of registers and under timing constraints. For the third problem, we focus on minimizing variables' lifetime in the context of software pipelining in the case of one loop as well as nested loops. We provide methods to solve these three problems, and show that a set of them have polynomial run-time. Once these methods are used, one may need to solve the problem of generating the transformed code and reducing its size. We provide algorithms to solve this fourth problem. Designers face these problems during hardware-software co-design, and in designing embedded systems as well as system-on-chip. Solving these problems is also useful in low power design. We exercise some of these methods on known benchmarks, and provide obtained numerical results that show their effectiveness.
We solve these problems using techniques related to retiming, an algorithm originally developed for hardware optimization.

1 Introduction

Loops are found in many real applications. This kind of applications can be found in the domain of image processing as well as in the domain of digital signal processing. For instance, an image is represented as a two dimensional array where each cell of the array is used to store one pixel of the image. Processing the image then transforms to doing some processing on the array using for instance "for" loops to access the array's elements.

Multimedia applications are another example of loop-intensive real applications. These applications require a high processor speed. Since some of them are portable applications, reducing their power consumption is required to increase the battery life. Note that prolonging battery life (for portable systems) is became a product differentiator in the market.

Loop-intensive applications can be classified as computational intensive, data intensive, or computational-and-data intensive applications. There are three ways to implement these applications: hardware, software or hardware-and-software implementations. To satisfy performance requirements, hardware implementations are the most targeted for the first and third classes, and imply a high design cost. Due to the low cost of the implementation and the ease of changing it, software implementations are targeted whenever they can satisfy the performance requirements. For all these three ways of implementing loop-intensive applications, there is a need for some optimizations as shown by Fig. 1 from [2].

One common optimization to these three kinds of implementations is the minimization of variables' lifetime. A variable is alive from the time it is produced until the last time it is used. For the case of software implementations for instance, a variable that is alive for long time may be transferred from the register where it is stored to the main memory if the number of simultaneous alive variables is greater than the number of the available registers; this implies an increase of run-time as well as power consumption due to the transfer between registers and the main memory. The size of memory has an implicit impact on both performance and power consumption. Also, it has an impact on the cost in dollars for the case of hardware implementations. The total size of memories must be reduced in the case of embedded systems as well as of system-on-chip. Sometimes, there is more than one way to minimize variables' lifetime for a target total size of memories. In this case, one needs to chose the way that may allow to increase performance and/or to reduce power consumption.

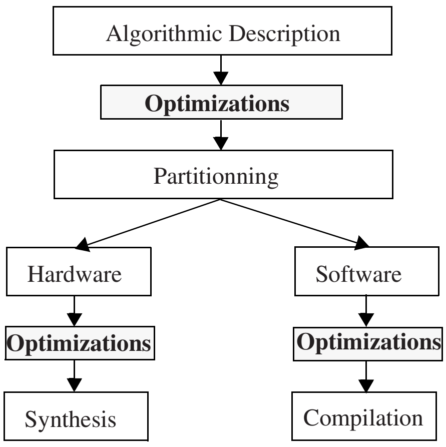


Fig. 1. Loop optimizations in the co-design flow.

In this paper, we address four research problems related to the minimization of variables' lifetime in loop-intensive applications. We first address the problem of minimizing variables' lifetime under timing constraints. We then investigate the problem of balancing variables' lifetime for a target total number of registers and under timing constraints. These two problems can be seen as platform independent, and hence approaches to solve them allow a source-to-source code transformation. Besides these two problems, we address the problem of minimizing variables' lifetime in the context of software pipelining in the case of one loop as well as nested loops. The latter problem is platform dependent. We provide methods to solve these three problems, and show that a set of them have polynomial run-time. Once these methods are used, one may need to solve the problem of generating the transformed code and reducing its size, which is useful for memory-size-constrained systems such as system-on-chip and embedded systems. We provide algorithms to solve this fourth problem. We exercise some of these methods on known benchmarks. Obtained numerical results show that the sum of variables' lifetime can be reduced by factors as high as 33.33%.

Basic retiming has been proposed in [1] as an optimization technique for synchronous sequential circuits. This technique changes the location of registers in the design in order to achieve one of the following goals: (*i*) minimizing the clock period, (*ii*) minimizing the number of registers, or (*iii*) minimizing the number of registers for a target clock period.

There is a relation ship between a register in the context of hardware implementations and an iteration in the context of software implementations. In the context of software implementations, moving for instance one register from the inputs of an operator to its outputs transforms to (*i*) delaying by one iteration the consummation of the result produced by this operator, and (*ii*) advancing by one iteration the consummation of the operands of that operator. Based on this relation ship, we exploit basic retiming to develop the methods for solving the four presented problems above. Using basic retiming to optimize loops is not new as we will present later in the paper. However, it is the first time that basic retiming is used in this paper to solve these problems that are to the best of our knowledge not addressed by any other paper.

2 Preliminaries

2.1 Cyclic Graph Model

In this paper, we are interested in "for"-type loops as the one in Fig. 2 (a). We assume that the body of the loop is constituted by a set of computational and/or assignment instructions only (i.e., no conditional or branch instruction like for instance *if-then-else* is inside the body).

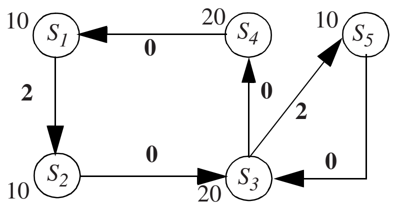
We model a loop by a directed cyclic graph $G = (V, E, d, w)$, where V is the set of instructions in the loop body, and E is the set of edges that represent data dependencies. Each vertex v in V has a non-negative integer execution delay $d(v) \in N$. Each edge $e_{u,v} \in E$, from vertex $u \in V$ to vertex $v \in V$, has a weight $w(e_{u,v}) \in N$, which means that the result produced by u at any iteration i is consumed by v at iteration $(i + w(e_{u,v}))$.

Fig. 2 presents a simple loop and its directed cyclic graph model. For Fig. 2 (b), the execution delay of each instruction S_i is specified as a label on the left of each node of the graph, and the weight $w(e_{u,v})$ of each arc $e_{u,v} \in E$ is in bold. For instance, the execution delay of S_1 is 10 and the value 2 on the arc e_{S_1,S_2} means that instruction S_2 at any iteration i uses the result produced by instruction S_1 at iteration $(i-2)$.

```
main () {
   int  a[U], b[U], c[U], d[U], e[U], i;

   for (i=2; i<= U; i++) {
      S1: a[i] = 1 + d[i];
      S2: b[i] = 1 + a[i-2];
      S3: c[i] = b[i] * e[i];
      S4: d[i] = 2 * c[i];
      S5: e[i] = 1 + c[i-2];
   }
}
```

(a) Simple loop.



(b) Directed cyclic graph model.

Fig. 2. A simple loop and its directed cyclic graph model.

We also address perfect nested “for”-type loops in this paper. For this kind of nested loops, no computational or assignment instructions must be between two successive “for”-type nested loops. All the instructions must be in the body of the innermost loop of the nest. We assume that this body does not contain conditional or branch instruction like *if-then-else* for instance. This kind of nested loops can be modeled as we did for the case of one loop. The weight of each arc in the graph transforms to a vector with a dimension equal to the number of loops in the nest. For an example of perfect nested “for”-type loops and its cyclic graph model, the reader is referred to [8].

2.2 Introduction to Basic Retiming

A synchronous sequential design can be modeled as a directed cyclic graph as we did for loops in Section 2.1. Instructions become computational elements of the design, edges become wires, and $w(e_{u,v})$ s become the number of registers on the wire between *u* and *v*.

Let $G = (V, E, d, w)$ be a synchronous sequential digital design. Basic retiming (or retiming for short in this paper) r [1] is defined as a function $r : V \rightarrow Z$, which transforms G to a functionally equivalent synchronous sequential digital design $G_r = (V, E, d, w_r)$. The set Z represents natural integers.

The weight of each edge $e_{u,v}$ in G_r is defined as follows:

$$w_r(e_{u,v}) = w(e_{u,v}) + r(v) - r(u), \forall e_{u,v} \in E \cdot \tag{1}$$

Since the weight of each edge in G_r represents the number of registers on that edge, then we must have:

$$w_r(e_{u,v}) \geq 0, \forall e_{u,v} \in E \cdot \tag{2}$$

Any retiming r that satisfies Equation (2) is called a valid retiming. From expressions (1) and (2) one can deduce the following inequality:

$$r(u) - r(v) \leq w(e_{u,v}), \forall e_{u,v} \in E \cdot \tag{3}$$

Let us denote by $P(u, v)$ a path from node *u* to node *v* in *V*. Equation (1) implies that for every two nodes *u* and *v* in *V*, the change in the register count along any path $P(u, v)$ depends only on its two endpoints:

$$w_r(P(u, v)) = w(P(u, v)) + r(v) - r(u),\ \forall u, v \in V, \tag{4}$$

where:

$$w(P(u, v)) = \sum_{e_{x,y} \in P(u,v)} w(e_{x,y}). \tag{5}$$

Let us denote by $d(P(u, v))$ the delay of a path $P(u, v)$ from node u to node v. $d(P(u, v))$ is the sum of the execution delays of all the computational elements that belong to $P(u, v)$.

A 0-weight path is a path such that $w(P(u, v)) = 0$. The minimal clock period of a synchronous sequential digital design is the longest 0-weight path. It is defined by the following equation:

$$\Pi = Max_{\forall u, v \in V}\{d(P(u, v)) \mid (w(P(u, v)) = 0)\}. \tag{6}$$

Two matrices called *W* and *D* are very important to the retiming algorithms. They are defined as follows [1]:

$$W(u, v) = Min\{(w(P(u, v)))\},\ \forall u, v \in V, \text{ and} \tag{7}$$

$$D(u, v) = Max\{d(P(u, v)) \mid w(P(u, v)) = W(u, v)\},\ \forall u, v \in V. \tag{8}$$

The matrices *W* and *D* can be computed as explained in [1].

One application of retiming is to minimize the clock period of synchronous sequential digital designs. For instance, by thinking on Fig. 2 (b) as a synchronous sequential design, the clock period of that design is $\Pi = 60$, which is equal to the sum of execution delays of computational elements $v_{i = 1, ..., 4}$ (i.e., $\Pi = 60 = d(v_2) + d(v_3) + d(v_4) + d(v_1)$). However, we can obtain $\Pi = 30$ if we apply the following retiming vector $\{2, 1, 1, 2, 0\}$ to the vector of nodes $\{1, 2, 3, 4, 5\}$ in G. The retimed graph G_r is presented by Fig. 3.

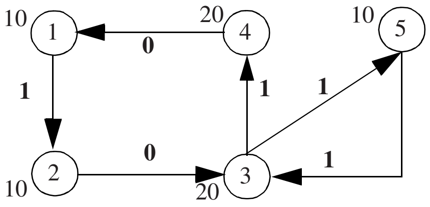


Fig. 3. A retimed graph with $\Pi = 30$.

For the purpose of this paper, we extract from [1] the following theorem, which is also proved in [1].

Theorem 1 : *Let $G = (V, E, d, w)$ be a synchronous digital design, and let Π be a positive real number. Then there is a retiming r of G such that the clock period of the resulting retimed design G_r is less than or equal to Π if and only if there exists an assignment of integer value r(v) to each node v in V such that the following conditions are satisfied: (1) $r(u) - r(v) \le w(e_{u,v}),\ \forall e_{u,v} \in E$, and (2) $r(u) - r(v) \le W(u, v) - 1,\ \forall u, v \in V$ such that $D(u, v) > \Pi$.*

❑

Once a retiming defined on Z is computed for solving a target problem, it can then be transformed to a non-negative retiming without impact on the solution of the problem. Indeed, let $m = Min\{r(v), \forall v \in V\}$. Then the function g defined as $g(v) = r(v) - m$, $\forall v \in V$, is a non-negative retiming (i.e., g takes non-negative values). In the rest of the paper, we consider non-negative retiming only.

3 Minimizing the Sum of Variables' Lifetime under Timing Constraints

In this section, we focus on minimizing the sum of the lifetime of all variables in the loop's body of unidimensional loops under a target latency constraint. This latency is defined here as the time required to execute, using unlimited number of computational elements, all the instructions of the loop's body at any iteration of the loop. We denote by Π the target latency. We represent the loop by a directed cyclic graph G as introduced in Section 2.1.

Assume that we want $\Pi = 60$ for the simple loop in Fig. 2. In this case, we have that variables produced by instructions S_1 and S_3 are alive for two iterations. The sum of variables' lifetime in this case is 4 iterations, and hence 4 registers are required to store these variables. Can one reduce the sum of variables' lifetime while still having $\Pi = 60$? The response is yes if we do not have constraints on the number of computational elements. Indeed, by thinking on G as a synchronous digital design, a retiming vector $\{2, 0, 0, 2, 0\}$ can be applied to obtain a sum of variables' lifetime equal to two. The resulted retimed graph is presented by Fig. 4. In this case, only variable produced by S_3 is alive for 2 iterations, which now requires 2 registers compared to 4 previously.

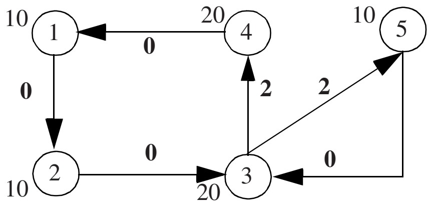


Fig. 4. Retiming Fig. 2 (b) using the retiming vector $\{2' 0' 0' 2' 0\}$.

By applying retiming on the graph modeling the loop, it is then possible to reduce the sum of variables' lifetime while still having a target latency Π. But, can this be always possible? If yes, how one can derive an algorithm to do it?

The response to the first question is yes, and our focus in the rest of this section is to provide an algorithm as a response to the second question. The proposed algorithm take the form of a mathematical linear program.

By thinking on G as a synchronous digital design and on Π as its clock period, one can derive from Theorem 1 the following theorem that will be used to derive the mathematical linear program.

Theorem 2 : *Let $G = (V, E, d, w)$ be a synchronous digital design, and let Π be a positive real number. Then there is a retiming r of G such that the clock period of the resulting retimed design G_r is less than or equal to Π if and only if there exists an assignment of non-negative integer values to r(v) and b(v) for each node v in V such that the following conditions are satisfied: (1) $b(u) + r(u) - r(v) \geq w(e_{u,v}), \forall e_{u,v} \in E$, (2) $r(v) - r(u) \geq -w(e_{u,v}), \forall e_{u,v} \in E$, and (3) $r(v) - r(u) \geq 1 - W(u, v), \forall u, v \in V$ such that $D(u, v) > \Pi$.*

❑

Proof: It can be easily derived from Theorem 1. Inequality (1) can always be satisfied by first applying Theorem 1.

❑

The second and third inequalities of Theorem 2 are the same as in Theorem 1. But, what is the objective from its first inequality? It allows to reduce the lifetime of the variable produced by the instruction at the node *u* by reducing the variable upper bound $b(u)$.

The system of inequalities of Theorem 2 may have an infinite number of possible solutions (regarding the unknown variables $b(v)$ and $r(v)$). Hence, to find a solution that allows to minimize the sum of the lifetime of variables in the loop's body, one can constraint this system by minimizing the objective function:

$$\sum_{v \in V} b(v), \tag{9}$$

which leads to the Integer Linear Program (ILP) constituted by (10)-(14).

Theorem 3 : *The ILP (10)-(14) has always a solution, and its optimal solution can be found in polynomial run-time.*

❑

Proof: If Theorem 1 applies, then Theorem 2 applies too. In this case, ILP (10)-(14) has always a solution since Theorem 2 applies. It can be optimally solved in polynomial run-time. Indeed, The right hand sides of inequalities (11)-(13) are always integers. From linear programming theory [12], we know that if the constraint matrix *A* of an ILP is totally unimodular and if the right hand sides of inequalities (as (11)-(13) in our case) are integers then the ILP and its relaxation obtained by ignoring the constraints integers (ignoring (14) in our case) have the same optimal solution. This relaxed ILP is a linear program, and hence it can be solved optimally in polynomial run-time by using for instance methods [13][14]. The constraint matrix *A* of ILP (10)-(14) is totally unimodular since it satisfies the following theorem from [15].

Theorem 4 : *A matrix X is totally unimodular if and only if for every square eulerian submatrix Y of X, we have that the sum of the entries of Y divides by 4.*

❑

$$Minimize\left(\sum_{v \in V} b(v)\right) \tag{10}$$

Subject to:

$$b(u) + r(u) - r(v) \geq w(e_{u,v}), \forall e_{u,v} \in E \tag{11}$$

$$r(v) - r(u) \geq -w(e_{u,v}),\ \forall e_{u,v} \in E \tag{12}$$

$$r(v) - r(u) \geq 1 - W(u,v),\ \forall u, v \in V \textit{ such that } D(u,v) > \Pi \tag{13}$$

$$b(v) \text{ and } r(v) \text{ are non-negative integers, } \forall v \in V \tag{14}$$

4 Balancing the Lifetime of Variables for a Target Sum of Variables' Lifetime and Timing Constraints

Again, a loop is modeled as a directed cyclic graph G as introduced in Section 2.1. In this section, we address a problem derived from the one targeted in Section 3. The problem consists in balancing the lifetime of variables under a target latency Π (as we defined in Section 3) while keeping the sum of variables' lifetime less than or equal to a target value say B. One can fix B to the minimal value of the objective function in (9) by solving (10)-(14). The problem can be viewed as the problem of maximizing the number M of edges of non-zero weight in G under a target latency Π while satisfying $\sum_{v \in V} b(v) \leq B$.

Is it important to solve this problem? It is important for instance in the case of hardware implementations that require reduction of power consumption. Indeed, an edge of non-zero weight means we have at least one register on that edge. Having registers distributed on the output of many computational elements allows to reduce the power consumption due to switching activities. In the case of software implementations, it allows to reduce the length of critical (or pseudo critical) paths, and hence to increase the number of instructions that can simultaneously be executed.

Let us give an example to show how solving this problem can help in optimizing some figures of merit. Let us use Fig. 2. Assume that we want *(i)* $\Pi = 30$ and *(ii)* $\sum_{v \in V} b(v) \leq 3$. By thinking again on G as a synchronous digital design and applying retiming on it, we can obtain Fig. 3, which satisfies both *(i)* and *(ii)*. In the case of hardware implementations, Fig. 3 is preferable since, besides reducing the switching activities, it allows to apply supply voltage scaling for node 5 that is off critical paths; hence, to reduce power consumption.

How to solve the problem? To solve it, we again use basic retiming and provide an integer linear program as we did in Section 3. Although ILPs are NP-hard in general, we show that this ILP can be solved in polynomial run-time as we did in Section 3.

After retiming G, let G_r be the resulted retimed graph. For every arc $e_{u,v}$ in G_r, let $m(e_{u,v})$ be a 0-1 unknown variable defined as follows. $m(e_{u,v})$ takes the value 1 if $w_r(e_{u,v}) \geq 1$, and 0 otherwise. Based on the definition of $m(e_{u,v})$'s and Inequality (2), we then have:

$$0 \leq m(e_{u,v}) \leq 1,\ \forall e_{u,v} \in E \text{, and} \tag{15}$$

$$m(e_{u,v}) \leq w_r(e_{u,v}),\ \forall e_{u,v} \in E. \tag{16}$$

Using Equation (3), we can transform (16) to:

$$m(e_{u,v}) + r(u) - r(v) \leq w(e_{u,v}),\ \forall e_{u,v} \in E. \tag{17}$$

From Theorem 2, we can derive the following theorem that we will use to derive a method to solve the problem.

Theorem 5 : *Let $G = (V, E, d, w)$ be a synchronous digital design, and let Π be a positive real number and B a non-negative integer. Then there is a retiming r of G such that the clock period of the resulted retimed design G_r is less than or equal to Π if and only if there exists an assignment of non-negative integer values to r(v) and b(v) for each node v in V, and 0/1 value $m(e_{u,v})$ to each arc $e_{u,v}$ in E, such that the following conditions are satisfied: (1) $0 \le m(e_{u,v}) \le 1, \forall e_{u,v} \in E$, (2) $m(e_{u,v}) + r(u) - r(v) \le w(e_{u,v}), \forall e_{u,v} \in E$, (3) $r(u) - r(v) \le W(u,v) - 1, \forall u, v \in V$ such that $D(u,v) > \Pi$, (4) $b(u) + r(u) - r(v) \ge w(e_{u,v}), \forall e_{u,v} \in E$ and (5) $\sum_{v \in V} b(v) \le B$.*

❑

Proof: It is omitted due to space limitation, but can be easily derived from Theorem 1.

❑

The number of edges of non-zero weight in G_r is:

$$M = \sum_{e_{u,v} \in E} m(e_{u,v}) . \tag{18}$$

The system of Inequalities in Theorem 5 may have many possible solutions. To have a solution that maximizes *M*, one can add formula

$$Maximize\left(\sum_{e_{u,v} \in E} m(e_{u,v})\right) \tag{19}$$

to that system as well as integrality constraints on the unknown variables. The resulted system is an Integer Linear Program (*ILP*), and given by (20)-(27).

Theorem 6 : *The ILP (20)-(27) has always a solution, and its optimal solution can be found in polynomial run-time.* ❑

Proof: It is omitted due to space limitation, but can be derived as we did for the case of Theorem 3.

❑

$$Maximize\left(\sum_{e_{u,v} \in E} m(e_{u,v})\right) \tag{20}$$

Subject to:

$$-m(e_{u,v}) \le 0, \forall e_{u,v} \in E \tag{21}$$

$$m(e_{u,v}) \le 1, \forall e_{u,v} \in E \tag{22}$$

$$m(e_{u,v}) + r(u) - r(v) \le w(e_{u,v}), \forall e_{u,v} \in E \tag{23}$$

$$r(u) - r(v) \le W(u,v) - 1, \forall u, v \in V \text{ such that } D(u,v) > \Pi \tag{24}$$

$$-b(u) - r(u) + r(v) \le -w(e_{u,v}), \forall e_{u,v} \in E \tag{25}$$

$$\sum_{v \in V} b(v) \leq B \tag{26}$$

$$m(e_{u,v}),\ b(v) \text{ and } r(v) \text{ are non-negative integers, } \forall e_{u,v} \in E,\ \forall v \in V \tag{27}$$

5 Transformed-Code Generation and Reduction of Its Size

For software implementations, the approaches for solving the problem of Section 3 or 4 constitute a source-to-source code transformation. The size of the transformed code is not unique. It depends on the retiming used. In this section, we show how to obtain this transformed code, and why the size of this code is not unique. We also provide a method to find a retiming that allows to obtain a code with a small size.

The transformed code is constituted by three parts: Prologue, Loop, and Epilogue. The Prologue is the segment of the original code that must be executed before a repetitive processing appears that corresponds to the Loop. The Epilogue is the segment of the original code that cannot be executed by the Loop and the Prologue.

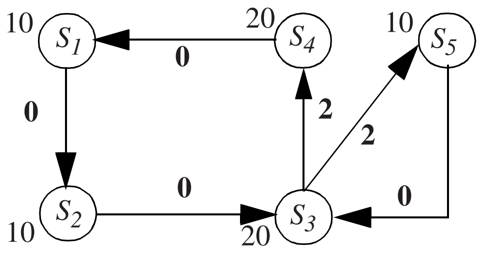
In Section 3, we showed that to minimize the sum of variables' lifetime for the loop in Fig. 2, the retiming vector $\{2, 0, 0, 2, 0\}$ can be applied on Fig. 2 (b). For this vector of retiming, the code for the transformed loop as well as its directed cyclic graph are given on Fig. 5.

```
main () {

    int a[U], b[U], c[U], d[U], e[U], i;

    /* Prologue */
    S2: b[2] = 1 + a[0];
    S3: c[2] = b[2] * e[2];
    S5: e[2] = 1 + c[0];
    S2: b[3] = 1 + a[1];
    S3: c[3] = b[3] * e[3];
    S5: e[3] = 1 + c[1];
    /* New Loop */
    for (i=4; i<= U; i++) {
      S1: a[i-2] = 1 + d[i-2];
      S2: b[i] = 1 + a[i-2];
      S3: c[i] = b[i] * e[i];
      S4: d[i-2] = 2 * c[i-2];
      S5: e[i] = 1 + c[i-2];
    }
    /* Epilogue */
    S1: a[U-1] = 1 + d[U-1];
    S4: d[U-1] = 2 * c[U-1];
    S1: a[U] = 1 + d[U];
    S4: d[U] = 2 * c[U];
}
```

(a) Code after retiming.



(b) Directed cyclic graph model for (a).

Fig. 5. Source code after application of a retiming on a loop.

Let $M = Max\{r(v), \forall v \in V\}$. With the help of Fig. 5, one can then easily double-check that the **P**rologue, the new **L**oop and the **E**pilogue for the transformed loop can be obtained by the algorithm PLE below.

From algorithm PLE, one can deduce that the sizes of the prologue and the epilogue as well as the number of iterations of the new loop depend on *M*. The value of *M* depends on the retiming used and is not unique. Indeed, for instance, to minimize the sum of variables' lifetime for the loop in Fig. 2, we have shown in Section 3 that the retiming vector $\{2, 0, 0, 2, 0\}$ can be applied on Fig. 2 (b). But, the retiming vector $\{2 + x, 0 + x, 0 + x, 2 + x, 0 + x\}$ can also be applied to obtain the same sum of variables' lifetime, where *x* is a non-negative integer. For the first vector, we have $M = 2$, and for the second one we have $M = 2 + x$. The last retiming vector is then not a good choice and can lead to completely unroll the loop if $x = U - 2$. To determine a retiming that leads to a small value of *M*, one may need to add an upper bound M_u on the retiming value for each computational element as described by (28).

$$r(v) \le M_u,\ \forall v \in V. \tag{28}$$

Inequality (28) can then be added to the constraints of the ILPs (10)-(14) and (20)-(27) to produce a retiming with a small *M*. Let ILP(M_u) be one of these ILPs once (28) is incorporated into its constraints. Of course the right value for M_u has to be determined. Finding the smallest value for M_u and solving ILP(M_u) can be done using the algorithm Solve_ILP(M_u) below.

Algorithm: PLE

```
/* The index i of the original loop is in [L, U].   $M = Max\{r(v), \forall v \in V\}$   */
1- Prologue Generation
    1.1 $i = L$
    1.2 While ( $i < (L + M)$ )
        1.2.1 $\forall v \in V$, if ( $(i + r(v)) < (L + M)$ ) then Generate a copy of v at iteration i as it was in the
              original loop's body.
        1.2.2 $i = i + 1$
2- New Loop Generation
    2.1   The index of the new loops is in [L + M, U]
    2.2   Its body: $\forall v \in V$, generate a copy of v where the index of each array in v of the original loop's
          body has now to be decreased by $r(v)$.
3- Epilogue Generation
    3.1 $i = U + 1$
    3.2 While ( $i \le (U + M)$ )
        3.2.1 $\forall v \in V$, if ( $(i - r(v)) \le U$ ) then Generate a copy of v by evaluating its expression
              derived from 2.2.
        3.2.2 $i = i + 1$
```

Algorithm: Solve_ILP(M_u)

```
1- Solve ILP($M_u$) without considering (28). Put the solution in $S_0$ and the value of the objective
   function in OptimalVal.
2- Using $S_0$, do $M_{max} = Max\{r(v), \forall v \in V\}$.
3- Let $M_u = M_{max}$ and $M_{min} = 0$.
4- while ($M_{min} \neq M_{max}$) do
  4.1 $M_u = (M_{min} + M_{max})/2$
  4.2 If ILP($M_u$) has a solution $S$ and the value of the objective function is $\leq$ OptimalVal
      then do $S_0 = S$ and $M_{max} = M_u$, else do $M_{min} = M_u$.
5- Solve ILP($M_u$). If it doesn't have a solution then use $S_0$ as a solution.
```

6 Minimizing Variables' Lifetime in Case of Software Pipelining

Software pipelining is a loop scheduling technique. It allows to increase the instruction level parallelism, which is useful for processors such as VLIW and superscalar processors. This instruction level parallelism is increased by reducing the difference of the start times of each two successive iterations of the loop. For an introduction to software pipelining and to its related techniques, the reader is referred to [7].

Software pipelining of single loop has been widely addressed in the literature [7]. Modulo scheduling is one of the approaches proposed for software pipelining. It is a periodic function $s : N \times V \to Q$ that determines the start time for each instruction of the loop's body. This function is defined as follows:

$$s_k(v) = s_0(v) + P \cdot k, \forall k \in N, \forall v \in V, \qquad (29)$$

where $s_0(v)$ is the start time of the first instance of the instruction v, and P is the period of s. Let C be the set of directed cycles in the directed cyclic graph modeling the loop. Based on data dependency constraints only, the minimum value of P is the inverse of the value in (30). This value can be computed in polynomial run-time, since (30) is an instance of a Minimum-Cost-to-Time Ratio Cycle Problem [10]. In the sequel, we assume that P is given.

$$Min_{c \in C}\left(\left(\sum_{e_{u,v} \in c} w(e_{u,v})\right) \Big/ \left(\sum_{\forall v \in V \text{ and } e_{u,v} \in c} d(u)\right)\right). \qquad (30)$$

Since software pipelining reduces the difference of the start times of each two successive iterations of the loop, it then increases the number of required registers to store alive variables. Our objective in the rest of this section is to show how to reduce the number of required registers for modulo scheduling.

Using the graph G modeling the loop, let v be any one of the successors of u. The result of the instruction u of iteration say k that is started at time $S_k(u)$ will be available at time $(S_k(u) + d(u))$. This result must still be alive until the start time of v at iteration $(k + w(e_{u,v}))$. Hence, the relative lifetime $t_v(u)$ of this result is:

$$t_v(u) = s_{k+w(e_{u,v})}(v) - (s_k(u) + d(u)),\ \forall k \in N,\ \forall e_{u,v} \in E. \tag{31}$$

Of course, we have:

$$t_v(u) \geq 0,\ \forall e_{u,v} \in E. \tag{32}$$

Using (29), we can then rewrite (31) as follows:

$$t_v(u) = s_0(v) - s_0(u) + P \cdot w(e_{u,v}) - d(u),\ \forall e_{u,v} \in E. \tag{33}$$

By considering all the successors of u, the lifetime $t(u)$ of any result of operation u then satisfies the following:

$$t(u) \geq t_v(u),\ \forall e_{u,v} \in E. \tag{34}$$

From (32), (33) and (34), we can derive the following system of inequalities (35) and (36):

$$t(u) + s_0(u) - s_0(v) \geq P \cdot w(e_{u,v}) - d(u),\ \forall e_{u,v} \in E \tag{35}$$

$$s_0(v) - s_0(u) \geq d(u) - P \cdot w(e_{u,v}),\ \forall e_{u,v} \in E \tag{36}$$

To find a modulo schedule that minimizes the sum of variables' lifetime, one then can solve the linear program (38)-(40). The number of registers required for each result produced by u is:

$$T(u) = \lceil t(u)/P \rceil,\ \forall u \in V. \tag{37}$$

Hence, linear program (38)-(40) allows to determine a schedule that reduces the required number of registers.

By dividing all the expressions (38)-(40) by P, replacing $((T(u))/P)$ by $T(u)$, and adding integrality constraints on $T(u)$, we obtain an MILP to determine a schedule that minimizes the required number of registers. MILPs cannot be in general solved with polynomial run-time. But, they can be solved with a small run-time when the size of the formulation is small. However, when $P = 1$ and all execution delays are integers, then one can prove as done for Theorem 3 that the MILP can be solved in polynomial run-time.

$$Minimize\left(\sum_{v \in V} t(v)\right) \tag{38}$$

Subject to:

$$t(u) + s_0(u) - s_0(v) \geq P \cdot w(e_{u,v}) - d(u),\ \forall e_{u,v} \in E \tag{39}$$

$$s_0(v) - s_0(u) \geq d(u) - P \cdot w(e_{u,v}),\ \forall e_{u,v} \in E \tag{40}$$

Although software pipelining can be extended to nested loops by applying it on one loop at each time, modulo scheduling has been used in the case of perfect nested loops

[8] to do software pipelining on all the loops of the nest in the same time. This modulo scheduling is as (29) but P and k become vectors. To determine the value of P, a linear program is proposed in [8]. This linear program computes also the start time of the first instance of each instruction v in the loop's body.

In the case of modulo scheduling for perfect nested loops, is it possible to minimize the sum of variables' lifetime? The response is yes, and our objective in the rest of this section is to show how it can be done.

A perfect nested loop can be modeled as we did for the case of one loop. The weight $w(e_{u,v})$ of each arc transforms to a vector $W(e_{u,v})$ (see [8] for more details). Without loss of generality, we assume that all components of $W(e_{u,v})$ are non-negative integers.

Basic retiming has been extended to the multidimensional case. Multidimensional retiming R has been used to reduce the length of critical paths in the case of perfect nested loops [11]. For this kind of retiming, (1) transforms to:

$$W_r(e_{u,v}) = W(e_{u,v}) + R(v) - R(u), \forall e_{u,v} \in E, \tag{41}$$

and (2) transforms to:

$$W_r(e_{u,v}) \supseteq 0, \forall e_{u,v} \in E, \tag{42}$$

where $\supseteq$ means lexicographically non-negative, which is required to exclude the case of consuming a result of an operation before this result becomes available. Since we assume that all the components of $W(e_{u,v})$ are non-negative integers, then instead of (42) we require in this paper that a valid multidimensional retiming must satisfy the following condition: all the components of $W_r(e_{u,v})$ are non-negative integers. We represent this condition by:

$$W_r(e_{u,v}) \gg 0, \forall e_{u,v} \in E. \tag{43}$$

The system of inequalities (35) and (36) holds for the case of modulo scheduling in the case of perfect nested loop. When a multidimensional retiming is applied on the nested loops, this system transforms to (44)-(46):

$$t(u) + s_0(u) - s_0(v) \geq P \cdot W_r(e_{u,v}) - d(u), \forall e_{u,v} \in E \tag{44}$$

$$s_0(v) - s_0(u) \geq d(u) - P \cdot W_r(e_{u,v}), \forall e_{u,v} \in E \tag{45}$$

$$W_r(e_{u,v}) \gg 0, \forall e_{u,v} \in E \tag{46}$$

For a given vector P that can be computed using the linear program in [8], to find a modulo schedule that minimizes the number of registers, one can then solve the MILP constituted by (48)-(52) and derived from (44)-(46), where:

$$c(e_{u,v}) = P \cdot W(e_{u,v}) - d(u), \forall e_{u,v} \in E. \tag{47}$$

This MILP also applies to the case of one loop only.

$$Minimize\left(\sum_{v \in V} t(v)\right) \tag{48}$$

Subject to:

$$t(u) + s_0(u) - s_0(v) + P \cdot (R(u) - R(v)) \geq c(e_{u,v}),\ \forall e_{u,v} \in E \tag{49}$$

$$s_0(v) - s_0(u) + P \cdot (R(v) - R(u)) \geq -c(e_{u,v}),\ \forall e_{u,v} \in E \tag{50}$$

$$R(v) - R(u) + W(e_{u,v}) \gg 0,\ \forall e_{u,v} \in E \tag{51}$$

$t(v)$ and all the components of $R(v)$ are non-negative integers, $\forall v \in V$ (52)

7 Related Work

An approach to minimize variables' lifetime without timing constraints has been proposed in [2]. An exact algorithm is proposed for one loop, and extended to address heuristically the case of nested loops. The disadvantages of this approach is that it can transform the loop body to an other one where the instructions can no longer be executed in parallel. Hence, it can increase the execution time. This approach does not address the problem of generating the transformed code as well as how to reduce its size. The transformed code by this approach is not useful for low power design, since it can generate non-balanced paths. Having non-balanced paths increases power consumption due to switching activities.

An approach to address a version of the problem of balancing variables' lifetime under timing constraints has been recently proposed in [3], and exploited in an algorithm [3] for reducing dynamic power consumption for sequential designs.

The problem of reducing code size when software pipelining is used to optimize loops is addressed in [4] and [5]. These papers have presented an approach based on retiming for this problem in the case of one loop only.

Code transformation regarding memories in the context of hardware and software has been investigated in [6] and [9].

8 Experimental Results

For any loop, let B and $\overline{M}$ be the sum of variable's lifetime and the number of zero-weight arcs, respectively. For the original loop, we denote B and $\overline{M}$ as B_o and $\overline{M}_0$, respectively. We denote by B^* the value of (10), and by $\overline{M}*$ the number of zero-weight arcs once (20)-(27) is solved. We define $RD1(\%)$ and $RD2(\%)$ as follows:

$$RD1(\%) = ((B_o - B^*)/B_o) \times 100 \text{, and}$$

$$RD2(\%) = ((\overline{M}_0 - \overline{M}*)/\overline{M}_0) \times 100 \,.$$

We have shown with the help of examples how it is important to solve the problems reported in this paper. Our objective in this section is to experimentally measure the values of $RD1(\%)$ and $RD2(\%)$ using some known benchmarks. For these benchmarks, we think on the cyclic graph modeling some circuits such as the correlator from [1] and circuits from ISCAS'89 benchmarks [17] as they are cyclic graphs modeling loops. The name of circuits used for this experiment are given in the first row of Table 1.

Rows 2 and 3 of this table report the values of $RD1(\%)$ and $RD2(\%)$, respectively. To obtained these values, we solve the ILPs (10)-(14) and (20)-(27), respectively. For each circuit, the ILPs are automatically generated by a module we coded in C++, and solved using [16]. For each ILP, the value of Π is fixed to its minimal possible value found by applying a retiming for minimal clock period on the corresponding circuit. For ILP (20)-(27), we first solve ILP (10)-(14), and then fix B in (27) to the value of (10).

As Table 1 reports, it is possible to obtain values as high as 33.33% and 28.57% for $RD1(\%)$ and $RD2(\%)$, respectively.

Table 1. Assessment of the approaches for the first two problems.

Circuit Name	**All Pole Lat-ice Filter**	**Second Avenh-aus Filter**	**Fifth Order Wave Digital Filter**	**Four Aven-haus Filter**	**Polyn -om Divid er**	**Second Order IIR**	**Corre lator**	**S344**	**S641**	**S713**	**S1238**	**S1423**
RD1(%)	33.33	0.00	0	12.50	20.00	20.00	20	0.00	15.79	15.0	0.00	0.00
RD2(%)	0.00	27.27	9.3	17.65	28.57	0.0	0	13.4	8.72	7.6	1.37	3.23

9 Conclusions

Many real applications are loop-intensive. Since many of them require high speed processing, low power or both of two, minimizing variables' lifetime is required when implementing these applications as hardware, software or hardware-and-software. Such a minimization is also useful for system-on-chip design and embedded systems.

In this paper, we have addressed three problems related to minimizing variables' lifetime in loop-intensive applications. We have proposed methods to solve these problems and showed that a set of them have polynomial run-time. We have also devised algorithms to the problem of deriving the code as well as reducing its size after application of these methods.

References

1. Leiserson C.E., Saxe J.B. : Retiming Synchronous Circuitry. Algorithmica (January 1995) 5–35.
2. Fraboulet A., Mignotte A., Huard G. : Loop alignment for memory accesses optimization. Proc. of the 12th International Symposium on System Synthesis (November 1999) 71–77.
3. Chabini N., Wolf W. : Reducing Dynamic Power Consumption in Synchronous Sequential Digital Designs Using Retiming and Supply Voltage Scaling. Submitted to IEEE Transactions on VLSI (March 2003).
4. Zhuge Q., Shao Z., Sha E. H.-M. : Optimal Code Size Reduction for Software-Pipelined Loops on DSP Applications. Proceedings of the International Conference on Parallel Processing, Vancouver, Canada (August 2002).
5. Zhuge Q., Xiao B., Sha E. H.-M. : Code Size Reduction Technique and Implementation for Software-Pipelined DSP Applications. Submitted to ACM Transcations in Embedded Computing Systems.
6. Catthoor F., Danckaert K., Wuytack S., Dutt N.-D. : Code transformations for data transfer and storage exploration preprocessing in multimedia processors. IEEE Design & Test of Computers,Vol.18, Issue 3 (May 2001) 70–82.
7. Allan V., Jones R.B., Lee R.M., Allan S.J. : Software Pipelining. ACM Computmg Surveys, Vol. 27, No. 3 (September 1995).
8. Ramanujam J. : Optimal Software Pipelining of Nested Loops. Proceeding 8th International Parallel Processing Symposium (April 1994) 335–342.
9. Panda P.-R., and *al.* : Data and memory optimization techniques for embedded systems. ACM Trans. on Design Automation of Electronic Systems, Vol.6, No.2 (April 2001).
10. Dasdan A., Gupta R. K. : Faster Maximum and Minimum Mean Cycle Algorithms for System Performance Analysis. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 17, No. 10 (October 1998).
11. Passos N.-L., Sha E.H.-M. : Achieving full parallelism using multidimensional retiming. IEEE Transactions on Parallel and Distributed Systems, Vol. 7, Issue 11, (November 1996) 1150–1163.
12. Schrijver A. : Theory of linear and integer programming. John Wiley and Sons (1986).
13. Khachian L.-G. : A polynomial algorithm in linear programming. Soviet Math Doklady, Vol.20 (1979).
14. Karmakar N. : A new polynomial-time algorithm for linear programming. Combinatorica, Vol.4 (1984).
15. Camion P. : Characterisation of totally unimodular matrices. Proc. of the Amer. Math. Soc., Vol. 16 (1965).
16. The LP_Solve Tool: `ftp://ftp.ics.ele.tue.nl/pub/lp_solve/`
17. ISCAS'89 Benchmark suite. North Carolina State University, Department of Computer Science. `http://www.cbl.ncsu.edu/benchmarks/`

Resource Interfaces*

Arindam Chakrabarti[1], Luca de Alfaro[2], Thomas A. Henzinger[1], and Mariëlle Stoelinga[2]

[1] Electrical Engineering and Computer Sciences, UC Berkeley
[2] Computer Engineering, UC Santa Cruz

Abstract. We present a formalism for specifying component interfaces that expose component requirements on limited resources. The formalism permits an algorithmic check if two or more components, when put together, exceed the available resources. Moreover, the formalism can be used to compute the quantity of resources necessary for satisfying the requirements of a collection of components. The formalism can be instantiated in several ways. For example, several components may draw power from the same source. Then, the formalism supports compatibility checks such as: can two components, when put together, achieve their tasks without ever exceeding the available amount of peak power? or, can they achieve their tasks by using no more than the initially available amount of energy (i.e., power accumulated over time)? The corresponding quantitative questions that our algorithms answer are the following: what is the amount of peak power needed for two components to be put together? what is the corresponding amount of initial energy? To solve these questions, we model interfaces with resource requirements as games with quantitative objectives. The games are played on state spaces where each state is labeled by a number (representing, e.g., power consumption), and a play produces an infinite path of labels. The objective may be, for example, to minimize the largest label that occurs during a play. We illustrate our approach by modeling compatibility questions for the components of robot control software, and of wireless sensor networks.

1 Introduction

In component-based design, a central notion is that of *interfaces*: an interface should capture those facts about a component that are necessary and sufficient for determining if a collection of components fits together. The formal notion of interface, then, depends on what "fitting together" means. In a simple case, an interface exposes only type information about the component's inputs and outputs, and "fitting together" is determined by type checking. In a more ambitious case, an interface may expose also temporal information about inputs and outputs. For example, the temporal interface of a file server may specify that the

* This research was supported in part by the DARPA grant F33615-00-C-1693, the MARCO grant 98-DT-660, the ONR grant N00014-02-1-0671, and the NSF grants CCR-0085949, CCR-0132780, CCR-0234690, and CCR-9988172.

`open_file` method must be called before the `read_file` method is invoked. If a client, instead, calls `read_file` before `open_file`, then an interface violation occurs. In [2], we argued that temporal interfaces are *games*. There are two players, Input and Output, and an objective, namely, the absence of interface violations. Then, an interface is *well-formed* if the corresponding component can be used in some environment; that is, player Input has a strategy to achieve the objective. Moreover, two interfaces are *compatible* if the corresponding components can be used together in some environment; that is, the composition of the two games is well-formed, and specifies the composite interface.

Here, we develop the theory of interfaces as games further, by introducing interfaces that expose resource information. Consider, for example, components whose power consumption varies. We model the interface of such a component as a control flow graph whose states are labeled with integers, which represent the power consumption while control is at that state. For instance, in the thread-based programming model for robot motor control presented in Section 5, the power consumption of a program region depends on how many motors and other devices are active. Now suppose that we want to put together two threads, each of which consumes power, but the overall amount of available peak power is limited to a fixed amount Δ. The threads are controlled by a scheduler, which at all times determines the thread that may progress. Then the two threads are Δ-*compatible* if the scheduler has a strategy to let them progress in a way so that their combined power consumption never exceeds Δ. In more detail, the game is set up as follows: player Input is the scheduler, and player Output is the composition of the two threads. At each round of the game, player Input determines which thread may proceed, and player Output determines the successor state in the control flow graph of the scheduled thread. In this game, in order to avoid a safety violation (power consumption greater than Δ), player Input may not let any thread progress. To rule out such trivial schedules, one may augment the safety objective with a secondary, liveness objective, say, in the form of a Büchi condition, which specifies that the scheduler must allow each thread to progress infinitely often. The resulting compatibility check, then, requires the solution of a Büchi game: the two threads are Δ-compatible iff player Input has a strategy to satisfy the Büchi condition without exceeding the power threshold Δ.

The basic idea of formalizing interfaces as such *Büchi threshold games* on integer-labeled graphs has many applications besides power consumption. For example, access to a mutex resource can be modeled by state labels 0 and 1, where 1 represents usage of the resource. Then, if we choose $\Delta = 1$, two or more threads are Δ-compatible if at any time at most one of the threads uses the resource. In Section 5, we will also present an interface model for the clients of a wireless network, where each state label represents the number of active messages at a node of the network, and Δ represents the buffer size. In this example, the Δ-compatibility check synthesizes not a scheduling strategy but a routing protocol that keeps the message buffers from overflowing.

A wide variety of other formalisms for the modeling and analysis of resource constraints have been proposed in the literature (e.g., [7,8,9,11]). The essential difference between these papers and our work is that we pursue a compositional approach, in which the models and analysis techniques are based on games. Once resource interfaces are modeled as games on graphs with integer labels, in addition to the *boolean* question of Δ-compatibility, for fixed Δ, we can also ask a corresponding *quantitative* question about resource requirements: What is the minimal resource level (peak power, buffer size, etc.) Δ necessary for two or more given interfaces to be compatible? To formalize the quantitative question, we need to define the *value* of an outcome of the game, which is the infinite sequence of states that results from playing the game for an infinite number of rounds. For Büchi threshold games, the value of an outcome is the supremum of the power consumption over all states of the outcome. The player Input (the scheduler) tries to minimize the value, while the player Output (the thread set) tries to maximize. The quantitative question, then, asks for the inf-sup of the value over all player Input and Output strategies.

The threshold interfaces, where an interface violation occurs if a power threshold is exceeded at any one time, provide but one example of how the compatibility of resource interfaces may be defined. We also present a second use of resource interfaces, where a violation occurs when an initially available amount Δ of energy (given, say, by the capacity of a battery) is exhausted. In this case, the value u of a finite outcome is defined as the *sum* (rather than maximum) over all labels of the states of the outcome, and player Input (the scheduler) wins if it can keep $\Delta - u$ nonnegative forever, or until a certain task is achieved. Note that in this game, negative state labels can be used to model a recharging of the energy source. Achieving a task might be modeled again by a Büchi objective, but for variety's sake, we use a quantitative *reward* objective in our formalization of such *energy interfaces.* For this purpose, we label each state with a second number, which represents a reward, and the objective of player Input is to obtain a total accumulated reward of Λ. The boolean Δ-compatibility question, then, asks if Λ can be obtained from the composition of two interfaces without exceeding the initial energy supply Δ. The corresponding quantitative resource-requirement question asks for the minimum initial energy supply Δ necessary to achieve the fixed reward Λ. Dually, a similar algorithm can be used to determine the maximal achievable reward Λ given a fixed initial energy supply Δ. In particular, if every state offers reward 1, this asks for the maximum runtime of a system (in number of state transitions) that a scheduler can achieve with a given battery capacity.

The paper is organized as follows. Section 2 reviews the definitions needed for modeling temporal (resourceless) interfaces as games and Section 3 adds resources to these games: we introduce integer labels on states to model resource usage, and we define boolean as well as quantitative objective functions on the outcomes of a game. As examples, we define four specific resource-interface theories: threshold games without and with Büchi objectives, and energy games without and with reward objectives. For these four theories, Section 4 gives algorithms for solving the boolean Δ-compatibility and the quantitative resource-requirement questions. These interface theories are also used in the two case studies of Section 5, one on scheduling embedded threads for robot control, and the other on routing messages across wireless networks.

2 Preliminaries

An *interface* is a system model that represents both the behavior of a component, and the behavior the component expects from its environment [2]. An interface communicates with its environment through input and output variables. The interface consists of a set of states. Associated with each state is an input assumption, which specifies the input values that the component is ready to accept from the environment, and an output guarantee, which specifies the output values that the component can generate. Once the input values are received and the output values are generated, these values cause a transition to a new state. In this way, both input assumptions and output guarantees can change dynamically. Formally, an *assume-guarantee* (A/G) *interface* [3] is a tuple $M = \langle V^i, V^o, Q, \hat{q}, \phi^i, \phi^o, \rho \rangle$ consisting of:

- Two finite sets V^i and V^o of boolean *input* and *output variables*. We require that $V^i \cap V^o = \emptyset$.
- A finite set Q of *states*, including an initial state $\hat{q} \in Q$.
- Two functions ϕ^i and ϕ^o which assign to each state $q \in Q$ a satisfiable predicate $\phi^i(q)$ on V^i, called *input assumption*, and a satisfiable predicate $\phi^o(q)$ on V^o, called *output guarantee*.
- A function ρ which assigns to each pair $q, q' \in Q$ of states a predicate $\rho(q, q')$ on $V^i \cup V^o$, called the *transition guard*. We require that for every state $q \in Q$, we have (1) $(\phi^i(q) \wedge \phi^o(q)) \Rightarrow \bigvee_{q' \in Q} \rho(q, q')$ and (2) $\bigwedge_{q', q'' \in Q}((\rho(q, q') \wedge \rho(q, q'')) \Rightarrow (q' = q''))$. Condition (1) ensures nonblocking; condition (2) ensures determinism.

We refer to the states of M as Q_M, etc. Given a set V of boolean variables, a *valuation* v for V is a function that maps each variable $x \in V$ to a boolean value $v(x)$. A valuation for V^i (resp. V^o) is called an *input* (resp. *output*) *valuation*. We write $\mathbb{V}^i$ and $\mathbb{V}^o$ for the sets of input and output valuations.

Interfaces as Games. An interface is best viewed as a game between two players, Input and Output. The game $G_M = \langle Q, \hat{q}, \gamma^i, \gamma^o, \delta \rangle$ associated with the interface M is played on the set Q of states of the interface. At each state $q \in Q$, player Input chooses an input valuation v^i that satisfies the input assumption, and simultaneously and independently, player Output chooses an output valuation v^o that satisfies the output guarantee; that is, at state q the moves available to player Input are $\gamma^i(q) = \{v \in \mathbb{V}^i \mid v \models \phi^i(q)\}$, and the moves available to player Output are $\gamma^o(q) = \{v \in \mathbb{V}^o \mid v \models \phi^o(q)\}$. Then the game proceeds to the state $q' = \delta(q, v^i, v^o)$, which is the unique state in Q such that $(v^i \uplus v^o) \models \rho(q, q')$. The result of the game is a run. A *run* of M is an infinite sequence $\pi = q_0, (v^i_0, v^o_0), q_1, (v^i_1, v^o_1), q_2, \ldots$ of alternating states $q_k \in Q$, input valuations $v^i_k \in \mathbb{V}^i$, and output valuations $v^o_k \in \mathbb{V}^o$, such that for all $k \geq 0$, we have (1) $v^i_k \in \gamma^i(q_k)$ and $v^o_k \in \gamma^o(q_k)$, and (2) $q_{k+1} = \delta(q_k, v^i_k, v^o_k)$. The run π is *initialized* if $q_0 = \hat{q}$. A *run prefix* is a finite prefix of a run which ends in a state. Given a run prefix π, we write $last(\pi)$ for the last state of π.

In a game, the players choose moves according to strategies. An *input strategy* is a function that assigns to every run prefix π an input valuation in $\gamma^i(last(\pi))$, and an *output strategy* is a function that assigns to every run prefix π an output valuation in $\gamma^o(last(\pi))$. We write Σ^i and Σ^o for the sets of input and output strategies. Given a state $q \in Q$, and a pair $\sigma^i \in \Sigma^i$ and $\sigma^o \in \Sigma^o$ of strategies, the *outcome* of the game *from* q is the run $out(q, \sigma^i, \sigma^o) = q_0, (v_0^i, v_0^o), q_1, (v_1^i, v_1^o), \ldots$ such that (1) $q_0 = q$ and (2) for all $k \geq 0$, we have $v_k^i = \sigma^i(q_0, \ldots, q_k)$ and $v_k^o = \sigma^o(q_0, (v_0^i, v_0^o), q_1, \ldots, q_k)$.

The *size* of the A/G interface M is taken to be the size of the associated game G_M: define $|M| = \sum_{q \in Q} |\gamma^i(q)| \cdot |\gamma^o(q)|$. Since the interface M is specified by predicates on boolean variables, the size $|M|$ may be exponentially larger than the syntactic description of M, which uses formulas for ϕ^i, ϕ^o.

Compatibility and Composition. The basic principle is that two interfaces are compatible if they can be made to work together correctly. When two interfaces are composed, the outputs of one interface may be fed as inputs to the other interface. Thus, the possibility arises that the output behavior of one interface violates the input assumptions of the other. The two interfaces are called compatible if the environment can ensure that no such I/O violations occur. The assurance that the environment behaves in a way that avoids all I/O violations is, then, the input assumption of the composite interface. Formally, given two A/G interfaces M and N, define $V^o = V_M^o \cup V_N^o$ and $V^i = (V_M^i \cup V_N^i) \setminus V^o$. Let $Q = Q_M \times Q_N$ and $\hat{q} = (\hat{q}_M, \hat{q}_N)$. For all $(p, q), (p', q') \in Q_M \times Q_N$, let $\phi^o(p, q) = (\phi_M^o(p) \wedge \phi_M^o(q))$ and $\rho((p, q), (p', q')) = (\rho_M(p, p') \wedge \rho_N(q, q'))$. The interfaces M and N are *compatible* if (1) $V_M^o \cap V_N^o = \emptyset$ and (2) there is a function ψ^i that assigns to all states $(p, q) \in Q$ a satisfiable predicate on V^i such that:

> For all initialized runs $(p_0, q_0), (v_0^i, v_0^o), (p_1, q_1), (v_1^i, v_1^o), \ldots$ of the A/G interface $\langle V^i, V^o, Q, \hat{q}, \psi^i, \phi^o, \rho \rangle$ and all $k \geq 0$, we have $(v_k^i \uplus v_k^o) \models (\phi_M^i(p_k) \wedge \phi_N^i(q_k))$. (†)

If M and N are compatible, then the *composition* $M \| N = \langle V^i, V^o, Q, \hat{q}, \phi^i, \phi^o, \rho \rangle$ is the A/G interface with the function ϕ^i that maps each state $(p, q) \in Q$ to a satisfiable predicate on V^i such that for all functions ψ^i that satisfy the condition (†), and all $(p, q) \in Q$, we have $\psi^i(p, q) \Rightarrow \phi^i(p, q)$; i.e., the input assumptions ϕ^i are the weakest conditions on the environment of the composite interface $M \| N$ which guarantee the input assumptions of both components. Algorithms for checking compatibility and computing the composition of A/G interfaces are given in [3]. These algorithms use the game representation of interfaces.

3 Resource Interfaces

Let $\mathbb{Z}_\infty = \mathbb{Z} \cup \{\pm\infty\}$. A *resource algebra* is a tuple $A = \langle \mathbb{L}, \oplus, \Theta \rangle$ consisting of:

– A set $\mathbb{L}$ of *resource labels*, each signifying a level of consumption or production for a set of resources.

– A binary *composition operator* $\oplus\colon \mathbb{L}^2 \to \mathbb{L}$ on resource labels.
– A *value function* $\Theta\colon \mathbb{L}^\omega \to \mathbb{Z}_\infty$, which assigns an integer value (or infinity) to every infinite sequence of resource labels.

A *resource interface* over A is a pair $R = (M, \lambda)$ consisting of an A/G interface $M = \langle \cdot, \cdot, Q, \hat{q}, \cdot, \cdot, \cdot \rangle$ and a labeling function $\lambda\colon Q \to \mathbb{L}$, which maps every state of the interface to a resource label. The *size* of the resource interface is $|R| = |M| + \sum_{q \in Q} |\lambda(q)|$, where $|\ell|$ is the space required to represent the label $\ell \in \mathbb{L}$. The runs of R are the runs of M, etc. Given a run $\pi = q_0, (v_0^i, v_0^o), q_1, (v_1^i, v_1^o), \ldots$, we write $\lambda(\pi) = \lambda(q_0), \lambda(q_1), \ldots$ for the induced infinite sequence of resource labels. Given a state $q \in Q$, the *value at* q is the minimum value that player Input can achieve for the outcome of the game from q, irrespective of the moves chosen by player Output: $val(q) = \inf_{\sigma^i \in \Sigma^i} \sup_{\sigma^o \in \Sigma^o} \Theta(\lambda(out(q, \sigma^i, \sigma^o)))$. The state q is *Δ-compliant*, for $\Delta \in \mathbb{Z}_\infty$, if $val(q) \leq \Delta$. We write $Q_\Delta^{rc} \subseteq Q$ for the set of Δ-compliant states. The resource interface R is *Δ-compliant* if the initial state $\hat{q}$ is Δ-compliant, and the *value* of R is $val(\hat{q})$.

Given two resource interfaces $R = (M_R, \lambda_R)$ and $S = (M_S, \lambda_S)$ over the same resource algebra A, define $\lambda\colon Q_R \times Q_S \to \mathbb{L}$ such that $\lambda(p, q) = \lambda_R(p) \oplus \lambda_S(q)$. The resource interfaces R and S are *Δ-compatible*, for $\Delta \in \mathbb{Z}_\infty$, if (1) the underlying A/G interfaces M_R and M_S are compatible, and (2) the resource interface $(M_R \| M_S, \lambda)$ over A is Δ-compliant. Note that Δ-compatibility does not require that both component interfaces R and S are Δ-compliant. Indeed, if R consumes a resource produced by S, it may be the case that R is not Δ-compliant on its own, but is Δ-compliant when composed with S. This shows that different applications call for different definitions of composition for resource interfaces, and we refrain from a generic definition. We use, however, the abbreviation $R \| S = (M_R \| M_S, \lambda)$.

The class of resource interfaces over a resource algebra A is denoted $\mathcal{R}[A]$. We present four examples of resource algebras and the corresponding interfaces.

Pure Threshold Interfaces. The resource labels of a threshold interface specify, for each state q, an amount $\lambda(q) \in \mathbb{N}$ of resource usage in q (say, power consumption). When the states of two interfaces are composed, their resource usage is additive. The number $\Delta \geq 0$ provides an upper bound on the amount of resource available at every state. A state q is Δ-compliant if player Input can ensure that, when starting from q, the resource usage never exceeds Δ. The value at q is the minimum amount Δ of resource that must be available at all states for q to be Δ-compliant. Formally, the *pure threshold algebra* A^t is the resource algebra with $\mathbb{L}^t = \mathbb{N}$ and $\oplus^t = +$ and $\Theta^t(n_0, n_1, \ldots) = \sup_{k \geq 0} n_k$. The resource interfaces in $\mathcal{R}[A^t]$ are called *pure threshold interfaces*. Throughout the paper, we assume that all numbers, including the state labels $\lambda(q)$ of pure threshold interfaces as well as Δ, can be stored in space of a fixed size. It follows that the size of a pure threshold interface $R = (M, \lambda)$ is equal to the size of the underlying A/G interface M.

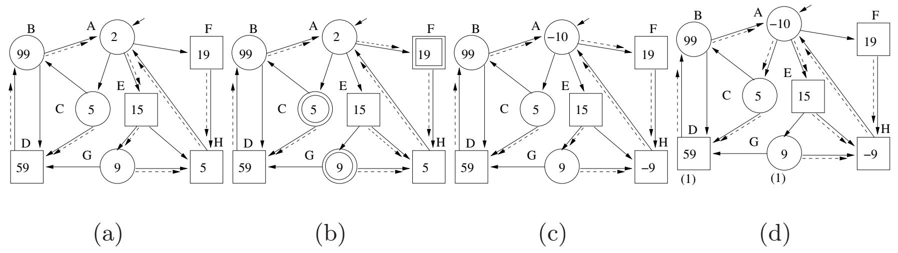


Fig. 1. Games illustrating the four classes of resource interfaces.

Example 1. Figure 1(a) shows the game associated with a pure threshold interface. For simplicity, the example is a turn-based game in which player Input makes moves in circle states, and player Output makes moves in square states. The numbers inside the states represent their resource labels. The solid arrows show the moves available to the players, and the dashed arrows indicate the optimal strategies for the two players. Note that at the initial state A, state E is a better choice than C for player Input in spite of having a greater resource label. It is easy to see that the value of the game (at A) is 15. □

Büchi Threshold Interfaces. While pure threshold interfaces ensure the safe usage of a threshold resource, they may allow some systems to never use the resource by not doing anything useful. To rule out this possibility, we may augment the pure threshold algebra with a generalized Büchi objective, which requires that certain state labels be visited infinitely often. A state q, then, is Δ-compliant if player Input can ensure that, when starting from q, the Büchi conditions are satisfied *and* resource usage never exceeds Δ. The formal definition of Büchi conditions within a resource algebra is somewhat technical. The *Büchi threshold algebra* A^{bt} is defined as follows, for a fixed set of labels $\mathcal{L}$:

- $\mathbb{L}^{bt}$ consists of triples $\langle n, \alpha, \beta \rangle \in \mathbb{N} \times 2^{\mathcal{L}} \times 2^{\mathcal{L}}$, where $n \in \mathbb{N}$ indicates the current level of resource usage, $\alpha \subseteq \mathcal{L}$ is a set of state labels that each need to be repeated infinitely often, and $\beta \subseteq \mathcal{L}$ is the set of state labels that are satisfied in the current state.
- $\langle n, \alpha, \beta \rangle \oplus^{bt} \langle n', \alpha', \beta' \rangle = \langle n + n', \alpha \uplus \alpha', \beta \uplus \beta' \rangle$.
- We distinguish two cases, depending on whether or not the generalized Büchi objective is violated: $\Theta^{bt}(\langle n_0, \alpha_0, \beta_0 \rangle, \langle n_1, \alpha_1, \beta_1 \rangle, \dots) = +\infty$ if there is an $\ell \in \alpha_0$ and a $k \geq 0$ such that for all $j \geq k$, we have $\ell \notin \beta_j$; otherwise, $\Theta^{bt}(\langle n_0, \alpha_0, \beta_0 \rangle, \langle n_1, \alpha_1, \beta_1 \rangle, \dots) = \sup_{k \geq 0} n_k$.

The resource interfaces in $\mathcal{R}[A^{bt}]$ are called *Büchi threshold interfaces.* The *number of Büchi conditions* of a Büchi threshold interface $R = (M, \lambda)$ is $|\hat{\alpha}|$, where $\hat{\alpha}$ is the second component of the label $\lambda(\hat{q})$ for the initial state $\hat{q}$ of M.

Example 2. Figure 1(b) shows a Büchi threshold game with a single Büchi condition. The graph is the same as in Example 1. The states with double borders are Büchi states, i.e., one of them needs to be repeated infinitely often. Note that the optimal output strategy at E has changed, because C is a Büchi state but H is not. This forces player Input to prefer at A state F over E in order to satisfy the Büchi condition. The value of the game is now 19. □

Pure Energy Interfaces. The resource labels of an energy interface specify, for each state q, the amount of energy $\lambda(q) \in \mathbb{Z}$ that is produced (if $\lambda(q) > 0$) or consumed (if $\lambda(q) < 0$) at q. When the states of two interfaces are composed, their energy expenditures are added. The number $\Delta \geq 0$ provides the initial amount of energy available. A state q is Δ-compliant if player Input can ensure that, when starting from q, the system can run forever without the available energy dropping below 0. The value at q is the minimum amount Δ of initial energy necessary for q to be Δ-compliant. Formally, the *pure energy algebra* A^e is the resource algebra with $\mathbb{L}^e = \mathbb{Z}$ and $\oplus^e = +$ and $\Theta^e(d_0, d_1, \dots) = -\inf_{k \geq 0} \sum_{0 \leq j \leq k} d_j$. The resource interfaces in $\mathcal{R}[A^e]$ are called *pure energy interfaces.* To characterize the complexity of the algorithms, we let the *maximal energy consumption* of a pure energy interface $R = (M, \lambda)$ be 1 if $\lambda(q) \geq 0$ for all states $q \in Q$, and $-\min_{q \in Q} \lambda(q)$ otherwise.

Example 3. Figure 1(c) shows a pure energy game. Player Input has a strategy to run forever when starting from the initial state A with 9 units of energy, but 8 is not enough initial energy; thus the game has the value 9. □

Reward Energy Interfaces. Some systems have the possibility of saving energy by doing nothing useful. To rule out this possibility, we may use a Büchi objective as in the case of threshold interfaces. For variety's sake, we provide a different approach. We label each state q not only with an energy expenditure, but also with a reward, which represents the amount of useful work performed by the system when visiting q. A reward energy algebra specifies a minimum acceptable reward Λ. A state q, then, is Δ-compliant if player Input can ensure that, when starting from q with energy Δ, the reward Λ can be obtained without the available energy dropping below 0. For $\Lambda \in \mathbb{N}$, the *Λ-reward energy algebra* A^{re}_{Λ} is defined as follows:

- $\mathbb{L}^{re} = \mathbb{Z} \times \mathbb{N}$. The first component of each resource label represents an energy expenditure; the second component represents a reward.
- $\langle d, n \rangle \oplus^{re} \langle d', n' \rangle = \langle d + d', n + n' \rangle$.
- There are two cases: $\Theta^{re}_{\Lambda}(\langle d_0, n_0 \rangle, \langle d_1, n_1 \rangle, \dots) = +\infty$ if $\sum_{j \geq 0} n_j < \Lambda$; otherwise, let $k^* = \min_{k \geq 0}(\sum_{0 \leq j \leq k} n_j \geq \Lambda)$ and define $\Theta^{re}_{\Lambda}(\langle d_0, n_0 \rangle, \langle d_1, n_1 \rangle, \dots) = -\inf_{0 \leq k \leq k^*} \sum_{0 \leq j \leq k} d_j$.

The resource interfaces in $\mathcal{R}[A^{re}_{\Lambda}]$ are called *Λ-reward energy interfaces.* The *maximal energy consumption* of a reward energy interface is defined as for pure energy interfaces, with the proviso that only the energy (i.e., first) components of resource labels are considered.

Example 4. Figure 1(d) shows a Λ-reward energy game with $\Lambda = 1$. The numbers in parentheses represent rewards; states that are not labeled with parenthesized numbers have reward 0. The optimal choice of player Input at state A is E, precisely the opposite of the pure energy case. If player Output chooses G at E, then the reward 1 is won, and player Input's objective is accomplished. If player Output instead chooses H at E, then 4 units of energy are gained in the cycle A,E,H,A. By pumping this cycle, player Input can gain sufficient energy to eventually choose the path A,C,D and win the reward 1. Hence the game has the value 5. Note that this example shows that reward energy games may not have memoryless winning strategies. □

4 Algorithms

Let A be a resource algebra. We are interested in the following questions:

Verification Given two resource interfaces $R, S \in \mathcal{R}[A]$, and $\Delta \in \mathbb{Z}_\infty$, are R and S Δ-compatible?

Design Given two resource interfaces $R, S \in \mathcal{R}[A]$, for which values $\Delta \in \mathbb{Z}_\infty$ are R and S Δ-compatible?

To answer these questions, we first need to check the compatibility of the underlying A/G interfaces M_R and M_S. Then, for the qualitative verification question, we need to check if the resource interface $R \| S \in \mathcal{R}[A]$ is Δ-compliant, and for the quantitative design question, we need to compute the value of $R \| S$. Below, for $A \in \{A^t, A^{bt}, A^e, A^{re}\}$, we provide algorithms for checking if a given resource interface $R \in \mathcal{R}[A]$ is Δ-compliant, and for computing the value of R. We present the algorithms in terms of the game representation $G_R = \langle Q, \hat{q}, \gamma^i, \gamma^o, \delta \rangle$ of the interface. The algorithms have been implemented in our tool Chic [12].

Pure Threshold Interfaces. For $n \in \mathbb{N}$, let $Q_{\leq n} = \{q \in Q \mid \lambda(q) \leq n\}$. For $\Delta \geq 0$, a pure threshold interface R is Δ-compliant iff player Input can win a game with the safety objective of staying forever in $Q_{\leq \Delta}$. Such safety games can be solved as usual using a *controllable predecessor* operator *CPre*: $2^Q \to 2^Q$, defined for all $X \subseteq Q$ by $\mathit{CPre}(X) = \{q \in Q \mid \exists v^i \in \gamma^i(q). \forall v^o \in \gamma^o(q). \delta(q, v^i, v^o) \in X\}$. The set of Δ-compliant states can then be written as the limit $Q^{rc}_\Delta = \lim_{k \to \infty} X_k$ of the sequence defined by $X_0 = Q$ and, for $k \geq 0$, by $X_{k+1} = Q_{\leq \Delta} \cap \mathit{CPre}(X_k)$. This algorithm can be written in μ-calculus notation as $Q^{rc}_\Delta = \nu X. (Q_{\leq \Delta} \cap \mathit{CPre}(X))$, where ν is the greatest fixpoint operator.

To compute the value of R, we propose the following algorithm. We introduce two mappings *lmax*: $2^Q \to \mathbb{N}$ and *below*: $2^Q \to 2^Q$. For $X \subseteq Q$, let $\mathit{lmax}(X) = \max\{\lambda(q) \mid q \in X\}$ be the maximum label of a state in X, and let $\mathit{below}(X) = \{q \in X \mid \lambda(q) < \mathit{lmax}(X)\}$ be the set of states with labels below the maximum. Then, define $X_0 = Q$ and, for $k \geq 0$, define $X_{k+1} = \nu X. (\mathit{below}(X_k) \cap \mathit{CPre}(X))$. For $k \geq 0$ and $q \in X_k \setminus X_{k+1}$, we have $\mathit{val}(q) = \mathit{lmax}(X_k)$.

While it may appear that computing the fixpoint $\nu X. (Q_{\leq \Delta} \cap \mathit{CPre}(X))$ requires quadratic time (computing *CPre* is linear in $|R|$, and we need at most $|Q|$ iterations), this can be accomplished in linear time. The trick is to use a refined version of the algorithm, where each move pair $\langle v^i, v^o \rangle$ is considered at most once. First, we remove from the fixpoint all states q' such that $\lambda(q') > \Delta$. Whenever a state $q' \in Q$ is removed from the fixpoint, we propagate the removal backward, removing for all $q \in Q$ any move pair $\langle v^i, v^o \rangle \in \langle \gamma^i(q), \gamma^o(q) \rangle$ such that $\delta(q, v^i, v^o) = q'$ and, whenever $\langle v^i, v^o \rangle$ is removed, removing also $\langle v^i, \hat{v}^o \rangle$ for all $\hat{v}^o \in \gamma^o(q)$. The state q is itself removed if all its move pairs are removed. Once the removal propagation terminates, the states that have not been removed are precisely the Δ-compliant states. In order to implement efficiently the algorithm for computing the value of a threshold interface, we compute X_{k+1} from X_k by removing the states having the largest label, and then back-propagating the removal. In order to compute $\mathit{below}(X_k)$ efficiently for all k, we construct a list of states sorted according to their label.

Theorem 1. *Given a pure threshold interface R of size n, and $\Delta \in \mathbb{Z}_\infty$, we can check the Δ-compliance of R in time $O(n)$, and we can compute the value of R in time $O(n \cdot \log n)$.*

Büchi Threshold Interfaces. Given a Büchi threshold interface R, let $\lambda(\hat{q}) = \langle \hat{n}, \hat{\alpha}, \hat{\beta} \rangle$, $|\hat{\alpha}| = m$, and $\hat{\alpha} = \{\alpha_1, \alpha_2, \ldots, \alpha_m\}$. Let $B^i = \{q \in Q \mid \lambda(q) = \langle n^q, \alpha^q, \beta^q \rangle$ and $\alpha_i \in \beta^q\}$ be the i-th set in the generalized Büchi objective, for $1 \leq i \leq m$. We can compute the set of Δ-compliant states of R by adapting the fixpoint algorithm for solving Büchi games [5] as follows. Given two sets $Z, T \subseteq Q$ of states, we define $\mathit{Reach}(Z,T) \subseteq Q$ as the set of states from which player Input can force the game to T while staying in Z. Formally, define $\mathit{Reach}(Z,T) = \lim_{k\to\infty} W_k$, where $W_0 = \emptyset$ and $W_{k+1} = Z \cap (T \cup \mathit{CPre}(W_k))$ for $k \geq 0$. Then, for $Z \subseteq Q$ and $1 \leq i \leq m$, we compute the sets $Y^i \subseteq Q$ as follows. Let $i' = (i \mod m) + 1$ be the successor of i in the cyclic order $1, 2, \ldots, m, 1, \ldots$ Let $Y^i_0 = Q$, and for $j \geq 0$, let $Y^i_{j+1} = \mathit{Reach}(Z, B^i \cap \mathit{CPre}(Y^{i'}_j))$. Intuitively, the set Y^i_{j+1} consists of the states from which Input can, while staying in Z, first reach B^i and then go to $Y^{i'}_j$. For $1 \leq i \leq m$, let the fixpoint be $Y^i = \lim_{j\to\infty} Y^i_j$: from Y^i, Input can reach B^i while staying in Z; moreover, once at B^i, Input can proceed to $Y^{i'}$. Hence, Input can visit the sets $B^1, B^2, \ldots, B^m, B^1, \ldots$ cyclically, satisfying the generalized Büchi acceptance condition. Denoting by $\mathit{GBüchi}(Z, B^1, \ldots, B^m) = Y^1 \cup Y^2 \cup \ldots \cup Y^m$, we can write the set of Δ-compliant states of the interface as $Q^{rc}_\Delta = \mathit{GBüchi}(Q_{\leq\Delta}, B^1, \ldots, B^m)$.

The algorithm for computing the value of a Büchi threshold interface can be obtained by adapting the algorithm for Δ-compliance, similarly to the case for pure threshold interfaces. Let $X_0 = Q$, and for $k \geq 1$, let $X_{k+1} = \mathit{GBüchi}(\mathit{below}(X_k), B^1, \ldots, B^m)$. Then, for a state $q \in X_k \setminus X_{k+1}$, we have $\mathit{val}(q) = \mathit{lmax}(X_k)$.

Since the set $\mathit{Reach}(Z,T)$ can be computed in time $O(m \cdot |R|)$, using again a backward propagation procedure, the computation of the set of Δ-compliant states of the interface requires time $O(m \cdot |R|^2)$, in line with the complexity for solving Büchi games. The value of Büchi threshold games can also be computed in the same time. In fact, Y^i for iteration $k+1$ (denoted $Y^i(k+1)$) can be obtained from Y^i for k (denoted $Y^i(k)$) by $Y^i_0(k+1) = Y^i(k)$ and, for $j \geq 0$, by $Y^i_{j+1}(k+1) = \mathit{Reach}(X_k \cap Y^i(k), B^i \cap \mathit{CPre}(Y^{i'}_j(k+1)))$. We then have $Y^i(k+1) = \lim_{j\to\infty} Y^i_j(k+1)$. Hence, for $1 \leq i \leq m$, the sets $Y^i(0)$, $Y^i(1)$, $Y^i(2)$, ... can be computed by progressively removing states. As each removal (which requires the computation of Reach) is linear-time, the overall algorithm is quadratic.

Theorem 2. *Given a Büchi threshold interface R of size n with m Büchi conditions, and $\varDelta \in \mathbb{Z}_\infty$, we can check the $\varDelta$-compliance of R and compute its value in time $O(n^2 \cdot m)$.*

Pure Energy Interfaces. Given a pure energy interface R, the value at state $q \in Q$ is given by $val(q) = \inf_{\sigma^i \in \varSigma^i} \sup_{\sigma^o \in \varSigma^o} \{\varTheta(\lambda(out(q, \sigma^i, \sigma^o)))\}$. To compute this value, we define an *energy predecessor operator* $EPre$: $(Q \to \mathbb{Z}_\infty) \to (Q \to \mathbb{Z}_\infty)$, defined for all $f\colon Q \to \mathbb{Z}_\infty$ and $q \in Q$ by

$$EPre(f)(q) = -\lambda(q) + \max\{0, \min_{v^i \in \gamma^i(q)} \max_{v^o \in \gamma^o(q)} f(\delta(q, v^i, v^o))\}.$$

Intuitively, $EPre(f)(q)$ represents the minimum energy Input needs for performing one step from q without exhausting the energy, and then continuing with energy requirement f. Consider the sequence of functions $f_0, f_1, \ldots\colon Q \to \mathbb{Z}_\infty$, where f_0 is the constant function such that $f_0(q) = -\infty$ for all $q \in Q$, and where $f_{k+1} = EPre(f_k)$ for $k \geq 0$. The functions in the sequence are pointwise increasing: for all $q \in Q$ and $k \geq 0$, we have $f_k(q) \leq f_{k+1}(q)$. Hence the limit $f_* = \lim_{k\to\infty} f_k$ (defined pointwise) always exists. From the definition of $EPre$, it can be shown by induction that $f_*(q) = val(q)$. The problem is that the sequence $f_0, f_1, \ldots$ may not converge to f_* in a finite number of iterations. For example, if the game has a state q with $\lambda(q) < 0$ and whose only transitions are self-loops, then $f_*(q) = +\infty$, but the sequence $f_0(q), f_1(q), \ldots$ never reaches $+\infty$. To compute the limit in finitely many iterations, we need a stopping criterion that allows us to distinguish between divergence to $+\infty$ and convergence to a finite value. The following lemma provides such a stopping criterion.

Lemma 1. *For all states q of a pure energy interface, either $val(q) = +\infty$ or $val(q) \leq -\sum_{p \in Q} \min\{0, \lambda(p)\}$.*

This lemma is proved in a fashion similar to a theorem in [4], by relating the value of the energy interface to the value along a loop in the game. Let $v^+ = -\sum_{p \in Q} \min\{0, \lambda(p)\}$. If $f_k(q) > v^+$ for some $k \geq 0$, we know that $f_*(q) = +\infty$. This suggests the definition of a modified operator $ETPre$: $(Q \to \mathbb{Z}_\infty) \to (Q \to \mathbb{Z}_\infty)$, defined for all $f\colon Q \to \mathbb{Z}_\infty$ and $q \in Q$ by

$$ETPre(f)(q) = \begin{cases} EPre(f)(q) & \text{if } EPre(f)(q) \leq v^+, \\ +\infty & \text{otherwise.} \end{cases}$$

We have $f_* = \lim_{k\to\infty} f_k$, where $f_0(q) = -\infty$ for all $q \in Q$, and $f_{k+1} = ETPre(f_k)$ for $k \geq 0$. Moreover, there is $k \in \mathbb{N}$ such that $f_k = f_{k+1}$, indicating that the limit can be computed in finitely many iterations. Once f_* has been computed, we have $val(q) = f_*(q)$ and $Q^{rc}_\varDelta = \{q \in Q \mid f_*(q) \leq \varDelta\}$.

Let ℓ be the maximal energy consumption of R. We have $v^+ \leq |Q| \cdot \ell$. Consider now the sequence $f_0, f_1, \ldots$ converging to f_*: for all $k \geq 0$, either $f_{k+1} = f_k$ (in which case $f_* = f_k$ and the computation terminates), or there must be $q \in Q$ such that $f_k(q) < f_{k+1}(q)$. Thus, the limit is reached in at most $v^+ \cdot |Q| \leq |Q|^2 \cdot \ell$ iterations. Each iteration involves the evaluation of the $ETPre$ operator, which requires time linear in $|R|$.

Theorem 3. *Given a pure energy interface R of size n with maximal energy consumption ℓ, and $\Delta \in \mathbb{Z}_\infty$, we can check the Δ-compliance of R and compute its value in time $O(n^3 \cdot \ell)$.*

Reward Energy Interfaces. Given a Λ-reward energy interface R and $\Delta \in \mathbb{Z}$, to compute Q^{rc}_Δ and *val*, we use a dynamic programming approach reminiscent of that used in the solution of shortest-path games [6]. We iterate over a set of *reward-energy allocations* $\mathcal{E}$: $Q \to (\{0, \dots, \Lambda\} \to \mathbb{Z}_\infty)$. Intuitively, for $f \in \mathcal{E}$, $q \in Q$, and $r \in \{0, \dots, \Lambda\}$, the value $f(q)(r)$ indicates the amount of energy necessary to achieve reward r before running out of energy. For $e_1, e_2 \in \mathbb{Z}$, let $\mathrm{Mxe}(e_1, e_2) = \max\{e_1, e_2\}$ if $\max\{e_1, e_2\} \leq v^+$, and $\mathrm{Mxe}(e_1, e_2) = +\infty$ otherwise. For $r \in \mathbb{N}$, let $\mathrm{Mxr}(r) = \max\{0, r\}$. For $q \in Q$, use $\lambda(q) = \langle d(q), n(q) \rangle$. We define an operator *ERPre*: $\mathcal{E} \to \mathcal{E}$ on energy-reward allocations by letting $g = \mathit{ERPre}(f)$, where $g \in \mathcal{E}$ is such that for all $q \in Q$ we have $g(q)(0) = 0$, and for all $r \in \{0, \dots, \Lambda - 1\}$,

$$g(q)(r) = \mathrm{Mxe}(-d(q), -d(q) + \min_{v^i \in \gamma^i(q)} \max_{v^o \in \gamma^o(q)} f(\delta(q, v^i, v^o))(\mathrm{Mxr}(r - n(q)))).$$

Intuitively, given an energy-reward allocation f, a state q, and a reward r, $\mathit{ERPre}(f)(q)(r)$ represents the minimum energy to achieve reward r from state q given that the next-state energy-reward allocation is f. Let $f_0 \in \mathcal{E}$ be defined by $f_0(q)(r) = +\infty$, for $q \in Q$ and $r \in \{0, \dots, \Lambda\}$, and for $k \geq 0$, let $f_{k+1} = \mathit{ERPre}(f_k)$. The limit $f_* = \lim_{k \to \infty} f_k$ (defined pointwise) exists; in fact, for all $q \in Q$ and $r \in \{0, \dots, \Lambda\}$, we have $f_{k+1}(q)(r) \leq f_k(q)(r)$. For all $q \in Q$, we then have $\mathit{val}(q) = f_*(q)(\Lambda)$, and $q \in Q^{rc}_\Delta$ if $f_*(q)(\Lambda) \leq \Delta$.

The complexity of this algorithm can be characterized as follows. For all $q \in Q$, $r \in \{0, \dots, \Lambda\}$, and $f \in \mathcal{E}$, the energy $f(q)(r)$ can assume at most $1 + v^+ \leq 1 + \ell \cdot |Q|$ values, where ℓ is the maximal energy consumption in R. Since each of these values is monotonically decreasing, the limit f_* is computed in at most $O(|Q|^2 \cdot \ell \cdot \Lambda)$ iterations. Each iteration has cost $|R| \cdot \Lambda$.

Theorem 4. *Given a Λ-reward energy interface R of size n with maximal energy consumption ℓ, and $\Delta \in \mathbb{Z}_\infty$, we can check the Δ-compliance of R and compute its value in time $O(n^3 \cdot \ell \cdot \Lambda^2)$.*

5 Examples

We sketch two small case studies that illustrate how resource interfaces can be used to analyze resource-constrained systems.

5.1 Distribution of Resources in a Lego Robot System

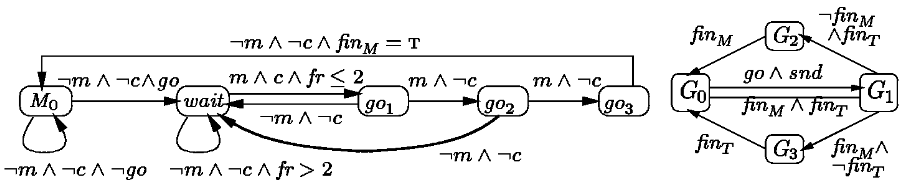
We use resource interfaces to analyze the schedulability of a Lego robot control program comprising several parallel threads. In this setup, player Input is a "resource broker" who distributes the resources among the threads. The system is compatible if Input can ensure that all resource constraints are met.

The Lego Robot. We have programmed a Lego robot that must execute various commands received from a base station through infrared (ir) communication, as well as recover from bumping into obstacles. Its software is organized in 5 parallel threads, interacting via a central repository. The thread Scan Sensors (S) scans the values of the sensors and puts these into the repository, Motion (M) executes the tasks from the base station, Bump Recovery (B) is executed when the robot bumps into an object, Telemetry (T) is responsible for communication with the base station and the Goal Manager (G) manages the various goals. There are 3 mutex resources: the motor (m), the ir sensor (s) and the central repository (c). Furthermore, energy is consumed by the motor and ir sensor. We model each thread as a resource interface; our model is open, as more threads can be added later.

Checking Schedulability Using Pure Threshold Interfaces. First, we disregard the energy consumption and consider the question whether all the mutex requirements can be met. To this end, we model each thread $i \in \{S, M, B, T, G\}$ as a threshold interface (M^i, λ^i) with threshold value $\Delta = 1$. The resource labeling $\lambda^i = (\lambda^i_m, \lambda^i_c, \lambda^i_s)$ is such that $\lambda^i_R(q)$ indicates whether, in state q, thread i owns resource R. The underlying A/G interface M_i has, for each resource $R \in \{m, c, s\}$, a boolean input variable gr^i_R (abbreviated R in the figures) indicating whether Input grants R to i. We also model a resource interface (M^E, λ^E) for the environment, expressing that bumps do not occur too often. This interface does not use any resources, i.e. $\lambda^E_r(q) = 0$ for all states q and all resources R. These resource interfaces are 1-compatible iff all mutex requirements are met.[1]

Due to space limitations, Figure 2 only presents the A/G interfaces for Motion and Goal Manager; the others be modeled in a similar fashion. Also, rather than with $\rho(p, q)$, we label the edges $\rho(p, q) \wedge \phi^i(p) \wedge \phi^o(q)$. The tread Motion in Figure (2a) has one boolean output variable fin_M, indicating whether it has finished a command from the base station. Besides the input variables gr^M_m, gr^M_c and gr^M_s discussed above, Motion has an input variable fr controlled by Scan Sensors that counts the steps since the last scanning of the sensors. In the initial location M_0, Motion waits for a command go from the Goal Manager. Its input assumption is $\neg m \wedge \neg c$, indicating that Motion needs neither the motor nor the repository. When receiving a command, Motion moves to the location *wait*, where it tries to get hold of the motor and of the repository. Since Motion needs fresh sensor values, it requires $fr \leq 2$ to move on the next location; otherwise it does not need either resource. In the locations go_1, go_2 and go_3, Motion executes the command. It needs the motor and repository in go_1, and the motor only in go_2 and go_3. If, in locations go_1 or go_2, the motor is retrieved from Motion (input $\neg m \wedge \neg c$, typically if Bump Recovery needs the motor), the thread goes back to location *wait*. When leaving location go_3, Motion sets $fin_M = \mathrm{T}$, indicating the completion of a command. We let $fin_M = \mathrm{F}$ on all other transitions. The labeling λ^M_r for $r \in R$ is given by: $\lambda^M_m(go_1) = \lambda^M_m(go_2) = \lambda^M_m(go_3) = \lambda^M_c(go_1)$.

[1] Note that the resource compliance of (Büchi) threshold games with multiple resource labelings can be checked along the same lines as the resource compliance of threshold games with single resource labelings.


$\neg m \wedge \neg c \wedge fin_M = \mathrm{T}$
$\neg m \wedge \neg c \wedge go$
$m \wedge c \wedge fr \leq 2$
$m \wedge \neg c$
$m \wedge \neg c$
M_0
$wait$
go_1
go_2
go_3
$\neg m \wedge \neg c$
$\neg m \wedge \neg c$
$\neg m \wedge \neg c \wedge \neg go$
$\neg m \wedge \neg c \wedge fr > 2$
fin_M
G_2
$\neg fin_M \wedge fin_T$
$go \wedge snd$
G_0
G_1
$fin_M \wedge fin_T$
fin_T
G_3
$fin_M \wedge \neg fin_T$

(a) Motion. **(b) Goal manager.**

Fig. 2. A/G interfaces modeling a Lego robot.

$\lambda_r^M(q) = 0$ in all other cases. (Note that $\lambda_R^i(q)$ is derivable from gr_R^i by considering edges leading to q.) The interface for *Goal Manager* (Figure(2b)) has output variables *go* and *snd* through which it starts up the threads Motion and Telemetry in location G_0 and then waits for them to be finished. It does not use any resources.

Checking Schedulability Using Büchi Threshold Interfaces. The threshold interfaces before express safety, but not liveness: the resource broker is not forced to ever grant the motor to Motion or Telemetry, in which case they stay forever in the locations *wait* or $wait_1$ respectively. To enforce the progress of the threads, we add a Büchi condition expressing that the locations G_0 should be visited infinitely often. Thus, each state is a state label and we define the location labeling of thread G by $\kappa^G(q) = (\lambda^G(q), \{G_0\}, \{q\})$ and for $i \in \{S, M, B, T, E\}$ by $\kappa^i(q) = (\lambda^i(q), \emptyset, \emptyset)$, where λ^i is as before. Then all mutex requirements can be met, with the state G_0 being visited infinitely often, iff the resource interfaces are 1-compatible.

Analyzing Energy Consumption Using Reward Energy Interfaces. Energy is consumed by the motor and the ir sensor. We define the energy expense for thread i at state q as $\lambda_e^i(q) = 5\lambda_m^i(q) + 2\lambda_s^i(q)$, expressing that the motor uses 5 energy units and the ir sensor 2. Currently, the system will always run out of energy because it is never recharged, but it is easy to add an interface for that. To prevent the system from saving energy by doing nothing at all, we specify a reward. A naive attempt would be to assign the reward to each location in each thread and sum the rewards upon composition. However, suppose that the reward acquired per energy unit is higher when executing Motion than when executing Telemetry. Then, the highest reward is obtained by always executing Motion and never doing Telemetry. This phenomenon is not a deficit of the theory, it is inherent when managing various goals. Since the latter is exactly the task of the goal manager, we reward the completion of a round of the goal manager. That is, we put $\lambda_r^G(G_0) = 1$ and $\lambda_r^i(q) = 0$ in all other cases. Then all mutex requirements can be met, while the system never runs out of power, iff the threshold interfaces interfaces (as defined before) are 1-compatible and their composition is 0-compliant as an energy reward interface.

5.2 Resource Accounting for the PicoRadio Network Layer

The PicoRadio [1] project aims to create large-scale, self-organizing sensor networks using very low-power, battery-operated *piconodes* that communicate over wireless links. In these networks, it is not feasible to connect each node individually to a power line or to periodically change its battery. Energy-aware routing [10] strategies have been found necessary to optimize use of scarce energy resources.We show how our methodology can be profitably applied to evaluate networks and synthesize optimal routing algorithms.

A PicoRadio Network. A *piconet* consists of a set of piconodes that can create, store, or forward packets using multi-hop routing. The piconet *topology* describes the position, maximal packet-creation rate, and packet-buffer capacity at each piconode, and capacity of each link. Each packet has a *destination*, which is a node in the network. A *configuration* of the network represents the number of packets of each destination currently stored in the buffer of each piconode. A configuration that assigns more packets to a node than its buffer size is not *legal.*The network moves from one configuration to another in a *round.* We assume that a piconode always uses its peak transmission capacity on an outgoing link as long as it has enough packets to forward on that link. Wireless transmission costs energy. Each piconode starts with an initial amount of energy, and can possibly gain energy by scavenging.

We are given a piconet with known topology and initial energy levels at each piconode. We wish to find a routing algorithm that makes the network satisfy a certain safety property, e.g., that buffer overflows do not occur (or that whenever a node has a packet to forward, it has enough energy to do so). A piconet together with such a property represents a concurrent finite-state safety game between player Packet Generator, and player Router. Each legal configuration of the network is represented by a game state; the state ERROR represents all illegal configurations. The guarded state transitions reflect the configuration changes the network undergoes from round to round as the players concurrently make packet creation and routing choices under the constraints imposed by the network topology. The state ERROR has a self-loop with guard T and no outgoing transitions. The initial state corresponds to the network configuration that assigns 0 packets to each node. The winning condition is derived from the property the network must satisfy. If player Router has a winning strategy σ, a routing algorithm that makes the network satisfy the given property under the constraints imposed by the topology exists and can be found from σ; else no such routing algorithm exists. We present several examples.

Finding a Routing Strategy to Prevent Buffer Overflows. Let $\lambda(q_c) = 0$ for each state q_c that represents a legal configuration c, and let $\lambda(\text{ERROR}) = 1$. If the pure threshold interface thus constructed is Δ-compliant for $\Delta = 0$, then a routing algorithm that prevents buffer overflow exists and can be synthesized from a winning strategy for player Router.

Finding the Optimal Buffer Size for a Given Topology. We wish to find out the smallest buffer capacity (less than a given bound) each piconode must have so that there exists a routing algorithm that prevents buffer overflows. Let $\lambda(q_c) = \max_i \sum_j c_{ij}$ for all nodes i and packet destinations j, where c_{ij} is the number of packets with destination j in node i in configuration c. The value of the pure threshold interface thus constructed gives the required smallest buffer size.

Checking if the Network Runs Forever Using Energy Interfaces. We wish to find if there exists a routing algorithm A_f that enables a piconet to run forever, assuming each piconode starts with energy e. Let e_{sc} be the energy scavenged by a piconode in each round. Let $\lambda(q_c) = e_{sc} - \max_i \sum_j (p \cdot \min(c_{ij}, l_i(r_i(j))))$, where q_c, i, j, and c_{ij} are as above, p is the energy spent to transmit a packet, $l_i(x)$ is the capacity of the link from node i to node x, and r_i is the routing table at node i; and $\lambda(\text{ERROR}) = -1$. If the pure energy interface thus constructed is Δ-compliant for $\Delta = e$, then A_f exists and is given by the Router strategy.

Finding the Minimum Energy Required to Achieve a Given Lifetime. We wish to find the minimum initial energy e such that there exists a routing algorithm A_r that makes each piconode run for at least r rounds. Let $\lambda(q_c) = (e_c, 1)$, where e_c is the energy label of configuration q_c defined in the pure energy interface above, and 1 is a reward; and $\lambda(\text{ERROR}) = (-1, 0)$. For $\Lambda = r$, the value of the Λ-reward energy interface thus constructed gives e, and the Router strategy gives A_r.

References

1. J.L. da Silva Jr., J. Shamberger, M.J. Ammer, C. Guo, S. Li, R. Shah, T. Tuan, M. Sheets, J.M. Rabaey, B. Nikolic, A. Sangiovanni-Vincentelli, and P. Wright. Design methodology for pico-radio networks. In *Proc. Design Automation and Test in Europe*, pp. 314–323. IEEE, 2001.
2. L. de Alfaro and T.A. Henzinger. Interface automata. In *Proc. Foundations of Software Engineering*, pp. 109–120. ACM, 2001.
3. L. de Alfaro and T.A. Henzinger. Interface theories for component-based design. In *Embedded Software*, LNCS 2211, pp. 148–165. Springer, 2001.
4. A. Ehrenfeucht and J. Mychielski. Positional strategies for mean-payoff games. *Int. J. Game Theory*, 8:109–113, 1979.
5. E.A. Emerson and C.S. Jutla. Tree automata, μ-calculus, and determinacy. In *Proc. Foundations of Computer Science*, pp. 368–377. IEEE, 1991.
6. J. Filar and K. Vrieze. *Competitive Markov Decision Processes*. Springer, 1997.
7. M. Hennessy and J. Riely. Information flow vs. resource access in the asynchronous π-calculus. In *Proc. Automata, Languages, and Programming*, LNCS 1853, pp. 415–427. Springer, 2000.
8. I. Lee, A. Philippou, and O. Sokolsky. Process-algebraic modeling and analysis of power-aware real-time systems. *Computing and Control Engineering J.*, 13:180–188, 2002.
9. M. Núñez and I. Rodríguez. PAMR: a process algebra for the management of resources in concurrent systems. In *Proc. Formal Techniques for Networked and Distributed Systems*, pp. 169–184. Kluwer, 2001.

10. R. Shah and J.M. Rabaey. Energy-aware routing for low-energy ad-hoc sensor networks. In *Proc. Wireless Communications and Networking Conference*, pp. 812–817. IEEE, 2002.
11. D. Walker, K. Crary, and G. Morrisett. Typed memory management via static capabilities. *ACM Trans. Programming Languages and Systems*, 22:701–771, 2000.
12. Chic: Checker for Interface Compatibility. www.eecs.berkeley.edu/∼tah/Chic.

Clocks as First Class Abstract Types

Jean-Louis Colaço[1] and Marc Pouzet[2]

[1] ESTEREL Technologies,
Park Avenue 9, Rue Michel Labrousse 31100 Toulouse, France
Jean-Louis.Colaco@esterel-technologies.com
[2] Laboratoire LIP6, Université Pierre et Marie Curie,
8 rue du Capitaine Scott, 75015 Paris, France.
Marc.Pouzet@lip6.fr

Abstract. Clocks in synchronous data-flow languages are the natural way to define several time scales in reactive systems. They play a fundamental role during the specification of the system and are largely used in the compilation process to generate efficient sequential code. Based on the formulation of clocks as *dependent types*, the paper presents a simpler clock calculus reminiscent to ML type systems with first order abstract types *à la* Laufer & Odersky. Not only this system provides clock inference, it shares efficient implementations of ML type systems and appears to be expressive enough for many real applications.

1 Introduction

1.1 Synchronous Data-Flow Programming

Synchronous languages have been introduced to deal with *real-time* systems, that is, systems requiring the ability to react to their environnement in bounded time and memory. These languages introduce a discrete and logical notion of time. In this logical time, events and the reaction of the system to their occurences have to be simultaneous and instantaneous. This hypothesis of instantaneous reaction, called the *synchronous hypothesis*, allow a concurrent but deterministic description of systems and efficient compilation of programs into sequential code.

Several languages have been designed on top of the synchronous hypothesis, imperative ones (Esterel [5]), graphical ones (Argos [18], SyncCharts [1], Ptolemy [7]) or data-flow ones (Lustre [14], Signal [3], Lucid Synchrone [23,11]). Synchronous data-flow languages manage (possibly) infinite sequences or *streams* as primitive values. Synchrony appears naturally in this model: streams are time indexed sequences and at time n, every stream takes its n^{th} value. Synchronous data-flow languages have proved to be well adapted to the design of sampled systems.

Lustre [14] has been successfully used by several industrial companies to implement safety critical systems in various domains like nuclear plants, civil aircrafts, transport and automotive systems. All these developments have been done using Scade, a graphical environment based on Lustre and distributed by Esterel Technologies [25].

1.2 Clocks

Synchronous systems can be described as the parallel composition of several processes [4]. Naturally, there arises a need to compose several processes computed on different rates. Rates in synchronous data-flow programming find a natural expression: the logical time defines the fastest possible rate in the system, called the *base clock*. All the other rates, or *clocks*, are derived from the base clock. The other clocks are slower than the base clock and are obtained by stating that a stream, on a given instant of the logical time, can be either absent or present. In the following example, x and y are two streams, and y is twice slower than x:

x	0	1	2	3	4	5	6	...
y	0		1		2		3	...

However, combining streams on different rates, such as x and y, can be tricky. In general, some synchronisation mechanism using possibly unbounded memory may be needed at run-time if such combinations are allowed [8]. Thus, data-flow synchronous languages provide a static analysis, checking that all combination of streams, even on different rates, can be evaluated without synchronisation mechanism (i.e, buffering mechanism). This analysis is known as the *clock calculus*.

Clocks have been originaly introduced in Lustre and Signal. In this paper, we focus our attention on the language Lucid Synchrone. Lucid Synchrone is as an extension of Lustre with features from ML-like languages, while retaining its fundamental properties: it is based on the same Kahn model [15]; it uses clocks as a way to deal with several time scales in a real-time system and programs are compiled into finite memory transition systems. Moreover, it provides powerfull extensions such as higher-order features, data-types, type and clock inference. Being an extension of Lustre, the results presented in the paper can be applied to Lustre as well. The language is used today by the Scade team at Esterel Technologies for the development of a new compiler of Scade, the industrial version of Lustre (see [12], for example).

1.3 Contribution

Previous works have shown that the clock calculus of Lustre could be defined as a classical *dependent type* system [10]. This clock calculus has been implemented in the first compiler of Lucid Synchrone (in 1996) and has proved to be both simple and expressive. It has served as a basis to implement a *clock verifier* inside a compiler for Scade at Esterel Technologies (in 2001). This compiler has been used in experiments on real-size examples (more than 50000 lines of code) and the dependent-based clock verifier appeared to be fast and satisfactory. Finally, a *shallow embedding* of Lucid Synchrone and its semantics into the Coq [13] proof assistant has been realised, establishing a correctness proof of the calculus [6].

Nonetheless, looking at real applications written in Scade, we observed that complex dependent clocks were not useful in many cases: most of the time, clocks are used locally in the so-called *activation condition* as a way to sample a process and clock inference could be obtained with a simpler calculus. Thus, this paper presents a new clock calculus which is expressive enough for most applications found in Scade. This calculus is based on classical Hindley-Milner type systems [20] with a restricted form of existential types as proposed by Laufer & Odersky [16]. This calculus shares efficient implementation techniques of ML type systems, it provides higher-order features and clock inference which is mandatory in a graphical environment such as Scade.

Section 2 gives an overview of this new calculus. Examples will be given in Lucid Synchrone syntax [1]. Examples will be kept first order such that they can be easily translated to Lustre syntax. In section 3, we define a (higher-order) synchronous data-flow kernel on which Lucid Synchrone is based and give it a Kahn semantics and a synchronous data-flow semantics. Section 4 is devoted to the clock calculus presentation. Section 5 illustrates its expressiveness on some typical examples. In section 7, we discuss related works and we conclude in section 8.

2 Overview

As any synchronous data-flow language, Lucid Synchrone is buit on top of a host language [2] used for writting primitive computations overs streams. Programs are written in an ML syntax but every ground type and value imported from the host language is implicitly lifted to streams. For example, `int` stands for the type of sequence of integers, `1` stands for the constant stream of values 1; `+` adds point-wise its two input streams and `if/then/else` is the point-wise conditional.

In the sequel we give on the right the graphical representation of some of the primitives in Scade.

`1`	1	1	1	$\dots$
`x`	x_0	x_1	x_2	$\dots$
`y`	y_0	y_1	y_2	$\dots$
`x + y`	$x_0 + y_0$	$x_1 + y_1$	$x_2 + y_2$	$\dots$
`c`	t	f	t	$\dots$
`if c then x else y`	x_0	y_1	x_2	$\dots$

Furthermore, `pre` (for "previous") is an instant delay operator and `->` is an initialization operator. The first value of `pre y` is undefined and noted *nil* [3]. `fby` is the initialised delay [4] and satisfies x `fby` $y = x$ `->` `pre` y.

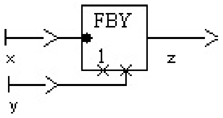
[1] A tutorial of Lucid Synchrone is available at `www-spi.lip6.fr/lucid-synchrone.html`.

[2] Ocaml [17] in the case of Lucid Synchrone.

[3] An initialisation analysis may check that the result of a program does not depend on these *nil* values [12].

[4] The operator was first introduced in Lucid [2].

`x -> y`	$x_0\ y_1\ y_2 \dots$
`pre y`	$nil\ y_0\ y_1 \dots$
`x fby y`	$x_0\ y_0\ y_1 \dots$



For example, we can compute the sum (such that, $s_n = \Sigma_{i=0}^{n} x_i$) of an input sequence `x` buy writting:

```
let sum x = s where
    rec s = x -> pre s + x
node sum : int -> int
node sum :: 'a -> 'a
```

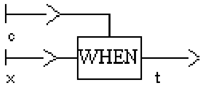
and the function which computes the minimum and the maximum of an input sequence:

```
let bounds x = (min, max) where
    rec min = x -> if x < pre min then x else pre min
    and max = x -> if x > pre max then x else pre max
node bounds : 'a -> 'a * 'a
node bounds :: 'a -> 'a * 'a
```

The two bounds are initialised to `x` and are modified step by step according to the current value of `x`. When this text is compiled, we obtain for each function a declaration (called `node` as in LUSTRE), the type (":") and clock inferences ("::"). For example, the clock `'a -> 'a` states that the function `sum` is length preserving: it returns a value every time it receives an input. Clocks are introduced below.

`when` is the *down*-sampling operator of LUSTRE which allows to extract a sub-stream from a stream according to a condition, *i.e.* a boolean stream.

`base`	$t\ \ t\ \ t\ \ t\ \dots$
`c`	$f\ \ t\ \ f\ \ t\ \dots$
`x`	$x_0\ x_1\ x_2\ x_3 \dots$
`x when c`	$x_1\ \ \ x_3 \dots$
`base on c`	$f\ \ t\ \ f\ \ t\ \dots$



The sampling operator introduces *clock types*. These clock types specify time behavior of stream programs.

The clock of a sequence s is a boolean sequence giving the instants where s is defined. Among these clocks, the constant boolean sequence `base` denotes the base clock of the system: a sequence on the base clock is present at every instant. In the above diagram, the current value of `x when c` is present when `x` and `c` are present and `c` is true. Since `x` and `c` are on the base clock `true`, the clock of `x when c` is noted `base on c`. Now the `sum` function can be used at a slower rate by down-sampling its input stream:

```
let sc x c = sum (x when c)
node sc : int -> clock -> int
node sc :: 'a -> (_c0:'a) -> 'a on _c0
```

`sc` has a function clock which follows its type structure. This clock says that for any clock `'a`, if the first argument `x` of the function has clock `'a` and the second argument named `_c0` has clock `'a`, then the result is on clock `'a on _c0` (variables are renamed in the clock type to avoid name conflicts). An expression on clock `'a on _c0` is present when the clock `'a` is true and the boolean stream `_c0` is present and true.

Now, the sampled sum can be instanciated with an actual clock. We first define a periodic clock which is true one instant of ten.

```
(* a sampler that counts modulo n *)
let sample n = ok where
    rec cpt = 0 -> if pre cpt = n - 1 then 0
                   else pre cpt + 1
    and ok = cpt = 0
node sample : int -> bool
node sample :: 'a -> 'a

(* defining a periodic 1/10 clock *)
let clock ten = sample 10
node ten : clock
node ten :: 'a

let sum_one_over_ten x = sc x ten
node sum_one_over_ten : int -> int
node sum_one_over_ten :: 'a -> 'a on ten
```

Thus, `sum_one_over_ten x` computes the sum of the sub-stream $(x_{10n})_{n \in I\!N}$. Here, boolean streams used to sample a stream have to be first introduced with the special construction `let clock`.

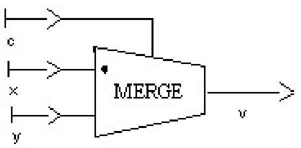
Programs must satisfy some clock constraints, as examplified on the following:

```
let unbounded x = x + (x when ten)
> x + (x when ten)
>      ^^^^^^^^^^
This expression has clock 'b on ten, but is used with clock 'b.
```

which adds the stream `x` to the sub-sequence of `x` made by filtering nine item over ten. Thus, it should compute the sequence $(x_n + x_{10n})_{n \in I\!N}$ which is clearly not bounded memory. This is why this program is rejected when dealing with real-time programming.

`merge` conversely allows slow processes to communicate with faster ones by merging sub-streams into "larger" ones:

`c`	f	t	f	t	$\ldots$
`x`		x_0		x_1	$\ldots$
`y`	y_0		y_1		$\ldots$
`merge c x y`	y_0	x_0	y_1	x_1	$\ldots$



The clock type of `merge` is:

```
let my_merge c x y = merge c x y
node my_merge : bool -> 'a -> 'a -> 'a
node my_merge :: (_c0:'a) -> 'a on _c0 -> 'a on not _c0 -> 'a
```

meaning that it combines *complementary* streams: when the second argument is present, the third one is absent and conversely.

The following operator is a classical control engineering operator, available in both the SCADE library and "digital" library of SIMULINK. This operator detects a rising edge (false to true transition). The output becomes true as soon as a transition has been detected and remains unchanged during `numberOfCycle` cycles. The output is initially false and a rising edge occuring while the output is true is detected. In LUCID SYNCHRONE syntax, this is written:

```
let count_down (reset, n) = cpt where
    rec cpt = if reset then n else (n -> pre (cpt - 1))

let risingEdgeRetrigger rer_Input numberOfCycle = rer_Output where
    rec rer_Output = (0 < v) & (c or count)
    and v = merge clk (count_down ((count,numberOfCycle) when clk))
                      ((0 fby v) whenot clk)
    and c = false fby rer_Output
    and clock clk = c or count
    and count = false -> (rer_Input & pre (not rer_Input))
```

When a clock name is introduced with the `let clock` constructor, the name is considered to be unique and does not take into account the expression on the right-hand side of the `let clock`. Thus, the following program is rejected:

```
let clock c = true in
let v = 1 when c in
let clock c = true in
let w = 1 when c in
v + w
> v + w
>     ^
This expression has clock 'a on ?_c0, but is used with clock 'a on ?_c1.
```

We stop the introductory examples here. Other examples can be found in the distribution [23]. Section 5 will give more examples of programs accepted (and rejected) with the proposed system. The main idea of this system is to replace expressions in clocks by abstract names introduced with the particular construction `let clock`. This is in contrast with previous versions of LUCID SYNCHRONE (as well as LUSTRE) where any boolean could be used to sample a stream. In term of a type system, the resulted clock calculus corresponds to a *standard* Hindley-Milner type system with a limited form of existential quantification as proposed by Laufer & Odersky [16]: the `let clock` construction corresponds to the access threw a `let` binding to a value belonging to an existential type.

3 A Synchronous Kernel and Its Semantics

We present here the core language used in this paper. It is a higher-order functional language over streams enriched with special operators for manipulating these streams:

$$\begin{array}{lll} e & ::= & i \mid x \mid op(e,e) \mid e \;\texttt{fby}\; e \\ & & \mid e \;\texttt{when}\; e \mid e \;\texttt{whenot}\; e \mid \texttt{merge}\; e\; e\; e \\ & & \mid e(e) \mid \lambda x.e \mid \texttt{rec}\; x = e \mid \texttt{let}\; x = e \;\texttt{in}\; e \\ & & \mid \texttt{let clock}\; x = e \;\texttt{in}\; e \\ & & \mid (e,e) \mid \texttt{fst}\; e \mid \texttt{snd}\; e \\ i & ::= & \texttt{true} \mid \texttt{false} \mid \texttt{0} \mid ... \\ op & ::= & + \mid ... \end{array}$$

An expression e may be an immediate constant (i), a variable (x), a point-wisely application of an operator to a tuple of inputs ($op(e,e)$) [5], an application of an initialised delay (`fby`) [6], an application of sampling operators (`when` and `whenot`) or the combination operator (`merge`), an application ($e(e)$), an abstraction $\lambda x.e$, a local definition (`let` $x = e$ `in` e), a local definition of a clock (`let clock` $x = e$ `in` e), a recursive expression (`rec` $x = e$), a pair (e,e) or an application of one of the pair access functions (`fst` and `snd`).

Other classical operators can be derived from this kernel. For example:

$$\begin{array}{ll} \texttt{if}\; x\; \texttt{then}\; y\; \texttt{else}\; z & = \texttt{let clock}\; c = x\; \texttt{in merge}\; c\; (y\; \texttt{when}\; c)\; (z\; \texttt{whenot}\; c) \\ x\; \texttt{->}\; y & = \texttt{if true fby false then}\; x\; \texttt{else}\; y \\ \texttt{pre}\, x & = nil\; \texttt{fby}\; x \end{array}$$

`->` is the initialisation operator and `pre` (for "previous") is the un-initialised delay. *nil* denotes any value with the correct type.

3.1 Data-Flow Kahn Semantics

We first give our core language a classical Kahn semantics [15] on sequences [7]. Let T^ω be the set of finite or infinite sequences of elements over the set T ($T^\omega = T^* + T^\infty$). The empty sequence is noted ϵ and $x.s$ denotes the sequence whose head is x and tail is s. Let $\leq$ be the prefix order over sequences, i.e., $x \leq y$ if x is a prefix of y. The ordered set $(T^\omega, \leq)$ is a cpo. If f is a continuous mapping from sequences to sequences, we shall write *fix* f for the smallest fix point of f.

If $T_1, T_2, ...$ are sets of scalar values (typically values imported from a host language), we define the domain V as the smallest set containing ${T_i}^\omega$ and closed by product and exponentiation.

For any assignment ρ (mapping values to variable names) and expressions e, we define the denotation of an expression e by $S_\rho(e)$. We first give an interpretation over sequences to every data-flow primitive and constant streams. Their definition is given in figure 1.

[5] For simplicity, we only consider binary operators in this presentation.
[6] This operator has been first introduced in Lucid [2].
[7] We keep here the original notation.

$$\begin{array}{ll}
i^{\#} & = i.i^{\#} \\
op^{\#}(s_1, s_2) & = \epsilon \text{ if } s_1 = \epsilon \text{ or } s_2 = \epsilon \\
op^{\#}(v_1.s_1, v_2.s_2) & = (v_1 \ op \ v_2).op^{\#}(s_1, s_2) \\
\\
\texttt{fby}^{\#}(\epsilon, ys) & = \epsilon \\
\texttt{fby}^{\#}(v.xs, ys) & = v.ys \\
\\
\texttt{when}^{\#}(s_1, s_2) & = \epsilon \text{ if } s_1 = \epsilon \text{ or } s_2 = \epsilon \\
\texttt{when}^{\#}(v_1.s_1, true.s_2) & = v_1.(\texttt{when}^{\#}(s_1, s_2)) \\
\texttt{when}^{\#}(v_1.s_1, false.s_2) & = \texttt{when}^{\#}(s_1, s_2) \\
\\
\texttt{merge}^{\#}(s_1, s_2, s_3) & = \epsilon \text{ if } s_1 = \epsilon \text{ or } s_2 = \epsilon \text{ or } s_3 = \epsilon \\
\texttt{merge}^{\#}(true.s_1, v_2.s_2, s_3) & = v_2.\texttt{merge}^{\#}(s_1, s_2, s_3) \\
\texttt{merge}^{\#}(false.s_1, s_2, v_3.s_3) & = v_3.\texttt{merge}^{\#}(s_1, s_2, s_3)
\end{array}$$

Fig. 1. Data-flow Kahn semantics

- immediate values (i) are lifted into infinite constant streams.
- operations are applied point-wisely to their arguments.
- the initialised delay (`fby`) *conses* the head of its first input to its second input.
- `when` and `merge` are filtering and combination operators overs sequences. The operation `when` emits a sub-sequence of its inputs corresponding to the instants where the condition is true. The operation `merge` merges two sub-sequences into a sequence: it emits the current value of its second inputs when the condition is true and the current value of its third input when the condition is false.

$$\begin{array}{rl}
S_\rho(op(e_1, e_2)) & = op^{\#}(S_\rho(e_1))(S_\rho(e_2)) \\
S_\rho(e_1 \ \texttt{fby} \ e_2) & = \texttt{fby}^{\#}(S_\rho(e_1), S_\rho(e_2)) \\
S_\rho(e_1 \ \texttt{when} \ e_2) & = \texttt{when}^{\#}(S_\rho(e_1), S_\rho(e_2)) \\
S_\rho(e_1 \ \texttt{whenot} \ e_2) & = \texttt{when}^{\#}(S_\rho(e_1), S_\rho(\texttt{not} \ e_2)) \\
S_\rho(\texttt{merge} \ e_1 \ e_2 \ e_3) & = \texttt{merge}^{\#}(S_\rho(e_1), S_\rho(e_2), S_\rho(e_3)) \\
S_\rho(x) & = \rho(x) \\
S_\rho(i) & = i^{\#} \\
S_\rho(\texttt{let} \ x = e_1 \ \texttt{in} \ e_2) & = S_{\rho[x \leftarrow S_\rho(e_1)]}(e_2) \\
S_\rho(\texttt{let clock} \ x = e_1 \ \texttt{in} \ e_2) & = S_{\rho[x \leftarrow S_\rho(e_1)]}(e_2) \\
S_\rho(\lambda x.e) & = \lambda y.S_{\rho[x \leftarrow y]}(e) \text{ where } y \notin Dom(\rho) \\
S_\rho(e_1(e_2)) & = S_\rho(e_1)(S_\rho(e_2)) \\
S_\rho(e_1, e_2) & = (S_\rho(e_1), S_\rho(e_2)) \\
S_\rho(\texttt{fst} \ e) & = v_1 \text{ if } S_\rho(e) = (v_1, v_2) \\
S_\rho(\texttt{snd} \ e) & = v_2 \text{ if } S_\rho(e) = (v_1, v_2) \\
S_\rho(\texttt{rec} \ x = e) & = \rho[x \leftarrow x^\infty] \text{ where } x^\infty = \mathit{fix} \ \lambda y.S_{\rho[x \leftarrow y]}(e)
\end{array}$$

We can easily check that the primitives are continuous for the prefix order. The denotational semantics for other constructions is defined as usual (see [24]). We recall it here shortly.

The semantics gives meaning to any Kahn network. Nonetheless, it is well known that simple data-flow networks cannot always be executed in bounded memory even when they are only composed of bounded memory operators. This problem appears when non length preserving functions (i.e, sampling functions) are considered [8] and has been exemplified in section 2.

3.2 Synchronous Data-Flow Semantics

In order to model synchronous execution, we describe a lower level data-flow semantics where absence of a stream is made explicit. The set of instantaneous values is enrichied with a special value *abs* representing the absence. The set of finite and infinite sequences of values belonging to T complemented with the absent value is noted T_{abs}^{ω} where $T_{abs} = T \cup \{abs\}$. That is:

$$\begin{array}{ll}
\mathit{Stream}\,(T) & = T.\mathit{Stream}\,(T) + \epsilon \\
\mathit{Value}\,(T) & = \mathit{abs} + T \\
\mathit{Clocked\text{-}Stream}\,(T) & = \mathit{Stream}\,(\mathit{Value}\,(T)) \\
\mathit{Clock} & = \mathit{Stream}\,(\mathit{bool})
\end{array}$$

A clocked stream (element of *Clocked-Stream* (T)) is made of present or absent values. We define the clock of a sequence s as a boolean sequence (without absent values) indicating when a value is present:

$$\begin{array}{ll}
\mathit{clock}\,(\epsilon) & = \epsilon \\
\mathit{clock}\,(\mathit{abs}.xs) & = \texttt{false}.\mathit{clock}\,(xs) \\
\mathit{clock}\,(x.xs) & = \texttt{true}.\mathit{clock}\,(xs)
\end{array}$$

We give a new interpretation to constant streams, to operators applied point-wisely and to synchronous primitives. This interpretation is given in figure 2 and is discussed below:

Constant Streams. This operator becomes polymorphic since it may produce a value (or not) according to the environment. For this reason, we add an extra argument giving its clock. Thus, $i[s]$ denotes a constant stream with clock s.

Point-Wise Application. We must decide here what to do, in the case of a binary operator, when only one of the argument is present. Three choices are possible:

1. store the available value in a state variable implementing a FIFO queue until the other input is present;
2. generate a run-time error;
3. reject statically this situation.

$$
\begin{array}{ll}
i^{\#}[\epsilon] & = \epsilon \\
i^{\#}[true.cl] & = v.i^{\#}[cl] \\
i^{\#}[false.cl] & = abs.i^{\#}[cl] \\
\\
op^{\#}(s_1, s_2) & = \epsilon \text{ if } s_1 = \epsilon \text{ or } s_2 = \epsilon \\
op^{\#}(abs.s_1, abs.s_2) & = abs.op^{\#}(s_1, s_2) \\
op^{\#}(v_1.s_1, v_2.s_2) & = (v_1\, op\, v_2).op^{\#}(s_1, s_2) \\
\\
\texttt{fby}^{\#}(\epsilon, ys) & = \epsilon \\
\texttt{fby}^{\#}(abs.xs, abs.ys) & = abs.\texttt{fby}^{\#}(xs, ys) \\
\texttt{fby}^{\#}(x.xs, y.ys) & = x.\texttt{fby1}^{\#}(y, xs, ys) \\
\texttt{fby1}^{\#}(v, \epsilon, ys) & = \epsilon \\
\texttt{fby1}^{\#}(v, abs.xs, abs.ys) & = abs.\texttt{fby1}^{\#}(v, xs, ys) \\
\texttt{fby1}^{\#}(v, w.xs, s.ys) & = v.\texttt{fby1}^{\#}(s, xs, ys) \\
\\
\texttt{when}^{\#}(s_1, s_2) & = \epsilon \text{ if } s_1 = \epsilon \text{ or } s_2 = \epsilon \\
\texttt{when}^{\#}(abs.xs, abs.cs) & = abs.\texttt{when}^{\#}(xs, cs) \\
\texttt{when}^{\#}(x.xs, true.cs) & = x.\texttt{when}^{\#}(xs, cs) \\
\texttt{when}^{\#}(x.xs, false.cs) & = abs.\texttt{when}^{\#}(xs, cs) \\
\\
\texttt{merge}^{\#}(s_1, s_2, s_3) & = \epsilon \text{ if } s_1 = \epsilon \text{ or } s_2 = \epsilon \text{ or } s_3 = \epsilon \\
\texttt{merge}^{\#}(abs.cs, abs.xs, abs.ys) & = abs.\texttt{merge}^{\#}(cs, xs, ys) \\
\texttt{merge}^{\#}(true.cs, x.xs, abs.ys) & = x.\texttt{merge}^{\#}(cs, xs, ys) \\
\texttt{merge}^{\#}(false.cs, abs.xs, y.ys) & = y.\texttt{merge}^{\#}(cs, xs, ys)
\end{array}
$$

Fig. 2. Synchronous data-flow semantics

In the context of real-time programming, and to be coherent with LUSTRE, only the third solution is retained. The point-wise application of an operator (*op*) needs its two arguments to be on the same clock. We need a static analysis — a *clock calculus* — to insure this property. This analysis will be presented in section 4.

Delay (`fby`). `fby` (for "followed by") is the unitary delay: it conses the head of its first argument to its second argument. The arguments and the result of `fby` must be on the same clock. `fby` corresponds to a two-state machine: while the two arguments are absent, it emits nothing and stays in its initial state `fby`. When the two arguments become present, it emits its first argument and goes into the new state `fby1` storing the previous value of its second argument. In this state, it emits a value every time its two arguments are present.

Sampling and Composition of Sequences (`When` *and* `Merge`*).* The sampling operator expects two arguments on the same clock. The clock of the result depends on the boolean condition (c). If the argument have a clock (cl), the clock of the result is (cl `on` c) such that:

$$\begin{array}{ll}
\texttt{on}^{\#}(\mathit{true}.cl, \mathit{true}.cs) & = \mathit{true}.\texttt{on}^{\#}(cl, cs) \\
\texttt{on}^{\#}(\mathit{true}.cl, \mathit{false}.cs) & = \mathit{false}.\texttt{on}^{\#}(cl, cs) \\
\texttt{on}^{\#}(\mathit{false}.cl, \mathit{abs}.cs) & = \mathit{false}.\texttt{on}^{\#}(cl, cs) \\
\texttt{on}^{\#}(cl, c) & = \epsilon \text{ if } cl = \epsilon \text{ or } c = \epsilon
\end{array}$$

We can notice that the clock is a simple sequence without absent values. The definition of `merge` states that one branch must be present when the other is absent.

Since this semantics is a partial semantics, we shall introduce clock constraints to reject programs which cannot be executed synchronously. This semantics has been described in the proof assistant Coq [6]. The use of an explicit absent value is part of the semantics of Signal [21]. Nonetheless, Signal cannot receive a Kahn semantics (functional and without absence) whereas in the case of Lustre and Lucid Synchrone, the use of absence is only done for characterising networks which can be executed synchronously.

4 Clock Calculus

We present now the clock calculus as a type system. The goal of the clock calculus is to produce jugments of the form $H \vdash e : cl$ meaning that "the expression e has clock cl in the environment H".

$$\begin{array}{lcl}
\sigma & ::= & \forall \beta_1, ..., \beta_n. \forall \alpha_1, ..., \alpha_m. \forall X_1, ..., X_k. cl \quad n, m, k \in \mathbb{N} \\
cl & ::= & cl \rightarrow cl \mid cl \times cl \mid \beta \mid s \mid (c : s) \\
s & ::= & \texttt{base} \mid s \ \texttt{on} \ c \mid s \ \texttt{on not} \ c \mid \alpha \\
c & ::= & X \mid n \\
 & & \\
H & ::= & [x_1 : \sigma_1, ..., x_n : \sigma_n]
\end{array}$$

Clock types are decomposed into two categories, clock schemes (σ) and clocks (cl). Clock schemes are regular clocks quantified over clock type variables (β), stream clock type variables (α) and carrier variables (X). Regular clocks (cl) may be functional ($cl \rightarrow cl$), products ($cl \times cl$), variables (β), stream clocks (s) and dependences ($c : s$). A stream clock may be the base clock (`base`), a sampled clock (s `on` c or s `on not` c) or a stream clock variable (α). A carrier (c) can be either a name (n) or a carrier variable (X).

We define the set of free variables $FV(cl)$ of a clock cl composed of free clock variables (β), free stream clock variables (α) and free carrier variables (X). We extend it to clock schemes and environments. $Dom(H)$ is the domain of H. $N(cl)$ returns the set of abstract names from cl. Their definitions are classical and not given in the paper.

Programs are clocked in an initial environment H_0 giving clocks to synchronous primitives. To make the clock calculus shorter, a synchronous expression like e_1 `fby` e_2 is treated as a regular application of a synchronous primitive to expressions and is clocked as if it were written $(\texttt{fby}\,(e_1))\,e_2$.

$$\begin{array}{rll} H_0 = [& \texttt{fby} & : \forall\alpha.\alpha \rightarrow \alpha \rightarrow \alpha \\ & \texttt{when} & : \forall\alpha.\forall X.\alpha \rightarrow (X:\alpha) \rightarrow \alpha \texttt{ on } X \\ & \texttt{whenot} & : \forall\alpha.\forall X.\alpha \rightarrow (X:\alpha) \rightarrow \alpha \texttt{ on not } X \\ & \texttt{merge} & : \forall\alpha.\forall X.(X:\alpha) \rightarrow \alpha \texttt{ on } X \rightarrow \alpha \texttt{ on not } X \rightarrow \alpha \\ & \texttt{fst} & : \forall\beta_1,\beta_2.\beta_1 \times \beta_2 \rightarrow \beta_1, \\ & \texttt{snd} & : \forall\beta_1,\beta_2.\beta_1 \times \beta_2 \rightarrow \beta_2] \end{array}$$

The first entry in this initial environment states that the clock of x `fby` y is the one of x and y. A boolean expression e with clock $(c:s)$ states that e is present when s is true and has the abstract value c. Thus, an expression e_1 `when` e_2 is well clocked if the two inputs are synchronous on the clock α. In that case, the clock of the result is a subclock α `on` X of α where X stands for the value of e_2. An expression `merge` $e\ e_1\ e_2$ is well clocked and on clock α if e is on clock α and is equal to some X, e_1 is on clock α `on` X and e_2 on clock α `on not` X.

Clocks may be instanciated and generalised in the following way:

$$inst(\forall\boldsymbol{\beta}.\boldsymbol{\alpha}.\boldsymbol{X}.cl') = cl'[\boldsymbol{cl}/\boldsymbol{\beta}][\boldsymbol{s}/\boldsymbol{\alpha}][\boldsymbol{c}/\boldsymbol{X}]$$

$$gen_H(cl) = \forall\beta_1,...,\beta_n.\forall\alpha_1,...,\alpha_m.\forall X_1,...,X_k.cl \quad \text{where } \beta_1,...,\beta_n,\alpha_1,...,\alpha_m,X_1,...,X_k = FV(cl)\backslash FV(H)$$

It states that a clock scheme can be instanciated by replacing clock variables by clocks, stream clock variables by stream clocks and carrier variables by carriers. Moreover, any variable can be generalized as soon as it does not appear free in the environment. The clocking rules are now given in figure 3.

– a constant stream may have any clock (rule (IM))
– the inputs of imported primitives must all be on the same clock (rule (OP))
– the rules for variables (INST), functions (FUN), applications (APP), local definitions (LET) and products (PAIR) are the classical typing rules of ML.
– a clock definition `clock` $x = e$ defines a new clock name n which has the clock type s. In doing this, n must be a fresh name, that is, it must not appear free in the environment nor in the returned clock cl_2. It states that if x evaluates to some value n and is on clock s, then the clock of e_2 is cl_2. The symbol n is an abstraction of the actual value of e_1 and is considered to be unique.

The clocking rule for the `let clock` construction is a particular case of the typing rule for `let` in the Laufer & Odersky type system. Here, the *abstract name* n corresponds to the introduction of a fresh skolem name.

4.1 Correctness Theorem and the Use of Clocks

The clock calculus is used for giving a synchronous data-flow semantics to programs and to establish a correctness property: well clocked programs can be executed synchronously in the synchrounous data-flow semantics without raising

$$\text{(IM)}\ H \vdash i : s \qquad \text{(OP)}\ \frac{H \vdash e_1 : s \quad H \vdash e_2 : s}{H \vdash op(e_1, e_2) : s} \qquad \text{(APP)}\ \frac{H \vdash e_1 : cl_2 \rightarrow cl_1 \quad H \vdash e_2 : cl_2}{H \vdash e_1\,(e_2) : cl_1}$$

$$\text{(FUN)}\ \frac{H, x : cl_1 \vdash e : cl_2}{H \vdash \lambda x.e : cl_1 \rightarrow cl_2} \qquad \text{(INST)}\ \frac{cl = inst(H(x))}{H \vdash x : cl} \qquad \text{(REC)}\ \frac{H, x : s \vdash e : s}{H \vdash \texttt{rec}\ x = e : s}$$

$$\text{(LET)}\ \frac{H \vdash e_1 : cl_1 \quad H, x : gen_H(cl_1) \vdash e_2 : cl_2}{H \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : cl_2} \qquad \text{(PAIR)}\ \frac{H \vdash e_1 : cl_1 \quad H \vdash e_2 : cl_2}{H \vdash (e_1, e_2) : cl_1 \times cl_2}$$

$$\text{(LET-clock)}\ \frac{H \vdash e_1 : s \quad H, x : (n : s) \vdash e_2 : cl_2 \quad n \notin N(H, cl_2)}{H \vdash \texttt{let clock}\ x = e_1\ \texttt{in}\ e_2 : cl_2}$$

Fig. 3. The Clock Calculus

execution errors. In practice, the clock information is used in synchronous compilers in the optimization process in order to avoid the expensive representation of "presence/absense" at run-time. Once the clock analysis is performed, every computation is annotated with its clock which serves as a guard: a computation is made only when its clock is true.

A detailed presentation of the use of clocks in the compilation process is out of scope of this paper. We simply define a translation of programs into programs where constants are annotated with their clocks. This translation follows exactly the clock calculus and is obtained by asserting the judgment:

$$H \vdash e : cl \Rightarrow ce$$

meaning that the expression e with clock cl can be transformed into the expression ce. The goal of this transformation is to produce a new program where expressions are annotated with their clocks. For example:

$$f = \lambda x.(0\ \texttt{fby}\ x) + 2 : \forall\alpha.\alpha \rightarrow \alpha$$

where + is the point-wise sum will be transformed into:

$$f = \lambda\alpha.\lambda x.(0[\alpha]\ \texttt{fby}\ x) + 2[\alpha]$$

The function f whose clock is polymorphic receives an extra argument α giving its clock. Then at instanciation point, the function will receive its clock as an extra argument. For example, if x is on clock s_1:

$$f\,(x\ \texttt{when}\ c)$$

is transformed into:

$$f\,s_1\,(x\ \texttt{when}\ c)$$

Our translation corresponds to the classical "library" method for compiling type classes in Haskell and first introduced in [26]. Clock variables are then abstracted at generalisation points and instanciated at application points. We first

$$\text{(IM)}\ H \vdash i : s \Rightarrow i[s] \qquad \text{(FUN)}\ \frac{H, x : cl_1 \vdash e : cl_2 \Rightarrow ce}{H \vdash \lambda x.e : cl_1 \to cl_2 \Rightarrow \lambda x.ce} \qquad \text{(INST)}\ \frac{cl = inst(H(x))}{H \vdash x : cl \Rightarrow instcode_{H(x)}(x)}$$

$$\text{(OP)}\ \frac{H \vdash e_1 : s \Rightarrow ce_1 \quad H \vdash e_2 : s \Rightarrow ce_2}{H \vdash op(e_1, e_2) : s \Rightarrow op(ce_1, ce_2)} \qquad \text{(PAIR)}\ \frac{H \vdash e_1 : cl_1 \Rightarrow ce_1 \quad H \vdash e_2 : cl_2 \Rightarrow ce_2}{H \vdash (e_1, e_2) : cl_1 \times cl_2 \Rightarrow (ce_1, ce_2)}$$

$$\text{(APP)}\ \frac{H \vdash e_1 : cl_2 \to cl_1 \Rightarrow ce_1 \quad H \vdash e_2 : cl_2 \Rightarrow ce_2}{H \vdash e_1\,(e_2) : cl_1 \Rightarrow ce_1\,(ce_2)} \qquad \text{(REC)}\ \frac{H, x : s \vdash e : s \Rightarrow ce}{H \vdash \mathtt{rec}\ x = e : s \Rightarrow \mathtt{rec}\ x = ce}$$

$$\text{(LET)}\ \frac{H \vdash e_1 : cl_1 \Rightarrow ce_1 \quad H, x : gen_H(cl_1) \vdash e_2 : cl_2 \Rightarrow ce_2}{H \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : cl_2 \Rightarrow \mathtt{let}\ x = gencode_H(cl_1, ce_1)\ \mathtt{in}\ ce_2}$$

$$\text{(LET-clock)}\ \frac{H \vdash e_1 : s \Rightarrow ce_1 \quad H, x : (n : s) \vdash e_2 : cl_2 \Rightarrow ce_2 \quad n \notin N(H, cl_2)}{H \vdash \mathtt{let\ clock}\ x = e_1\ \mathtt{in}\ e_2 : cl_2 \Rightarrow \mathtt{let}\ x = ce_1\ \mathtt{in}\ \mathtt{let}\ n = x\ \mathtt{in}\ ce_2}$$

Fig. 4. Transforming streams into clocked-streams

introduce two auxiliary functions:

$$\begin{array}{ll}
gencode_H(cl, ce) & = \lambda\alpha_1, ..., \alpha_m.X_1, ..., X_k.ce \\
& \quad \text{where } \beta_1, ..., \beta_n, \alpha_1, ..., \alpha_m, X_1, ..., X_k = FV(cl)\backslash FV(H) \\
instcode_{cl}(x) & = x \\
instcode_{\forall\boldsymbol{\beta}.\boldsymbol{\alpha}.\boldsymbol{X}.cl'}(x) & = x\, s_1...s_n.c_1...c_k \text{where } inst(\forall\boldsymbol{\beta}.\boldsymbol{\alpha}.\boldsymbol{X}.cl') = cl'[\boldsymbol{cl}/\boldsymbol{\beta}][\boldsymbol{s}/\boldsymbol{\alpha}][\boldsymbol{c}/\boldsymbol{X}]
\end{array}$$

The predicate is defined in figure 4.

- immediate constants receive an extra argument giving their clock.
- rules for abstractions (FUN), applications (APP), recursions (REC), products (PAIR) are simple morphisms.
- clocks are passed at instantiation points (rule (INST)) and abstracted at generalization points (rule (LET)). Here, only stream clock variables (s) or carrier variables (X) from the scheme clock are used since only they may appear in the sampling of a stream value.
- for clock definitions (rule (LET-clock)), the name n is introduced in the generated code since it may appear in some computation of a clock.

We then prove that this transformation will not provide programs producing incomplete pattern failure. For this purpose, we define a valuation $\mathcal{V}$ from variables to boolean streams and lift it to clocks such that:

$$\begin{array}{ll}
\mathcal{V}(\mathtt{base}) & = \mathtt{true} \\
\mathcal{V}(s\ \mathtt{on}\ c) & = \mathtt{on}^{\#}(\mathcal{V}(s), \mathcal{V}(c)) \\
\mathcal{V}(s\ \mathtt{on\ not}\ c) & = \mathtt{on}^{\#}(\mathcal{V}(s), \mathtt{not}^{\#}(\mathcal{V}(c)))
\end{array}$$

We define interpretation functions $\mathcal{I}_.(.)$ relating clock schemes and set of values. An interpretation is such that:

$v \in \mathcal{I}_\mathcal{V}(s)$	iff $clock\,(v) \leq \mathcal{V}(s)$ where $\leq$ stands for the prefix order
$v \in \mathcal{I}_\mathcal{V}((c : s))$	iff $v \in \mathcal{I}_\mathcal{V}(s)$ and $v = \mathcal{V}(c)$
$v \in \mathcal{I}_\mathcal{V}(cl_1 \rightarrow cl_2)$	iff for all v_1 such that $v_1 \in \mathcal{I}_\mathcal{V}(cl_1), v(v_1) \in \mathcal{I}_\mathcal{V}(cl_2)$
$(v_1, v_2) \in \mathcal{I}_\mathcal{V}(cl_1 \times cl_2)$	iff $v_1 \in \mathcal{I}_\mathcal{V}(cl_1)$ and $v_2 \in \mathcal{I}_\mathcal{V}(cl_2)$
$v \in \mathcal{I}_\mathcal{V}(\forall \beta_1, ..., \beta_k.\sigma)$	iff for all $cl_1, ..., cl_k, v \in \mathcal{I}_\mathcal{V}(\sigma[cl_1/\beta_1, ..., cl_k/\beta_k])$
$v \in \mathcal{I}_\mathcal{V}(\forall \alpha_1, ..., \alpha_n.X_1, ..., X_k.\sigma)$	iff for all $s_1, ..., s_n, c_1, ..., c_k$, $v\,(\mathcal{V}s_1), ..., (\mathcal{V}s_n)\,(\mathcal{V}c_1), ..., (\mathcal{V}c_k) \in \mathcal{I}_\mathcal{V}(cl[s_1/\alpha_1, ..., s_n/\alpha_n, c_1/X_1, ..., c_k/X_k])$

Theorem 1 (Clock Soundness). *If $[x_1 : \sigma_1, ..., x_n : \sigma_n] \vdash e : cl \Rightarrow ce$ then for all valuation $\mathcal{V}$., for all interpretation function $\mathcal{I}_.(.)$, for all $v_1, ..., v_n$ such that $v_1 \in \mathcal{I}_\mathcal{V}(\sigma_1), ..., v_n \in \mathcal{I}_\mathcal{V}(\sigma_n)$, we have $\mathrm{S}_{\mathcal{V}[v_1/x_1,...,v_n/x_n]}(ce) \in \mathcal{I}_\mathcal{V}(cl)$.*

The proof is given in appendix A.

The clock calculus presented in this paper is implemented in the Lucid Synchrone compiler and is in used for more than one year. Classical implementation techniques for ML type systems have been used (e.g, destructive unification). The technique for checking that the introduced name in the rule (LET-clock) is a fresh name and does not escape its scope is due to Pottier [19] and is used for the efficient implementation of the Laufer & Odersky system.

Practical experiments show that the clock calculus is as fast as the type inference which means that it can be applied to real-size examples.

5 Examples

We illustrate the expressivity of this calculus on some typical examples. A Lucid Synchrone program is a sequence of global declarations. A global declaration defines a name which can be used after its declaration. Every global declaration is analysed sequentially and compiled.

5.1 Activation Conditions

A classical primitive in a block diagram framework (such as the one of Scade) is the *activation condition* (also known as *enable sub-system* in Simulink). It consists of executing a node only when a condition is true. In term of clocks, it corresponds to filtering the input of that node on a certain clock `clk` and to project the result on the base clock. Such an operation is a higher-order construct and can be defined like the following:

```
let condact clk f input init =
    let rec u = merge clk (f (input when clk))
                          ((init fby u) whenot clk) in
    u
node condact : clock -> ('a -> 'b) -> 'a -> 'b -> 'b
node condact :: (clk:'a) -> ('a on clk -> 'a on clk) -> 'a -> 'a -> 'a
```

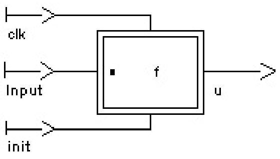


Fig. 5. SCADE notation for activation condition.

Its graphical representation in SCADE is given in figure 5. Using the `condact` primitive, we can rewrite the *rising edge retrigger* as it is written in the SCADE library.

```
let risingEdgeRetrigger rer_Input numberOfCycle = rer_Output where
    rec rer_Output = (0 < v) & (c or count)
    and v = condact clk count_down (count,numberOfCycle) 0
    and c = false fby rer_Output
    and clock clk = c or count
    and count = false -> (rer_Input & pre (not rer_Input))
```

Its graphical representation is given in figure 6.

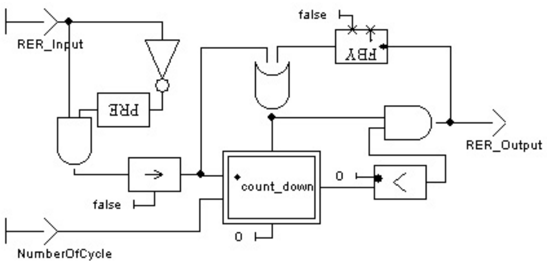


Fig. 6. The `risingEdgeRetrigger` in SCADE

5.2 Iteration

The symetric operation of the activation condition is an *iterator*. It corresponds to the writting of an internal `for` or `while` loop. This operator consists in iterating a function on an input.

```
let iter clk init f input =
    let rec o = f i
    and i = merge clk input ((init fby o) whenot clk) in
    o when clk
node iter :: (clk:'a) -> 'a -> ('a -> 'a) -> 'a on clk -> 'a on clk
```

5.3 Scope Restriction

Being based on the Laufer & Odersky type system, the clock calculus suffers from the same limitations. Mainly, when clock names are introduced with the `let clock` construction, these names must not escape their lexical scope. For example:

```
let escape x =
    let clock c = (x = 0) in
    x when c

>....escape x =
> let clock c = (x = 0) in
> x when c
The clock of this expression depends on ?c_0 which escape its scope.
```

Thus, the clock calculus is less expressive than the previous clock calculus of Lucid Synchrone or the one of Lustre where a result can depend on a clock computed internally as soon as the clock is returned as a result. This could be considered as a *serious* restriction. Quite surprisingly, this is not the case in practice mostly because most (near all!) programs found in Scade use clocks in a limited way corresponding to the *activation condition* and because these languages do not provide higher-order features.

6 Extension: Clocks Defined at Top-Level

We have defined (and implemented) an extension of the presented clock calculus with *top-level clocks*. We call *top-level* or *constant* clocks, clocks defined globally at top-level of a program. These clocks do not depend on any input of the program but they may, themselves, be instanciated on different clocks. Consider, for example:

```
let clock sixty = sample 60 (* 1/60 *)
node sixty : clock
node sixty :: 'a
```

It defines a periodic clock `sixty`. Using it, we can define a (real) clock in the following way:

```
let hour_minute_second second =
    let minute = second when sixty in
    let hour = minute when sixty in
    hour,minute,second

node hour_minute_second : 'a -> 'a * 'a * 'a
node hour_minute_second ::
     'a -> 'a on sixty on sixty * 'a on sixty * 'a
```

A stream on clock `'a on sixty on sixty` is only present one instant over 3600 instants which match perfectly what we are expecting.

Using clocks defined globally, we can write simple over-sampling functions. Consider, for example, the computation of the sequence $(o_n)_{n \in \mathbb{N}}$ such that:

$$o_{2n} = x_n$$
$$o_{2n+1} = x_n$$

It can be programmed in the following way:

```
let clock half = h where
    rec h = true -> not (pre h)
let stuttering x = o where
    rec o = merge half x (0 -> (pre o) whenot half)
node stuttering :: 'a on half -> 'a
```

This is a true example of oversampling, that is, a function whose internal clock is *faster* than the clock of its input. This example shows that some limited form of oversampling — which is possible in SIGNAL and not in LUSTRE — can be achieved with simple typing techniques.

It appears that top-level clocks are sufficient to express many programs appearing in the SCADE environment. In particular, many clocks are periodic [8]. Periodic (constant) clocks are useful for specifying hard real-time constraints from the environment and to direct the scheduling strategy of the compiler [9]. It is an open question to know whether constant clocks — which only need a very modest, dependent-less type system — are sufficient for this purpose.

In term of type system, these constant clocks defined at top-level do not raise any technical difficulty. Clock names defined at top-level are constant names and act as new type constructors with arity zero defined in ML languages. Then, as it is the case for clock names defined locally, two clocks are equal if they have the same name.

7 Related Works

LUCID SYNCHRONE has been originally developed as a functional extension of LUSTRE and to serve as an experimental language for prototyping extensions for it. It is based on the same semantic model and clocks are largely reminiscent of the ones in LUSTRE. Up to syntactic details, the clock calculus presented in this paper can be applied directly to LUSTRE programs. Nonetheless, the clock calculus is expressed here as a typing problem: this allows clocks to be automatically inferred using standard techniques and is compatible with higher-order. In comparison, LUSTRE is first-order and clocks are verified instead of being inferred. Clock inference is mandatory in a graphical programming environment such as the one of SCADE (the clock of intermediate wires must be computed

[8] Typically, an application is made of several main behaviors, each of them being executed at different periodic (constant) clocks, e.g., corresponding to 60hz, 200hz.

automatically) and one of our motivation is to design an efficient clock inference mechanism for SCADE. Finally, higher-order features appeared to be very useful for the prototyping of special purpose primitives before their ad-hoc incoding inside the SCADE compiler.

The present clock calculus can also be related to the one of SIGNAL. Clocks in SIGNAL can be arbitrary boolean expressions and the language supports full over-sampling. As a result, the clock calculus of SIGNAL reaches an impressive expressive power. Two streams with unrelated clocks can be combined thru the operator `default`, and the clock of the resulting stream is the "union" of the clocks of the two arguments. However, this expressiveness comes at the price of a greater complexity. In particular, the clock calculus of SIGNAL calls for boolean resolution techniques and fix-point iteration whereas we use simpler unification techniques.

This new calculus is less expressive than the previous clock calculus of LUCID SYNCHRONE, based on dependent types. The main difference is that clock types only contain *abstract names* introduced with the special primitive `let clock` whereas any boolean expression could be used for sampling a stream. From a SCADE user point of view, the need to name each clock and introduce it with a special construct (here `let clock`) is not a problem; people developing safety critical applications are pretty familiar with this kind of discipline.

8 Conclusion

In this paper we have presented a simple type-based clock inference calculus for a synchronous data-flow language providing higher-order features such as LUCID SYNCHRONE. The system is based on the extension of an ML type system with first class abstract types proposed by Laufer & Odersky. This calculus has been obtained by forbiding general boolean expressions in clock types, allowing them to contain only abstract names. These abstract names denote special boolean streams which are used to sample a stream. They can be introduced through the dedicated construction `let clock`.

We discovered recently that the idea of replacing boolean expressions by names in clocks has been already suggested by other implementors of synchronous compilers [22]. Nonetheless, it does not seem that the resulting clock calculus has been identified nor implemented.

Our motivation in doing such a clock calculus was mainly pragmatic. After several years of use of a dependent-type based clock calculus and looking at programs written in SCADE and LUSTRE, we observed that most of the time complex dependences in clock are useless and that the simplification proposed here is expressive enough for many real applications. The new system is simpler to use and it shares standard theory and implementation techniques of ML type systems.

Finally, we believe that bridging together the clock calculus and standard ML typing may contribute to the use of clocks as a good programming discipline in synchronous data-flow languages.

References

1. Charles André. Representation and Analysis of Reactive Behaviors: A Synchronous Approach. In *CESA*, Lille, july 1996. IEEE-SMC. Available at: `www-mips.unice.fr/~andre/synccharts.html`.
2. E. A. Ashcroft and W. W. Wadge. *Lucid, the data-flow programming language.* Academic Press, 1985.
3. 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.
4. G. Berry. Real time programming: Special purpose or general purpose languages. *Information Processing*, 89:11–17, 1989.
5. G. Berry and G. Gonthier. The Esterel synchronous programming language, design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.
6. Sylvain Boulmé and Grégoire Hamon. Certifying Synchrony for free. In *International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, volume 2250, La Havana, Cuba, December 2001. Lecture Notes in Artificial Intelligence, Springer-Verlag. Short version of *A clocked denotational semantics for Lucid-Synchrone in Coq*, available as a Technical Report (LIP6), at `http://www-spi.lip6.fr/lucid-synchrone`.
7. J. Buck, S. Ha, E. Lee, and D. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *International Journal of computer Simulation*, 1994. special issue on Simulation Software Development.
8. P. Caspi. Clocks in dataflow languages. *Theoretical Computer Science*, 94:125–140, 1992.
9. P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis, P. Niebert From Simulink to SCADE/Lustre to TTA: a layered approach for distributed embedded applications. LCTES'03.
10. Paul Caspi and Marc Pouzet. Synchronous Kahn Networks. In *ACM SIGPLAN International Conference on Functional Programming*, Philadelphia, Pensylvania, May 1996.
11. Paul Caspi and Marc Pouzet. Lucid Synchrone, a functional extension of Lustre. submitted to publication, 2001.
12. Jean-Louis Colaço and Marc Pouzet. Type-based Initialization of a Synchronous Data-flow Language. In *Synchronous Languages, Applications, and Programming*, volume 65. Electronic Notes in Theoretical Computer Science, 2002.
13. The coq proof assistant, 2002. `http://coq.inria.fr`.
14. 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.
15. G. Kahn. The semantics of a simple language for parallel programming. In *IFIP 74 Congress.* North Holland, Amsterdam, 1974.
16. Konstantin Läufer and Martin Odersky. An extension of ML with first-class abstract types. In *ACM SIGPLAN Workshop on ML and its Applications, San Francisco, California*, pages 78–91, June 1992.
17. Xavier Leroy. The Objective Caml system release 3.06. Documentation and user's manual. Technical report, INRIA, 2002.
18. Florence Maraninchi and Yann Rémond. Argos: an automaton-based synchronous language. *Computer Languages*, (27):61–92, 2001.

19. Michel Mauny and François Pottier. An implementation of Caml Light with existential types. Technical Report 2183, INRIA, October 1993.
20. R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Science*, 17:348–375, 1978.
21. D. Nowak, J. R. Beauvais, and J. P. Talpin. Co-inductive axiomatisation of synchronous language. In *International Conference on Theorem Proving in Higher-Order Logics (TPHOLs'98)*. Springer Verlag, October 1998.
22. Lecheck Olendersky. Private communication, December 2002. Workshop Synchron, Toulon, France.
23. Marc Pouzet. *Lucid Synchrone, version 2. Tutorial and reference manual.* Université Pierre et Marie Curie, LIP6, Mai 2001. Distribution available at: `www-spi.lip6.fr/lucid-synchrone`.
24. John C. Reynolds. *Theories of Programming Languages.* Cambridge University Press, 1998.
25. SCADE. `http://www.esterel-technologies.com/scade/`.
26. P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 60–76. ACM, January 1989.

A Proof of Theorem 1 (Clock Soundness)

The property is proved by induction on the structure of expressions.

Case $e' = i$. We have $\vdash i : s \Rightarrow i[s]$. By definition, $\mathit{clock}\,(S_{\mathcal{V}}(i[s])) = \mathcal{V}(s)$.

Case $e' = x$. We have $H, x : \sigma \vdash x : cl \Rightarrow \mathit{instcode}_H(x)$. There are several cases. Either $\sigma = cl$ or some variables are universally quantified.

– If $\sigma = cl$ then $H, x : cl \vdash x : cl \Rightarrow x$ and the property holds by definition.
– Let $\sigma = \forall \beta_1, ..., \beta_n.\alpha_1, ..., \alpha_m.X_1, ..., X_k.cl$. Let $H = [\sigma_1/x_1, ..., \sigma_p/x_p, \sigma/x]$ be the typing environment and $[v_1/x_1, ..., v_p/x_p, v/x]$ the corresponding environment for the evaluation. Let $\rho = \mathcal{V}[v_1/x_1, ..., v_p/x_p, v/x]$.

  Let v such that $v \in \mathcal{I}_{\mathcal{V}}(\forall \beta_1, ..., \beta_n.\alpha_1, ..., \alpha_m.X_1, ..., X_k.cl)$. According to the definition of $\mathcal{I}_.(.)$, for all $cl_1, ..., cl_n, s_1, ..., s_m, c_1, ..., c_k$, we have
  $v\,(\mathcal{V}s_1)...(\mathcal{V}s_n).(\mathcal{V}c_1)...(\mathcal{V}c_k) \in \mathcal{I}_{\mathcal{V}}(cl[cl_1/\beta_1, ..., cl_n/\beta_n][s_1/\alpha_1, ..., s_m/\alpha_m]$
  $[c_1/X_1, ..., c_k/X_k])$.
  The property holds since:
  $v\,(\mathcal{V}s_1)...(\mathcal{V}s_n).(\mathcal{V}c_1)...(\mathcal{V}c_k) = (S_\rho(x\,s_1...s_n.c_1...c_k)) = S_\rho(\mathit{instcode}_H(x))$.

Case of Primitives. Direct recurrence.

Case $e' = e_1(e_2)$. Let H be the typing environment and ρ, the corresponding evaluation environment. Suppose the property holds for e_1 and e_2, that is $S_{\mathcal{V}}(ce_1) \in \mathcal{I}_{\mathcal{V}}(cl_1 \rightarrow cl_2)$ and $S_{\mathcal{V}}(ce_2) \in \mathcal{I}_{\mathcal{V}}(cl_1)$. According to the definition of the interpretation $\mathcal{I}_.(.)$, for any $v \in \mathcal{I}_{\mathcal{V}}(cl_1)$ we have $(S_{\mathcal{V}}(ce_1))(v) \in \mathcal{I}_{\mathcal{V}}(cl_2)$. Thus, $(S_{\mathcal{V}}(ce_1))(S_{\mathcal{V}}(ce_2)) = S_{\mathcal{V}}(ce_1(ce_2)) \in \mathcal{I}_{\mathcal{V}}(cl_2)$. Thus the property holds.

Case $e' = \lambda y.e$. Let H be the typing environment and ρ, the corresponding evaluation environment. Suppose that $H, y : cl_y \vdash e : cl \Rightarrow ce$. Applying the recurrence hypothesis, for all ρ corresponding to H and for all $v_y \in \mathcal{I}_{\mathcal{V}}(cl_y)$ we have $S_{\mathcal{V}[v_y/y]}(ce) \in \mathcal{I}_{\mathcal{V}}(cl)$. Thus, for all $v_y \in \mathcal{I}_{\mathcal{V}}(cl_y)$, we have $S_{\mathcal{V}[v_y/y]}((\lambda y.ce)\, y) \in \mathcal{I}_{\mathcal{V}}(cl)$. Thus, for all $v_y \in \mathcal{I}_{\mathcal{V}}(cl_y)$, we have $(S_{\mathcal{V}}(\lambda y.ce))(v_y) \in \mathcal{I}_{\mathcal{V}}(cl)$. Thus the property holds.

Case of $e' = \texttt{let}\ y = e_1\ \texttt{in}\ e_2$. Direct recurrence and combination of the preceding rules.

Case of Clock Declarations $e = \texttt{let clock}\ x = e_1\ \texttt{in}\ e_2$. Suppose that the property holds for $H \vdash e_1 : s_1 \Rightarrow ce_1$, that is, for all $\mathcal{V}$ and $[v_1/x_1, ..., v_n/x_n]$ such that $v_1 \in \mathcal{I}_{\mathcal{V}}(\sigma_1), ..., v_n \in \mathcal{I}_{\mathcal{V}}(\sigma_n)$, $S_{\mathcal{V}[v_1/x_1,...,v_n/x_n]}(ce_1) \in \mathcal{I}_{\mathcal{V}}(s_1)$. Let n be a fresh name $n \notin N(H)$. Let $\mathcal{V}'$ be an extension of $\mathcal{V}$ such that $\mathcal{V}'(z) = \mathcal{V}(z)$ if $z \neq n$. Then for all $\mathcal{V}'$ extending $\mathcal{V}$, for all $v_1 \in \mathcal{I}_{\mathcal{V}'}(\sigma_1), ..., v_n \in \mathcal{I}_{\mathcal{V}'}(\sigma_n)$ we have $S_{\mathcal{V}'[v_1/x_1,...,v_n/x_n]}(ce_1) \in \mathcal{I}_{\mathcal{V}'}(s_1)$.

Suppose that the property holds for $H, x : (n : s_1) \vdash e_2 : cl_2 \Rightarrow ce_2$ where $n \notin N(H)$ and $n \notin N(cl_2)$, that is for all $\mathcal{V}'$, for all $v_1 \in \mathcal{I}_{\mathcal{V}'}(\sigma_1), ..., v_n \in \mathcal{I}_{\mathcal{V}'}(\sigma_n), v_x \in \mathcal{I}_{\mathcal{V}'}((n : s_1))$, we have $S_{\mathcal{V}'[v_1/x_1,...,v_n/x_n,v_x/x]}(ce_2) \in \mathcal{I}_{\mathcal{V}'}(cl_2)$. $v_x \in \mathcal{I}_{\mathcal{V}'}((n : s_1))$ means that $v_x \in \mathcal{I}_{\mathcal{V}'}(s_1)$ and $\mathcal{V}'(n) = v_x$. Since n is a fresh name, this means that for all z $\mathcal{V}'(z) = \mathcal{V}(z)$ is $z \neq n$ and $\mathcal{V}'n = v_x$ otherwise, that $S_{\mathcal{V}'[v_1/x_1,...,v_n/x_n,v_x/x]}(ce_2) = S_{\mathcal{V}[v_1/x_1,...,v_n/x_n,v_x/x,v_x/n]}(ce_2)$ and finally, that $\mathcal{I}_{\mathcal{V}'}(cl_2) = \mathcal{I}_{\mathcal{V}}(cl_2)$. Thus, $S_{\mathcal{V}[v_1/x_1,...,v_n/x_n,v_x/x,v_x/n]}(ce_2) \in \mathcal{I}_{\mathcal{V}}(cl_2)$, that is $S_{\mathcal{V}[v_1/x_1,...,v_n/x_n]}(\texttt{let}\ x = ce_1\ \texttt{in let}\ n = x\ \texttt{in}\ ce_2) \in \mathcal{I}_{\mathcal{V}}(cl_2)$ which is the expected result.

Case of Recursions $e' = (\texttt{rec}\ x = e)$. Let $f : y \mapsto S_{\mathcal{V}[y/x]}(ce)$. We have $\epsilon \in \mathcal{I}_{\mathcal{V}}(s)$. For all $v_1 \in \mathcal{I}_{\mathcal{V}}(\sigma_1), ..., v_n \in \mathcal{I}_{\mathcal{V}}(\sigma_n), v \in \mathcal{I}_{\mathcal{V}}(s)$, $S_{\mathcal{V}[v/x]}(ce) \in \mathcal{I}_{\mathcal{V}}(s)$, that is $f(v) \in \mathcal{I}_{\mathcal{V}}(s)$. Thus, for all n, $f^n(\epsilon) \in \mathcal{I}_{\mathcal{V}}(s)$. For all n, $f^n(\epsilon) \leq lim_{n\to\infty}(f^n(\epsilon))$. $\lim_{n\to\infty}(f^n(\epsilon)) \notin \mathcal{I}_{\mathcal{V}}(s)$, means that $clock\,(\lim_{n\to\infty}(f^n(\epsilon))) \not\leq \mathcal{V}(s)$. Thus, there exists a k such that $f^k(\epsilon) \not\leq \mathcal{V}(s)$ which is contradictory. Thus, we get the expected result. □

Energy-Conscious Memory Allocation and Deallocation for Pointer-Intensive Applications

Victor De La Luz[1], Mahmut Kandemir[1], Guangyu Chen[1], and Ibrahim Kolcu[2]

[1] Department of Computer Science and Engineering
The Pennsylvania State University
University Park, PA 16802, USA
{delaluzp,kandemir,gchen}@cse.psu.edu
[2] Computation Department
UMIST
Manchester M60 1QD, UK
ikolcu@co.umist.ac.uk

Abstract. A multi-bank memory architecture is composed of multiple memory banks, each of which can be energy-managed independently. In this paper, we present a set of strategies for reducing energy consumption in a multi-bank memory architecture using energy-conscious dynamic memory allocation/deallocation. Applications that make dynamic memory allocations are used very frequently in mobile computing/networking area. Our strategies focus on such applications and try to cluster dynamically created data with temporal affinity in the physical address space such that the data occupy a small number of memory banks. The remaining banks can be shut off, saving energy. All of our strategies have been implemented and tested using an in-house energy simulator and an application suite that consists of nine pointer-intensive real-life applications. Our results show that all the strategies considered in this paper save energy (e.g., our user-initiated strategy saves 49% leakage energy on the average). The results also indicate that the best savings are obtained when energy-aware memory allocation/deallocation is combined with automatic data migration.

1 Introduction

In many embedded/portable systems, memory subsystem is a major energy consumer. For example, recent studies (e.g., [1,26]) indicate that main memory consumes nearly 90% of overall system energy excluding I/O units. In particular, in mobile/embedded environments where energy is at premium, memory energy consumption can be a real problem. Consequently, software optimizations that reduce energy consumption in memory during the execution of a mobile application can be very useful in practice. One way of reducing energy consumption in memory subsystem is to employ a multi-bank memory architecture. In a multi-bank memory architecture, the memory is divided into banks and each bank can be power-managed independently. Chen et al. [3] discuss that up to 70% savings in overall leakage energy consumption are possible if unused SRAM banks are turned off by the garbage collector in a Java-based environment. While dynamic (switching) energy is consumed only when the bank is accessed, the leakage energy is consumed as long as the bank is powered on (active), even if it is not used by the current computation. Optimizing leakage energy consumption is particularly important as the current trends [2,22,18,17,21,12] indicate that it will form a large portion of overall energy consumption in upcoming process technologies.

In this paper, we present a set of strategies for reducing leakage energy consumption in a multi-bank memory architecture. Our strategies focus on applications (mainly pointer-based codes) that make frequent dynamic memory allocations (using `malloc()` or similar routines) and deallocations (using `free()` or similar routines). Such applications are very frequently used in mobile environments and networking area (e.g., see [7, 24,13] and the references therein). The optimization algorithms proposed in this paper cluster dynamically-created data with temporal affinity in the physical address space such that the data occupy a small number of memory banks. The remaining banks can be shut off, saving leakage energy. We investigate this issue at several levels. First, we study a user-initiated energy-saving scheme where the user indicates (by using a modified form of conventional `malloc()`) which data objects are related (i.e., exhibit temporal affinity). Based on this, our strategy clusters these objects (i.e., the corresponding pages in the physical address space) to minimize the number of banks active (powered on) at any given time. Our experience with this implementation indicates that it requires minimal change to the program source and is very effective in practice. Second, we explore a user-transparent scheme where dynamically-created data are clustered automatically. An important advantage of this scheme is that it requires no change to the program at hand; only the implementations of `malloc()` and `free()` need to be modified. Our compiler analyzes the code and replaces the original memory allocation calls with their energy-aware counterparts. Our energy-conscious memory allocation strategies work in conjunction with a memory-aware deallocation (a modified form of the `free()` utility). Finally, we investigate the support for automatic (user-transparent) data migration for saving energy. All our strategies have been implemented and tested using an in-house energy simulator and an application suite that consists of nine pointer-intensive real-life applications. Our experimental results show that all the strategies considered in this paper save leakage energy. They also indicate that the best savings are obtained when memory allocation/deallocation is used in conjunction with automatic data migration.

2 Architecture and Background

2.1 Architecture

Integrated circuit technology continues to advance at an unrelenting pace. With the advent of 0.18 micron and finer process technologies, it is now possible to pack an entire electronic system onto a small set of chips. In this paper, we focus on an architecture with multiple SRAM banks (as main memory) and investigate necessary software support to reduce memory energy consumption. SRAMs (static RAMs) are popular main memory building blocks in some embedded/mobile environments [1]. In our multi-bank memory architecture, each memory bank can be powered on and off independently using a sleep signal (per bank) issued by the memory controller. In particular, unused memory banks can be turned off to save energy. The sleep signals can be activated/deactivated by setting appropriate bits in a register in the memory controller. Apart from the banked memory architecture, our architecture also has a CPU core, a data cache, an instruction cache, a hardware accelerator (e.g., a co-processor), and a custom logic (ASIC). The last two components are optional and not considered in this work.

It should be made clear that, in our work, memory banking is exclusively used for improving energy consumption. This is an entirely different strategy than memory interleaving where banking is used to increase memory-level parallelism. In memory-interleaving, the main objective is to keep as many banks active at the same time as possible (to maximize parallelism). This is exactly the opposite of our goal, which is to limit the number of accesses to small set of banks (at a given time period) so that we can save energy by placing the unused banks into a power-saving mode.

2.2 malloc() and free()

In pointer-intensive mobile applications, dynamic memory space (heap) is manipulated using memory allocation (`malloc()`) and deallocation (`free()`) calls. Since these calls manipulate memory space, their implementation can be modified to make better use of the banked nature of the memory architecture from the energy viewpoint. Therefore, we first discuss the functionalities of `malloc()` and `free()`.

In a virtual memory based system, operating system (OS) governs the virtual-to-physical address mapping. If a requested physical page is not in memory, a page fault is generated and subsequently processed by the page fault handler [23]. In this subsection, we present a brief overview of `malloc()` and `free()` implementations in current systems. The `malloc()` and `free()` routines provide a memory allocation/deallocation package. The `malloc()` function allocates uninitialized space for a data object whose size is specified by the `size` parameter:

```
void * malloc(size_t size)
```

The BSD Unix `malloc()` function maintains multiple lists of free blocks according to size, allocating space from the appropriate list [16]. The allocated space is suitably aligned (after possible pointer coercion) for storage of any type of object. If the space is of page size or larger, the memory returned will be page-aligned. One important fact about this routine is that when invoked it allocates only virtual memory space. Later, when the allocated region is accessed, a page fault occurs and physical memory is allocated. In other words, the operating system comes into picture only when a reference to the allocated region is made. There is one important exception, though. If the virtual memory space is full, an OS call called `brk()` is invoked to increase the virtual address space (so that the allocation request can be completed). There are many other implementations of `malloc()`. For example, Lea [19] implemented an enhanced form of the standard first-fit algorithm by using an array of free lists segregated by object size. Alternative memory/heap/region allocation algorithms include Haertel's hybrid algorithm [10], Weinstock and Wulf's fast segregated algorithm (called quickfit [28]), and others (e.g., [27,25,11]). Among the important objectives of any memory allocation algorithm are maximizing portability and compatibility, minimizing memory allocation time, maximizing locality (in virtual address space), and improving error messaging [19]. All these implementations mentioned above, however, differ in how they manage virtual address space. In this paper, we are more interested in managing physical address space. Consequently, we focus on the BSD Unix `malloc()` function only. The argument to `free()`

is a pointer to a block previously allocated by `malloc()`. This space is made available for further allocation after the `free()` routine is executed. In this work, we modify the implementations of `malloc()` and `free()` routines as well as the default OS page fault handler to make them energy-conscious (i.e., more suitable for a mobile environment). This can be achieved by taking into account the banked nature of the memory system. We also show how an optimizing compiler can make use of these new memory allocation and deallocation routines.

Benchmark	Allocated Memory	Memory Allocations	Description	Data Structure	Lines
`Atr`	812KB	324	Network address translation	Double-linked list	626
`SP`	1,068KB	620	All-nodes shortest path algorithm	Array of double-linked lists	1,028
`Encr`	792KB	335	Digital signature for security	Array of single-linked lists	1,411
`Hyper`	1,285KB	928	Machine simulation	Quad-tree	583
`Wood`	1,010KB	669	Color-based surface inspection method	Array of double-linked lists	978
`Usonic`	1,044KB	830	Feature-based estimation algorithm	Binary tree	1,005
`Jpegview`	1,308KB	796	JPEG image display	Array of double-linked lists	665
`Pscheduler`	1,181KB	841	Monthly personal scheduler	Array of double-linked lists	821
`Wbrowser`	1,477KB	918	Wed-browser for hand-held devices	Arbitrary graph of linked-lists	1,266

Fig. 1. *Applications used in our experiments. The third column gives the total number of dynamic memory allocations made during execution and the second column shows the total memory space allocated (when the entire program execution is considered). The fifth column indicates the main dynamic data structure maintained by each application and the last column gives the number of lines in the code.*

2.3 Energy Consumption and Leakage Optimization

Energy consumption has two major components [2]: dynamic energy and static energy. While in current CMOS circuits dynamic energy is the dominant part (between 80% and 90%), trends indicate that the contribution of static energy (also called leakage) to the overall energy budget will increase exponentially in upcoming circuit generations [2, 17].

The leakage energy is consumed as long as the circuit is powered on (whether it is accessed or not). This is in contrast to dynamic energy which is spent only when there is a bit transition activity. The leakage energy consumption of SRAM modules (banks) can be optimized using several circuit techniques. One of these techniques is gating the supply voltage [2]. This is achieved through a sleep signal which can be controlled by hardware, software, or a mix of both. When this signal is activated, the supply voltage to the circuit is gated and the leakage energy consumption of the bank is eliminated. In this study, we assume that each memory bank can be in one of the three states: R/W (Read/Write), Active, and Inactive. In the R/W state, the memory bank is being read or written. It consumes full dynamic energy as well as full leakage energy. In the active state, the memory bank is active (powered on) but not being read or written. It consumes some amount of dynamic (pre-charge) energy and full leakage energy. Finally, in the inactive state, the bank does not contain any useful data and is supply-gated. We conservatively assume that when in the inactive state (i.e., when supply-gating is used) a memory bank consumes 5% of its original leakage energy (normally, it should consume even less). While placing unused memory banks in inactive state may save significant leakage energy, it may also cause some performance degradation. Specifically, when a bank in the inactive state is accessed (to service a memory request), it takes some time to transition to the active state [3]. In this paper, we term this time as the resynchronization time (or resynchronization penalty), and it is assumed to be 350 cycles (to be on the conservative side). It should be mentioned that both inactive state leakage energy consumption and resynchronization time are highly implementation dependent and are affected by the sizing of the driving transistors. Finally, we also assume that the per cycle leakage energy consumed during resynchronization is the average of per cycle leakage energy consumed when the system is in the active state and that consumed when it is in the inactive state. To be fair, any evaluation of a leakage optimization scheme based on supply gating should take into account the resynchronization time and energy as well. All absolute energy numbers presented in this paper are for 0.18 micro technology.

3 Benchmarks, Simulation Environment, and Energy Distribution

To test the effectiveness of our strategies in reducing energy consumption, we used a suite of nine real-life applications. A common characteristic of these applications is that all of them are pointer-intensive. Figure 1 lists important characteristics of these applications. To perform our experiments, we used an in-house simulation environment built on top of the Shade tool-set [5]. Shade is an execution-driven ISA and cache memory simulator for Sparc architectures. We simulated a mobile architecture composed of a Sparc-IIep based embedded core processor, data and instruction caches, TLB, memory controller, and a banked memory architecture. The simulator outputs the leakage and dynamic energy consumptions in these components and overall application execution time. In particular, it simulates a data access in detail (by enhancing Shade); it tracks the virtual-to-physical address translation and records the accesses to memory controller, TLB (which has 32 entries), and page table. It also counts the number of page faults. In this work, we assume that each page fault costs 4.1 mJ energy and 14 msec time penalty. These values are similar to those given for an IBM Travelstar 48GH Disk for mobile and laptop systems [14]. Obviously, the secondary storage might be an off-chip DRAM (in an embedded environment), in which case these values are highly conservative. We assume that when a page fault occurs, the processor does not do any useful work until the data arrives. The energy consumed in main memory is divided into static and dynamic components. For dynamic energy in caches and SRAM banks, the model given by Kamble and Ghose [15] is used. In computing the leakage energy, we assumed that the leakage energy per cycle of the entire memory is equal to the dynamic energy consumed per access (note that several prior work also operate with similar assumptions e.g., [17]). According to Chandrakasan et al. [2], the leakage energy is expected to be the dominant component in energy consumption in the future; consequently, this is a reasonable assumption. The energy consumed in the processor core is estimated by counting the number of instructions of each type and multiplying the count by the base energy consumption of the corresponding instruction. The base energy consumption of different instruction types is obtained using a publicly-available, cycle-accurate, architectural-level energy simulator [26].

The default bank configuration used in our simulations has 16 SRAM banks, each of which is 32KB (a total memory capacity of 512KB), with a page size of 4KB. The default configuration also has an 8KB, 2-way set-associative data cache and an 8KB, direct-mapped instruction cache, both with a cache line size of 32 bytes. In this work, we focus only on "data accesses" and assume that program code, OS code, and page table reside in a separate set of banks (i.e., 512KB is used only for the application data, which is a realistic assumption as, for example, Java Virtual Machine for embedded devices (KVM) uses even smaller heap sizes for objects). Figure 2 shows the energy consumption breakdown for our architecture when no leakage optimization is applied. The energy consumption of each application is divided into several parts: the energy consumed in the CPU core, the energy consumed in data cache, the leakage energy consumed in main memory, the dynamic energy consumed in main memory, the energies spent in TLB, page table accesses, and memory controller, and the energy consumed in handling page faults (off-chip energy). We observe that the main memory energy (leakage + dynamic) dominates the overall energy consumption (72.6% on the average). Since the number of page faults is low (due to high locality), the contribution of off-chip accesses is around 5.7%. The contribution of energy consumed in TLB, page table, and memory controller (due to data accesses) is 8.6%, mainly because these units are small in size. We see from these results that memory leakage energy constitutes a significant portion of the energy budget (45.8% on average), and therefore, it is a suitable target for energy optimization.

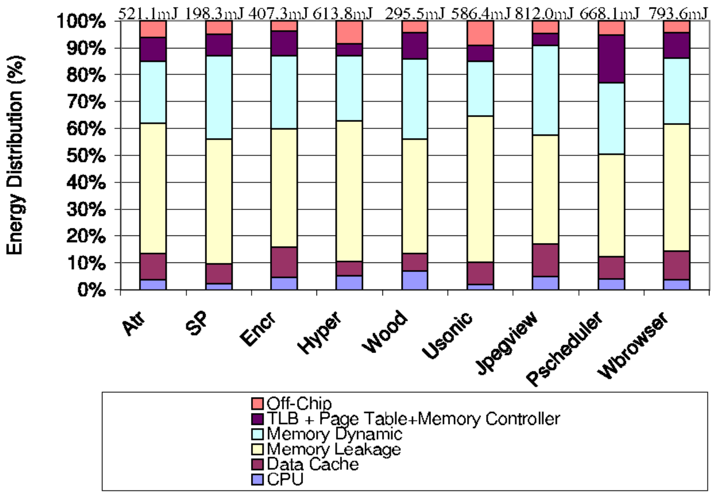


Fig. 2. *Energy breakdown due to data accesses (16 × 32KB banks). Above each bar is the total energy consumption (in millijoules).*

4 Memory Allocation Strategies

4.1 Comparison to Previous Work

There are numerous published work in memory allocation/deallocation. Some of these techniques [10,28,19,16] focus on efficient (fast) `malloc()` implementations, whereas others specifically focus on data locality [8,9]. There is an important difference between our work and these. While all these studies target specifically the virtual address space, our work targets the physical address space. This difference arises because of the fact that in many of the previous studies, the main objective is optimizing cache (or page) locality. This objective can be achieved (to some extent) by focusing on the virtual address space, incurring inefficiencies only at page boundaries. Our objective, however, is very different. We would like to shut off idle (inactive) memory banks for saving energy in a mobile environment. This can only be achieved by working with physical addresses, a requirement which makes our implementation and optimization strategy entirely different from the related work. Chilimbi et al. [4] present two semi-automatic tools, namely, cache-conscious reorganization and cache-conscious allocation, for optimizing cache locality in pointer-intensive mobile applications. The compiler analysis performed in our work is similar to that would be required by the cache-conscious allocation but targets the physical address space. Also, in addition to a user-assisted approach, we also present a fully-automatic, compiler-based strategy to take advantage of energy-aware memory allocator. Finally, we also optimize memory deallocation. We believe that cache locality optimizations and data clustering for energy optimization are complementary; and the best energy results can be obtained by employing both the techniques. To the best of our knowledge, no previous paper studied energy-efficient `malloc()/free()` implementations for mobile computing environments and their use in a given application.

A related group of studies focus on optimizing dynamic energy consumption of a multi-bank memory architecture using OS-based techniques (e.g., [20]) and compiler-based approaches (e.g., [6]). Lebeck et al. [20] present an elegant strategy that changes the OS page allocation policy to minimize dynamic energy consumption. While such an approach is very effective in some cases, it is not an application-sensitive strategy. Instead, as will be illustrated in the following sections, our approach can be used in conjunction with a compiler analysis to reduce leakage energy (The advantage of the strategy discussed in [20] over ours is that it can be used even if the source code of the application to be optimized is not available.) It should also be noted that, in addition to memory allocation, we also energy-optimize memory deallocation. The previous compiler work [6], on the other hand, focuses on program transformations for array-intensive applications (static data allocation) and assumes that there is no virtual memory support (some embedded systems work without virtual memory).

4.2 ea_malloc(), ea_free(), and User-Specified Affinity

Ideally, an energy-conscious memory bank management strategy for a mobile environment should have two main objectives: (i) using as few memory banks as possible and (ii) colocating data with temporal affinity as much as possible. Both memory allocation

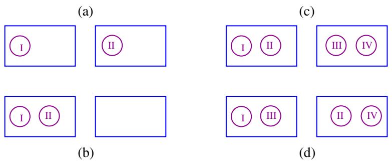
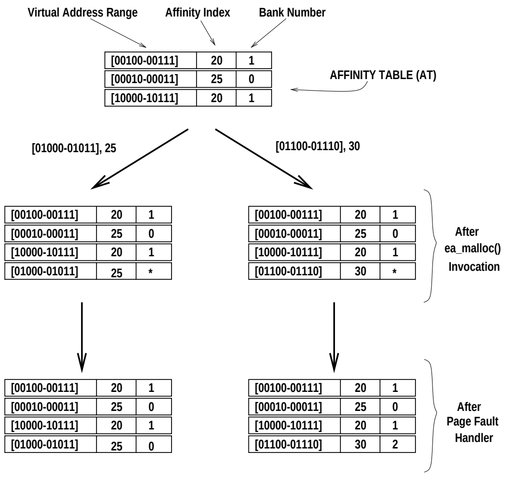


Fig. 3. *Different memory allocations for four objects (I, II, III, and IV) in a two-bank memory system.*



[00100-00111]	20	1
[00010-00011]	25	0
[10000-10111]	20	1

[00100-00111]	20	1
[00010-00011]	25	0
[10000-10111]	20	1
[01000-01011]	25	*

[00100-00111]	20	1
[00010-00011]	25	0
[10000-10111]	20	1
[01100-01110]	30	*

[00100-00111]	20	1
[00010-00011]	25	0
[10000-10111]	20	1
[01000-01011]	25	0

[00100-00111]	20	1
[00010-00011]	25	0
[10000-10111]	20	1
[01100-01110]	30	2

Fig. 4. *Affinity table and two different entry insertion scenarios (assuming no space problem on the banks in question). Virtual address ranges are given as the start and end page numbers (in binary).*

and deallocation can help in achieving these objectives. To show these points, let us consider two scenarios. Figures 3(a) and (b) show two example memory allocations for two objects (I and II) in a two-bank memory system. We can clearly see that the allocation in (b) is much better than that in (a) in terms of energy consumption (as only one bank is activated), illustrating the importance of memory allocation. In another scenario, let us consider the memory allocation of four objects (I, II, III, IV) in a two-bank system (see Figures 3(c) and (d)). If objects I and III have the same lifetime (i.e., they die at around the same time), the allocation in Figure 3(d) is much better than that in (c), as in the former, when the two objects (I and III) die, we can shut off the bank. This last scenario shows that even during deallocation we can save energy. In this section, we present our energy-aware memory allocator (`ea_malloc()`) and deallocator (`ea_free()`), and show how they can be used by an application programmer.

Our new energy-aware memory allocation routine has the following interface:

```
ea_malloc(size_t size, long int affinity_index)
```

Here, `size` denotes the size of the data object to be allocated (in bytes) and `affinity_index` is a positive integer number used to "capture the temporal affinity" between different allocations. In other words, two different dynamic data allocations with the same affinity index are assumed to have temporal affinity; that is, the associated data will be used together in the course of execution. So, these two allocations should be made from the same bank if it is possible to do so. While an optimizing compiler can analyze a given code and replace the original `malloc()` calls with their energy-aware counterparts (as discussed in the next subsection), it is also not very difficult for the programmer to use `ea_malloc()` directly. All she/he simply needs to do is to use the same affinity index for each related memory allocation. Therefore, it requires minimal modification to the source code being optimized.

In order to implement `ea_malloc()`, we need to establish a mapping from affinity indices to memory banks. Establishing the index-to-bank mapping is not very trivial as a typical implementation of `malloc()` does not allocate physical memory at `malloc()` invocation time; it allocates only virtual memory (see Section 2.2). Physical memory is later allocated (following a page fault) when the data in the allocated region is referenced. Consequently, in order to energy-manage physical memory, we need to "propagate" some information from the `malloc()` invocation time to the data reference time. Our implementation achieves this by using the following approach. In the `ea_malloc()` invocation time, we record the virtual address space region requested by `ea_malloc()` along with the corresponding affinity index in a data structure called "affinity table (AT)." An affinity table entry contains the triplet:

```
(virtual address range, affinity index, bank number)
```

The top portion of Figure 4 shows a typical AT that contains three entries. Since in `ea_malloc()` invocation time the bank number is not known, only the first two fields are entered in the AT; the third field is marked using '*'. The middle part of Figure 4 illustrates two different scenarios when different entries are entered into AT shown in the top portion. For example, when the virtual address range is [01000-01011] and the affinity index is 25, we enter ([01000-01011],25,*) in the AT. Later, when an access to that region is made, we incur a page fault. In our modified page fault routine, we take into account the affinity index and perform an energy-aware page allocation.

In order to explain the activities in our modified page fault handler, let us first assume that we do not experience a space problem for the bank which we want to allocate the data from. If we already have a bank number (in AT) associated with the current affinity index, the new allocation also gets the same bank number, and the allocation is made from that bank. The left-bottom part of Figure 4 depicts this case. Since when we incur a page fault for the range [01000-01011], we already have the affinity index 25 associated with bank number 0 in AT (i.e., the entry ([00010-00011],25,0)), we simply rewrite ([01000-01011],25,*) as ([01000-01011],25,0) and perform the allocation in question from bank 0. If, on the other hand, we do not have a bank number associated with the current affinity index, we select an unused bank number and associate that bank number with the current affinity index. This is illustrated in the right-bottom portion of Figure 4. When we incur a page fault with the virtual address region [01100-01110] and the affinity index 30, we first check whether the affinity index 30 appears in the AT. Since it does not, we pick up a new bank number, which is 2 in our example, and rewrite the entry ([01100-01110],30,*) as ([01100-01110],30,2); that is, the allocation is made from bank 2. Note that, in this case, we select a bank number "not" associated in the AT with any entry. This is because selecting an already used bank number would colocate data that do not exhibit temporal affinity. In some cases, it might be beneficial to do such colocations, but this should be indicated by the programmer/compiler using the "same" affinity index. To summarize, in the page fault handler, we record the bank number from which the current allocation has been made along with the virtual address space region and affinity index.

In order to illustrate the case where we experience space problem, we consider once more the AT shown in the top portion of Figure 4. Assume that the next memory allocation request has the entry ([11000-11011],20,*). When the corresponding page fault occurs, the page fault handler would want to rewrite this entry as ([11000-11011],20,1); that is, it would want to allocate the data from the bank 1. Assume now that this bank is full and cannot accept more data. In this case, the handler selects a new bank, say bank 2, rewrites the entry as ([11000-11011],20,2), and allocates data from this new bank.

```
void * ea_malloc(size_t size, long int ai)
{
  v_type * va;
  r_type var;
  va = malloc(size);
  var = compute_va_region(va,size);
  insert(var,ai,*) in AT;
  return(va);
}
         (a)

void ea_fault_handler(v_type va, i_type pi, ...)
{
 /* save the faulting instruction state */
 compute the address range var;
 if AT already contains a (var,ai,bm) sufficient space
   perform physical page allocation from bank bm;
 else {
       find the entry (var,ai,*) in AT;
         if there exists a (vbr,ai,bj) in AT and
          bj has sufficient space then
           rewrite (var,ai,*) as (var,ai,bj);
           perform physical page allocation from bank bj;
         else
           if there is a bank bk (not in AT) then
             rewrite (var,ai,*) as (var,ai,bk);
             perform physical page allocation from bank bk;
           else
            if there is a bank bl in AT with sufficient
             space then
              rewrite (var,ai,*) as (var,ai,bl);
              perform physical page allocation from bank bl;
            else /* the physical memory is full */
             run the page replacement algorithm to
                replace a page from the bank associated
                with the affinity index ai (if any)
             otherwise replace a page using the page
              replacement algorithm for the entire memory;
             let br be the selected bank number;
             rewrite (var,ai,*) as (var,ai,br);
             perform physical page allocation from bank br;
      }
  /* restore state information to restart instruction */
}
        (b)
```

Fig. 5. *(a)* `ea_malloc()` *and (b) page fault handler.* `var` *and* `vbr` *are virtual address regions, and* `pi` *is the id of the current process.* `v_type`, `r_type`, *and* `i_type` *are the types for virtual addresses, virtual address regions, and process id, respectively.*

```
synch1 = ...;
synch2 = ...;
...
while (synch1+synch2<...){
  code1 = (struct code *) malloc(sizeof(struct code));
  code2 = (struct code *) malloc(sizeof(struct code));
  code3 = (struct code *) malloc(sizeof(struct code));
  ...
  synch1 = Synch_Comp(code1->pin,code3->pin,...);
  ...
  synch2 = Synch_Comp(code2->pin,code2->pin,...);
  ...
}
             (a)

synch1 = ...;
synch2 = ...;
...
while (synch1+synch2<...){
  code1 = (struct code *) ea_malloc(sizeof(struct code),20);
  code2 = (struct code *) ea_malloc(sizeof(struct code),20);
  code3 = (struct code *) ea_malloc(sizeof(struct code),20);
  ...
  synch1 = Synch_Comp(code1->pin,code3->pin,...);
  ...
  synch2 = Synch_Comp(code2->pin,code2->pin,...);
  ...
}
            (b)

synch1 = ...;
synch2 = ...;
...
while (synch1+synch2<...){
  code1 = (struct code *) ea_malloc(sizeof(struct code),20);
  code2 = (struct code *) ea_malloc(sizeof(struct code),30);
  code3 = (struct code *) ea_malloc(sizeof(struct code),20);
  ...
  synch1 = Synch_Comp(code1->pin,code3->pin,...);
  ...
  synch2 = Synch_Comp(code2->pin,code2->pin,...);
  ...
}
            (c)
```

Fig. 7. *(a) Original code fragment from* `Encr`. *(b) Loop-based affinity assignment. (c) Expression-based affinity assignment.*

```
void  ea_free(void * va)
{
  find the corresponding entry (var,ai,bj);
  free(va);
  delete (var,ai,bj) from the AT;
  none=true;
  current_entry = the first AT entry;
  while(none && current_entry is valid){
    if (current_entry.bank_id == bj) none=false;
    current_entry++;
  }
  if(none) turn off bank bj;
}
```

Fig. 6. *Energy-conscious memory deallocation.*

```
void  ea_free_migrate(void * va)
{
  find the corresponding entry (var,ai,bj);
  free(va);
  delete (var,ai,bj);
  none=true;
  current_entry = the first AT entry;
  while(none && current_entry is valid){
    if (current_entry.bank_id == bj) none=false;
    current_entry++;
  }
  if(none) turn off bank bj;
  else
    if (bj's total data size < MT)
       if possible select a set of banks B,
       migrate data from bj to B,
       modify the associated TLB entry, and
       turn off bank bj;
}
```

Fig. 8. *Energy-conscious deallocation with migration support.*

Figures 5(a) and (b) give sketches for `ea_malloc()` and the modified page fault handler, respectively. The `ea_malloc()` routine allocates virtual memory, inserts an entry to the AT, and returns. The `ea_fault_handler()` routine, on the other hand, first checks whether there is already a bank number associated with the affinity index of the data. We refer to this bank as the "first-try bank." If so and if that bank has available space to accommodate the data, data is allocated from that bank. Otherwise, a new bank number (which does not appear in any entry in the AT) is selected and the data is allocated from that bank. If there is no such a bank, a bank that appears in the AT with sufficient space is selected (if there exists one). If all these fail, that means the physical memory is full. In this case, we return to the first-try bank and, using the page replacement algorithm, replace a page from that bank. If there is no first-try bank for the current data allocation (i.e., its affinity index is not associated with any bank number), we run the original (default) page replacement algorithm for the entire memory. It should be noted that our `ea_malloc()` implementation tries not to colocate data with different affinity indices as much as possible. Therefore, it is an "affinity-based" clustering strategy. A pure data colocation algorithm (one that does not consider temporal affinity between data objects), on other hand, would first try to fill the banks that are listed in AT as much as possible (before going to a new bank). It our scheme, it is entirely user's responsibility to associate affinity indices with memory allocations. The system just obeys what the user indicates and tries to colocate as many related data allocations as possible in a small number of banks. The allocations with different affinity indices, on the other hand, go to different banks (if possible). Therefore, if the user wants to colocate unrelated allocations, she should enforce it using the same affinity index for these allocations.

It should be emphasized that memory allocation can save energy only until the whole memory space is filled up. After that point, to continue saving energy, we need to be able to "shut off" the banks that are not occupied by any live data. Consequently, we implemented an energy-aware version of the `free()` routine. This routine, called `ea_free (void *)`, has the same interface as the original `free()` routine; its implementation, however, is different. Specifically, when invoked, it first performs the deallocation required, removes the entry from the AT, and then "checks whether there is any live data in the current bank" after this deallocation. This is done by checking the last fields of the entries in the AT. If the deallocated data was the last data in the bank, the bank is turned off. A sketch of our `ea_free(void *)` implementation appears in Figure 6. It is easy to see that it requires a minimal modification to the original `free()` routine.

We evaluated the effectiveness of our user-specified strategy using the nine applications in our experimental suite. By analyzing the memory allocations and their sizes in each code carefully, we converted the original `malloc()` calls to `ea_malloc()` calls and the original `free()` calls to `ea_free()` calls by hand. That is, we identified which memory allocations are related and assigned the affinity indices accordingly. It should be mentioned that in some cases we assigned the same affinity index even to unrelated data allocations if we felt that there is sufficient space on the bank to hold all data. Note, however, that being able to make such colocation decisions requires some knowledge about program access pattern, sizes of dynamically allocated data, number of banks, and bank capacities.

Our results are presented in Figures 9 and 10 as percentage leakage energy savings and percentage overall energy savings, respectively, over the default memory management strategy (which treats the entire memory space as a big monolithic bank). In both our strategy and the default strategy, however, if a bank is not used at all, it is kept in

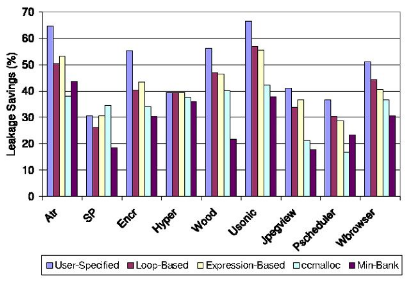


Fig. 9. *Leakage energy savings using different optimization strategies.*

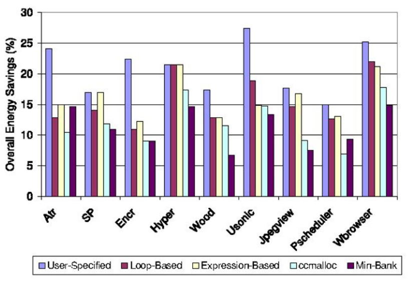


Fig. 10. *Overall energy savings using different optimization strategies.*

the inactive state. Therefore, *the energy benefits reported come solely from our energy-aware allocation/deallocation.* The default strategy can allocate a page from anywhere in memory (depending on the current state of its free list queue), even from an otherwise inactive bank. It should also be noted that our optimization strategy might affect the dynamic energy consumption due to additional code executed in the modified memory allocator and page fault handler. These variations are also included in the results shown in Figures 9 and 10. That is, *the reported energy results also include all the energy overheads involved in our strategy.* In Figure 9, the first bar (of each application) gives the percentage reduction in memory leakage energy due to this "user-initiated strategy." We see that on the average 49.0% leakage energy is saved. The first bar in Figure 10, on the other hand, shows the percentage savings in overall energy (which includes energies expended in the cache, TLB, memory controller and processor core as well as the dynamic energy spent in main memory and in handling the page faults). It can be observed from Figure 10 that the overall energy savings range from 14.9% to 27.4%, averaging in 20.8%, clearly showing the effectiveness of our energy optimization strategy. We also note that all applications in our experimental suite benefit from our energy-aware memory allocation. In `SP` and `Pscheduler`, the original memory allocation is relatively effective. Consequently, the overall energy improvement is not too high. In `Usonic`, on the other hand, frequent memory allocations are done within the innermost search-and-insert loop. In this application, by using affinity indices carefully, we were able to improve the overall energy consumption by 27.4%.

4.3 User-Transparent Clustering

It is also possible to convert the original `malloc()` (`free()`) calls in a given code to `ea_malloc()` (`ea_free()`) calls automatically. The main task of such a strategy is to capture the temporal affinity relations from the code. To achieve this, in this paper, we focus on two compiler-based strategies. The first strategy, called "loop-based," assumes temporal affinity between memory allocations occurring in the same loop and, conversely, assumes that no temporal affinity exists between allocations made in different (independent) loops. The memory allocations that occur outside of any loop are assumed to be enclosed within an imaginary loop that iterates only once. The rationale behind this approach is that the loop iteration counts are in general large[1] and that the data allocated within the same loop are expected to be used (together) during the entire loop execution. The second strategy, referred to as "expression-based," considers two memory allocations related if and only if the associated data accesses occur in the same statement. This strategy might perform better than the loop-based strategy if the loop bodies are very large and not all dynamically-allocated data are used uniformly across the loop body. In order to illustrate the difference between these two strategies, we consider a code fragment from `Encr`. The original code fragment, the loop-based version, and the expression-based version are shown in Figures 7(a), (b), and (c), respectively. The loop-based version assumes that all memory allocations in this fragment have temporal affinity, whereas the expression-based version associates only the first and third allocations.

The second bar in Figure 9 (resp. Figure 10) for each application show the percentage memory leakage saving (resp. overall energy saving) for the loop-based strategy. The third bar, on the other hand, gives the corresponding savings for the expression-based method. From these results, we observe the following. The average leakage savings for the loop-based and expression-based strategies are 39.4% and 41.6%, respectively. The corresponding overall energy savings are 15.3% and 16.1%. In general, the expression-based strategy generates better results than the loop-based strategy. The reason for this is that not all memory allocations done in a given loop are correlated (see for example Figure 7). Consequently, the loop-based strategy sometimes tries to colocate unrelated data as it happens in `Atr`, `SP`, and `Encr`. Note that the potential negative impact of fine-grain behavior of the expression-based allocation is eliminated (in most cases) by the high-level grouping discussed in the previous paragraph. In two applications (`Hyper` and `Wood`), these two strategies result in the same code. In `Usonic`, on the other hand, the loop-based strategy performs better as the allocations done in innermost loops do not occupy large space (two very large memory allocations are done outside inner loops); therefore, clustering the inner-loop allocations in the same set of banks generates a better result. In fact, it is possible to achieve even better results by capturing the affinity by one of the large allocations occurring outside innermost loop and the allocations done in the innermost loop (this is the additional benefit brought by the user-specified method over the loop-based strategy in this application). The expression-based strategy generates the same result as the user-specified method in two applications (`SP` and `Hyper`).

Our energy-conscious memory allocation approach improves energy due to two reasons: "energy-aware bank assignment" and "colocation of data that exhibit temporal affinity." We now show how energy-aware bank assignment alone and colocation alone would perform. That is, we study the individual contributions of colocation and energy-aware bank assignment. The colocation technique that we use clusters the data with temporal affinity in the virtual address space, and is similar to the Chilimbi's `ccmalloc()` strategy [4]. The energy-aware bank assignment strategy is, on the other hand, a sequential first-touch policy (discussed in [20]) that tries to minimize the number banks used. It achieves this by filling an entire bank before allocating any object from another bank. However, all the versions use `ea_free()`. The last two bars (for each application) in Figures 9 and 10 give the percentage leakage and overall energy savings due to these

[1] Our experience shows that this is the case even with the pointer-intensive codes.

two strategies. We can see from these results that, except for the `Atr` and `Pscheduler` benchmarks, the energy-aware bank assignment (denoted `Min-Bank` in the figures) performs worse than the rest. The reason for this is that trying to co-locate the objects without affinity (while it can improve short-term energy behavior due to initialization phases in codes) does not lead to affinity-based clustering in general. While we may have good bank locality until the memory is filled (i.e., we fill the banks slowly), this strategy does not necessarily place the relevant objects in the same bank. Consequently, one has less opportunity to turn off the banks. Also, such a bank assignment degrades the performance more than other energy-sensitive assignments. The average leakage and overall energy improvements due to `Min-Bank` are 28.8% and 11.2%, respectively. The `ccmalloc()` version, on the other hand, performs better than `Min-Bank` in most of the cases, resulting in an average leakage (resp. overall) energy saving of 33.4% (resp. 12.1%). However, its energy performance is still lower than our strategies in general. This is mainly because the `ccmalloc()` focuses only on the virtual address space, and there is no guarantee that the objects colocated in the virtual address space will also be colocated in the physical address space. Based on these results, we can conclude that for the best energy behavior both colocation and careful bank assignment are crucial.

4.4 Automatic Data Migration for Energy

Our memory allocation strategy tries to use as fewer banks as possible while ensuring colocation of data with temporal affinity. However, in the course of execution, as dynamic data are allocated and deallocated, significant bank fragmentation might occur. A bank fragmentation corresponds to a case where the live data occupies larger number of banks than necessary. This occurs when the data objects are scattered in the physical address space and there are large gaps between different objects residing in the same bank. In some cases, a bank needs to be powered on for a long time just to maintain a small amount of data. In such cases, migrating such data to other (relatively full) active banks might enable us to shut off the bank in question and save energy.

Our automatic migration strategy achieves this by using a "migration threshold (MT)." An MT is the minimum amount of total data in a bank that prevents migration of data from that bank to others. If the total size of the data residing in a given bank reduces below MT, all the data in that bank are migrated to other active banks (if there are any) using memory-to-memory copy operation. To implement this strategy, we need to keep track of the amount of data in each bank, which is done by modifying the page fault handler. Also, after migrating (copying) the data, we need to update the associated TLB entries. The sketch of our energy-conscious memory deallocator with migration support (`ea_free_migrate()`) is given in Figure 8. Since migration is a costly operation from both energy and execution cycles perspectives (as it involves bank-to-bank data copies), `ea_free_migrate()` attempts migration only if the bank in question cannot be turned off after the last `ea_free()` operation. It should be mentioned that due to aliasing and parameter passing, in C programs, dynamically allocated data cannot be relocated in general in the virtual address space. However, it should also be noted that, in our case, we are relocating data in the physical address space (by moving the data physically and updating the TLB entry), which should not cause any problem.

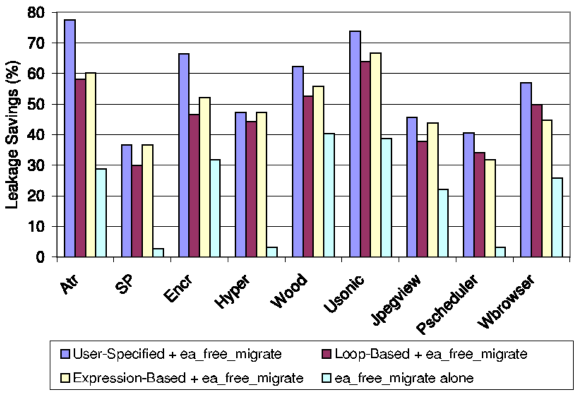


Fig. 5. *Percentage leakage savings when automatic data migration is employed (MT=4KB).*

Selecting a suitable MT that works across multiple applications is an important issue. Working with small MT values postpones migration activity and when MT is reached it might be very late to take any advantage of. Similarly, in very large MT values, it might be difficult to find an active bank (or a set of banks) which has sufficient empty space. We believe that it is difficult to determine the optimum MT value statically. In this work, we experimented with different MT values and found that 4KB is a reasonable MT value. We show in Figure 5 the migration results (percentage energy savings) when `ea_free_migrate()` is used alone and together with `ea_malloc()/ea_free()` with an MT value of 4KB. We see that in two applications, `SP` and `Hyper`, `ea_free_migrate()` could not find an opportunity for migration. The reason for this is that in these two applications when a bank reaches the migration threshold, the other active banks are almost full and cannot accept data. In `Atr`, data migration increased the energy consumption (with respect to `ea_free()`) as high energy cost of migration could not be compensated for by the additional bank turn off. In the rest of our applications, automatic migration increased energy savings. Overall, when it is used alone, the average leakage energy saving is 21.8%. When used in conjunction with user-specified, loop-based, and expression-based strategies, the average leakage energy savings are, respectively, 56.3%, 46.3%, and 48.7% (the overall energy savings are not presented in detail due to lack of space).

5 Concluding Remarks

A contribution of this work is the substantial reduction in leakage energy consumption that our optimizations enable, with tolerable impact on performance. As mobile systems are becoming more and more energy-sensitive and circuit-based energy management is approaching to its limits, we believe that software-based energy optimization strategies are very important. Our energy-aware memory management routines can be used by application programmer as well as an optimizing compiler.

References

1. F. Catthoor, S. Wuytack, E. D. Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle. *Custom Memory Management Methodology – Exploration of Memory Organization for Embedded Multimedia System Design*. Kluwer Academic Publishers, June 1998.
2. A. Chandrakasan, W. J. Bowhill, and F. Fox. *Design of High-Performance Microprocessor Circuits*. IEEE Press, 2001.
3. G. Chen, R. Shetty, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and M. Wolczko. Tuning garbage collection in an embedded Java environment. In Proc. *the 8th International Symposium on High-Performance Computer Architecture,* Cambridge, MA, February 2–6, 2002.
4. T. M. Chilimbi, M. D. Hill, and J. R. Larus. Cache-conscious structure layout. In Proc. *the ACM Conference on Programming Languages Design and Implementation,* May 1999.
5. B. Cmelik and D. Keppel. Shade: A Fast Instruction-Set Simulator for Execution Profiling. In Proc. *the ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems*, pp. 128–137, May 1994.
6. V. Delaluz, M. Kandemir, N. Vijaykrishnan, A. Sivasubramaniam, and M. J. Irwin. DRAM energy management using software and hardware directed power mode control. In Proc. *the 7th International Conference on High Performance Computer Architecture,* Monterrey, Mexico, January 2001.
7. P. Ellervee, M. Miranda, F. Catthoor, and A. Hemani. System-level data format exploration for dynamically allocated data structures. In Proc. *the 37th ACM Design Automation Conference,* Los Angeles, CA, June 2000, pp. 556–559.
8. D. Grunwald and B. Zorn. CUSTOMALLOC: efficient synthesized memory allocators. *Technical Report CU-CS-602-92,* Department of Computer Science, University of Colorado, Boulder, CO, July 1992.
9. D. Grunwald, B. Zorn, and R. Henderson. Improving the cache locality of memory allocation. In Proc. *the ACM Conference on Programming Languages and Implementation*, pp. 177–186, June 1993.
10. M. Haertel. GNU malloc. ftp://ftp.cs.colorado.edu/pub/misc/malloc-implementations/
11. D. R. Hanson. Fast allocation and deallocation of memory based on object lifetimes. *Software-Practice and Experience,* 20(1), pp. 5–12, January 1990.
12. H. Hanson, M. S. Hrishikesh, V. Agarwal, S. W. Keckler, and D. Burger. Static energy reduction techniques for microprocessor caches. In Proc. *the 2001 International Conference on Computer Design,* 2001.
13. A. Hemani, B. Svantesson, P. Ellervee, A. Postula, J. Oberg, A. Jantsch, and H. Tenhunen. High-level synthesis of control and memory intensive communication systems. In Proc. *the 8th Annual IEEE International ASIC Conference and Exhibit,* pp. 185–191, September 1995.
14. *IBM Datasheet for Travelstar 48GH Disk,* 2000.
15. M. Kamble and K. Ghose. Analytical energy dissipation models for low power caches. In Proc. *the International Symposium on Low Power Electronics and Design*, page 143, August 1997.
16. C. Kingsley. Description of a very fast storage allocator. *Documentation of 4.2 BSD Unix malloc implementation,* February 1982.
17. S. Kaxiras, Z. Hu, and M. Martonosi. Cache decay: exploiting generational behavior to reduce cache leakage power. In Proc. *the 28th International Symposium on Computer Architecture,* Sweden, June 2001.
18. N. S. Kim, K. Flautner, D. Blaauw, and T. Mudge. Drowsy instruction caches. In Proc. *the 35th International Symposium on Microachitecture,* Istanbul, Turkey, 2002.
19. D. Lea. G++ malloc. http://g.oswego.edu/dl/html/malloc.html.

20. A. R. Lebeck, X. Fan, H. Zeng, and C. S. Ellis. Power aware page allocation. In Proc. *the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems,* November 2000.
21. L. Li, I. Kadayif, Y-F. Tsai, N. Vijaykrishnan, M. Kandemir, M. J. Irwin, and A. Sivasubramaniam. Leakage energy management in cache hierarchies. In Proc. *the 11th International Conference on Parallel Architectures and Compilation Techniques,* Charlottesville, Virginia, September, 2002.
22. M. D. Powell, S. Yang, B. Falsafi, K. Roy, and T. N. Vijaykumar. Reducing Leakage in a High-Performance Deep-Submicron Instruction Cache. IEEE Transactions on VLSI, Vol. 9, No. 1, February 2001.
23. A. Silberschatz, P. Galvin, and G. Gagne. *Applied Operating System Concepts,* John Wiley & Sons, Inc., 2000.
24. P. Slock, S. Wuytack, F. Catthoor, and G. de Jong. Fast and extensive system-level memory exploration for ATM applications. In Proc. *the 10th ACM/IEEE International Symposium on System Level Synthesis,* pp. 74–81, September 1997.
25. D. Stoutamire. Portable, modular expression of locality. *Ph.D. Thesis,* University of California at Berkeley, CA, 1997.
26. N. Vijaykrishnan, M. Kandemir, M. J. Irwin, H. Y. Kim, and W. Ye. Energy-driven integrated hardware-software optimizations using SimplePower. In Proc. *the International Symposium on Computer Architecture,* June 2000.
27. K.-P. Vo. Vmalloc: a general and efficient memory allocator. *Software Practice & Experience,* vol.26, pp.1–18, 1996.
28. C. B. Weinstock and W. A. Wulf. Quickfit: an efficient algorithm for heap storage allocation. *ACM Notices,* 23(10):141–144, October 1988.

Space Reductions for Model Checking Quasi-Cyclic Systems*

Matthew B. Dwyer, Robby, Xianghua Deng, and John Hatcliff

Department of Computing and Information Sciences, Kansas State University
234 Nichols Hall, Manhattan KS, 66506, USA.
{dwyer,robby,deng,hatcliff}@cis.ksu.edu

Abstract. Despite significant research on state-space reductions, the poor scalability of model checking for reasoning about behavioral models of large, complex systems remains the chief obstacle to its broad acceptance. One strategy for making further progress is to exploit characteristics of classes of systems to develop domain-specific reductions. In this paper, we identify a structural property of system state-spaces, which we call *quasi-cyclic* structure, that can be leveraged to significantly reduce the space requirements of model checking. We give a formal characterization of quasi-cyclic state-spaces and describe a state-space exploration algorithm for exploiting that structure. In addition, we describe a class of real-time embedded systems that are quasi-cyclic, we outline how we customized an existing model checking framework to implement space-efficient search of quasi-cyclic systems, and we present experimental data that demonstrate multiple orders of magnitude reductions in space consumption.

1 Introduction

Model checking is maturing into an effective technology for automated analysis of system designs. It has been used widely in validating hardware designs and is beginning to be applied in reasoning about behavioral models of software (e.g., [3,9,10]). Model checking is an attractive validation technology because it is automatic and it is significantly more thorough in considering the possible behaviors of a system than traditional testing methods. Thus, model checking holds the promise of providing high-levels of assurance of system correctness properties without the need for significant human effort; existing methods for high-assurance development, in contrast, are very labor intensive (e.g., [6]).

Despite a decade of intensive research on general techniques for reducing the complexity of model checking (e.g., see [4]), scalability remains the chief obstacle to its wide-spread adoption. It is common-place for model checkers to exhaust available memory when analyzing even highly-abstract models of real systems.

* This work was supported in part by the U.S. Army Research Office (DAAD190110564), by DARPA/IXO's PCES program (AFRL Contract F33615-00-C-3044), by NSF award CCR-0306607, and by Rockwell-Collins.

While general reduction methods are effective (e.g., partial order reductions can lead to order of magnitude space reductions), we believe that information about the *system domain* can be exploited to enable further significant space reductions. In recent work on model checking system design models, we have explored several strategies for exploiting information about the design and run-time environment of distributed real-time embedded (DRE) systems built on CORBA-compliant middleware platforms [14]. These techniques have been implemented in a model checking framework, called Bogor, that is designed for ease of extension and customization [13]. We have customized Bogor to exploit information about the scheduling policy, the abstract behavior of middleware services, and the time-triggered system environment. This has resulted in orders of magnitude space reductions [5,9], however, even more reduction is needed before these techniques can be applied effectively to design models of production systems.

In this paper, we present an algorithm for decomposing a global state-space search into a number of independent *predicate-bounded* sub-searches whose scopes are defined by a predicate over program states. We have identified a class of systems, which we term *quasi-cyclic*, for which a projection of the system's state-space is cyclic. Defining a predicate that identifies the recurring states in those cycles leads to a *quasi-cyclic* predicate bounded search. Intuitively, a quasi-cyclic system is one whose traces are characterized by repeated visits to states that share the same values for a subset $\{t_1, \ldots, t_n\}$ of state variables. The state-space of a quasi-cyclic system consists of *regions* that are bounded by states containing *repeated* $\{t_1, \ldots, t_n\}$*-sub-states* (i.e., states where projecting onto the values for $\{t_1, \ldots, t_n\}$ yields a sub-state that was encountered previously in the search). One can decompose the state-space search of a quasi-cyclic system into independent searches of these bounded regions. Conceptually, one can use any form of search for the different parts of a decomposed search, but for clarity in our presentation we consider a version that performs depth-first search (DFS) within regions and breadth-first search (BFS) across region boundaries. Each DFS is limited to the states within a single bounded region; such a search results in the generation of system states that define new region boundaries.

Those boundary states generate additional regions that are searched in first-in-first-out order, thereby forming a BFS of the reachable regions. In addition to following this search strategy, we apply a key heuristic: we only store the boundary states plus the states generated by the DFS on the *current* region – after a DFS search of a region is completed the stored states for that region are purged.

This offers the potential for significant reduction in the memory requirements of model checking since search storage is localized to region state-spaces. The size of the reduction depends on several factors including the number of region boundary states and the number of states calculated for each region. When the former is large and the latter small the reductions can be many orders of magnitude, as discussed in Section 5. As is shown in Section 2, memory consumption when model checking cyclic systems is bounded by the sum of the number of region boundary states and the number of states in the largest individual region search. On many realistic scalable examples this results in sub-linear space growth as system size increases. This space reduction enables checking properties of systems that could not otherwise be checked without exhausting available memory, but it may incur a significant overhead in the time required to check a system. This stems from the fact that individual region state-spaces may overlap and force the combined BFS-DFS algorithm to revisit states multiple times. This redundancy may actually increase state-space search time, but we demonstrate that our decomposition results in sub-searches that are completely independent and thus amenable to parallel execution; we present data on the effectiveness of parallelism in combating increased execution time.

Decomposed searches of quasi-cyclic systems preserve reachability properties. They also allow region-bounded properties to be checked in individual region DFS. Global state-space properties can also be preserved, but we do not consider that issue in this paper.

A broad range of realistic systems can be characterized as quasi-cyclic. Any system with a *control loop* (e.g., graphical user interfaces, web-servers) can be considered quasi-cyclic. Periodic real-time systems repeatedly wait for time-triggered interrupts to initiate processing in a frame and are thus quasi-cyclic. In this paper, we consider behavioral design models of DRE systems from Boeing's Bold Stroke avionics framework. These systems can be classified as quasi-cyclic, and our decomposed search algorithm enables multiple orders-of-magnitude reduction in memory consumption over existing highly-optimized search algorithms. This is part of our larger effort on Cadena – an integrated development environment that we are building to provide support for modelling, analysis, and implementation of DRE systems built using the CORBA Component Model (CCM).

The specific contributions of this paper are the: (*i*) definition of a method for decomposing state-space search into a set of independent predicate-bounded sub-searches that are amenable to effective parallelization, (*ii*) identification of quasi-cyclic-ness as a state-space characteristic that enables predicate-bounded search decomposition and yields significant memory reduction, (*iii*) characterization of a class of periodic real-time systems as quasi-cyclic, and (*iv*) description of an implementation and evaluation of the effectiveness of quasi-cyclic state-space search over a range of example systems.

This paper continues in Section 2 with the definition of quasi-cyclic state-spaces and presentation of the decomposed search algorithm. Section 3 describes Cadena design models for DRE systems built in the Bold Stroke framework and characterizes them as quasi-cyclic. Sections 4 and 5 describe the implementation and evaluation of decomposed search, respectively, over a collection of Cadena design models. Section 6 discusses related work, and Section 7 concludes.

2 Quasi-Cyclic State Space Search

A *transition system* is a tuple $(S, s_0, E, \rightarrow)$, where S is a set of states, $s_0 \in S$ is the initial state, $E \subseteq S$ are designated end states, and $\rightarrow \subseteq S \times S$ is the state transition relation; $\rightarrow^*$ denotes the reflexive and transitive closure of the transition relation. The *state-space* of a system is the set of states, $S_{reach} \subseteq S$, that are reachable from s_0 (i.e., $S_{reach} = \{s \mid s_0 \rightarrow^* s\}$).

States S are defined by the values of a set of *state variables* V. Variables are partitioned into two sub-groups: $t_i \in T$ for variables whose values are *transient* in that they return to the same fixed values at region boundaries, and $g_i \in G$ for variables whose values may vary *globally* across entire system executions. A state $s \in S$ binds values c_v to variables $v \in V$ written as $s = [c_{t_1}, \ldots, c_{t_n}, c_{g_1}, \ldots, c_{g_m}]$. Let $\pi_U(s)$ be the projection onto variables $U \subseteq V$ from state s, written $[c_{u_1}, \ldots, c_{u_n}]$. Thus, $\pi_T(s)$ is the projection onto transient variables and $\pi_G(s)$ the projection onto global variables. An equality predicate $p_U(s) \stackrel{\text{def}}{=} (\pi_U(s) = [c_{u_1}, \ldots, c_{u_n}])$ defines values for a subset of the state variables and is true in and only in states that have those state variable values; we sometimes write predicates as $p_{[c_{u_1}, \ldots, c_{u_n}]}$. We refer to the set of states satisfying a predicate p as *p-states*.

For convenience, we can evaluate a predicate p_U to access the values tested by a predicate (i.e., $[c_{u_1}, \ldots, c_{u_n}]$). For disjoint sets of variables, we can combine predicates/projections to generate new predicates/projections. In the subsequent presentation, we use the combination $p_T.\pi_G(s)$ to generate a new state $s' = [c^p_{t_1}, \ldots, c^p_{t_n}, c^s_{g_1}, \ldots, c^s_{g_m}]$, where the c^p values are those tested by the predicate and the c^s values are projected from s.

We say that a p-state *leads-to* a p'-state when all sequences of transitions starting at the p-state pass through some p'-state. A state-space is *quasi-cyclic* if there exists a predicate p_T such that: s_0 leads-to a p_T-state, every p_T-state leads to a p_T-state or an end state. Intuitively, a quasi-cyclic state-space is one whose execution traces are characterized by regions that are bounded by *transient* variables reaching fixed values, defined by p_T, and where *global* variables are free to retain their values across region boundaries.

Consider a simple guarded assignment system:

```
l1: y = 0; goto l2;
l2: x = 0; goto l3;
l3: true -> x = 2; goto l4;
    true -> x = 3; goto l4;
l4: y = y + x; goto l5;
l5: y>5 -> skip; goto end;
    y<=5 -> skip; goto l2;
end:
```

The state of this system is described by three variables: a location counter (before executing the location), ranging over values li, and integer variables x and y. The left side of Figure 1 illustrates the full state-space of the example, where states are of the form $[pc, x, y]$ and $T = \{pc, x\}$. We note that this state-space is quasi-cyclic since $p_T = [l3, 0]$ satisfies the condition defined above. Given this predicate one can decompose the state-space into regions bounded by the 11 p_T-states (and end states) as illustrated on the right side of Figure 1. We call

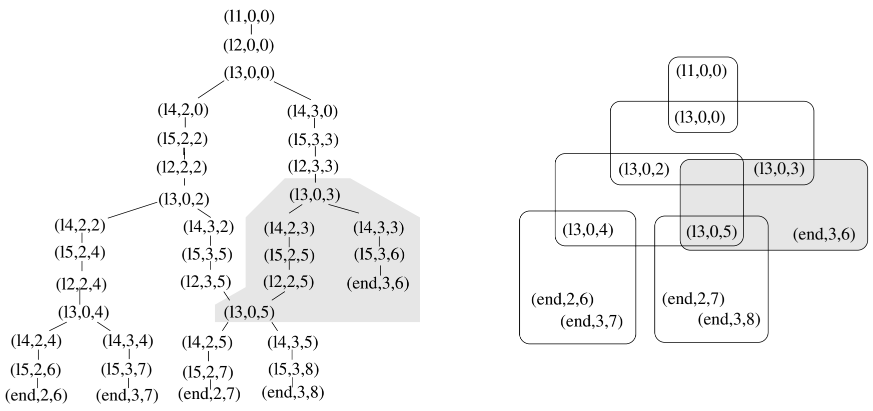


Fig. 1. Quasi-Cyclic State Space Example

the global projections of these boundary states the *projected global state-space.* Note that the shaded areas in the left and right figures correspond to the same region.

The key insight of our method is that one can search these regions independently and still preserve the ability to check a broad range of properties. This can provide a space savings when the size of the overall state-space is larger than the size of the state-space of the largest region; for this to happen, there must be some transient data. In Section 5 we discuss the potential increase in run-time that arises from duplicating explorations from common states in different regions. There is only one such common state in the example of Figure 1, $(l3, 0, 5)$, and since it is a region boundary state it will be explored a single time.

Algorithm 1(a) depicts the classic explicit depth-first state-space search algorithm [4]. We contrast this with our *quasi-cyclic state-space search*, which is shown in Algorithm 1(b). The chief difference between the algorithms is that the quasi-cyclic search performs a series of DFSs (lines b.3 and b.8), one for each global state projection that is encountered (line b.8), rather than a single DFS from the initial state (line a.2). The quasi-cyclic search employs a *predicate-bounded* DFS where successor-less end states and states satisfying p_T define the maximum depth of the search. Each DFS instance in the quasi-cyclic search starts with a set containing only the initial state as the $seen_t$ set (line b.7); thus, a quasi-cyclic search requires only as much memory as is needed for the largest DFS search in addition to the number of boundary states. Each DFS collects the set of p_T-states that bounded the search and adds them to the queue (lines b.19-b.22). The global projection onto those states that have not been seen before ($seen_g$) and that are not queued for search are enqueued (line b.22) for a BFS of quasi-cycles.

The action of this algorithm on the above example would be to perform $p_{[l3,0]}$-bounded DFS instances for the regions shown on the right side in Figure 1 in the following breadth-first order: $(l1, 0, 0)$, $(l3, 0, 0)$, $(l3, 0, 2)$, $(l3, 0, 3)$, $(l3, 0, 4)$, $(l3, 0, 5)$. This search requires worst-case storage of five projected global states,

```
(a) Classic State-space Search
1 $seen := \{s_0\}$
2 $DFS(s_0)$

$DFS(s)$
3 $workSet(s) := enabled(s)$
4 while $workSet(s) \neq \emptyset$
5     let $\alpha \in workSet(s)$
6     $workSet(s) := workSet(s) \setminus \{\alpha\}$
7     $s' := \alpha(s)$
8     if $s' \notin seen$ then
9         $seen := seen \cup \{s'\}$
10        $pushStack(s')$
11        $DFS(s')$
12        $popStack()$
end $DFS$
```

```
(b) Quasi-Cyclic State-space Search
1 $seen_g := \emptyset$
2 $seen_t := \{s_0\}$
3 $DFS(p_T, s_0)$
4 while $\neg queueEmpty()$
5     $s_g := dequeue()$
6     $seen_g := seen_g \cup \{s_g\}$
7     $seen_t := \{p_T.s_g\}$
8     $DFS(p_T, p_T.s_g)$

$DFS(p_T, s)$
9 $workSet(s) := enabled(s)$
10 while $workSet(s) \neq \emptyset$
11    let $\alpha \in workSet(s)$
12    $workSet(s) := workSet(s) \setminus \{\alpha\}$
13    $s' := \alpha(s)$
14    if $s' \notin seen_t \wedge \neg p_T(s')$ then
15        $seen_t := seen_t \cup \{s'\}$
16        $pushStack(s')$
17        $DFS(p_T, s')$
18        $popStack()$
19    if $p_T(s')$ then
20        $s'_g := \pi_G(s')$
21        if $s'_g \notin seen_g \wedge \neg inQueue(s'_g)$ then
22            $enqueue(s'_g)$
end $DFS$
```

Algorithm 1: Classical Search vs. Quasi-Cyclic Search

$seen_g = \{0, 2, 3, 4, 5\}$, and nine states in the largest DFS instance (the instance headed by $(l3, 0, 0)$). The total space consumption for this quasi-cyclic search is much less than the forty-one states that are stored by the classic state-space search shown on the left side of the figure.

It is interesting to note that the behavior of Algorithm 1(b) reduces to breadth-first search if p_T is true (and all state variables are considered global) and to depth-first search if p_T is false. Thus, for non-trivial predicate definitions the algorithm performs a search that is a hybrid of DFS and BFS. Regardless of the definition of p_T it is guaranteed to reach every state that is reached by the classic search.

Proposition 1 (State Coverage). *For a given system, quasi-cyclic state-space search visits all of the states that are visited in the classic DFS state-space search.*

Proof: Every state s' produced at line 7 of Algorithm 1(a) is the result of exploring a trace $[s_0, s_1, \ldots, s']$ whose states are stored on the stack of recursive `DFS()` calls. Algorithm 1(b) explores each such trace in segments $[s_0, \ldots, s_{b_1}]$, $[s_{b_1}, \ldots, s_{b_2}]$, $\ldots$, $[s_{b_n}, \ldots, s']$ where s_{b_i} is a p_T-state. Each segment is explored in a separate recursive DFS traversal

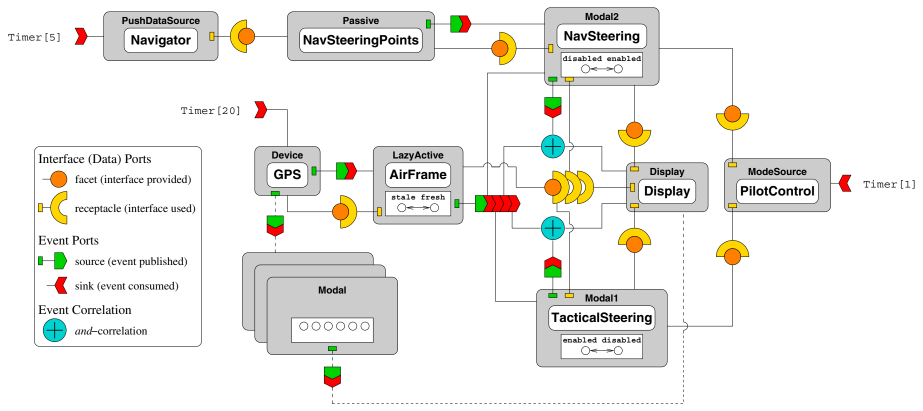


Fig. 2. Simple avionics system

and the projection onto a boundary p_T-state is stored in the queue and used to initiate the DFS traversal of the subsequent segment. The exhaustive consideration of all such boundary states in the loop beginning at line 6 assures that all segments will be explored.

Proposition 1 implies that quasi-cyclic state-space search preserves reachability properties of systems. It is also clear that any intra-region safety property is also preserved since those properties would be checked independently during the course of each `DFS()` call initiated at line 11 of Algorithm 1(b). We are working on extending quasi-cyclic state-space search to efficiently analyze LTL properties.

Proposition 2 (Space Consumption). *Quasi-cyclic state-space search requires memory bounded asymptotically by the sum of the number of distinct states in the projected global state-space and the maximal size of a region DFS.*

Proof: The maximal space consumption of Algorithm 1(b) is bounded by the size of $seen_g$ and $seen_t$. $seen_g$ is at its largest at the end of the algorithm when the global projection of all boundary states have been stored. Each `DFS()` call is executed in sequence, thus the space for $seen_t$ of each depth-first search is reused across the set of searches. This means that the largest $seen_t$ set for all DFS determines the maximal space consumption.

3 Cadena Designs

We now give a brief overview of the structure of DRE systems that are designed using Cadena, and we explain how they give rise to quasi-cyclic state-spaces.

Figure 2 presents the CORBA component model (CCM) architecture for a very simple avionics system that shows steering cues on a pilot's navigational display. Although quite small, this system is indicative of the kind of system structure found in Boeing Bold Stroke designs. In the system, the pilot can choose between two different display modes: a *tactical* display mode displays steering cues related to a tactical (*i.e.*, mission) objective, while a *navigation* display mode displays cues related to a navigational objective. Cues for the navigation display are derived in part from navigation steering points data that can be entered by the navigator.

The system is realized as a collection of components coupled together via interface and event connections. Input position data is gathered periodically at a rate of 20 Hz in the GPS component and then passed to an intermediate AirFrame component, which in a more realistic system would take position data from a variety of other sensors. Both the NavSteering and TacticalSteering component produce cue data for Display based on air frame position data. The Navigator component polls for inputs from the plane's navigator at a rate of 5 Hz that are used to form NavSteeringPoints data. This data is then used to form navigational steering cues in NavSteering. PilotControl polls for a pilot steering mode at a rate of 1 Hz and enables or disables NavSteering and TacticalSteering accordingly. NavSteering and TacticalSteering are referred to as *modal components* since they each contain a *mode variable* (represented as a component attribute) whose value (enabled,disabled) determines the component's behavior. When a steering component is in enabled mode, it processes data from AirFrame and notifies the Display component that its data is ready to be retrieved over one of the modal component's interface connections. When a steering component is in disabled mode, all incoming events from AirFrame are ignored and the component takes no other action.

There are many interesting aspects to this system and its development in Cadena that we cannot explain here due to lack of space (see [5] for more details). We focus here on issues related to the quasi-cyclic structure of the state-spaces of these systems.

In Bold Stroke applications, even though at a conceptual level component event source ports are connected to event sink ports, in the implementation, event communication is factored through a real-time CORBA event channel. Use of such infrastructure is central to Bold Stroke computation because it provides not only a mechanism for communicating events, but also a pool-based threading model, time-triggered periodic events, and event correlation. In order to shield application components from the physical aspects of the system, for product-line flexibility, and for run-time efficiency, all components are *passive* (i.e., they do not contain threads) – instead, component methods are run by event-channel threads that dispatch events by calling the event handlers ("*push methods*" in CORBA terminology) associated with event sink ports. Thus, the event channel layer is the engine of the system in the sense that the threads from its pool drive all the computation of the system. The Event Channel also provides event correlation and event filtering mechanisms. In the example system of Figure 2, *and*-correlation is used, for instance, to combine event flows from NavSteering and AirFrame into Display. The semantics of *and*-correlation on two events e_1

and e_2 is that the event channel waits for an instance of both e_1 and e_2 to be published before creating a notification event that is dispatched to the consumer of the correlation.

Periodic processing in Bold Stroke applications is achieved by having a component such as GPS subscribe to a periodic time-out (e. g. `Timer[20]`) that is published by the real-time event-channel (the event-channel contains dedicated timer threads to publish such events). The time between two occurrences of a timeout of rate r is referred to as the *frame* of r (e.g., the length of the frame associated with the 5 Hz rate is 200 milliseconds).

In constructing transition system models of Bold Stroke applications, we take advantage of the fact that in rate-monotonic scheduling theory, which is used in Bold Stroke systems, the frame associated with a rate r can be evenly divided into some whole number of r'-frames for each rate r' that is higher than r. In the example system of Figure 2, the frame of the slowest rate (1 Hz) can be divided into 5 5 Hz frames, and each 5 Hz frame can be divided into 4 20 Hz frames. The longest frame/period (the frame associated with the lowest rate) is called the *hyper-period.*

We do not keep an explicit representation of clock ticks, but instead our model enforces the following constraints related to issuing of timeouts:

– a single timeout is issued for the slowest rate group in the hyper-period,
– timeouts for rate groups, r_i and r_j where $r_i > r_j$, are issued such that r_i/r_j timeouts of rate r_i are issued in a r_j frame.

These constraints determine the total number and relative ordering of instances of timeouts that may occur in the hyper-period. Combining these constraints with the scheduling strategy implemented in Bogor for Cadena models safely approximates all interleavings of timeouts and component actions (given the assumption that no frame overruns occur) [5].

Systems such as the one in Figure 2 are captured in Cadena using specifications phrased in three parts: (1) component behavioral descriptions the describe the interface and transition semantics of component types such as the LazyActive component type in Figure 2 of which AirFrame is an instance, (2) a reusable model of a real-time CORBA event channel, and (3) system configuration information that describes the allocation of component instances and the port connections made between each of these instances as diagramed in Figure 2 (see [5] for a detailed explanation).

From these three parts, the model checker builds a state vector (pc, c, q, a, t) where the tuple components are as follows.

$pc.d = (pc.d_{r_1}, \ldots, pc.d_{r_n})$ stores program counter of each of the event-channel dispatch threads (one thread per each rate group r_i)

$pc.t$ stores the program counter for the event-channel timer thread that responds to system timer interrupts and places timeout events in dispatch queues q_{r_i}.

$c = (c_1, \ldots, c_k)$ stores the data states of component instances, each of which consists of a, possibly empty, set of mode attributes as defined by c_i's component type.

$q = (q_{r_1}, \dots, q_{r_n})$ are rate-specific queues of pairs, (c, e), recording the dispatch of event e to port c.

$a = (a_1, \dots, a_l)$ stores the current states of each of the event correlation recognition automata.

t records an abstraction of time used to trigger timeouts.

Note that in a real avionics system, there would be significant numeric computation to transform raw GPS data into a form that is useful for other components such as AirFrame. We do not represent this computation in our model for several significant reasons. First, in the actual systems supplied to us by Boeing, all such computation is stripped out for security reasons and to avoid dissemination of propriety information. Second, Boeing engineers are often concerned with reasoning about control properties associated with modes, and the elided data computations rarely influence the modal behavior of the system. In essence, Boeing engineers have by happenstance performed a manual abstraction of the system – an abstraction that produces a system that is very well-suited for model checking in that remaining mode data domains are finite and small.

Other information such as the connection information between component instances is required to define the semantics of Cadena systems. However, we do not store connection information in the state vector at all since this information remains constant through the lifetime of Bold Stroke systems.

Of the state vector components listed above, only the values $\boldsymbol{c}$ of the component mode variables are considered *global* according to the classification scheme of Section 2 – all the remaining values are *transient*. State spaces generated from Cadena models are quasi-cyclic in the sense of Section 2 because there exists a predicate p_T^C on transient components of the state vector that holds when

- each of the dispatch thread program counters $pc.d_{r_i}$ holds the value of the initial thread control point,
- the program counter $pc.t$ holds the value of the initial thread control point,
- each of the dispatch queues q_{r_i} are empty,
- each of the automata a_j is in its initial state, and
- the abstraction of time t is zero (i.e., start of hyper-period).

To explore how our proposed state-space optimization techniques scale along various axes, we consider a parameterized family of systems built off of the basic system shown in Figure 2. Those extensions were obtained by instantiating parameters m, c, and v from the following description:

- m modal components are added between GPS and NavDisplay.
- Each new mode variable from the m components above is changed non-deterministically by a new component ModalControl that is running at 1 Hz. We put a bound c on the maximum number of modal values that can be changed during a single run of ModalControl's 1 Hz timeout handler.
- Each new mode variable from the m-components above can have v different values.

4 Implementation in Bogor

Bogor [13] is an extensible and highly modular explicit-state model checking framework. It provides a rich modeling language including features that allow for dynamic creation of objects and threads, garbage collection, virtual method calls and exception handling. It also provides extension mechanisms that ease the task of customization to provide domain-specific abstraction layer and to accommodate, for example, variations in scheduling policies, search modes for state exploration, state encodings, and checkers for specification languages. Bogor employs state-of-the-art reduction techniques such as collapse compression [11], heap symmetry [12], thread symmetry [2], and partial-order reductions.

We have customized Bogor to support checking of Cadena models [5] that gives orders of magnitude space reduction from previous approaches [9]. This was done by providing a builtin abstraction of CORBA event channel semantics, and by customizing Bogor's scheduler to enforce constraints on time-triggered event occurrences and scheduling policy described in the previous section, In addition, Bogor was easily configured to identify *static* component connection data that did not need to be stored in the state vector.

4.1 Quasi-Cyclic Search

We have implemented quasi-cyclic state-space search for Cadena on top of the previous customized implementation. We describe how we extended our existing Cadena-oriented checker to implement Algorithm 1(b).

We modified the classical DFS algorithm used in [5] so that it only explores *one hyper-period* for each invocation of the DFS algorithm. This modification is done in the search module of Bogor. We use the predicate p_T^C defined in the previous section as the condition to determine when the end of the hyper-period is encountered in the search (lines b.19-b.22). The p_T^C predicate is implemented in a straight-forward manner because the interface to the state representation in Bogor provides methods to query the program counter of each thread, global and local variables.

After each transition in the search, the current state is inspected to see whether it satisfies p_T^C. If so, we then apply the π_G function to extract each modal component's modal value from the state. The relatively small state vector consisting of only modal values is then enqueued (lines b.21-b.22) if it is not in the queue and it has not been explored before. Once the state vector is enqueued, then the search backtracks to continue its search of the current hyper-period.

For each state vector of modal values s_g in the queue, the DFS algorithm is then started with a state that is constructed using p_T and s_g (lines b.8). In the actual implementation, a new instance of the Bogor model checker is created for each of these searches, thus, the stored states of previous hyper-period are garbage-collected and only the state vectors of mode values are preserved through-out the complete search.

Bogor's open design allowed us to implement this extension in less than 200 lines of Java code and less than one day. An additional 40 lines of Java code were needed to implement the parallel search.

4.2 Parallel Quasi-Cyclic Search

As will be shown in the next section, the quasi-cyclic search may potentially have a significant runtime overhead. This happens when there are many overlapping states between the quasi-cyclic regions. Since the $seen_t$ is always reset for each DFS call, overlapping states will be revisited. In contrast, the classical DFS will backtrack when such overlapping states are encountered.

In order to alleviate the time overhead, we have implemented a parallel version of the quasi-cyclic search. Note that because of the decomposition of the search that is performed by quasi-cyclic search, each DFS call (line b.8) is completely independent and may be executed in a separate thread. The DFS searches only interact by updating the shared queue and the shared set $seen_g$, which can be accessed in parallel; we synchronize those specific access points (lines b.4-b.6 and b.21-b.22). In our implementation, the user can specify how many threads to use to perform parallel DFS searches.

5 Evaluation on Realistic Designs

Our goal is to evaluate the extent to which quasi-cyclic state-space search is effective for reasoning about behavioral models of real systems. We have performed an evaluation in the context of a collection of Cadena design models that encode Bold Stroke system designs provided by Boeing engineers. The original Bold Stroke designs were developed to *illustrate* structural aspects found in real designs as a means of evaluating a wide variety of design-time analyses, but as standalone designs they reflect neither the complexity nor scale of realistic designs. In this study, we have adapted several of these Cadena designs to make them scalable in ways that correspond to the way that real system designs scale (as explained to us by Boeing engineers); specifically as systems scale the following trends are typically followed:

1. the number of modal components increases to comprise as much as one-third of the total number of system components;
2. modal components are rarely independent and can often be grouped into dependent clusters that change value in the same hyper-period (e.g., components that are always enabled/disabled as a group);
3. the domain of values for mode variables can become larger in realistic systems; boolean modes remain, but the progression of mode variable values in general becomes more complex (e.g., implementing a state machine); and
4. the number of modal components that change value independently during a hyper-period is generally bounded by a small constant that depends on the specific system.

We have adapted two existing modal design models, ModalSP and MediumSP, to introduce new modal and non-modal components in a manner that is consistent with these constraints. We experimented with variations on these models to understand the scalability of quasi-cyclic state-space search. For each model we varied the number of components added (denoted m), the number of modal components that change in a hyper-period (c), and the number of values that the added modal components will take on (v); a model configuration is denoted by the triple (m, c, v).

We summarize results for a selection of checks on these models; the complete data set is available online [1]. The *classic* model checks were run using the customization of Bogor for Cadena models [5], discussed in Section 3, and as such they represent highly-optimized state-space searches. The *quasi-cyclic* checks used the Bogor implementation described in Section 4.1. Data on total run-time and space usage were obtained on a single user workstation with a 2.53Mhz Pentium 4 and running JDK 1.4.1 on Windows XP Pro with the maximum heap of 1350Mbytes allowed for the JVM.

Figure 3 plots memory and space consumption for classical checks of three variations of the ModalSP; in its original form the ModalSP has 8 components and 125 event publications per hyper-period. The shape of these plots is characteristic of what we found for further scaling of the models; we show these *smaller* variations since for the larger ones the classic searches run out of memory very quickly and are therefore not suitable for comparative evaluation. From these plots three trends are clear[1]: (1) quasi-cyclic search requires significantly less space than the classic search, (2) the consumption of memory for quasi-cyclic search scales extremely well with increasing system size and complexity, and (3) quasi-cyclic search can incur a significant run-time overhead relative to classic search.

The MediumSP is significantly more complex than the ModalSP having 50 components and 820 events publications per hyper-period. Its state-space is more than an order of magnitude larger and consequently classic checks run out of memory even for small model configurations. Data for the $(3, 1, 2)$ and $(3, 1, 3)$ configurations of MediumSP are representative of the overall data we collected. Classical search of $(3, 1, 2)$ required 9.5 hours and used more than 650 Mbytes of memory, whereas quasi-cyclic search required 20 hours, but only 150 Mbytes of memory. Classical search of the $(3, 1, 3)$ configuration exhausted available memory, while quasi-cyclic search takes several days, but uses only slightly more than 150 Mbytes of memory. The rate of growth of memory for quasi-cyclic search suggests that scaling to significantly larger systems is possible, but the run-time cost of sequential search quickly becomes prohibitive.

The prototype parallel quasi-cyclic search was used to check several configurations of the ModalSP system on a quad-processor 500Mhz Pentium Xeon system with 4 Gbytes of memory. For ModalSP $(3, 1, 4)$ classical checking took

[1] Note that the dashed-line for the space consumption of quasi-cyclic search is a bit difficult to see. It is just above and nearly parallel to the horizontal axis in the plots. The actual memory used for each of those checks was less than 8 Mbytes.

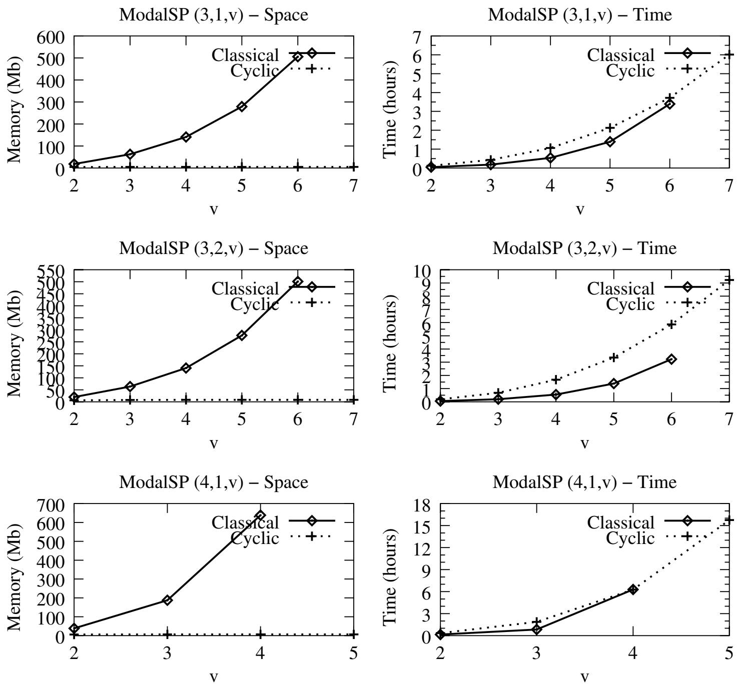


Fig. 3. Quasi-cyclic Performance for ModalSP Configurations

201 minutes, sequential quasi-cyclic checking took 360 minutes, and four-way parallel quasi-cyclic checking took 152 minutes.

5.1 Discussion

It is difficult to quantify the extent to which quasi-cyclic search will effectively scale with system size. The data collected to date suggests that explosive memory growth can be moderated significantly, but that large-scale parallelism will be required to control the growth in analysis time. Based on an informal analysis of the *available parallelism* in quasi-cyclic search we believe that fine-tuning of our parallel implementation (e.g., to work around the sequentiality of garbage collection in the JVM) will allow near linear speedup to be achieved for large numbers of processors. We are aggressively pursuing this line of research and conducting a broad empirical evaluation of the scalability of parallel quasi-cyclic search's run-time performance.

Our study was limited to Bold Stoke system design models for which quasi-cyclic search appears to be very effective at reducing memory requirements. We believe that quasi-cyclic search can also be effective for reasoning about other classes of periodic real-time systems as well as more general *control-loop* based software systems (e.g., user interface systems, web-servers). We are working to apply quasi-cyclic search to this broader class of systems.

This form of search is especially effective when a significant portion of the program data is transient and when the global data transitions deterministically across a series of quasi-cyclic regions. This latter observation suggests that careful modeling of the way that a system's environment can influence transitions of its global data is warranted. We experimented with some different environment models including a completely non-deterministic environment that simply *chooses* among possible mode variable settings and noticed a negative impact on performance. We found that capturing assumptions about environment behavior that *determinize* the transition of mode values (e.g., encoding sequencing of flight mode values from takeoff through ascent, flight, descent and landing) reduced run-time.

6 Related Work

Godefroid et. al. [8] introduced state-space caching techniques to increase the size of systems for which model checking is feasible. The main idea of the work is that one can perform a state-space search by storing only a portion of the seen before set. Studies have shown that, in general, randomly selecting the states to preserve in the seen set yields the best performance. The algorithm may visit some states several times because they have been removed from the seen set. In addition, to detect termination the algorithm needs at least as much memory as is required to store the states of the longest DFS stack. In contrast, we use specific knowledge of the system, encoded as a predicate, to identifying quasi-cyclic regions in the state-space. Quasi-cyclic search is guaranteed to find all states that the classical DFS (or state-caching search) finds and only those states; hence, it terminates whenever the classical DFS can terminate. Our algorithm may also visit states multiple times, however, based on our preliminary experiments, the ability to parallelize the search compensates for this redundant calculation.

Godefroid [7] introduced state-less search to model check systems that have a relatively small number of states that are revisited in the search and that have little non-determinism. State-less search is also effective for systems where partial order reduction significantly reduces the number of paths that need to be explored. It does not work well for systems such as the Cadena models because of the non-determinism used for safely modeling timeouts and priority-based scheduling. The non-determinism in modeling the scheduler introduces many states that need to be revisited. We implemented a state-less search for a single quasi-cyclic region of the $(3, 1, 2)$ ModalSP configuration; it ran for several hours before we terminated it in contrast to the 30 seconds required by the stateful quasi-cyclic search.

7 Conclusions

We have developed a search algorithm that decomposes a state-space search into local regions, defined by a predicate over state vectors, that can be searched independently. The algorithm is quite general, but in this paper we have emphasized how the manual definition of a predicate that captures the cyclic structure of a projection of a system's state-space can significantly reduce the memory requirements of the search. Ongoing work is seeking to automate the identification of region-defining predicates, thereby significantly increasing the breadth of applicability of our algorithm.

We have identified a class of quasi-cyclic design models for event-driven component-based designs of distributed real-time embedded systems. We described how these systems can naturally be mapped onto the quasi-cyclic search algorithm which can be implemented in the Bogor model checking framework. Preliminary experimental evaluation confirmed that the memory consumption of the algorithm scales extremely well with increasing system complexity; in fact, for our experiments memory consumption was nearly constant. While quasi-cyclic search does incur significant overhead, up to a factor of three slowdown for the systems we considered, that overhead can be overcome by even a modest parallelization of the search. We believe that significantly greater performance can be achieved through large-scale parallelism and are undertaking an experimental study to assess that hypothesis.

References

1. http://cadena.projects.cis.ksu.edu, 2003.
2. D. Bosnacki, D. Dams, and L. Holenderski. Symmetric spin. In *International Journal on Software Tools for Technology Transfer*. Springer-Verlag, 2002.
3. W. Chan, R. J. Anderson, P. Beame, D. Notkin, D. H. Jones, and W. E. Warner. Optimizing symbolic model checking for statecharts. *IEEE Transactions on Software Engineering*, 27(2):170–190, 2001.
4. E. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, 2000.
5. W. Deng, M. Dwyer, J. Hatcliff, G. Jung, and Robby. Model-checking middleware-based event-driven real-time embedded software. In *Proceedings of the 1st International Symposium on Formal Methods for Component and Objects*, 2002.
6. M. Dyer. *The Cleanroom Approach to Quality Software Development.* Wiley, 1992.
7. P. Godefroid. Model-checking for programming languages using VeriSoft. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL'97)*, pages 174–186, Jan. 1997.
8. P. Godefroid, G. J. Holzmann, and D. Pirottin. State-space caching revisited. In G. von Bochmann and D. K. Probst, editors, *Proc. Fourth International Conference on Computer Aided Verification*, volume 663 of *Lecture Notes in Computer Science*, pages 178–191. Springer, June 1993.
9. J. Hatcliff, W. Deng, M. Dwyer, G. Jung, and V. Prasad. Cadena: An integrated development, analysis, and verification environment for component-based systems. In *Proceedings of the 25th International Conference on Software Engineering*, 2003.
10. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages*, pages 58–70, Jan. 2002.
11. G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–294, May 1997.

12. R. Iosif. Symmetry reduction criteria for software model checking. In *Proceedings of Ninth International SPIN Workshop*, volume 2318 of *Lecture Notes in Computer Science*, pages 22–41. Springer-Verlag, Apr. 2002.
13. Robby, M. B. Dwyer, and J. Hatcliff. Bogor: An extensible and highly-modular model checking framework. In *Proceedings of the 9th European Software Engineering Conference held jointly with the 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2003.
14. D. Sharp. Reducing avionics software cost through component based product line development. In *Proceedings of the Software Technology Conference*, Apr. 1998.

Intelligent Editor for Writing Worst-Case-Execution-Time-Oriented Programs [⋆]

Janosch Fauster, Raimund Kirner, and Peter Puschner

Institut für Technische Informatik,
Technische Universität Wien,
Treitlstraße 3/182/1,
A-1040 Wien, Austria

Abstract. To guarantee timeliness in hard real-time systems the knowledge of the worst-case execution time (WCET) for its time-critical tasks is mandatory. Accurate and correct WCET analysis for modern processor is a quite complex problem. Path analysis is required to identify a minimal set of possible execution paths. Further, the modeling of a processor's internal states for features like caches or pipelines requires to consider possible interferences of these features.
This paper presents a new software engineering paradigm tailored to the development of real-time software. This paradigm results into more predictable programs and is therefore well-suited for the development of real-time systems. New software development tools are necessary to support developers in writing efficient code for this new paradigm. In this paper an editor is described that highlights all code that is not conform with this programming paradigm.

1 Introduction

The knowledge of the worst-case execution time (WCET) of tasks is crucial for the design of real-time systems. Only if safe upper bounds for the WCET of all time-critical tasks have been established, it becomes possible to verify the timeliness of the whole real-time system. Over the last one-and-a-half decades research in WCET analysis and real-time computing has solved many sub-problems of WCET analysis. Despite this progress in WCET analysis, there still exist three fundamental problems in the current state of the art in WCET analysis [8]:

First, WCET analysis needs exact knowledge about the possible execution paths through the analyzed code. Deriving this information automatically is, however, not possible in the general case. This is due to the fact that the control flow of a program typically depends on the input data of the program and

[⋆] This work has been supported by the IST research project "High-Confidence Architecture for Distributed Control Applications (NEXT TTA)" under contract IST-2001-32111.

a WCET bound thus cannot be predicted purely from code analysis. Further, the fully automatic program analysis to derive descriptions about possible control flows automatically is in conflict to the halting problem. Therefore, current WCET analysis tools rely on the provision of the lacking path information [2,3].

The second major problem is obtaining correct and accurate models about the timing behavior of modern processors. These processors typically use features like caches or pipelines to improve their peak-performance. The effects of these hardware features interfere with each other and are therefore hard to predict. Even worse, the behavior of a processor generally is scarely documented [1]. These facts taken together make it difficult if not impossible, to build for WCET analysis tools a correct and accurate hardware model of the target processor.

The third major problem is the complexity of the WCET analysis. Beside the problems in identifying the possible execution paths and obtaining detailed hardware-timing data, the complexity of WCET analysis itself is a problem. The number of paths that have to be analyzed to calculate a precise WCET bound are growing exponentially with the number of consecutive branches. Full path enumeration therefore becomes infeasible, except for programs having a very simple control flow [4]. To overcome this problems, approximating analysis techniques are used. These approximations causes overestimation and consequently lead to a system design with decreased utilization of hardware resources.

A possible solution to the above problems is the use of new software engineering paradigms tailored to the development of real-time software. A recently developed paradigm for this new area of software engineering is WCET-oriented programming [8,7]. It represents an unconventional way on how to write programs. The fundamental motivation of WCET-oriented programming is to reduce the number of program statements input-data dependent control flow. New software development tools are necessary to support software developers in writing efficient programs for this new software engineering paradigm.

In this paper we describe an editor that is able to highlight code that is not conform with this programming paradigm. The described analysis method to highlight the code is integrated as plug-in into a popular editor.

This article is structured as follows: Section 2 discusses WCET-oriented programming and introduces the *single-path approach* to increase predictability in real-time programs. Section 3 describes an analysis method to detect program statements causing an input-data dependent control flow. The integration of this analysis into an editor is explained in section 4. Examples are shown in section 5 to demonstrate the features of this program analysis methods. Section 6 gives a conclusion to the article.

2 Writing WCET-Oriented Programs

WCET-oriented programming is a software engineering paradigm especially tailored to the development of real-time software. This section discusses its novel aspects compared to traditional performance oriented programming. Further, the *single path approach* – a paradigm to increase predictability of real-time software – is described.

2.1 Traditional Performance-Oriented Coding

Non real-time programmers typically aim at a good average performance to allow for a high throughput. Therefore, the primary performance goal of non real-time programmers is the speed optimization for the most probable (i.e., frequent) scenarios. In order to be able to favor the frequent cases the code tests the properties of input-data sets and chooses the actions to be performed during an execution based on input data. Using input-data dependent control decisions is an effective way to achieve short execution times for the favored input-data sets. This approach is therefore suitable for optimizing the average execution time. In contrast to this, a programming style that is based on input-data dependent control decisions adversely affects the quality of the achievable WCET. This is due to the following reasons:

– *Tests to identify the current input data*: Even if an input-data set is not among the "favored" inputs it has to be tested at the points where the control flow between favored and non favored inputs splits. While the fast code makes up for the cost of the control decisions in the case of favored inputs, the execution time of the input-data tests add up to the execution time without compensation for all other data.
– *Branching costs*: Similarly to the previous argument, not just the costs for testing the properties of input data but also the costs for branching to the respective code sections increase the total execution time of non-favored cases.
– *Information-theoretical imbalance*: Every functionality on a defined input-data space and available data memory has a specific complexity. The overall problem complexity determines the number and types of operations needed to solve the problem for the given input-data space. Performance oriented, non real-time programming spreads this overall complexity unevenly over the input-data scenarios: to facilitate a high throughput, the frequent input-data scenarios are treated at computational cost that are below the complexity that would result if the total complexity would be evenly distributed to all scenarios. As the complexity inherent to a problem is constant, a cost reduction for some part of the input-data space necessarily causes higher costs for the rest of the inputs. Again, this impairs the achievable WCET (example: average versus worst-case transmission time of a string coded in Huffman code respectively a constant-length code).

Data-dependent control decisions are the results of traditional performance-optimization patterns. In the following we show that we have to apply a completely different and not so common optimization strategy, if we aim at optimizing the worst-case completion time.

2.2 Programming for the Worst-Case

As shown in the previous section, traditional (non real-time) programming tends to produce code that has a high WCET. We observe that it is the different treatment of scenarios, i.e., favoring certain input-data sets over others, that causes an increased WCET. The reasons for this were detailed above. In order to write code that has a good WCET the shortcomings of the traditional programming style have to be avoided. A novel programming strategy is needed. WCET-oriented programming (i.e., programming that aims at generating code with a good WCET) tries to produce code that is free from input-data dependent control flow decisions or, if this cannot be completely achieved, restricts operations that are only executed for a subset of the input-data space to a minimum. Note that in some applications it is impossible to treat all inputs identically. This can be due to the inherent semantics of the given problem or the limitations of the programming language used. WCET-oriented programming needs a way of thinking that is quite different from the solution strategies we normally use. As a consequence, it produces unconventional algorithms that may not look straightforward at the first sight. The resulting pieces of code, however, are characterized by competitive WCETs due to the small number of tests (and branches) on input data and the minimal information-theoretical imbalance. A small number of input-data dependent alternatives does not only keep the WCET down. It also keeps the total number of different execution paths through a piece of code low. Identifying and characterizing a smaller number of paths for WCET analysis is easier and therefore much less error-prone than dealing with a huge number of alternatives. In this way, WCET-oriented programming does not only produce code with better WCET performance but also yields more dependable WCET-analysis results and thus more dependable real-time code than traditional programming.

2.3 The Single-Path Approach for Predictable Code

As mentioned before, the problem of WCET analysis is in general complex because programs behave differently for different input data, i.e., different input data cause the code to execute on different execution paths with differing execution times. In the following we propose an approach that avoids this complexity by ensuring that the code to be WCET-analyzed has only a single execution path. This approach uses code transformations to transform input-data dependent branches and their alternatives into sequential code (input-data independent branches are not transformed). To be precise, the code resulting from the transformation avoids data dependencies in execution times by keeping input-data dependent branching local to single operations with data-independent execution times.

Constant-Time Conditional Expression. The key feature to our predictable-programming approach is the so-called constant-time conditional expression operation (CTCE). A CTCE consists of a boolean expression and two expressions. It evaluates the two expressions and returns one out of the two results. Which of the two results is actually selected depends on the truth value of the condition. Throughout this paper we use the following notation for CTCEs:

$$\mathit{cond} \ \# \ \mathit{expr}_1 \ : \ \mathit{expr}_2$$

cond represents the condition of the constant-time conditional, expr_1 and expr_2 stand for the two expressions that are evaluated. If *cond* yields true then the value of the CTCE is the result of expr_1. If *cond* evaluates to false the CTCE returns the value of expr_2. What has been described above may remind the reader of the conditional assignment operator "?:" of the C programming language. Indeed, as we assume that expr_1 and expr_2 do not have any side effects, the final result of both types of statements is the same. That is also why the syntax of the CTCE has been chosen similar to that of the C conditional expression (?:). Note, however, that there are significant differences in the control flow of the two constructs. The C conditional expression works like an *if-then-else* construct. It first evaluates the condition and then executes one of the two branches. In contrast to the conditional expression in C, the CTCE evaluates both expressions. Having computed the expressions it evaluates the condition and returns one of the two expression results as the value of the whole constant-time conditional expression. In [8] the different semantics of the two conditional operators are described in more detail.

Obviously, evaluating a conditional expression with the new operator takes longer than using an operator with the semantics of the C construct (the old operator evaluates only one expression while the new one has to evaluate both). The big advantage of the CTCE shows, however, when it comes to execution time prediction. In the C conditional the two alternatives of the branch perform different operations, and this usually implies that the alternatives have different execution times. In the general case it is therefore impossible to predict the exact execution time of the conditional. The constant-time implementation has a single, constant execution time. Its execution time is therefore predictable. This is achieved as follows: First, both alternative expressions execute in sequence. Executing both expressions unconditionally avoids the problem of finding out how the conditional behaves in a worst-case execution. Second, the result of the overall conditional is selected and returned in a simple operation with constant execution time.

The CTCE can be realized with a *conditional move instruction*, which is implemented on a number of modern processors [8].

2.4 Converting WCET-Analyzable Code into Single-Path Code

The CTCE is a construct to make the *single-path approach* explicit. In the following we illustrate how every well structured and WCET-analyzable piece of program code can be translated into code with a single execution path. By the term WCET-analyzable code we understand code for which the maximum number of iterations of every loop is known; a WCET bound is thus computable.

The translation replaces all input-data dependent branches by sequential code that uses CTCEs [6].

We only consider structured data dependent branchings of high-level languages like conditional statements (e.g., if statement) and loops; gotos or exit statements are leaved out. In order to translate a piece of code into temporally predictable code we transform these two statement types into non-branching code.

- Conditional branching statements conditionally change the values of a number of variables. The transformation of such conditional branches is straightforward. The translation process generates sequential code with constant-time conditional assignments for each of the conditionally changed variables. When translating assignments in nested conditional branches, the conditions of all nested branches have to combined in the conditions of the generated conditional assignments.
- Loops with input-data dependent termination conditions are translated in two steps. First, the loop is changed into a simple counting loop with a constant iteration count. The iteration count of the new loop is set to the maximum iteration count of the original loop. The old termination condition is used to build a new branching statement inside the new loop. This new conditional statement is placed around the body of the original loop and simulates the data dependent termination of the original loop in the newly generated counting loop. The second step of the loop translation transforms the new conditional statement, that has been generated from the old loop condition, into a constant-time conditional assignment. This way the entire loop executes in constant time.

Note that applying the described transformation to existing real-time code may yield temporal predictability at a very high cost in terms of execution time. Thus we consider the illustration of the transformation as a demonstration of the general applicability of our approach, rather than proposing to use the transformation for generating temporally predictable code from arbitrary real-time programs.

In order to come up with code that is both temporally predictable and well performing the programmer needs to use adequate algorithms, i.e., algorithms with no or minimal input-data dependent branching. This new paradigm tailored to the development of real-time software is called WCET-oriented programming. Special software engineering tools can help the software developer in writing WCET-oriented software. In the following sections a special feature for an editor is described that provides the developer with additional information about the predictability of the code. Statements that cause an input-data dependent change of the control flow (i.e., statements that violates the single-path approach) are highlighted. This information allows the developer to check the predictability of the current code. Based on this information, a refinement of the code may be possible to avoid any performance decrease caused by the automatic translation of the program into single-path code.

3 Control Flow Analysis

We developed a plug-in for an editor to support the development of WCET oriented code. This plug-in can analyze the source code and mark every control-flow that violates the *single-path paradigm.*

In order to analyze the source code, our first task is to detect all *basic blocks* in the code and to construct a *control flow graph* representing all control flows between the basic blocks. The second step is to extract data information from input code and to analyze the *data flow* in the function. For that reason we need a data structure that supports a correct and efficient analysis of the code. The data flow information we need, can be represented as a *semi-lattice*, where the elements of the lattice constitute abstract properties of the program. In our case each variable is mapped to a lattice shown in figure 1 containing the following three elements:

- the *bottom* element ($\perp$) that marks an "unreachable value"
- *maybe input dependent* (MBID) that marks a variable as "possibly input dependent" and finally
- *not input dependent* (NID) that marks an element as surly "independent from the input"

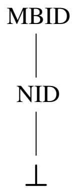


Fig. 1. Hasse Diagram of Lattice

This semi-lattice induces a partial order on its elements. The diagram represents the information content of the values, i.e. upper values have a less precise information content than lower values.

The initial state of the semi-lattice is NID for all elements in all basic blocks. But there is an exception. The *start* node contains all the informations that we have at the beginning of the algorithm and so it gets initialized as follows:

- all global variables are marked as "maybe input dependent", because we consider only intra-procedural control flows and so we don't know what happens outside a function.
- all parameters of the function are considered as input dependent too. This is evident, because these parameters are the input for the function.
- all locally defined variables are initialized as "not input dependent".

After this initialization we have a complete control flow graph and we can solve the problem with a modified fixpoint iteration scheme. For that purpose we have developed an analysis algorithm which is subdivided into three levels:

1. basic block level analysis
2. statement level analysis
3. expression level analysis

Each of this steps will now be covered in detail.

3.1 Basic Block Analysis

First of all we take an arbitrary basic block out of the control flow graph. Each basic block has an input vector and an output vector, in which the state (MBID, NID or $\perp$) at the entry point and at the exit point of the basic block for all variables is saved. Then we have to compute the current input vector. If a basic block can be reached by only one other basic block, we simply set the input vector to the output vector of that basic block. Otherwise, if the basic block can be reached by more than one control path, we have to apply an union operation on all output vectors of the preceding basic blocks in order to get the input vector of the current node. This union operation is defined in the following way:

$$\bigcup\nolimits_{bb}(p_1 \dots p_n) = \begin{cases} \perp & \text{if } \forall j \in \{1 \dots n\} : i_j = \perp \\ \text{MBID} & \text{if } \exists j \in \{1 \dots n\} : i_j = \text{MBID} \\ \text{NID} & \text{otherwise} \end{cases} \tag{1}$$

which means that if the state of a variable is NID in all vectors, it remains NID in the input vector, but if the state in only one of the considered output vectors is MBID, the new value is MBID.

After we have computed the input vector, all statements and all expressions in these statements are analyzed. Details regarding those analyses are given in the next sections. The whole procedure for calculating the output vectors of all basic blocks in the control flow graph is repeated until the output values are stable for all nodes. It can be shown that this algorithm always terminates.

3.2 Statement Analysis

As mentioned before, each statement in a basic block must be analyzed separately. It is important to notice, that a vector is associated to each statement, in which the state of all variables in that particular code location is stored. So the first task is to update the current vector. If the statement to be analyzed is the first in the basic block it takes over the input vector of the basic block, otherwise it uses the vector of the statement preceding it.
The statement analysis module mainly extracts all expressions from the statement and passes them to a dedicated expression analyzer. This is true for simple arithmetic expressions, but also for the conditional expressions in while-, do/while-, for-, if/else- and switch/case-statements. The second job for this module is to detect indirect control flow dependencies.

The example in figure 2 shows such an indirect dependency. The first if-statement in the illustration depends on a global value. That means, that the assignment-expression `b=2` is also input dependent, although `b` is assigned only

```
main()
{
  int b;
  if (a)
      b=2;
  if (b)
      doSomething;
}
```

Fig. 2. Code Example for Indirect Flow Dependency

a constant. That means furthermore that also the second if-statement depends on global input data and should be marked accordingly.

3.3 Expression Analysis

This is the central part of the analysis, because here we actually set the state of the variables. Each expression has a set of input parameters $in = \{in_1 \dots in_n\}$ and a set of output parameters $out = \{out_1 \dots out_n\}$, where it is possible to deduce the state of all output parameters from the state of the input parameters using following function:

$$\forall out_i \in out : \\ out_i = \bigcup_{expr}\{in_1 \dots in_n\} = \begin{cases} \bot & \text{if } \forall j \in \{1 \dots n\} : in_j = \bot \\ \text{MBID} & \text{if } \exists j \in \{1 \dots n\} : in_j = \text{MBID} \\ \text{NID} & \text{otherwise} \end{cases} \tag{2}$$

We distinguish the following cases:

– *assignments*, like `a=b+c`. $in = \{b, c\}$, $out = \{a\}$.
All elements at the right side of the assignment are input elements, the variable at the left side is the output element. This means, that if all elements at the right side of the assignment are NID, then also the variable at the left side is NID. But if at least one element at the right side has the value MBID, then also the variable at the left side gets MBID.
– *functions*. $in = \{\text{MBID}\}$, $out = \{$global variables, return-value, referenced values of pointers in arguments$\}$.
Functions need a more complicated handling. At this point it is important to remember that we make only intraprocedural and no interprocedural analysis. Thus after a function call we must assume, that all globally declared variables could have been changed during the function call and for that reason the value of all those variables is set to MBID. Further, the return value of the function is always input dependent. Finally, another case must be considered: if one of the arguments of the called function is a pointer, its referenced value is set to MBID, too.

- *constants*. *in* = $\emptyset$, *out* ={constants}.
  Constants are considered to be always NID.
- *structures/unions*. Each member of a structure is treated separately. Thus a single member can have the value MBID, while other ones are in the state NID. Special care must be taken for recursive structures, e.g. datastructures used for trees and lists. Here we restrict the accuracy of the analysis to assure the correctness of the algorithm. We assume that each locally defined structure is NID at the beginning. But if at least one member gets MBID, the whole structure turns to MBID and cannot be changed any longer to NID.
- *pointers*. Each pointer is represented by two state values. One for the pointer itself and a second one for the referenced variable. The pointer itself can be treated like a common variable, but the referenced variable needs special attention, because it could be referenced by more than one pointer. For that purpose we have two different strategies. The easier one assumes that all referenced variables have always MBID as their current state, while the more sophisticated strategy uses alias-informations to handle references by pointers. More informations about the alias-analysis can be found in section 3.4.
- *arrays*. Arrays are handled similarly as pointers, i.e. each array is represented with two states, one for the array itself and one for *all* its references. Thus, if one array entry changes its state from NID to MBID, the whole array is considered to be MBID and remains in this states forever. This is necessary because the analysis doesn't keep track of individual array cells and so if one value changes its state to MBID, later we don't know which cell was affected by this change and thus we must assume, that all cells could be in the state MBID. We have chosen this way to handle arrays, because it's quite simple, whereas developing a method that considers each element of an array individually would be complex and leads to higher computation and memory consumption. Arrays can be subject to aliasing too and therefore we use our aliasing analysis with arrays as well.

3.4 Alias Analysis

One or more pointers can reference the same memory location. If one pointer changes the value of that location, all other pointers that reference the same location will see this new value, too. Thus we must ensure that our analysis takes into consideration this behavior. As mentioned before, a simple strategy would be that all referenced variables are always set to MBID. By assuming the "worst case", we can be sure that our analysis will always be correct. In most cases this treatment of pointers (and arrays) is sufficient, but there are other cases were a more accurate view is needed. For more accurate results we have developed an alias analysis, which recognizes pointers that could reference the same memory location. To keep things simple, we have put a restriction on our implementation: all global variables (including globally declared pointers) are assumed to refer to the same memory location. In most cases this will not be true, but our analysis is “safe”, i.e. if an element is set to NID it is surly not input dependent, whereas if it is set to MBID it *may* depend on the input values. Following this restriction, all these global variables have to be marked as “aliased”.

```
int a;
main()
{
  int *x, *y, *z;

  y=z;
  x=y;

  *w=a;
}
```

Fig. 3. Example for Aliasing Analysis

The example in figure 3 shows a typical case, where an alias analysis should be used. The first statement (y=z) means that both y and z reference the same memory location and thus have to be aliased. The result is shown in figure 4a, where each variable has its own list containing those variables to which it could be aliased (for a more detailed description of the used data structures refer to section 4).

x $\rightarrow$
y $\rightarrow$ z $\rightarrow$
z $\rightarrow$ y $\rightarrow$

a) aliased variables 1

x $\rightarrow$ y $\rightarrow$ z $\rightarrow$
y $\rightarrow$ z $\rightarrow$ x $\rightarrow$
z $\rightarrow$ y $\rightarrow$ x $\rightarrow$

b) aliased variables 2

Fig. 4. Aliased Variables for the Example Code

The second statements adds further aliasing information. After the execution of the statement (x=y), x, y and z are aliased. The result is shown in figure 4b.

Now, the last statement in the example will need the collected aliasing informations to produce a correct result. Here the referenced variable of the pointer w gets a new, input dependent, value and so it is set to MBID. Now the analyzer sees that aliasing information is connected with this variable and therefore updates also the states of the other two pointers. At the end of this piece of code the output vector which stores the input dependency information is shown in figure 5.

	a	x	*x	y	*y	z	*z
MBID	1	0	1	0	1	0	1

Fig. 5. Flow Dependency Output Vector

4 Editor with Integrated CFA

We have implemented a plug-in for an editor that supports the data flow analysis described in the previous chapter. By using this enhanced editor the programmer has two possibilities:

1. writing programs in such a manner that nothing will be highlighted. With this proceeding one can be sure, that the program is conform to the *single path approach*.
2. often the first procedure is not possible, because a program already exists in large parts or the restrictions implied to are too big. In this case the advanced features of the editor are use to highlight all code that could violate the *single path approach* and one can verify, whether they are really critical or still tolerable.

We have implemented a plug-in for *vim*[1], that is capable of analyzing ANSI C89 code, but for more programming conveniences it supports also many ANSI C99 [5] and GNU C-expansions [9], for example additional datatypes and the //-comment. The CTCE, which is described in section 2.3, is also supported by the analysis. It has the following syntax:

$$\texttt{<expr> = <cond>} \sharp \texttt{ <expr1> : <expr2>.}$$

This expression is translated into two conditional expressions, where `<expr1>` or `<expr2>` is assigned to `<expr>`. According to the *single path approach* `<expr1>` and `<expr2>` have to be free of side effects. In languages with pointers its very difficult to locate such side effects and so we put the following constraints on these expressions:

$$\texttt{<expr1>, <expr2>} \in \{\texttt{CONST, VAR}\},$$

where `CONST` stands for a numerical constant (independent from the type) and `VAR` is an identifier representing a variable name.

The plug-in for *vim* is written in the vi scripting language. It executes the following steps:

– First of all it starts an external program, which analyzes the source code. This external program writes the results of the analysis, i.e. those line numbers in which there is code that violates the *single path approach*, in a file.
– Now we use a small program called "sequencer", which extracts from this file and returns the next line to be highlighted.

[1] `http://www.vim.org`

– We use the built-in features of vim to highlight this line.
– The sequencer-program is called in a loop until all flow dependent lines are highlighted.

The program that analyzes the code reconstructs a control flow graph after parsing the code. Each node in the control flow graph contains its assigned statements and pointers to its successor and predecessor nodes. Besides each node has two bitvectors storing the states of all variables at the entry and exit point of the basic block (`input` and `output` vectors). The dataflow analysis was implemented using the three-level algorithm presented in section 3. At statement level the detection of indirect control flow dependencies in nested loops was implemented using a global stack storing flags. If the flag at the top of the stack is set, the analyzer knows that he is currently inside a conditional expression dependent on the input value. At expression level we have some more datastructures: a *current vector* that stores the state of all variables after the execution of that expression and a field of lists storing the collected alias-informations by using the algorithm presented in section 3.4. The datastructure for storing aliasing information was initially implemented as a simple bit-matrix, where an entry x/y is true if the variables x and y are aliased. Tests have shown that this matrix could become large in functions with many variables, but is typically only sparse. Therefore we decided to implement that matrix in form of a vector with references to simple lists. Each variable has its own list, where each list entry represents a variable to which this variable is aliased.

An expression is analyzed by going recursively to the innermost subexpression. This one is inspected and the results are passed back to the surrounding expression. By using the results of all subexpressions, the current expression can be examined. This mechanism is repeated until the whole expression is analyzed. After all variable states are stable, i.e. the output vector of all nodes does not change for a whole iteration, the algorithm terminates by collecting the results and highlighting them in the editor.

5 Examples

The examples given in this section illustrate the software engineering paradigm of WCET-oriented programming and demonstrate the program analysis to test whether a code is conform to this paradigm.

5.1 Example for WCET Oriented Programming

WCET-oriented programming is intended to increase the predictability of a real-time programs. Figure 6a) shows a traditional performance-oriented implementation of find_first. Its behavior is that the loop is left as the first occurrence of a key value is found within the array. In contrast, the WCET-oriented implementation shown in figure 6b) has an almost input-data independent runtime behavior. Its only control-flow variance comes from the (?:) operator. Therefore, also conventional code generated for this implementation has a reduced variance in the execution time. As shown in [7], for processors with hardware support for conditional move instructions the execution time becomes data-independent and its WCET is even better than the WCET for the traditional performance-oriented implementation.

```
int find_first
  (int key, int a[])
{
  int i;
  int pos = 100;

  for (i=0; i<=SIZE-1;i++)
  {
    if (a[i] == key)
    {
      pos = i;
      break;
    }
  }
}
```

a) Standard version

```
int find_first_wcet
  (int key, int a[])
{
  int i;
  int pos = 100;

  for (i=SIZE-1;i>=0;i--)
  {
    pos = ((a[i]==key) ?
      i : pos);
  }
}
```

b) WCET oriented version

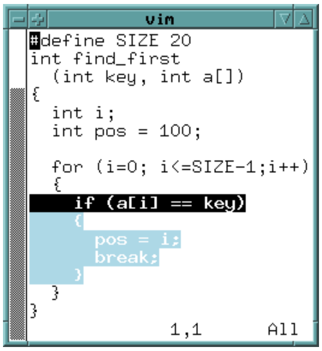
Fig. 6. Example Code: find_first

The result of the conformance analysis for the WCET-oriented paradigm is shown in figure 7. The result for the traditional performance oriented implementation given in figure 7a) shows that there exists an input-data dependent control-flow path. The result for the WCET-oriented implementation given in figure 7b) shows that there does not exist any input-data dependent control-flow path. A technical detail is that in case the (?:) operator given in figure 7b would contain other assignments than simple numeric expressions, the statement would be classified as input-data dependent control flow.

The runtime behavior of the WCET-oriented implementation is therefore more predictable than the traditional performance-oriented implementation. Information like this can be used by the software developer to implement real-time code with increased predictability.

5.2 Example for Alias Analysis

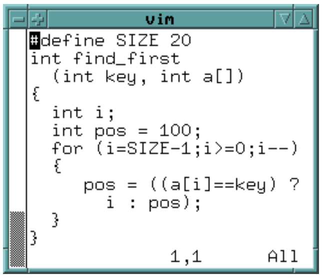
The capability of the integrated alias analysis is shown by a simple example given in figure 8. The code in this example iterates over a local array having an input-data independent content.



```
#define SIZE 20
int find_first
  (int key, int a[])
{
  int i;
  int pos = 100;

  for (i=0; i<=SIZE-1;i++)
  {
    if (a[i] == key)
    {
      pos = i;
      break;
    }
  }
}
                1,1      All
```

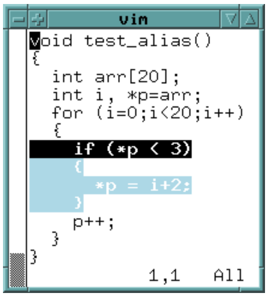
a) traditional performance-oriented implementation



```
#define SIZE 20
int find_first
  (int key, int a[])
{
  int i;
  int pos = 100;
  for (i=SIZE-1;i>=0;i--)
  {
     pos = ((a[i]==key) ?
       i : pos);
  }
}
                1,1      All
```

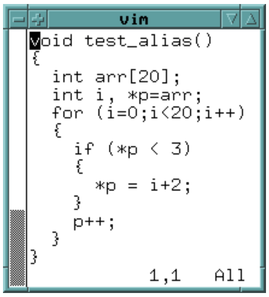
b) WCET-oriented implementation

Fig. 7. Input Dependency Analysis for find_first



```
void test_alias()
{
  int arr[20];
  int i, *p=arr;
  for (i=0;i<20;i++)
  {
    if (*p < 3)
    {
      *p = i+2;
    }
    p++;
  }
}
              1,1   All
```

a) without alias analysis



```
void test_alias()
{
  int arr[20];
  int i, *p=arr;
  for (i=0;i<20;i++)
  {
    if (*p < 3)
    {
      *p = i+2;
    }
    p++;
  }
}
              1,1   All
```

b) with alias analysis

Fig. 8. Input Dependency Analysis with Alias Information

As shown in figure 8a the conformance analysis for the WCET-oriented paradigm will classify the if statement as input-data dependent. The result for the conformance analysis with enabled alias analysis shown in figure 8b detects that this conditional control flow based on a pointer reference is input-data independent.

Due to the inherent complexity of an alias analysis for a programming language like C, our implementation is not able to precisely detect every input-data independent control-flow. But the alias analysis is safe in the sense that no input-data dependent control-flow will be classified as input-data independent.

6 Summary and Conclusion

To guarantee the timeliness of hard real-time systems it is necessary to perform WCET analysis for their time-critical tasks. Programs can be easily analyzed for their WCET, if they are translated into *single-path code*. To increase the efficiency of the translated code, a new programming paradigm – called WCET-oriented programming – has been developed. New software development tools are necessary to support developers in writing efficient code for this new paradigm.

In this paper we presented a method based on control and data flow analysis to analyze the conformance of a code with the *single-path approach*. The analysis has been implemented as a plug-in for the editor *vim*.

Experiments have been done to illustrate the application of the conformance analysis to sample algorithms. The results of the implementations for the traditional performance-oriented programming and WCET-oriented programming have been compared to demonstrate the improved predictability of WCET-oriented programming.

References

1. Pavel Atanassov, Raimund Kirner, and Peter Puschner. Using real hardware to create an accurate timing model for execution-time analysis. In *International Workshop on Real-Time Embedded Systems RTES (in conjunction with 22nd IEEE RTSS 2001)*, London, UK, Dec. 2001.
2. Antoine Colin and Isabelle Puaut. Worst case execution time analysis for a processor with branch prediction. *Real-Time Systems*, 18(2):249–274, May 2000.
3. Jakob Engblom and Andreas Ermedahl. Modeling complex flows for worst-case execution time analysis. In *Proc. 21st IEEE Real-Time Systems Symposium (RTSS)*, Orlando, Florida, USA, Dec. 2000.
4. Thomas Lundqvist and Per Stenström. Timing analysis in dynamically scheduled mircoprocessors. In *Proc. 20th IEEE Real-Time Systems Symposium (RTSS)*, pages 12–21, Dec. 1999.
5. American National Standards Insitute/International Standards Organisation. *ISO/IEC 9899:1999 Programming Languages – C.* American National Standards Institute, New York, USA, 2 edition, Dec. 1999.
6. Peter Puschner. Transforming execution-time boundable code into temporally predictable code. In Bernd Kleinjohann, K.H. (Kane) Kim, Lisa Kleinjohann, and Achim Rettberg, editors, *Design and Analysis of Distributed Embedded Systems*, pages 163–172. Kluwer Academic Publishers, 2002. IFIP 17th World Computer Congress - TC10 Stream on Distributed and Parallel Embedded Systems (DIPES 2002).
7. Peter Puschner. Algorithms for Dependable Hard Real-Time Systems. In *Proc. 8th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, Jan. 2003.
8. Peter Puschner and Alan Burns. Writing Temporally Predictable Code. In *Proceedings of the 7th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, pages 85–91, Jan. 2002.
9. Richard Stallman. *Using and Porting the GNU Compiler Collection (GCC).* iUniverse.com, Inc., USA, 2000. gcc-2.96, ISBN 0-595-10035-X.

Clock-Driven Automatic Distribution of Lustre Programs

Alain Girault[1] and Xavier Nicollin[2]

[1] INRIA Rhône-Alpes, POP ART project, France,
`Alain.Girault@inrialpes.fr`
[2] INPG/VERIMAG, France,
`Xavier.Nicollin@imag.fr`

Abstract. Data-flow programming languages use clocks as powerful control structures to manipulate data, and clocks are a form of temporal types. The clock of a flow defines the sequence of logical instants where it bears a value. This is the case of the synchronous language Lustre. We propose a solution for distributing Lustre programs, such that the distribution is driven by the clocks of the source program. The motivation is to take into account long duration tasks inside Lustre programs: these are tasks whose execution time is long compared to the other computations in the application, and whose maximal execution rate is known and bounded. Such a long duration task could be given a slow clock, but this would violate the synchronous abstraction. Distributing Lustre programs can solve this problem: the user gives a partition of the set of clocks into as many subsets as he desires computing locations, and our distribution algorithm produces one program for each such computing location. Each program only computes the flows whose clock belongs to it, therefore giving time to each long duration task to complete.

Keywords. Automatic distribution, synchronous abstraction, data-flow languages, clocks, long duration tasks, reactive systems.

1 Introduction

1.1 Reactive Systems

Reactive systems are computer systems that react continuously to their environment, at a speed determined by the latter [15]. This class of systems contrasts, with *transformational systems* (whose inputs are available at the beginning of their execution, and which deliver their outputs when terminating: e.g., compilers), and with *interactive systems* (which react continuously to their environment, but at their own speed: e.g., operating systems). Most industrial real-time systems are reactive systems: control, supervision, signal-processing systems … They must meet the following requirements:

Temporal requirements. This concerns the input rate as well as the input/output response time. To check their satisfaction on the implementation, it is necessary to know bounds on the execution time of each computation as well as on the maximal input rate.

Safety requirements. These systems are often critical ones, hence they require rigorous design methods and languages as well as formal verification and validation of their behavior.

Parallelism requirements. At least, the design must take into account the parallelism between the system and its environment. These systems are also often implemented on parallel architectures, for reasons of processor load, performance increase, fault tolerance or geographical distribution. Finally, it is convenient and natural to design such systems as sets of cooperating parallel components.

1.2 The Synchronous Data-Flow Approach and Clocks

The synchronous approach has been proposed to *ease* the design of reactive systems. It is based on the *synchronous abstraction* [4], which is similar to the abstraction made when designing synchronous circuits at the gate level. Concerning the implementation, synchronous programs are embedded inside a periodic execution loop of the form presented to the right.

```
loop each tick
   read inputs
   compute next state
   write outputs
end loop
```

There are numerous languages based upon the synchronous abstraction, among which three are data-flow languages: LUSTRE [14], SIGNAL [12], and LUCID Synchrone [11]. Such languages use *clocks* as powerful control structures to manipulate data [7]. In data-flow languages, each variable manipulated by the program is a *flow*, which is an infinite sequence of typed data, and clocks are a form of *temporal types*. The clock of a flow defines the sequence of logical instants where the flow bears a value. Unlike many hardware modelling tools, in LUSTRE any Boolean flow can be a clock. A predefined clock always exists: it is the *base clock* of the program, which is the sequence of its activation instants. That is, the base clock is the flow of `tick`s of the periodic execution loop. LUSTRE, SIGNAL, and LUCID Synchrone offer operators to upsample and downsample flows. Downsampling allows the definition of a *slower* clock, while upsampling allows the projection of a slow flow onto a faster clock. All the clocks of a program can be represented within a single *clocks tree*, whose root is the base clock.[1] A clock `C1` is then said to be *faster* than another clock `C2` iff `C2` is in the subtree whose root is `C1`.

Below is an example of a LUSTRE program (to the left) along with its clock tree (to the right). Each node of this tree is decorated with the inputs and outputs whose clock matches the node. Note that `when`, `current`, `pre`, and `->` are respectively the downsampling, upsampling, delay, and initialisation operators. The `FILTER` program has *two clocks* (the base clock of the program, and the Boolean `CK`), *two inputs* (the Boolean `CK` whose clock is the base clock, and the integer `IN` whose clock is `CK`), and *two outputs* (the integer `RES` whose clock is the base clock, and the integer `OUT` whose clock is `CK`; `OUT` is computed by calling

[1] Actually, this is not true in SIGNAL, where it is a *forest* instead of a *tree*.

the external function SLOW, and this occurs each time CK is true). The following table gives an example of a run of this program:

```
node FILTER (CK : bool; (IN : int) when CK)
     returns (RES : int; (OUT : int) when CK);
let
  RES = current ((0 when CK) -> pre OUT);
  OUT = SLOW (IN);
tel.
function SLOW (A : int) returns (B : int);
```

$$\begin{tikzcd}
\text{base } \{\mathtt{RES}, \mathtt{CK}\} \arrow[d] \\
\mathtt{CK}\ \{\mathtt{IN}, \mathtt{OUT}\}
\end{tikzcd}$$

Fig. 1. The LUSTRE FILTER program and its clock tree.

base clock cycle number	1	2	3	4	5	6	7	8	9	...
CK	T	F	F	T	F	F	T	F	F	...
IN	13			9			40			...
OUT	42			27			69			...
pre OUT	nil			42			27			...
0 when CK	0			0			0			...
(0 when CK) -> pre OUT	0			42			27			...
RES	0	0	0	42	42	42	27	27	27	...

1.3 Clock-Driven Automatic Distribution

In this paper, we present a method to achieve the *clock-driven automatic distribution* of LUSTRE programs.

By *automatic distribution*, we mean that, given a centralised source program and some distribution specifications, we will automatically build as many programs as required by the distribution specifications[2]. These programs will communicate harmoniously, so that their combined behaviour be functionally equivalent to the behaviour of the initial centralised source program.

By *clock-driven*, we mean that the distribution specifications will be expressed as a partition of the set of clocks of the centralised source program into several subsets, one for each computing location desired by the user.

1.4 Motivation: Long Duration Tasks

One of the main motivations for clock-driven distribution is the handling of *long duration tasks*, which have the following characteristics: their execution time is *long* compared to the other computations in the application; their execution time is *known* and *bounded*; and their maximal execution rate is *known* and *bounded*.

The function SLOW invoked within the FILTER program of Fig. 1 is an example of a long duration task, whose rate is given by the clock CK.

[2] This is sometimes called *semi*-automatic distribution since the user has to provide the distribution specifications.

Of course, the execution time and execution rate of a long duration task must be consistent with the performance of the hardware running the system. Otherwise the temporal requirements of the system will never be met. We will come back to this point in § 5.1.

Many reactive systems involve long duration tasks. This is the case of the CO3N4 software control system, developed at Schneider Electric for nuclear plants. One of CO3N4's subsystems had a very tight timing constraint, and its first implementation could not meet it. This subsystem consisted of two parts, a slow one performing a long duration task, and a fast one performing everything else. The engineers came up with a solution where the slow part was cut in two subparts, with the system performing at each of its cycles the fast part, and alternately the first slow subpart and the second slow subpart.

Consider a system with three tasks: task A performs slow computations (duration=8, period=deadline=32); task C performs the fast and urgent computations (duration=4, period=deadline=8); and finally task B performs medium and less urgent computations (duration=6, period=deadline=24). There are two ways to implement such a system:

1. Manual task slicing. Tasks A and B are sliced into small chunks, which are interleaved with task C (Fig. 2). This is very hard to achieve, error prone, and difficult to debug. The main reasons are that the slicing itself is complex, in the general case various subparts may communicate with each other, and finally it is difficult to come up with a correct and deadlock-free implementation.

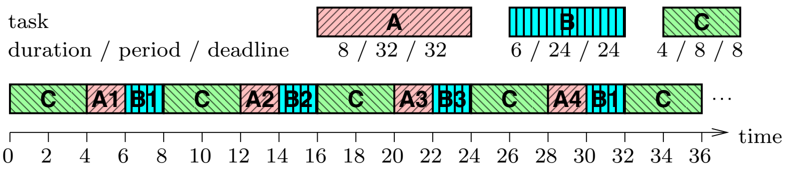


Fig. 2. Manual task slicing.

2. Distribution into three processes. Tasks A, B, and C are performed by one process each, and the task slicing is done by the scheduler of the underlying RTOS, with some priority policy.[3] There are two ways to achieve this:

(a) Manually programming three asynchronous tasks. The example of the Mars Rover Pathfinder[4] shows the limitation of this approach for critical systems: several parallel tasks had to share common resources; during execution, a priority inversion occurred [22], causing a total system reset! The system's intrinsic non-determinism combined with the fact that each subsystem was designed separately made the problem all the more difficult to debug. We claim that this approach makes the program harder to debug, test, and verify.

[3] This priority problem is *orthogonal* to the scope of this paper.

[4] Story report at `http://www.cs.cmu.edu/afs/cs/user/raj/www/mars.html`.

(b) Distributing a centralised program into three processes. To achieve automatically a distributed implementation, we choose the *object code distribution method*: first the centralised program is compiled into a single object code (a single task program), which is then distributed according to the system designer's specifications. The main advantage is that it is harder and error-prone to design directly a distributed system, hence the recent success of automatic distribution methods [13,10]. The other advantage is the possibility to debug and formally verify the centralised program *before* its distribution, which is easier and faster than debugging a distributed program. Finally, there remains the issue of the correctness of this approach: it will be addressed in § 3.1.

In other words, the solution we propose avoids, **2(b)**, not only the manual partition of A and B into small chunks A_i and B_i (as in solution **1**), but also the manual partition of the whole program into A, B, and C (as in solution **2(a)**). Moreover, since the distribution will be automated, it will allow the user to test several partitions along with several priority policies.

1.5 Outline of the Paper

Section 2 introduces the internal format of our programs. Section 3 presents a first attempt of automatic distribution and discusses why this does not work. Section 4 details our clock-driven automatic distribution method. Finally, Section 5 discusses issues like worst case execution time, implementation, and related work, and provides some concluding remarks.

2 Preliminaries

Our method for achieving the clock-driven distribution of Lustre programs heavily uses past work on their automatic distribution, based on the OC internal format. In this section, we present first this format, then the basic distribution algorithm, the communication primitives we use, and finally an OC program that will serve as a running example in future sections.

2.1 Internal Format

Our distribution algorithm [10] uses an internal format called OC (for Object Code [20]). OC is an object code format common to several synchronous languages such as Esterel [5] and Lustre [14]. An OC program is a finite deterministic automaton. This state graph can be cyclic, but in each state, there is sequential acyclic code, represented by a rooted binary directed acyclic graph (DAG) of actions. A program manipulates three kinds of variables: *input* variables can only be used as r-values; *local* and *output* variables can also be used as l-values; *output* variables can also be written to the environment.

Each DAG has one root (graphically represented by a circled dot), several unary and binary nodes, and one or more leaves:

- Unary nodes are sequential actions, which can be either an *assignment* to a local or output variable: `var:=exp`, where `exp` can contain external function calls; an *output writing*: `write(var)`; or an *external procedure call*: `proc(...,var`$_i$`,...)(...,val`$_j$`,...)`, where `var`$_i$ and `val`$_j$ are respectively the variable and value parameters.
- Binary nodes are deterministic branchings: `if var then p else q endif`, where `p` and `q` are subdags.
- Leaves, and only leaves, denote the next state number: `goto n`.

The outcome of the Lustre compiler is an OC program that performs exclusively the `compute next state` and `write outputs` phases of the execution loop (see § 1.2). It is a transformational function that terminates after completing *one* transition of the automaton. The loop itself and the `read inputs` phase are performed by a special synchronous/asynchronous interface that samples the inputs, stores their values in the local memory, and invokes the OC program. The same behaviour holds for the Esterel compiler [1].

This synchronisation of the program with its environment is not explicit in OC. Since we will need to manipulate precisely the OC code, including its interactions with the environment, we add an explicit action at the root of each DAG: it materialises the `read inputs` phase. We call this special action `go`, with the input signals as arguments. The `go` action indicates that the inputs *have been* read and that their values are available in the local memory.

This internal format is quite general since programs written in a classical imperative programming language can be compiled into this format. In fact, any OC program can be translated into a flow graph of basic blocks, and vice-versa.

2.2 The `FILTER` OC Program

We consider the OC program `FILTER`, obtained by compiling the Lustre program of Fig. 1. It has two states, depicted in Fig. 3 below along with the DAG of state 1. Besides the clocks, inputs, and outputs of the Lustre program, the OC program has one local variable, the integer `V`. The program stays in state 0 until the first time `CK` bears the value `true`; then it moves to state 1. This is due to the initialisation operator `->` in the Lustre program.

Below is an example of a run of the `FILTER` program (Fig. 4). The subscripts indicate the cycle number (logical time). Note that $\texttt{RES}_4$ is equal to $\texttt{OUT}_1$. Throughout the paper, we suppose that `SLOW` takes 7 time units to complete and that, altogether, the other computations of the program take 1 time unit. Thus, the Worst Case Execution Time (WCET) of `FILTER` is 8 time units (7+1). In the run of Fig. 4, it is reached at cycles 1 and 4. So the period of the execution loop (that is, the rate of the base clock) cannot be smaller than 8 time units. If the temporal requirements impose a smaller period, then the synchronous abstraction cannot be validated. The purpose of our distribution method is to achieve a smaller period for the computations scheduled on the base clock. In Section 3, we show that the simple distribution of `FILTER` in two communicating tasks does not solve the problem because these two tasks still share the same logical

```
state 0:
go(CK,IN);
if (CK) then {
  RES:=0;
  write(RES);
  V:=0;
  OUT:=SLOW(IN);
  write(OUT);
  goto 1;
} else {
  RES:=V;
  write(RES);
  goto 0;
} endif;
```

```
state 1:
go(CK,IN);
if (CK) then {
  RES:=OUT;
  V:=OUT;
  OUT:=SLOW(IN);
  write(OUT);
} else {
  RES:=V;
} endif;
write(RES);
goto 1;
```

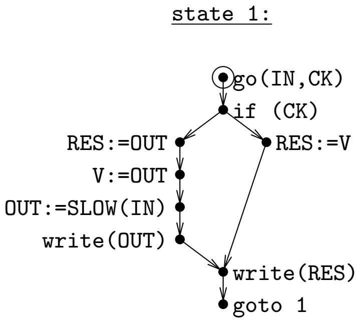


Fig. 3. The centralised FILTER program: textual representations of states 0 and 1 and graphical representations of state 1.

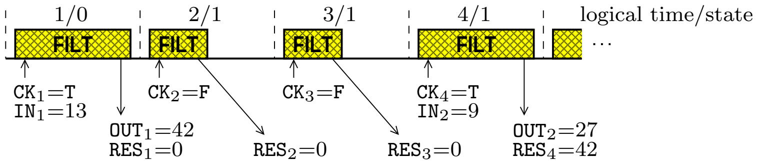


Fig. 4. A run of the centralised FILTER program.

time, and hence the same period. Next, in Section 4, we present our method to desynchronise further the two tasks, so that the fast computations be performed at a faster rate.

3 A First Attempt of Clock-Driven Automatic Distribution

3.1 Automatic Distribution Algorithm

The distribution algorithm we use in this paper is fully presented in [10]. We only outline it here. It involves the following successive steps:

1. assign a unique computing location to each sequential action, according to the *distribution specifications*; these specifications are a partition of the set of inputs and outputs of the program into n subsets, one for each computing location of the final parallel program; note that achieving the "best" localisation (whatever the optimisation criterium) of the sequential actions is out of the scope of this paper: readers interested in these topics can refer to [13];
2. replicate the program on each location;
3. on each location, suppress the sequential actions not belonging to the considered location;
4. on each state of the automaton, insert sending actions in order to solve the data dependencies between two distinct locations;

5. on each state, insert receiving actions in order to match the sending actions;
6. if required by the user, on each state, add the needed dummy communications in order to resynchronise the distributed program; we will not consider this step until Section 4.3.

This algorithm has been implemented and tested in the `ocrep` tool. It acts as a post-processor for the various synchronous languages compilers. Concretely, it takes as input an OC program `foo.oc`, and a file `foo.rep` containing the distribution specifications (see item 1 above). Also, its correctness has been formally proved in [6]: the obtained distributed program is functionally equivalent to the initial centralised one.

3.2 Communication Primitives

We choose to have two FIFO channels for each pair of locations, one in each direction. This is quite cheap in terms of execution environment, and has proved to work satisfactorily [9]. Concretely, we use two communication primitives:

The *send* primitive `send(dst,var)` sends the current value of variable `var` to location `dst` by inserting it into the queue directed towards `dst`.

The *receive* primitive `var:=receive(src)` extracts the head value from the queue starting at location `src` and assigns it to variable `var`, which is the local copy of the distant variable maintained by the distant computing location `src`.

These primitives perform both the data-transfer and the synchronisation needed between locations: when the queue is empty, `receive` is blocking. The only requirement on the network is that it must preserve ordering and integrity of messages.

3.3 Clock-Driven Distribution

From now on, let us assume that the user wishes his `FILTER` program to run over two computing locations `L` and `M`, according to the *clock distribution specifications* on the right.

location name	assigned clocks
`L`	base
`M`	`CK`

We now need to derive the corresponding distribution specifications, expressed on the inputs and outputs of the program instead of the assigned clocks (as required by `ocrep`). To do this, we just assign to a given computing location all the inputs and outputs whose clock belongs to it.

We also infer the clock of each location: this is necessary in order to know the *rate* of each location. Indeed, the knowledge of these rates will be required to compute the final WCET (in § 5.1). To compute the inferred clock of any given location, we take the root of the smallest subtree containing all the clocks assigned by the user. For the above distribution specifications, we obtain:

location name	assigned clocks	infered inputs & outputs	infered location clock
`L`	base	`CK`, `RES`	base
`M`	`CK`	`IN`, `OUT`	`CK`

Applied to the FILTER program of Fig. 3, these distribution specifications yield the following distributed program, shown in Fig. 5.

```
    location L (clock base)                       location M (clock CK)
state 0:          state 1:                  state 0:            state 1:
go(CK);           go(CK);                   go(IN);             go(IN);
send(M,CK);       send(M,CK);               CK:=receive(L);     CK:=receive(L);
if (CK) then {    if (CK) then {            if (CK) then {      if (CK) then {
  RES:=0;           OUT:=receive(M);          OUT:=SLOW(IN);      send(L,OUT);
  write(RES);       RES:=OUT;                 write(OUT);         OUT:=SLOW(IN);
  V:=0;             V:=OUT;                   goto 1;             write(OUT);
  goto 1;         } else {                  } else {            } else {
} else {            RES:=V;                   goto 0;           } endif;
  RES:=V;         } endif;                  } endif;            goto 1;
  write(RES);     write(RES);
  goto 0;         goto 1;
} endif;
```

Fig. 5. The FILTER program distributed over two locations L and M.

The distributed program of Fig. 5 calls for the following remarks:

1. The go(CK,IN) action from the centralised FILTER program has been split into *two* actions: go(CK) on location L and go(IN) on location M. This is a direct consequence of our distribution specifications.
2. On location M, all computations are scheduled on its clock CK. Hence, they are necessarily inside the then branch of the test if(CK). Only a goto can appear in the else branch, which is the case of state 0.
3. The value of CK is sent by location L to location M *at each cycle* of the base clock. As a result, location M actually runs at the speed of the base clock instead of CK.

Fig. 6 below is an example of a run of the distributed FILTER program, where L and M are both embedded in their own periodic execution loop.

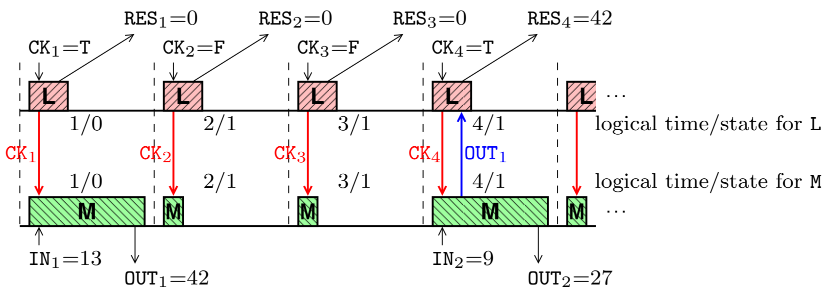


Fig. 6. A run of the distributed FILTER program.

Assume that the communications take 1 time unit. The new WCETs are respectively 2 and 8 time units for L and M. On a multi-processor architecture, the global WCET is 8, hence the situation is the same as before; on a single-processor architecture, it is worse since the global WCET is 10. One reason is that `CK` is sent by `L` to `M` at each cycle. Would the period of `L` be smaller than that of `M`, the size of the FIFO queue would be unbounded.

Instead of the distributed program of Fig. 5, we would like location `M` to run at the speed of its assigned clock, that is `CK`. This would give enough time for the computation of the long duration task embedded in `SLOW`. In the next section, we explain in details our method to achieve this.

4 The Proposed Method

In location `M` of the distributed program of Fig. 5, the `go(IN)` action could be performed *inside* the `then` branch of the test `if(CK)`. Indeed, the clock of `IN` is `CK`, and therefore a new value of `IN` is only expected at those cycles where `CK` is `true`. If we manage to move the `go(IN)` action inside the `then` branch of the test, then a carefully chosen bisimulation should be able to detect useless branchings. Suppressing such a useless `if(CK)` test on location `M` will obviate the sending of `CK` by location `L`, therefore allowing the two programs to be desynchronised.

4.1 Moving the `go` Actions Downward

Moving the `go` actions downward concerns only the programs of the computing locations whose clock is *not* the base clock. For each such program, let `CK` be its clock and $\mathtt{IN}_1, \ldots, \mathtt{IN}_n$ be its n inputs, as specified in the distribution specifications. Hence, the current action at the root of its DAGs is $\mathtt{go(IN}_1, \ldots, \mathtt{IN}_n\mathtt{)}$. Then, we traverse each of its DAGs downward, starting from the root, as follows:

– On a unary node, continue in the next node.
– On a branching `if(var)`, if `var` is `CK`, then insert $\mathtt{go(IN}_1, \ldots, \mathtt{IN}_n\mathtt{)}$ at the beginning of the `then` branch, mark the DAG, and continue in the branching closure; otherwise continue in both branches `then` and `else`.
– On a branching closure, continue in the next node.
– On a leaf, do nothing.
– At the end of the traversal, if the DAG is marked, then remove the `go` at its root. It means that a `go` action has been inserted somewhere in the DAG.

To illustrate this algorithm, we apply it to the `FILTER` program. According to our distribution specifications (see § 3.3), only the DAG of location `M` needs to be traversed. Fig. 7 shows the result for state 0 (it is similar for state 1).

```
loc. M (clock CK) - state 0
  go(IN);
  if (CK) then {
    OUT:=SLOW(IN);
    write(OUT);
    goto 1;
  } else {
    goto 0;
  } endif;
```

$\rightsquigarrow$

```
loc. M (clock CK) - state 0
  if (CK) then {
    go(IN);
    OUT:=SLOW(IN);
    write(OUT);
    goto 1;
  } else {
    goto 0;
  } endif;
```

Fig. 7. State 0 of location M: before and after the traversal.

4.2 Suppressing Useless Branchings

In [8], we have presented an algorithm for suppressing binary branching whose branches are observationally equivalent. Explaining this algorithm in details is out of the scope of the present article. Shortly, it involves traversing each DAG downward from its root, and for each branching encountered, in checking whether or not both branches are observationally equivalent. If they are, then the branching is suppressed; otherwise, it is kept. Checking if two branches are observationally equivalent is done with an on-the-fly bisimulation, called *test bisimulation*. This test bisimulation is formally expressed by three axioms and six inference rules, whose soundness and completeness have been proved in [8].

For our method to work, we divide Step 4 of our distribution algorithm (see § 3.1) into two parts. The test bisimulation and branching suppression will take place between these two parts. During the first part, we only insert the `send` actions concerning the unary nodes of the DAG, that is, everything but the branchings. During the second part, we insert the `send` actions concerning the binary nodes, that is the branchings that were not suppressed.

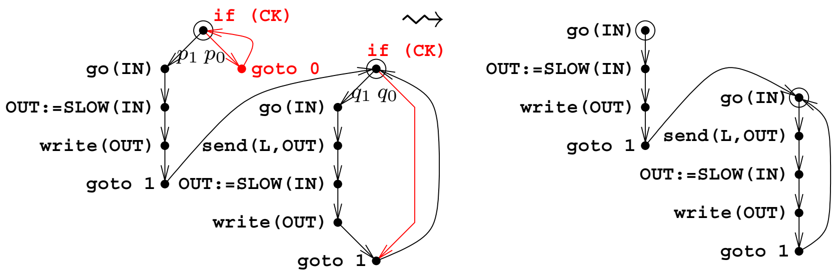


Fig. 8. Suppression of the branchings in the DAGs of location M.

Fig. 8 above shows the result of applying our test bisimulation to the DAGs of location M. In state 1, the parts labelled q_1 and q_0 are found to be bisimilar, hence the branching is suppressed. The same goes in state 0 for p_1 and p_0, even though there is a `goto` in the `else` branch.

4.3 Final Result

At this point, there remains to insert the `send` actions concerning the branchings that have not been suppressed. For the `FILTER` program, the only remaining branching is the `if(CK)` of location `L`, and since `CK` belongs to location `L`, nothing is inserted. Finally, all the `receive` actions need to be inserted. For the `FILTER` program, the final result is shown in Fig. 9:

```
      location L (clock base)                    location M (clock CK)
state 0:           state 1:                 state 0:          state 1:
go(CK);            go(CK);                  go(IN);           go(IN);
if (CK) then {     if (CK) then {           OUT:=SLOW(IN);    send(L,OUT);
  RES:=0;            OUT:=receive(M);       write(OUT);       OUT:=SLOW(IN);
  write(RES);        RES:=OUT;              goto 1;           write(OUT);
  V:=0;              V:=OUT;                                  goto 1;
  goto 1;          } else {
} else {             RES:=V;
  RES:=V;          } endif;
  write(RES);      write(RES);
  goto 0;          goto 1;
} endif;
```

Fig. 9. The final `FILTER` program distributed over two locations `L` and `M`.

Fig. 10 shows a run of the new distributed `FILTER` program on a distributed architecture, to be compared with Fig. 6.

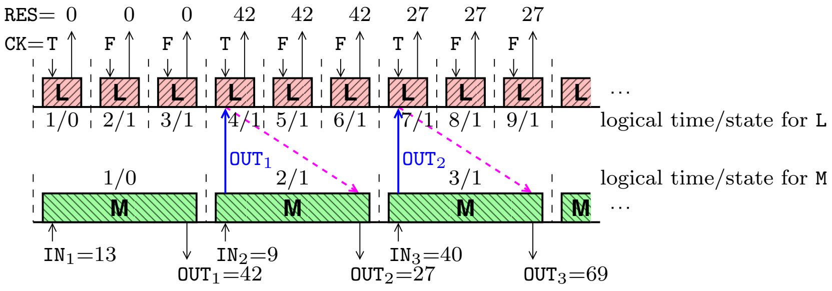


Fig. 10. A run of the clock-driven distributed `FILTER` program.

Observe that the period of `L` can now be smaller than that of `M`. In this run, the former is one third of the latter, `CK` being `true` once every three instants. For instance, with the same durations as in Section 3.3, the periods of `L` and `M` can be respectively 3 and 9.

Now, tasks `L` and `M` could even be run at a faster rate, not taking into account their clocks. In such a case, `receive` being blocking when the queue is empty, `L` will not be able to run more than three times faster than `M`. In contrast, `M`'s period could perfectly be smaller than three times `L`'s period. This would result in an unbounded FIFO queue between `M` and `L`. To prevent this, we can add a dummy communication, sent by `L` in state 1 at the beginning of its `then` branch, and received by `M` in state 1 just before its `goto` (represented by the dashed line in Fig. 10). This is exactly the purpose of Step 6 of our distribution algorithm. Such a dummy communication would force `L` and `M` to be loosely synchronised.

On a centralised architecture, Fig. 11 shows the schedule obtained when the periods of `L` and `M` are respectively 5 and 15. No communication takes place before the second occurrence of `CK`, which corresponds to cycle 4 of task `L` and to the beginning of cycle 2 of task `M`. At this point, task `L` begins and is suspended as soon as it reaches the `OUT:=receive(M)` action since the queue is empty. Then, `M` begins and immediately sends `OUT`, which resumes `L`. Since `L` has a higher priority, it preempts `M` and completes its execution. Then, `M` resumes and continues until the beginning of `L`'s cycle 5 when it is preempted by `L`. This results in slicing `M` into `M1`, `M2`, and `M3`. Here, the deadlines are met thanks to the distribution in two processes.

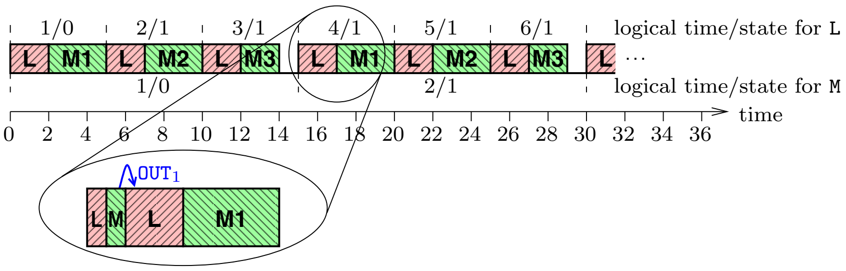


Fig. 11. Centralised schedule obtained with the clock-driven distribution.

4.4 The New Distribution Algorithm

Taking into account the two modifications described above, our new distribution algorithm involves the following successive steps:

1. locate each sequential action; same as § 3.1-Step 1;
2. replicate the program; same as § 3.1-Step 2;
3. suppress the sequential actions; same as § 3.1-Step 3;
4. on each state and each location, move the `go` action downward according to the clock of the computing location; see § 4.1;
5. on each state of the automaton, insert `send` actions to solve the data dependencies between two distinct locations, *except* when concerning branching actions;
6. on each state and each location, suppress the useless branchings; see § 4.2;
7. on each state, insert `send` actions to solve the data dependencies concerning the branching actions that were not suppressed during Step 6;
8. insert `receive` actions; same as § 3.1-Step 5;
9. add dummy communications; same as § 3.1-Step 6.

5 Discussion

5.1 Validating the Synchronous Abstraction

Validating the synchronous abstraction is always crucial when programming reactive systems. This involves computing the WCET of the object code generated by the compiler, and comparing it to the execution loop period it is embedded in (see § 1.2). In our case, it is more complex because the program is distributed into n processes. We thus have n WCET, each corresponding to a program whose execution rate is the location's clock (see § 3.3). We then compute the utilisation factor of the processor and check the Liu&Layland condition to know whether a static or dynamic priority schedule is feasible or not [17]: namely Rate Monotonic (RM) for the static priority policy and Earliest Deadline First (EDF) for the dynamic priority policy. Note that to avoid uncontrollable context switching between tasks, static priorities are preferable for hard real-time systems.

Concerning our `FILTER` example, the WCET of locations `L` and `M` are respectively 2 and 8. Since the rates of locations `L` and `M` are respectively 5 and 15, the Liu&Layland condition for RM holds, see besides. As a result, the schedule shown in Fig. 11 above is feasible with static priorities (RM policy). Note that, 15 being a multiple of 5, the bound on the utilisation factor is 1 instead of $2(\sqrt{2}-1)$.

$$\boxed{\frac{2}{5}+\frac{8}{15}=\frac{14}{15}\leq 1}$$

5.2 Implementation

The method presented in this paper has been implemented in two software tools: on one hand the `ocrep` tool[5], to distribute automatically OC programs with the branching reduction, and on the other hand the `ludivin`[6] GUI [21], to build automatically the clocks tree of a Lustre program, help the user specify a desired distribution, and call `ocrep` to do the job.

5.3 Related Work

Our work is related to Giotto [16], a compiler for embedded systems. Its abstraction consists of instantaneous communication, time-deterministic computation, and value-deterministic computation. This is very much like the synchronous abstraction. Giotto's basic functional unit is the task, which is a periodically executed piece of code. One important point is that the period must be *static*. In contrast, clocks can specify *dynamic* periods. A Giotto program can also be annotated with *platform constraints*, which are similar to our distribution specifications: a constraint may map a particular task to a particular CPU. The Giotto compiler schedules the tasks on the target architecture, and guarantees that the logical semantics is preserved (both functionality and timing). However, a Giotto program might be *over constrained* when it does not permit any execution consistent with the platform constraints. In such a case, the compiler

[5] `ocrep` is available at `www.inrialpes.fr/pop-art/people/girault/Ocrep`.

[6] `ludivin` is available at `www.inrialpes.fr/pop-art/people/girault/Ludivin`.

rejects it as non valid. In contrast, our method always produces an executable distributed program. Then, determining whether this program meets the desired timing constraints is left to the user. Since we use finite deterministic automata, computing such a worst case execution time is straightforward.

Also perhaps related is the code generator Real-Time Workshop [23] for Simulink, which includes multi-tasking support for multi-rate models. The authors do not know if this solution would allow a similar schedulability analysis for real-time applications.

Another related work is Esterel's *asynchronous tasks* mechanism [19]. Basically, a synchronous Esterel program can launch an asynchronous task by means of a dedicated output signal (called `exec`), and then be warned of its termination by means of a dedicated input signal (called `return`). Although not as powerful and flexible as clocks, this mechanism allows long duration tasks to be taken into account. However, they are handled *externally* w.r.t. the synchronous program, while our method allows them to be handled *inside* the program.

Finally, there has been a lot of work done on the automatic distribution of Signal programs [18,2,3]. Distribution in Signal is performed by first compiling the program into a *hierarchical data-flow graph* with *conditional dependency equations*, called a Synchronised Data-Flow Graph (SDFG). Vertices of this SDFG are signals and variables, while the edges are labelled with clocks. Therefore, each vertex is located in a hierarchy of clocks. In the most general case, the set of all the program's clocks form a *forest* instead of a *tree*. This means that the program *does not* have a base clock, which raises code generation problems since the periodic rate of the program cannot be statically determined. Yet, for a wide class of programs, the forest is reduced to a single tree and the program does have a base clock. Such programs are called *endochronous*. Once the SDFG is built, communications are inserted in it, then subgraphs are extracted corresponding to different computing locations, and finally sequential code is generated from each subgraph.

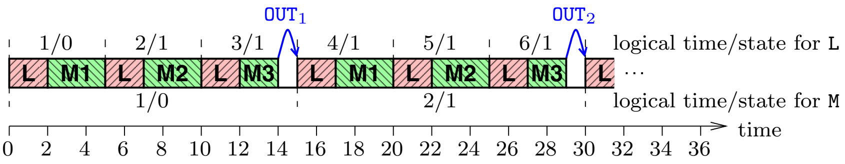


Fig. 12. Centralised schedule obtained when `OUT` is sent as soon as possible.

5.4 Concluding Remarks

To conclude, we would like to stress the original contributions of our work. We offer a method and tool to automatically distribute Lustre programs according to the clocks of the source code, hence avoiding the manual partition of an entire system into several tasks with different periods (the user only needs to partition the set of the program's clocks). This allows long duration tasks to be taken into account inside a synchronous program, which would not be otherwise possible. Once the distribution is done, the Liu&Layland conditions can be checked to choose the most suited priority policy.

Finally, in the program of Fig. 9, a standard data-flow analysis could allow us to move the sending of `OUT` from `M` to `L` backwards, that is just after the assignment to `OUT`. As a result, the preemptions taking place in Fig. 11 would not occur, therefore resulting in the run shown in Fig. 12. This future extension would have to take place as the very last step of our distribution algorithm, that is, even after the optional resynchronisation step.

References

1. C. André, F. Boulanger, and A. Girault. Software implementation of synchronous programs. In *International Conference on Application of Concurrency to System Design, ICACSD'01*, pages 133–142, Newcastle, UK, June 2001. IEEE.
2. P. Aubry, P. Le Guernic, and S. Machard. Synchronous distributions of Signal programs. In *29th Hawaii International Conference on System Sciences, HICSS-29*, pages 656–665, Honolulu, USA, January 1996. IEEE.
3. A. Benveniste, B. Caillaud, and P. Le Guernic. Compositionality in dataflow synchronous languages: Specification and distributed code generation. *Information and Computation*, 163:125–171, 2000.
4. A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages twelve years later. *Proceedings of the IEEE*, 2002. Special issue on embedded systems.
5. G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.
6. B. Caillaud, P. Caspi, A. Girault, and C. Jard. Distributing automata for asynchronous networks of processors. *European Journal of Automation (RAIRO-APII-JESA)*, 31(3):503–524, 1997. Research report INRIA 2341.
7. P. Caspi. Clocks in data-flow languages. *Theoretical Computer Science*, 94:125–140, 1992.
8. P. Caspi, J.-C. Fernandez, and A. Girault. An algorithm for reducing binary branchings. In P.S. Thiagarajan, editor, *15th Conference on the Foundations of Software Technology and Theoretical Computer Science, FST&TCS'95*, volume 1026 of *LNCS*, Bangalore, India, December 1995. Springer-Verlag.
9. P. Caspi and A. Girault. Execution of distributed reactive systems. In S. Haridi, K. Ali, and P. Magnusson, editors, *1st International Conference on Parallel Processing, EURO-PAR'95*, volume 966 of *LNCS*, pages 15–26, Stockholm, Sweden, August 1995. Springer-Verlag.
10. P. Caspi, A. Girault, and D. Pilaud. Automatic distribution of reactive systems for asynchronous networks of processors. *IEEE Trans. on Software Engineering*, 25(3):416–427, May/June 1999.
11. P. Caspi and M. Pouzet. Lucid synchrone: une extension fonctionnelle de Lustre. In *Journées Francophones des Langages Applicatifs (JFLA)*, Morzine, France, February 1999. Inria.
12. P. Le Guernic, T. Gautier, M. Le Borgne, and C. Lemaire. Programming real-time applications with Signal. *Proceedings of the IEEE*, 79(9):1321–1336, September 1991.

13. R. Gupta, S. Pande, K. Psarris, and V. Sarkar. Compilation techniques for parallel systems. *Parallel Computing*, 25(13):1741–1783, 1999.
14. 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.
15. D. Harel and A. Pnueli. On the development of reactive systems. In *Logic and Models of Concurrent Systems, NATO*. Springer-Verlag, 1985.
16. T.A. Henzinger, B. Horowitz, and C.M. Kirsch. Giotto: A time-triggered language for embedded programming. In *First International Workshop on Embedded Software, EMSOFT'01*, volume 2211 of *LNCS*, Tahoe City, USA, October 2001. Springer Verlag.
17. C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in hard real-time environnement. *Journal of the ACM*, 20(1):46–61, January 1973.
18. O. Maffeïs. *Ordonnancements de graphes de flots synchrones ; Application à la mise en œuvre de Signal.* PhD Thesis, University of Rennes I, Rennes, France, January 1993.
19. J.-P. Paris. *Exécution de tâches asynchrones depuis Esterel.* PhD Thesis, University of Nice, Nice, France, 1992.
20. J.A. Plaice and J.-B. Saint. *The Lustre-Esterel Portable Format.* INRIA, Sophia-Antipolis, France, 1987. Unpublished report.
21. F. Salpétrier. Interface graphique utilisateur pour la répartition de programmes Lustre dirigée par les horloges. Master's thesis, ESISAR, Valence, France, June 2002.
22. L. Sha, R. Rajkumar, and J.P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. on Computers*, 39:1175–1185, September 1990.
23. The MathWorks, Inc. *Real-Time Workshop User's Guide, Version 3*, January 1999.

Reasoning about Abstract Open Systems with Generalized Module Checking

Patrice Godefroid

Bell Laboratories, Lucent Technologies,
god@bell-labs.com

Abstract. We present a framework for reasoning about abstract open systems. Open systems, also called "reactive systems" or "modules", are systems that interact with their environment and whose behaviors depend on these interactions. Embedded software is a typical example of open system. Module checking [KV96] is a verification technique for checking whether an open system satisfies a temporal property no matter what its environment does. Module checking makes it possible to check adversarial properties of the "game" played by the open system with its environment (such as "is there a winning strategy for a malicious agent trying to intrude a secure system?"). We study how module checking can be extended to reason about 3-valued abstractions of open systems in such a way that both proofs and counter-examples obtained by verifying arbitrary properties on such abstractions are guaranteed to be sound, i.e., to carry over to the concrete system. We also introduce a new verification technique, called *generalized module checking*, that can improve the precision of module checking. The modeling framework and verification techniques developed in this paper can be used to represent and reason about abstractions automatically generated from a static analysis of an open program using abstraction techniques such as predicate abstraction. This application is illustrated with an example of open program and property that cannot be verified by current abstraction-based verification tools.

1 Introduction

Software verification via automatic abstraction and model checking is currently an active area of research (e.g., [BR01,CDH$^+$00,DD01,HJMS02,VHBP00]). This approach consists of automatically extracting a model out of a program by statically analyzing its code, and then of analyzing this model using model-checking techniques. If the model-checking results are inconclusive due to too much information being lost in the current abstraction, the model can then be automatically refined into a more detailed one provided the abstraction process can be parameterized and adjusted dynamically guided by the verification needs, as is the case with predicate abstraction [GS97] for instance. Current frameworks and tools that follow the above paradigm typically use traditional formalisms (such as Kripke structures or Labeled Transition Systems) for representing models, while the soundness of their analysis is based on using a simulation relation for relating the abstract model to the concrete program being analyzed. Two well-known drawbacks of these design choices are that the scope of verification is then limited to universal properties, and that counter-examples are generally unsound since abstraction usually introduces unrealistic behaviors that may yield spurious errors being reported when analyzing the model. In practice, the second limitation is perhaps more severe since the relative popularity of model checking (especially in industry) is due to its ability to detect errors that would be very hard to find otherwise, and not so much to its ability to prove "correctness".

Recently [GJ02,GHJ01,HJS01,BG00,BG99], it was shown how automatic abstraction can be performed to verify arbitrary formulas of the propositional μ-calculus [Koz83] in such a way that both correctness proofs and counter-examples are guaranteed to be sound. The key to make this possible is to represent abstract systems using richer models that distinguish properties that are true, false and unknown of the concrete system. Reasoning about such systems thus requires 3-valued temporal logics [BG99], i.e., temporal logics whose formulas may evaluate to *true*, *false* or $\perp$ ("unknown") on a given model. Then, by using an automatic abstraction process that generates by construction an abstract model which is less complete than the concrete system with respect to a completeness preorder logically characterized by 3-valued temporal logic, every temporal property ϕ that evaluates to *true* (resp. *false*) on the abstract model automatically holds (resp. does not hold) of the concrete system, hence guaranteeing soundness of both proofs and counter-examples. In case ϕ evaluates to $\perp$ on the model, a more precise verification technique called *generalized model checking* [BG00,GJ02] can be used to check whether there exist concretizations of the abstract model that satisfy ϕ or violate ϕ; if a negative answer is obtained in either one of these two tests, ϕ does not hold (resp. holds) of the concrete system. Otherwise, the analysis is still inconclusive and a more complete (i.e., less abstract) model is then necessary to provide a definite answer concerning this property of the concrete system. This approach can be used to both prove and refute arbitrary formulas of the propositional μ-calculus.

In this paper, we investigate how the scope of program verification via automatic abstraction can be extended to deal with programs implementing *open systems*. An *open system*, also called "reactive system" or "module", is a system that interacts with its environment and whose behavior depends on this interaction. Embedded software is a typical example of open system. It has been argued [KV96] that temporal properties of an open system should be verified with respect to all possible environments for that system. This problem is known as the *module checking* problem [KV96]. Module checking makes it possible to check adversarial properties of the "game" played by the open system and its environment (such as "is there a winning strategy for a malicious agent trying to intrude a secure system?").

We study how module checking can be extended to reason about 3-valued abstractions of open systems in such a way that both proofs and counter-examples obtained by verifying arbitrary properties of such abstractions are guaranteed to be sound, i.e., to carry over to the concrete system. We also introduce a new verification technique, called *generalized module checking*, that can improve the precision of module checking. The practical motivation of this paper is thus to develop a framework for representing and reasoning about abstractions automatically generated from a static analysis of an open program using abstraction techniques such as predicate abstraction. Our framework is illustrated by an example of application in Section 6. Note that existing software verification-by-abstraction frameworks and tools (e.g., [BR01,CDH$^+$00,DD01, HJMS02,VHBP00]) do not currently support verification techniques for open programs.

2 Background

2.1 3-Valued Models and Generalized Model Checking

In this section, we recall the main ideas and key notions behind the framework of [BG99,BG00,GHJ01,HJS01] for reasoning about partially defined systems. Examples of modeling formalisms for representing such systems are *partial Kripke structures* (PKS) [BG99], *Modal Transition Systems* (MTS) [LT88,GHJ01] or *Kripke Modal Transition Systems* (KMTS) [HJS01].

Definition 1. – *A* KMTS *M is a tuple* $(S, P, \xrightarrow{must}, \xrightarrow{may}, L)$*, where S is a nonempty finite set of states, P is a finite set of atomic propositions,* $\xrightarrow{may} \subseteq S \times S$ *and* $\xrightarrow{must} \subseteq S \times S$ *are transition relations such that* $\xrightarrow{must} \subseteq \xrightarrow{may}$*, and* $L : S \times P \to \{true, \bot, false\}$ *is an* interpretation *that associates a truth value in* $\{true, \bot, false\}$ *with each atomic proposition in P for each state in S.*
- *An MTS is a KMTS where* $P = \emptyset$.
- *A PKS is a KMTS where* $\xrightarrow{must} = \xrightarrow{may}$.
- *A Kripke structure (KS) is a PKS where* $\forall s \in S : \forall p \in P : L(s, p) \neq \bot$.

The third value $\bot$ (read "unknown") and *may*-transitions unmatched by *must*-transitions are used to model explicitly a loss of information due to abstraction concerning, respectively, state or transition properties of the concrete system being modeled. A standard, *complete* Kripke structure is a special case of KMTS where $\xrightarrow{must} = \xrightarrow{may}$ and $L : S \times P \to \{true, false\}$, i.e., no proposition takes value $\bot$ in any state. It is worth noting that PKSs, MTSs and KMTSs are all equally expressive (i.e., one can translate any formalism into any other) [GJ03].

In interpreting propositional operators on KMTSs, we use Kleene's strong 3-valued propositional logic [Kle87]. Conjunction $\wedge$ in this logic is defined as the function that returns *true* if both of its arguments are *true*, *false* if either argument is *false*, and $\bot$ otherwise. We define negation $\neg$ using the function '*comp*' that maps *true* to *false*, *false* to *true*, and $\bot$ to $\bot$. Disjunction $\vee$ is defined as usual using De Morgan's laws: $p \vee q = \neg(\neg p \wedge \neg q)$. Note that these functions give the usual meaning of the propositional operators when applied to values *true* and *false*.

Propositional modal logic (PML) is propositional logic extended with the modal operator AX (which is read "for all immediate successors"). Formulas of PML have the following abstract syntax: $\phi ::= p \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid AX\phi$, where p ranges over P. The following 3-valued semantics generalizes the traditional 2-valued semantics for PML.

Definition 2. *The value of a formula ϕ of 3-valued PML in a state s of a KMTS $M = (S, P, \xrightarrow{must}, \xrightarrow{may}, L)$, written $[(M,s) \models \phi]$, is defined inductively as follows:*

$$\begin{aligned}
[(M,s) \models p] &= L(s,p)\\
[(M,s) \models \neg\phi] &= comp([(M,s) \models \phi])\\
[(M,s) \models \phi_1 \wedge \phi_2] &= [(M,s) \models \phi_1] \wedge [(M,s) \models \phi_2]\\
[(M,s) \models AX\phi] &= \begin{cases} true \;\; if\, \forall s' : s \xrightarrow{may} s' \Rightarrow [(M,s') \models \phi] = true\\ false \; if\, \exists s' : s \xrightarrow{must} s' \wedge [(M,s') \models \phi] = false\\ \bot \quad\;\; otherwise \end{cases}
\end{aligned}$$

This 3-valued logic can be used to define a preorder on KMTSs that reflects their degree of completeness. Let $\leq$ be the *information ordering* on truth values, in which $\bot \leq$ *true*, $\bot \leq$ *false*, $x \leq x$ (for all $x \in \{true, \bot, false\}$), and $x \not\leq y$ otherwise.

Definition 3 ($\preceq$). *Let $M_A = (S_A, P, \xrightarrow{must}_A, \xrightarrow{may}_A, L_A)$ and $M_C = (S_C, P, \xrightarrow{must}_C, \xrightarrow{may}_C, L_C)$ be KMTSs. The* completeness preorder *$\preceq$ is the greatest relation $\mathcal{B} \subseteq S_A \times S_C$ such that $(s_a, s_c) \in \mathcal{B}$ implies the following:*

- *$\forall p \in P : L_A(s_a, p) \leq L_C(s_c, p)$,*
- *if $s_a \xrightarrow{must}_A s'_a$, there is some $s'_c \in S_C$ such that $s_c \xrightarrow{must}_C s'_c$ and $(s'_a, s'_c) \in \mathcal{B}$,*
- *if $s_c \xrightarrow{may}_C s'_c$, there is some $s'_a \in S_A$ such that $s_a \xrightarrow{may}_A s'_a$ and $(s'_a, s'_c) \in \mathcal{B}$.*

This definition allows to abstract M_C by M_A by letting truth values of propositions become $\bot$ and by letting *must*-transitions become *may*-transitions, but all *may*-transitions of M_C must be preserved in M_A. We then say that M_A is *more abstract*, or *less complete*, than M_C. The inverse of the completeness preorder is called *refinement preorder* in [LT88,HJS01,GHJ01]. Note that relation $\mathcal{B}$ reduces to a simulation relation when applied to MTSs with *may*-transitions only.

It can be shown that 3-valued PML logically characterizes the completeness preorder [BG99,HJS01,GHJ01].

Theorem 1. *Let $M_A = (S_A, P, \xrightarrow{must}_A, \xrightarrow{may}_A, L_A)$ and $M_C = (S_C, P, \xrightarrow{must}_C, \xrightarrow{may}_C, L_C)$ be KMTSs such that $s_a \in S_A$ and $s_c \in S_C$, and let Φ be the set of all formulas of 3-valued PML. Then,*

$$s_a \preceq s_c \text{ iff } (\forall \phi \in \Phi : [(M_A, s_a) \models \phi] \leq [(M_C, s_c) \models \phi]).$$

In other words, KMTSs that are "more complete" with respect to $\preceq$ have more definite properties with respect to $\leq$, i.e., have more properties that are either *true* or *false*. Moreover, any formula ϕ of 3-valued PML that evaluates to *true* or *false* on a KMTS has the same truth value when evaluated on any more complete structure. This result also holds for PML extended with fixpoint operators, i.e., the propositional μ-calculus [BG99,BG00].

In [GHJ01], we showed how to adapt the abstraction mappings of [Dam96] to construct abstractions that are less complete than a given concrete program with respect to the completeness preorder.

Definition 4. *Let $M_C = (S_C, P, \overset{must}{\longrightarrow}_C, \overset{may}{\longrightarrow}_C, L_C)$ be a (concrete) KMTS. Given a set S_A of abstract states and a total*[1] *abstraction relation on states $\rho \subseteq S_C \times S_A$, we define the (abstract) KMTS $M_A = (S_A, P, \overset{must}{\longrightarrow}_A, \overset{may}{\longrightarrow}_A, L_A)$ as follows:*

- $s_a \overset{must}{\longrightarrow}_A s'_a$ *if* $\forall s_c \in S_C : s_c \rho s_a \Rightarrow (\exists s'_c \in S_C : s'_c \rho s'_a \wedge s_c \overset{must}{\longrightarrow}_C s'_c)$*;*
- $s_a \overset{may}{\longrightarrow}_A s'_a$ *if* $\exists s_c, s'_c \in S_C : s_c \rho s_a \wedge s'_c \rho s'_a \wedge s_c \overset{may}{\longrightarrow}_C s'_c$*;*
- $L_A(s_a, p) = \begin{cases} \textit{true} \ \textit{if} \ \forall s_c : s_c \rho s_a \Rightarrow L_C(s_c, p) = \textit{true} \\ \textit{false} \ \textit{if} \ \forall s_c : s_c \rho s_a \Rightarrow L_C(s_c, p) = \textit{false} \\ \bot \quad \textit{otherwise} \end{cases}$

The previous definition can be used to build abstract KMTSs.

Theorem 2. *Given a KMTS M_C, any KMTS M_A obtained by applying Definition 4 is such that $M_A \preceq M_C$.*

Given a KMTS M_C, any abstraction M_A less complete than M_C with respect to the completeness preorder $\preceq$ can be constructed using Definition 4 by choosing the inverse of ρ as $\mathcal{B}$ [GHJ01]. When applied to MTSs with *may*-transitions only, the above definition coincides with traditional "conservative" abstraction. Building a 3-valued abstraction can be done using existing abstraction techniques at the same computational cost as building a conservative abstraction [GHJ01].

Since by construction $M_A \preceq M_C$, any temporal-logic formula ϕ that evaluates to *true* (resp. *false*) on M_A automatically holds (resp. does not hold) on M_C. It is shown in [BG00] that computing $[(M_A, s) \models \phi]$ can be reduced to two traditional (2-valued) model-checking problems on regular fully-defined systems (such as Kripke structures or Labeled Transition Systems), and hence that 3-valued model-checking for any temporal logic L has the same time and space complexity as 2-valued model checking for the logic L.

However, as argued in [BG00], the semantics of $[(M, s) \models \phi]$ returns $\bot$ more often than it should. Consider a KMTS M consisting of a single state s such that the value of proposition p at s is $\bot$ and the value of q at s is *true*. The formulas $p \vee \neg p$ and $q \wedge (p \vee \neg p)$ are $\bot$ at s, although in all complete Kripke structures more complete than (M, s) both formulas evaluate to *true*. This problem is not confined to formulas containing subformulas that are tautological or unsatisfiable. Consider a KMTS M' with two states s_0 and s_1 such that $p = q = \textit{true}$

[1] That is, $(\forall s_c \in S_C : \exists s_a \in S_A : s_c \rho s_a)$ and $(\forall s_a \in S_A : \exists s_c \in S_C : s_c \rho s_a)$.

Logic	**MC**	**SAT**	**GMC**
Propositional Logic	Linear	NP-complete	NP-complete
PML	Linear	PSPACE-complete	PSPACE-complete
CTL	Linear	EXPTIME-complete	EXPTIME-complete
μ-calculus	NP∩co-NP	EXPTIME-complete	EXPTIME-complete
LTL	PSPACE-complete	PSPACE-complete	EXPTIME-complete

Fig. 1. Known results on the complexity in the size of the formula for (2-valued and 3-valued) model checking (MC), satisfiability (SAT) and generalized model checking (GMC).

in s_0 and $p = q = \mathit{false}$ in s_1, and with a *may*-transition from s_0 to s_1. The formula $AXp \wedge \neg AXq$ (which is neither a tautology nor unsatisfiable) is $\perp$ at s_0, yet in all complete structures more complete than (M', s_0) the formula is *false*. This observation is used in [BG00] to define an alternative 3-valued semantics for temporal logics called the *thorough* semantics since it does more than the other semantics to discover whether enough information is present in a KMTS to give a definite answer. Let the *completions* $\mathcal{C}(M, s)$ of a state s of a KMTS M be the set of all states s' of complete Kripke structures M' such that $s \preceq s'$.

Definition 5. *Let ϕ be a formula of any two-valued logic for which a satisfaction relation $\models$ is defined on complete Kripke structures. The truth value of ϕ in a state s of a KMTS M under the* thorough *interpretation, written $[(M, s) \models \phi]_t$, is defined as follows:*

$$[(M,s) \models \phi]_t = \begin{cases} \mathit{true} \text{ if } (M', s') \models \phi \text{ for all } (M', s') \text{ in } \mathcal{C}(M,s) \\ \mathit{false} \text{ if } (M', s') \not\models \phi \text{ for all } (M', s') \text{ in } \mathcal{C}(M,s) \\ \perp \quad \text{otherwise} \end{cases}$$

It is easy to see that, by definition, we always have $[(M, s) \models \phi] \leq [(M, s) \models \phi]_t$. In general, interpreting a formula according to the thorough three-valued semantics is equivalent to solving two instances of the generalized model-checking problem [BG00].

Definition 6 (Generalized Model-Checking Problem). *Given a state s of a KMTS M and a formula ϕ of a temporal logic L, does there exist a state s' of a complete Kripke structure M' such that $s \preceq s'$ and $(M', s') \models \phi$?*

This problem is called *generalized model-checking* since it generalizes both model checking and satisfiability checking. At one extreme, where $M = (\{s_0\}, P, \xrightarrow{must} = \xrightarrow{may} = \{(s_0, s_0)\}, L)$ with $L(s_0, p) = \perp$ for all $p \in P$, all complete Kripke structures are more complete than M and the problem reduces to the satisfiability problem. At the other extreme, where M is complete, only a single structure needs to be checked and the problem reduces to model checking.

Algorithms and complexity bounds for the generalized model-checking problem for various temporal logics were presented in [BG00]. In the case of branching-time temporal logics, generalized model checking has the same complexity in the size of the formula as satisfiability. In the case of linear-time

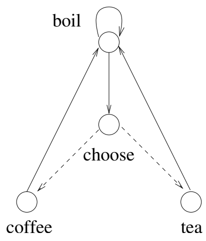


Fig. 2. Example of open system

temporal logic, generalized model checking is EXPTIME-complete in the size of the formula, i.e., harder than both satisfiability and model checking, which are both PSPACE-complete in the size of the formula for LTL. Figure 1 summarizes the complexity results of [BG00]. These results show that the complexity in the size of the formula of computing $[(M, s) \models \phi]_t$ (GMC) is always higher than that of computing $[(M, s) \models \phi]$ (MC).

Regarding the complexity in the size of the model $|M|$, it is shown in [GJ02] that generalized model checking can in general require quadratic running time in $|M|$, but that the problem can be solved in time linear in $|M|$ in the case of (LTL or BTL) *persistence* properties, i.e., properties recognizable by (word or tree) automata with a co-Büchi acceptance condition. Persistence properties include several important classes of properties of practical interest, such as all safety properties.

2.2 Module Checking

The framework described in the previous section assumes that the abstract system M is *closed*, i.e., that its behavior is completely determined by the state of the system. But what if M is an abstraction of an *open* system?

An *open system* is a system that interacts with its environment and whose behavior depends on this interaction. Such systems are also often called *reactive*. In [KV96], Kupferman and Vardi argued that verifying temporal properties (especially those specified in branching temporal logics) of open systems must be handled differently than traditional model checking. Their argument is best illustrated with a simple example. Consider a vending machine M which repeatedly boils water, asks the environment to choose between coffee and tea, and deterministically serves a drink according to the external choice. This machine is depicted in Figure 2, where dotted transitions represent transitions taken by the environment. The machine has four states: *boil, choose, coffee* and *tea.* When M is in state *boil*, we know exactly what its possible next states are (namely, *choose* and *boil*). In contrast, when M is in state *choose*, the set of possible next states is unknown since it depends on the environment: it could be any subset of the set $\{$*coffee, tea*$\}$. To see the difference this makes semantically, consider the property "is it always possible for M to eventually serve tea?", which can be represented as the CTL formula[2] $AGEF tea$. If we evaluate this formula on

[2] See [Eme90] for a general introduction to the temporal logics used in this paper.

M viewed as a traditional Kripke structure, thus ignoring whether transitions are taken by the system or its environment, we obtain $[M \models AGEF tea] = true$, since, for every state of M, there exists a path from that state to state *tea*. In contrast, if we take into account transitions taken by the environment, the answer should be *false* since, if the environment decides to always choose *coffee*, the machine M will never serve tea. This observation prompted the introduction of a variant of the model-checking problem for reasoning about open systems, called the *module checking* problem:

> Given a module (M, s) and a temporal logic formula ϕ, does (M, s) satisfy ϕ in all possible environments?

Formally, a *module* is defined in [KV96] as a Kripke structure whose set of states is partitioned into two sets: a set of system states, representing the states where the system can make transitions (such as the states *boil, coffee* and *tea* in the example above), and a second set of environment states, where the environment can make transitions (such as the state *choose* in the previous example). Here, we will use an alternative formalization (already suggested in [KV96]) and represent modules as Modal Transition Systems instead: *must*-transitions will model system transitions while *may*-transitions will model environment transitions. Note that this second formalization is more general since it allows states from which both system and environment transitions exist. To simplify notations, we also use a set of atomic propositions and associate with each state a labeling from every propositions to the set $\{true, false\}$.

Definition 7 (Module). *A* module *is a tuple* $(S, P, \xrightarrow{must}, \xrightarrow{may}, L)$*, where* S *is a nonempty finite set of states,* P *is a finite set of atomic propositions,* $\xrightarrow{may} \subseteq S \times S$ *and* $\xrightarrow{must} \subseteq S \times S$ *are transition relations such that* $\xrightarrow{must} \subseteq \xrightarrow{may}$*, and* $L : S \times P \to \{true, false\}$ *is an* interpretation *that associates a truth value in* $\{true, false\}$ *with each atomic proposition in* P *for each state in* S*.*

A module is thus modeled as a KMTS where no proposition takes the value $\perp$. In this context, module checking can then be formally defined as follows. Recall that $\mathcal{C}(M, s)$ denotes the set of completions of a KMTS/module M (see previous section): $\mathcal{C}(M, s) = \{(M', s') | s \preceq s'$ and M' is a KS$\}$.

Definition 8 (Module Checking). *Given a module* (M, s) *and a temporal logic formula* ϕ*, we say that the module* (M, s) *satisfies* ϕ*, denoted* $(M, s) \models_r \phi$*, if* $\forall (M', s') \in \mathcal{C}(M, s) : (M', s') \models \phi$*. Checking whether* $(M, s) \models_r \phi$ *is called the* module-checking problem*.*

Formalized this way[3], it is clear that module checking (ModC) is related to generalized model checking (GMC). In the next section, we study this relationship.

[3] The definition of module checking in [KV96] actually requires the transition relation of any (M', s') in $\mathcal{C}(M, s)$ to be total, which has the effect of preventing the environment from blocking the system by refusing to execute any transition; this assumption is eliminated here to simplify the presentation.

3 Comparing Module Checking and GMC

We first consider the case of properties specified in linear temporal logic (LTL) or in universal branching temporal logics such as $\forall CTL$ and $\forall CTL^*$. In this case, [KV96] shows that module checking and model checking coincide. Indeed, by definition, an LTL or universal property holds of a model if and only if all paths in that model satisfy the given (path) property. In the case of a module, module checking can be reduced to checking whether the property holds of the module when placed in the maximal environment consisting of all possible environment transitions, which in turns is equivalent to model checking. This implies that LTL module checking has the same complexity as LTL model checking: it is PSPACE-complete in the size of the formula and can be done in linear time in the size of the model (it is also known to be NLOGSPACE-complete in $|M|$). In contrast, it is shown in [BG00,GJ02] that GMC for LTL can be more precise but also more expensive than LTL model checking: GMC for LTL is EXPTIME-complete in the size of the formula [BG00] and can be done in quadratic time in the size of the model [GJ02].

In the case of properties specified in branching temporal logics (BTL), thus including existential quantification, [KV96] shows that the complexity of module checking is higher than that of model checking, and is in fact as hard as satisfiability.

The following theorem states that, in the BTL case, GMC and module checking are interreducible.

Theorem 3. *For any branching temporal logic L, the GMC problem and the module checking problem for L are interreducible in linear time and logarithmic space.*

Proof. Consider a formula ϕ of the logic L and a module (M, s) represented by a KMTS as defined in Section 2.2. We have $(M, s) \models_r \phi$ iff $\forall (M', s') \in \mathcal{C}(M, s) : (M', s') \models \phi$ (by Definition 8) iff $\neg\exists (M', s') \in \mathcal{C}(M, s) : (M', s') \not\models \phi$. For any branching temporal logic, the latter is equivalent to $\neg GMC((M, s), \neg\phi)$ (by Definition 6). Thus, we have $(M, s) \models_r \phi$ iff $GMC((M, s), \neg\phi)$ does not hold. Conversely, $GMC((M, s), \phi)$ holds iff $(M, s) \not\models_r \neg\phi$.

In the BTL setting, generalized model checking and module checking are thus very closely related. One could even say that they are the "*dual*" of each other, in the same sense as the quantifier $\forall$ is the dual of $\exists$ since $\forall = \neg\exists\neg$. The previous theorem also explains why both problems have the same complexity in the case of BTL. For instance, [KV96] pointed out that the complexity in the size of the formula of module checking for a BTL L is the same as that of satisfiability for L, while a similar result was proved independently for GMC in [BG00].

This close correspondence also makes it possible to transfer unmatched results obtained for one problem to the other. For instance, [KV96] only shows that module checking for CTL can be done in PTIME in the size of M. Using the results of [GJ02] concerning the complexity of GMC in the size of M, we immediately obtain that module checking for CTL can require quadratic running time in $|M|$, but that it can be solved in time linear in $|M|$ in the case of

CTL *persistence* properties, i.e., properties recognizable by tree automata with a co-Büchi acceptance condition. Conversely, [GJ02] does not provide a lower bound on the complexity of GMC in the size of M for BTL persistence properties. Using the proof of Theorem 2 in [KV96] which states that the program complexity of CTL and CTL* module checking is PTIME-complete, we obtain that GMC for BTL persistence properties (and hence CTL and CTL* in general) is PTIME-hard. (Moreover, it is easy to show that this proof carries over to GMC for LTL, which is thus also PTIME-hard.)

In summary, generalized model checking and module checking are different, yet related, problems. The former is a framework for reasoning about partially-specified, i.e., abstract, systems, while the latter is a framework for reasoning about open systems. It is then natural to ask whether these two techniques could be combined into a framework for reasoning about abstract open systems. The rest of this paper investigates this idea.

4 Modeling and Reasoning about Abstract Open Systems

We now discuss how to model abstract open systems. Our goal is to define a modeling formalism for representing abstractions of programs implementing open systems. Such abstractions could then be automatically generated from source code by static analysis tools using abstraction techniques like predicate abstraction. We thus focus here on a semantic model to represent abstract open systems, not on a modeling language (with a specific syntax).

Combining the ideas of Section 2, an abstract open system can simply be represented as a KMTS with two distinct types of "unknowns": the third truth value $\perp$ can model loss of information due to abstraction, while *may*-transitions unmatched by *must*-transitions can model uncertainty due to environment transitions. Formally, an *abstract module*, or *3-valued module*, can be defined as follows.

Definition 9 (Abstract Module). *An* abstract module *is a KMTS, i.e., a tuple* $(S, P, \xrightarrow{must}, \xrightarrow{may}, L)$ *where* S *is a nonempty finite set of states,* P *is a finite set of atomic propositions,* $\xrightarrow{may} \subseteq S \times S$ *and* $\xrightarrow{must} \subseteq S \times S$ *are transition relations such that* $\xrightarrow{must} \subseteq \xrightarrow{may}$*, and* $L : S \times P \rightarrow \{true, \perp, false\}$ *is an* interpretation *that associates a truth value in* $\{true, \perp, false\}$ *with each atomic proposition in* P *for each state in* S*. A module is an abstract module such that* $\forall s \in S : \forall p \in P : L(s, p) \neq \perp$*. An abstract model is an abstract module such that* $\xrightarrow{must} = \xrightarrow{may}$*.*

It is worth emphasizing that modeling abstract open systems this way does not restrict expressiveness of either kind of "unknowns" since KMTSs have the same expressiveness as PKSs and MTSs [GJ03]. In other words, any modeling of incomplete information using $\perp$ can also be done using *may*-transitions instead, and vice versa. This implies that it does not matter which of $\perp$ or *may*-transitions are used to model abstraction or the environment. What matters is that the two sources of incomplete information are modeled differently, so that they can be distinguished in the model. Indeed, these two types of partial information are treated differently, as we will see later.

We now turn to the definition of *abstract module checking*, or *3-valued module checking*. For doing so, we first need to define the set of possible environments in which an abstract module can be executed. We formally represent this set as the set of completions of the abstract module with respect to a preorder $\preceq_{MTS}$, which we define as follows.

Definition 10 ($\preceq_{MTS}$). *Let* $M_A = (S_A, P, \xrightarrow{must}_A, \xrightarrow{may}_A, L_A)$ *and* $M_C = (S_C, P, \xrightarrow{must}_C, \xrightarrow{may}_C, L_C)$ *be KMTSs. The preorder* $\preceq_{MTS}$ *is the greatest relation* $\mathcal{B} \subseteq S_A \times S_C$ *such that* $(s_a, s_c) \in \mathcal{B}$ *implies the following:*

- $\forall p \in P : L_A(s_a, p) = L_C(s_c, p)$,
- *if* $s_a \xrightarrow{must}_A s'_a$, *there is some* $s'_c \in S_C$ *such that* $s_c \xrightarrow{must}_C s'_c$ *and* $(s'_a, s'_c) \in \mathcal{B}$,
- *if* $s_c \xrightarrow{may}_C s'_c$, *there is some* $s'_a \in S_A$ *such that* $s_a \xrightarrow{may}_A s'_a$ *and* $(s'_a, s'_c) \in \mathcal{B}$.

The preorder $\preceq_{MTS}$ is similar to the completeness preorder on MTSs [LT88], but is defined on KMTSs and leaves truth values of atomic propositions unchanged. Definition 10 is also similar to Definition 3 defining $\preceq$ on KMTSs except for the first condition which prevents the refinement of unknown truth values. Let $\mathcal{C}_{MTS}((M, s)) = \{(M', s') | s \preceq_{MTS} s' \text{ and } M' \text{ is a PKS}\}$ denote the set of completions of a module (M, s) with respect to $\preceq_{MTS}$. We can now define abstract module checking as follows.

Definition 11 (Abstract Module Checking). *Given an abstract module* (M, s) *and a temporal logic formula* ϕ, *computing the value*

$$[(M, s) \models_r \phi] = \begin{cases} \mathit{true} \ \ \mathit{if}\ \forall (M', s') \in \mathcal{C}_{MTS}(M, s) : [(M', s') \models \phi] = \mathit{true} \\ \mathit{false}\ \mathit{if}\ \exists (M', s') \in \mathcal{C}_{MTS}(M, s) : [(M', s') \models \phi] = \mathit{false} \\ \perp \quad \mathit{otherwise} \end{cases}$$

is defined as the abstract module checking problem.

The previous definition generalizes the definition of module checking: when (M, s) is a concrete module (i.e., a module where no atomic proposition has the value $\perp$ in any state), Definition 11 coincides with Definition 8 defining module checking.

Abstract module checking defines a new 3-valued logic for reasoning about abstract modules: its syntax is as usual and its semantics is defined by $[(M, s) \models_r \phi]$. The following preorder $\preceq_{PKS}$ on KMTSs measures the degree of completeness of abstract modules with respect to the new semantics derived from abstract module checking.

Definition 12 ($\preceq_{PKS}$). *Let* $M_A = (S_A, P, \xrightarrow{must}_A, \xrightarrow{may}_A, L_A)$ *and* $M_C = (S_C, P, \xrightarrow{must}_C, \xrightarrow{may}_C, L_C)$ *be KMTSs. The preorder* $\preceq_{PKS}$ *is the greatest relation* $\mathcal{B} \subseteq S_A \times S_C$ *such that* $(s_a, s_c) \in \mathcal{B}$ *implies the following:*

1. $\forall p \in P : L_A(s_a, p) \leq L_C(s_c, p)$,

2. *if* $s_a \xrightarrow{may}_A s'_a$, *there is some* $s'_c \in S_C$ *such that* $s_c \xrightarrow{may}_C s'_c$ *and* $(s'_a, s'_c) \in \mathcal{B}$,
3. *if* $s_a \xrightarrow{must}_A s'_a$, *there is some* $s'_c \in S_C$ *such that* $s_c \xrightarrow{must}_C s'_c$ *and* $(s'_a, s'_c) \in \mathcal{B}$,
4. *if* $s_c \xrightarrow{may}_C s'_c$, *there is some* $s'_a \in S_A$ *such that* $s_a \xrightarrow{may}_A s'_a$ *and* $(s'_a, s'_c) \in \mathcal{B}$,
5. *if* $s_c \xrightarrow{must}_C s'_c$, *there is some* $s'_a \in S_A$ *such that* $s_a \xrightarrow{must}_A s'_a$ *and* $(s'_a, s'_c) \in \mathcal{B}$.

The preorder $\preceq_{PKS}$ is similar to the completeness preorder on PKSs [BG99], but is defined on KMTSs and requires a bisimulation-like relation on both *may* and *must* transitions. The above definition extends Definition 3 defining $\preceq$ on KMTSs by also requiring conditions (2) and (5). An important property of $\preceq_{PKS}$ is the following.

Lemma 1. *Let* $M_A = (S_A, P, \xrightarrow{must}_A, \xrightarrow{may}_A, L_A)$ *and* $M_C = (S_C, P, \xrightarrow{must}_C, \xrightarrow{may}_C, L_C)$ *be KMTSs. Given any two states* $s_a \in S_A$ *and* $s_c \in S_C$, $s_a \preceq_{PKS} s_c$ *implies the two following properties:*

- $\forall (M'_A, s'_a) \in \mathcal{C}_{MTS}(M_A, s_a) : \exists (M'_C, s'_c) \in \mathcal{C}_{MTS}(M_C, s_c) : s'_a \preceq_{PKS} s'_c$, *and*
- $\forall (M'_C, s'_c) \in \mathcal{C}_{MTS}(M_C, s_c) : \exists (M'_A, s'_a) \in \mathcal{C}_{MTS}(M_A, s_a) : s'_a \preceq_{PKS} s'_c$.

Intuitively, the previous lemma states that, if $s_a \preceq_{PKS} s_c$, then the set of "possible environments" (i.e., $\mathcal{C}_{MTS}$) for s_a and s_c are equivalent: any environment of s_a is a possible environment of s_c and vice versa. This lemma is useful to prove our next theorem.

Theorem 4. *Let* $M_A = (S_A, P, \xrightarrow{must}_A, \xrightarrow{may}_A, L_A)$ *and* $M_C = (S_C, P, \xrightarrow{must}_C, \xrightarrow{may}_C, L_C)$ *be two abstract modules (KMTSs) such that* $s_a \in S_A$ *and* $s_c \in S_C$, *and let* Φ *be the set of all formulas of 3-valued PML. Then,*

$$s_a \preceq_{PKS} s_c \textit{ iff } (\forall \phi \in \Phi : [(M_A, s_a) \models_r \phi] \leq [(M_C, s_c) \models_r \phi]).$$

Proof. Omitted due to space limitations.

The previous theorem thus states that 3-valued PML defined with the semantics $[(M,s) \models_r \phi]$ logically characterizes the preorder $\preceq_{PKS}$.[4] This implies that abstract module checking cannot distinguish abstract modules that are equivalent with respect to the preorder $\preceq_{PKS}$. Another important corollary of this theorem is that, by generating an abstraction (M_A, s_a) from (a static analysis of) a concrete module (M_C, s_c) such that $s_a \preceq_{PKS} s_c$, we can then *both prove and disprove arbitrary properties* of s_c by doing module checking of its abstraction s_a. How to automatically generate abstractions preserving $\preceq_{PKS}$ is discussed with an example later in Section 6.

In the case of LTL, abstract (3-valued) module checking, i.e., computing $[(M,s) \models_r \phi]$, reduces to abstract (3-valued) model checking, i.e., computing $[(M,s) \models \phi]$ with M viewed as a PKS where all the *may*-transitions are also *must*-transitions. In the BTL case, 3-valued module checking can be *approximated* by 3-valued model checking using the 3-valued semantics on KMTS

[4] As in the 2-valued case, this result also holds for the propositional μ-calculus.

defined in Definition 2. Indeed, $[(M, s) \models \phi] = true$ (respectively *false*) implies that $\forall(M', s') \in \mathcal{C}(M, s) : (M', s') \models \phi$ (resp., $(M', s') \not\models \phi$), which in turn implies that $[(M, s) \models_r \phi] = true$ (resp., *false*). We thus always have $[(M, s) \models \phi] \leq [(M, s) \models_r \phi]$. Since 3-valued model checking can be done at the same cost as traditional 2-valued model checking [BG00], computing $[(M, s) \models \phi]$ is less computationally expensive than computing $[(M, s) \models_r \phi]$ in the BTL case (see Theorem 3), and can thus be used as a cheaper but less precise partial algorithm for testing whether $[(M, s) \models_r \phi]$ is *true* or *false* in that case.

5 Generalized Module Checking

As in the case of 3-valued model checking, precision of 3-valued module checking can be improved by defining a 3-valued *thorough semantics*, denoted $[(M, s) \models_r \phi]_t$. Let $\mathcal{C}_{PKS}((M, s)) = \{(M', s') | s \preceq_{PKS} s' \text{ and } M' \text{ is a module }\}$ denote the set of possible (concrete) modules for an abstract module (M, s).

Definition 13 (Thorough Abstract Module Checking). *Given an abstract module (M, s) and a temporal logic formula ϕ, computing the value*

$$[(M, s) \models_r \phi]_t = \begin{cases} true \;\; if\; (M', s') \models_r \phi \; for\; all\; (M', s') \; in\; \mathcal{C}_{PKS}(M, s) \\ false \; if\; (M', s') \not\models_r \phi \; for\; all\; (M', s') \; in\; \mathcal{C}_{PKS}(M, s) \\ \perp \quad\;\; otherwise \end{cases}$$

is defined as the thorough abstract module checking problem.

The next theorem states that thorough abstract module checking is consistent with abstract module checking.

Theorem 5. *For any abstract module (M, s) and temporal logic formula ϕ, we have $[(M, s) \models_r \phi] \leq [(M, s) \models_r \phi]_t$.*

Proof. By definition, $[(M, s) \models_r \phi] = true$ implies $[(M, s) \models_r \phi]_t = true$. Moreover, $[(M, s) \models_r \phi] = false$ implies $[(M, s) \models_r \phi]_t = false$ since $\exists(M', s') \in \mathcal{C}_{MTS}(M, s) : \forall(M'', s'') \in \mathcal{C}_{PKS}(M', s') : (M'', s') \not\models \phi$ implies $\forall(M', s') \in \mathcal{C}_{PKS}(M, s) : \exists(M'', s'') \in \mathcal{C}_{MTS}(M', s') : (M'', s') \not\models \phi$.

Checking whether $[(M, s) \models_r \phi]_t = true$ can be reduced to solving an instance of the generalized model checking problem for (M, s) and $\neg\phi$ with respect to the completeness preorder $\preceq$ on KMTSs, because of the two universal quantifications defining $[(M, s) \models_r \phi]_t = true$. In contrast, checking whether $[(M, s) \models_r \phi]_t = false$ involves an alternation between $\exists$ and $\forall$, and requires solving an instance of the following problem, which we call *generalized module checking.*

Definition 14 (Generalized Module-Checking Problem). *Given a state s of an abstract module M and a formula ϕ of a temporal logic L, does there exist a state s' of a module M' such that $s \preceq_{PKS} s'$ and $(M', s') \models_r \phi$?*

Clearly, generalized module checking (GModC) generalizes both module checking and generalized model checking for any (i.e., BTL or LTL) temporal logic since it includes both as particular sub-problems. Hence, generalized module checking is at least as hard as generalized model checking, which is itself harder than module checking in the LTL case and as hard as module checking in the BTL case (see Section 3). Is GModC harder than GMC? The next theorem shows that this is not the case by providing a reduction from GModC to GMC.

Theorem 6. *Given any (LTL or BTL) temporal logic L, any instance of the generalized module checking problem for L can be reduced in linear time and logarithmic space to an instance of the generalized model checking problem for L.*

Proof. (Sketch) Consider a formula ϕ of the logic L and an abstract module (M, s) represented by a KMTS $M = (S, P, \xrightarrow{must}, \xrightarrow{may}, L)$ as defined in Section 4. Without loss of generality, let us assume that ϕ is in positive form in which negation can apply only to atomic propositions. We first define a PKS $M' = (S', P', \to', L')$ that simulates exactly the *may* and *must* transitions of M with an additional proposition p_{must} as follows: $S' = S \times \{must, may\}$, $P' = P \cup \{p_{must}\}$, $\to' = \{((s,x),(s',x')) | (s,s') \in \xrightarrow{may}, x' = must$ if $(s,s') \in \xrightarrow{must}$ or $x' = may$ otherwise$\}$, $\forall (s,x) \in S' : \forall p \in P : L'((s,x),p) = L(s,p)$, and $L'((s,x),p_{must}) = true$ if $x = must$ or $L'((s,x),p_{must}) = false$ otherwise. Second, we translate the formula ϕ into the formula $T(\phi)$ defined with a set $P' = P \cup \{p_{must}\}$ of atomic propositions by applying recursively the following rewrite rules: for all $p \in P$, $T(p) = p$ and $T(\neg p) = \neg p$; $T(\phi_1 \wedge \phi_2) = T(\phi_1) \wedge T(\phi_2)$; $T(\phi_1 \vee \phi_2) = T(\phi_1) \vee T(\phi_2)$; $T(AX\phi) = AX(T(\phi))$; $T(EX\phi) = EX(p_{must} \wedge T(\phi))$. (Note that the above rules are for PML; in the case of the mu-calculus, fixpoint operators are left unchanged; a similar translation exists for LTL). Third, we show that $GModC((M,s),\phi) = true$ iff $GMC((M',(s,must)),T(\phi)) = true$.

To summarize, GModC has the same complexity as GMC. This implies the following, maybe surprising, corollary: even though computing $[(M,s) \models_r \phi]_t$ (thorough abstract module checking) can be more precise than computing $[(M,s) \models_r \phi]$ (abstract module checking), it has the same complexity in the BTL case.

6 Application

The practical motivation for this paper is to provide a framework for verifying properties of programs implementing open systems using automatic abstraction techniques such as predicate abstraction.

An example of program implementing an open system is shown on the left of Figure 3. In this program, x and y denote variables controlled by the program (system), f denotes some unknown function of the system, while z denotes a variable controlled by the environment. The notation "$x, y = 1, 0$" means that variables x and y are simultaneously assigned to values 1 and 0, respectively.

Imagine we are interested in checking the property "(eventually y is odd) and (at any time, x is odd or y is even)". From this property, we define two predicates p : "is x odd?" and q : "is y odd?". Given these two predicates, the property can be represented by the LTL formula $\phi = \Diamond q \wedge \Box(p \vee \neg q)$.

```
program C() {
  x,y= 1,0;
  if (read(z) == 0) then {
    x,y = 2*f(x),f(y);
    x,y = 1,0;
  } else
    x,y = f(x),f(y);
}
```

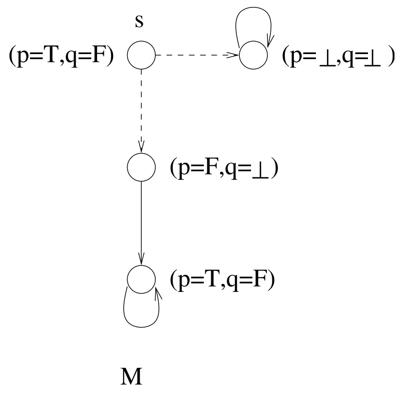


Fig. 3. Example of open program and abstract module

Using predicate abstraction techniques and predicates p and q, one can automatically compute an abstract module for this program that satisfies the preorder $\preceq_{PKS}$ and thus Theorem 4, by construction. An example of such an abstract module (M, s) for this program is shown on the right of Figure 3. The truth values of atomic propositions p and q is defined in each state as indicated in the figure. Dotted transitions indicate *may*-transitions unmatched by *must*-transitions and represent transitions controlled by the environment. Note how the unknown function f is modeled using $\perp$, while uncertainty due to the environment is modeled using *may*-transitions.

In this example, we have $[(M, s) \models_r \phi] =\perp$, while $[(M, s) \models_r \phi]_t = \mathit{false}$. In other words, using the thorough interpretation and generalized module checking is needed to obtain a definite answer in this case. Indeed, this result is obtained by proving that, for all possible completions (M', s') such that $(M, s) \preceq_{PKS} (M', s')$, there exists an environment of (M', s') (namely the one which forces the system down the leftmost path) where ϕ is violated. We are not aware of any decision procedure more efficient than generalized module checking to prove automatically that the open program C of Figure 3 violates property ϕ.[5]

Finally note how GModC differ from GMC in the presence of *may*-transitions: while $GModC((M, s), \phi) = \mathit{false}$, we have $GMC((M, s), \phi) =\perp$.

7 Conclusions

We have presented a framework for representing and reasoning about abstract open systems. Our framework is designed to be used in conjunction with automatic abstraction tools for generating abstractions from static program analysis. We identified the preorder $\preceq_{PKS}$ as the one being logically characterized by the

[5] Note that, since GMC for LTL can be reduced to SAT for CTL* (using Theorem 23 of [BG00]), the above verification results could be obtained using a SAT solver for CTL*, but at a much higher complexity both in $|M|$ and $|\phi|$.

Logic	**MC**	**ModC**	**GModC**
Propositional Logic	Linear	Linear	NP-complete
PML	Linear	PSPACE-complete	PSPACE-complete
CTL	Linear	EXPTIME-complete	EXPTIME-complete
μ-calculus	NP$\cap$co-NP	EXPTIME-complete	EXPTIME-complete
LTL	PSPACE-complete	PSPACE-complete	EXPTIME-complete

Fig. 4. Complexity in the size of the formula of model checking (MC), module checking (ModC), and generalized module checking (GModC).

3-valued semantics derived from the definition of module checking of [KV96]. Any abstraction that preserves $\preceq_{PKS}$ can then be used to both prove and disprove arbitrary temporal properties of the concrete program. We introduced new variants of the module checking problem suitable for reasoning about such abstractions, namely abstract, thorough abstract and generalized module checking. We also studied the complexity of these problems. The complexity of generalized module checking is summarized in the last column of Figure 4. The precision of generalized module checking was illustrated with an example of program and property that is beyond the scope of current abstraction-based verification tools.

Note that generating an abstraction (M_A, s_a) preserving $\preceq_{PKS}$ assumes that it is possible to determine which transitions of the concrete module (M_C, s_c) are controlled by the system and which transitions are controlled by the environment. Our framework does not currently support a way to safely approximate (M_C, s_c) in the case this information is unknown (i.e., cannot be determined exactly by a static analysis). Previous work on alternating refinement relations [AHKV98,AH01] provides a way to conservatively abstract a game (like the one played between a system and its environment) while preserving the existence of a winning strategy for one of the players. However, such game abstractions do not preserve the existence of a winning strategy for the other player: they are *conservative* in the same sense as a simulation relation defines a conservative abstraction of a system, which can be used for proving universal properties but not refuting these. An interesting topic for future work is therefore to study how to combine our framework with techniques for abstracting games with the goal of designing "3-valued game abstractions" that would preserve winning strategies for both players while allowing abstraction. Such a way, one could extend the framework developed in this paper to allow sound approximations of the partitioning between system and environment transitions while preserving the ability to prove and disprove any temporal property of the interaction of the system with its environment.

References

[AH01] L. de Alfaro and T. Henzinger. Interface Automata. In *Proceedings of the 9th ACM Symposium on the Foundations of Software Engineering (FSE'01)*, 2001.

[AHKV98] R. Alur, T. Henzinger, O. Kupferman, and M. Vardi. Alternating Refinement Relations. In *Proc. 10th Conference on Concurrency Theory*, volume 1466 of *Lecture Notes in Computer Science*, pages 163–178. Springer-Verlag, 1998.

[BG99] G. Bruns and P. Godefroid. Model Checking Partial State Spaces with 3-Valued Temporal Logics. In *Proceedings of the 11th Conference on Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 274–287, Trento, July 1999. Springer-Verlag.

[BG00] G. Bruns and P. Godefroid. Generalized Model Checking: Reasoning about Partial State Spaces. In *Proceedings of CONCUR'2000 (11th International Conference on Concurrency Theory)*, volume 1877 of *Lecture Notes in Computer Science*, pages 168–182, University Park, August 2000. Springer-Verlag.

[BR01] T. Ball and S. Rajamani. The SLAM Toolkit. In *Proceedings of CAV'2001 (13th Conference on Computer Aided Verification)*, volume 2102 of *Lecture Notes in Computer Science*, pages 260–264, Paris, July 2001. Springer-Verlag.

[CDH$^+$00] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng. Bandera: Extracting Finite-State Models from Java Source Code. In *Proceedings of the 22nd International Conference on Software Engineering*, 2000.

[Dam96] D. Dams. *Abstract interpretation and partition refinement for model checking.* PhD thesis, Technische Universiteit Eindhoven, The Netherlands, 1996.

[DD01] S. Das and D. L. Dill. Successive Approximation of Abstract Transition Relations. In *Proceedings of LICS'2001 (16th IEEE Symposium on Logic in Computer Science)*, pages 51–58, Boston, June 2001.

[Eme90] E. A. Emerson. Temporal and Modal Logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science.* Elsevier/MIT Press, Amsterdam/Cambridge, 1990.

[GHJ01] P. Godefroid, M. Huth, and R. Jagadeesan. Abstraction-based Model Checking using Modal Transition Systems. In *Proceedings of CONCUR'2001 (12th International Conference on Concurrency Theory)*, volume 2154 of *Lecture Notes in Computer Science*, pages 426–440, Aalborg, August 2001. Springer-Verlag.

[GJ02] P. Godefroid and R. Jagadeesan. Automatic Abstraction Using Generalized Model Checking. In *Proceedings of CAV'2002 (14th Conference on Computer Aided Verification)*, volume 2404 of *Lecture Notes in Computer Science*, pages 137–150, Copenhagen, July 2002. Springer-Verlag.

[GJ03] P. Godefroid and R. Jagadeesan. On the Expressiveness of 3-Valued Models. In *Proceedings of VMCAI'2003 (4th Conference on Verification, Model Checking and Abstract Interpretation)*, volume 2575 of *Lecture Notes in Computer Science*, pages 206–222, New York, January 2003. Springer-Verlag.

[GS97] S. Graf and H. Saidi. Construction of Abstract State Graphs with PVS. In *Proceedings of the 9th International Conference on Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83, Haifa, June 1997. Springer-Verlag.

[HJMS02] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages*, pages 58–70, Portland, January 2002.

[HJS01] M. Huth, R. Jagadeesan, and D. Schmidt. Modal Transition Systems: a Foundation for Three-Valued Program Analysis. In *Proceedings of the European Symposium on Programming (ESOP'2001)*, volume 2028 of *Lecture Notes in Computer Science*. Springer-Verlag, April 2001.

[Kle87] S. C. Kleene. *Introduction to Metamathematics.* North Holland, 1987.

[Koz83] D. Kozen. Results on the Propositional Mu-Calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[KV96] O. Kupferman and M. Vardi. Module Checking. In *Proc. 8th Conference on Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 75–86, New Brunswick, August 1996. Springer-Verlag.

[LT88] K. G. Larsen and B. Thomsen. A Modal Process Logic. In *Proceedings of Third Annual Symposium on Logic in Computer Science*, pages 203–210. IEEE Computer Society Press, 1988.

[VHBP00] W. Visser, K. Havelund, G. Brat, and S. Park. Model Checking Programs. In *Proceedings of ASE'2000 (15th International Conference on Automated Software Engineering)*, Grenoble, September 2000.

Schedule-Carrying Code[*]

Thomas A. Henzinger, Christoph M. Kirsch, and Slobodan Matic

EECS, University of California, Berkeley

Abstract. We introduce the paradigm of *schedule-carrying code* (SCC). A hard real-time program can be executed on a given platform only if there exists a feasible schedule for the real-time tasks of the program. Traditionally, a *scheduler* determines the existence of a feasible schedule according to some scheduling strategy. With SCC, a *compiler* proves the existence of a feasible schedule by generating executable code that is attached to the program and represents its schedule. An SCC executable is a real-time program that carries its schedule as code, which is produced once and can be revalidated and executed with each use. We evaluate SCC both in theory and practice. In theory, we give two scenarios, of nonpreemptive and distributed scheduling for Giotto programs, where the generation of a feasible schedule is hard, while the validation of scheduling instructions that are attached to the programs is easy. In practice, we implement SCC and show that explicit scheduling instructions can reduce the scheduling overhead up to 35% and can provide an efficient, flexible, and verifiable means for compiling Giotto programs on complex architectures, such as the TTA.

1 Introduction

Giotto is a high-level programming language for hard real-time applications [4]. A Giotto program consists of a collection of modes, each specifying the release times and deadlines of a set of periodic tasks. In Giotto, the semantics of value propagation between tasks is defined independent of the system scheduler, and therefore deterministic: as long as the scheduler maintains all deadlines, the outputs of all task invocations are determined by the sensor inputs, and do not depend on the task ordering, distribution, or preemption mechanism of a particular RTOS. A Giotto program is executable only if there exists a schedule that meets all deadlines on a given hardware platform, with given resources and performance. An executable combination of Giotto program and platform data (primarily worst-case execution times for all tasks) is called *time-safe* [5]. The Giotto compiler must, in addition to generating code, prove time safety [6]. The proof of time safety establishes the existence of a feasible schedule, and in doing so, produces the schedule. We introduce the idea of *schedule-carrying code* (SCC): once a feasible schedule has been produced, it can be attached to the code

[*] This work was supported by the AFOSR MURI grant F49620-00-1-0327, the California MICRO grant 01-037, the DARPA grant F33615-C-98-3614, the MARCO grant 98-DT-660, and the NSF grants CCR-0208875, CCR-0085949, and CCR-0225610.

to serve as a witness for time safety, in case the code source is untrusted, or if the code is moved to multiple targets. Thus, SCC is the paradigm of proof-carrying code [10] extended from traditional safety properties, such as memory safety, to real-time properties. SCC, however, offers an even more important capability than the reuse of time-safety proofs. If the schedule itself is provided in the form of executable instructions, properly attached to the code generated from Giotto, then SCC renders the system scheduler of the target platform obsolete. This leads to dramatic performance improvements in executing Giotto programs [7].

Let us be more precise. We define two virtual execution engines called the *E(mbedded) machine*, and the *S(cheduling) machine*. The E machine executes E code generated from a Giotto program [5]. E code is reactive code: it manages the release times and deadlines of software tasks in reaction to environment events, such as clock ticks. When an environment interrupt occurs, E code may call a driver that reads a sensor port, or writes an actuator port, or transfers a value between ports, and it may release a task to the system. If an RTOS is used, the released task enters the ready queue, and can be dispatched by the system scheduler. A Giotto compiler that proves time safety is a *schedule-generating compiler*: it generates, in addition to E code, also S code, which represents a feasible schedule for the generated E code. S code is executed by the S machine, which replaces the system scheduler. S code is proactive code: it manages the execution of released tasks on the available CPUs. S code may dispatch a task for a certain amount of time (time-slice preemptive scheduling), or until another task is released (priority preemptive), or until the task completes (nonpreemptive). In other words, the S language is an expressive, executable schedule description language. Our implementation executes intertwined E and S code produced by the Giotto compiler, and thus provides the kernel functionality of an RTOS.

The usefulness of SCC rests on two premises. First, the execution of S code is more efficient, and at least as flexible, as the use of the scheduler provided by an RTOS. This is substantiated by our experiments, where we achieve up to 35% overhead reduction and demonstrate the ability to change scheduling strategies when switching Giotto modes [7]. Second, it is often more efficient to check the feasibility of a schedule than to generate the schedule ("proof checking is easier than proof generation"). Whenever this is the case, then it is beneficial to have the Giotto compiler generate a feasible schedule once, and attach it to the generated E code in the form of S code. Then, before execution, the target platform may check the schedule in order to be sure that it can execute the code without time-safety violations (i.e., without missing any deadlines specified by the original Giotto program). While preemptive single-CPU scheduling for Giotto is simple [6], it is NP-hard to generate nonpreemptive or distributed schedules for Giotto programs, even if the program has only a single mode. We show that these schedules, once expressed in S code, can be checked in time linear in the size of the E code and the frequency of events. More generally, SCC provides an efficient implementation of Giotto in the presence of many scheduling constraints. In particular, in Section 4 we show how Giotto can be compiled onto a time-triggered architecture using SCC.

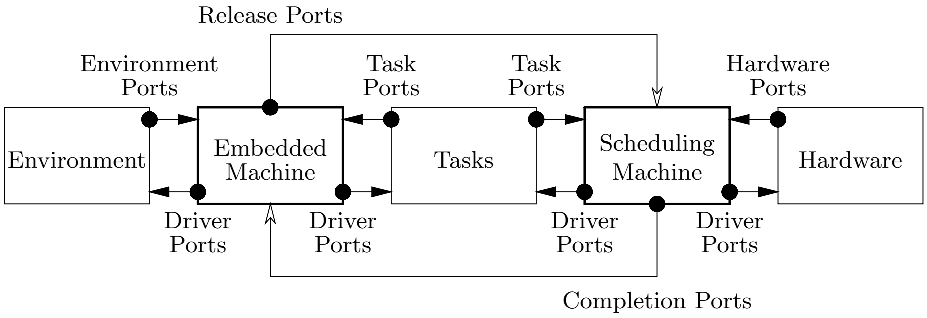


Fig. 1. The E machine and the S machine

2 SCC: Embedded Code Plus Scheduling Code

The E machine [5] is a virtual machine that interpretes E code, which supervises the execution of software tasks in response to physical events. The S machine is a virtual machine that interpretes S code, which specifies the temporal order of task execution. Figure 1 shows how E and S machine interact with the physical environment, software tasks, and hardware platform. We first review the E machine and then introduce the S machine.

2.1 The Embedded Machine

Interface. Physical environment processes communicate information to the E machine through *environment ports*, such as clocks and sensors, and application software processes, called *tasks*, communicate information to the E machine through *task ports*. The E machine communicates information to the environment and the tasks by calling system software processes, called *drivers*, which write to *driver ports*, for instance actuators. The E machine releases tasks for execution to the task scheduler (the S machine, or the scheduler of an RTOS) by writing to *release ports*, and the scheduler signals the completion of tasks to the E machine through *completion ports*. Hence, the environment, task, and completion ports are the *input ports* of the E machine. A change of value at an input port is an *input event* and causes an interrupt. The E machine monitors the occurrence of input events through *triggers*. In this paper, we consider only *time triggers*. A time trigger can be specified as a positive integer δ; it watches an environment clock and becomes enabled δ clock ticks after its activation. Tasks, drivers, and triggers are external to the E machine and must be implemented in some programming language like C. Tasks are preemptive, user-level code without internal synchronization points; drivers are system-level code during whose execution all interrupts that correspond to input events are disabled. The task idle is a special task that never completes.

E code. There are three non-control-flow E code instructions. The `call`(d) instruction initiates the execution of a driver d, and the E machine waits until d

```
while $ProgramCounter \neq \bot$ do
  $i := Instruction(ProgramCounter)$
  if call($d$) $= i$ then
    if driver $d$ accesses a port of a task $t$ that has been released but not completed
    then throw a time-safety exception else execute $d$
  else if schedule($t$) $= i$ then
    if task $t$ has already been released but not yet completed
    then throw a time-safety exception else emit a signal on the release port of $t$
  else if future($g, a$) $= i$ then
    append the trigger binding $(g, a, s)$ to $TriggerQueue$, where $s$ is the current
    state of the input ports that occur in $g$
  end if
  $ProgramCounter := Next(ProgramCounter)$
end while
```

Algorithm 1: The E code interpreter

is finished before proceeding to the next E code instruction. The `schedule`(t) instruction releases a task t to be executed, concurrently with other released tasks, and then the E machine proceeds immediately to the next E code instruction. The `schedule` instruction does not order the execution of tasks, nor does it relinquish control of the CPU to the scheduler. The `future`(g, a) instruction activates the trigger g and marks the E code at address a for execution at the future time instant when g becomes enabled. In order to handle multiple active triggers, the E machine maintains a queue of *trigger bindings* (g, a, s), where s is the current state of the input ports watched by the trigger g, which is required for evaluating g in the future. E code has also control-flow instructions such as `if`(c, a) and `return`. In the former case, if the condition c (a predicate on input ports) evaluates to true, then the E machine proceeds to address a; otherwise it proceeds to the next instruction. The `return` instruction terminates the execution of E code.

Algorithm 1 summarizes the E code interpreter. For each input event, the E machine checks the trigger bindings in *TriggerQueue*. The first trigger binding (g, a, s) in the queue with an enabled trigger g is removed from the queue and the interpreter is invoked with *ProgramCounter* set to address a. This is repeated until the queue contains no trigger binding with an enabled trigger. Then, the E machine relinquishes control of the CPU to the task scheduler, which is either provided by an RTOS [5] or implemented as S machine (see below). The goal of the scheduler is to execute the released tasks so that they complete before their deadlines. E code specifies task deadlines in two ways: once released, a task t must complete (1) before any driver accesses a port of t, and (2) before t is released again. If a task violates one of these conditions, then the E machine throws a *time-safety exception*; otherwise the execution is *time-safe*.

Example. Figure 2 shows a Giotto program [4] in the left column and, in the middle and right column, E code generated by the Giotto compiler [6]. The Giotto program is a simplified version of a program that implements the flight

```
start hover {
  mode hover() period 120ms {
    exitfreq 3 do cruise(switch);
    taskfreq 1 do pilot();
    taskfreq 2 do control();
    taskfreq 3 do lieu(); }
  mode cruise() period 120ms {
    exitfreq 2 do hover(switch);
    taskfreq 1 do pilot();
    taskfreq 2 do control();
    taskfreq 4 do move(); }
```

```
  H0: if(switch, C0 + 1)
      schedule(pilot)
      schedule(control)
      schedule(lieu)
      future(40ms, H40a)
      return[h0]
H40a: if(switch, H40b)
      schedule(lieu)
      future(20ms, H60)
      return[h40]
H40b: future(20ms, C60)
      return
 H60: schedule(control)
      future(20ms, H80a)
      return[h60]
H80a: if(switch, H80b)
      schedule(lieu)
      future(40ms, H0)
      return[h80]
H80b: future(10ms, C90)
      return
```

```
 C0: if(switch, H0 + 1)
     schedule(pilot)
     schedule(control)
     schedule(move)
     future(30ms, C30)
     return[c0]
C30: schedule(move)
     future(30ms, C60)
     return[c30]
C60: if(switch, H60)
     schedule(control)
     schedule(move)
     future(30ms, C90)
     return[c60]
C90: schedule(move)
     future(30ms, C0)
     return[c90]
```

Fig. 2. A Giotto program with two modes, and the generated E code

controller of a model helicopter [8]. The program consists of a *hover* mode, in which the helicopter maintains its airborne position, and a *cruise* mode. A Giotto mode specifies a set of periodic tasks. The task periods are specified through frequencies relative to the mode period, which is 120ms for both modes shown here. The *pilot* task is invoked, in both modes, every 120ms to perform path planning. The task outputs flight directions to the *control* task, which controls the servos and is invoked every 60ms in both modes. In the *hover* mode, the *control* task reads the current position estimation from the *lieu* task, which is invoked every 40ms. In the *cruise* mode, the *control* task receives position and velocity information from the *move* task, which is invoked every 30ms. The system can switch mode every 40ms from *hover* to *cruise*, and every 60ms from *cruise* to *hover*. A mode switch is initiated when the *switch* condition evaluates to true. For simplicity, we omitted all sensor and actuator code.

The E code in the middle column of Figure 2 implements the *hover* mode, and the right column implements the *cruise* mode. The execution of the program starts in the *hover* mode, thus the E code execution starts with the `if`(*switch*, `C0` + 1) instruction at address `H0`. If *switch* evaluates to true, then the E machine proceeds to the `schedule`(*pilot*) instruction that follows the instruction at address `C0`. This corresponds to a switch to the *cruise* mode. If *switch* evaluates to false, then the E machine proceeds to the `schedule`(*pilot*) instruction that follows the `if` instruction. The subsequent `schedule` instructions release the *control* and *lieu* tasks for execution. Then, the `future`(40*ms*, `H40a`) instruction makes the E machine execute the E code at address `H40a` after 40ms elapse. For now, the `return`[`h0`] instruction terminates the E code execution and relinquishes control of the CPU to the task scheduler. The expression [`h0`] is an *E code annotation*, which will be explained later.

```
while $ProgramCounter \neq \perp$ do
  $i := Instruction(ProgramCounter)$
  $ProgramCounter := Next(ProgramCounter)$
  if $\texttt{call}(d) = i$ then
  if driver $d$ accesses a port of a task $t$ that has been released but not completed
  then throw a time-safety exception else execute $d$
  else if $\texttt{dispatch}(t, h, a) = i$ then
    if there is a thread instance in $ThreadSet$ with a non-idle task then
      throw a time-sharing exception
    else
      insert the thread instance $(t, ProgramCounter, h, a, ReferenceTime)$
      into $ThreadSet$ and set $ProgramCounter$ to $\perp$
    end if
  else if $\texttt{idle}(h) = i$ then
    insert the thread instance $(\mathsf{idle}, \perp, h, ProgramCounter, ReferenceTime)$
    into $ThreadSet$ and set $ProgramCounter$ to $\perp$
  else if $\texttt{fork}(a) = i$ then
    insert the thread instance $(\mathsf{idle}, \perp, \mathsf{true}, a, s)$ into $ThreadSet$, where $s$ is the
    current value of the system clock
  end if
end while
```

Algorithm 2: The S code interpreter

2.2 The Scheduling Machine

Interface. The hardware on which the S machine runs communicates information to the S machine through *hardware ports*, such as clocks and message buffers, and the tasks communicate information to the S machine through task ports. The S machine communicates information to the hardware and the tasks by calling drivers. An external task handler (in our case, the E machine) signals the release of tasks to the S machine through release ports, and the S machine signals the completion of tasks to the task handler by writing to completion ports. Hardware, task, and release ports are input ports for the S machine, and changes in their values are input events. The S machine monitors input events through *timeouts*. In this paper, we consider two kinds of timeouts. A *clock timeout* is specified by a nonnegative integer δ; it watches a system clock and expires δ clock ticks into the S code thread that contains the timeout. The *release timeout* Θ expires as soon as any task is released.

S code. There are four non-control-flow S code instructions. The $\texttt{call}(d)$ instruction initiates the execution of a driver d. Similar to the E machine, the S machine waits until d is finished before proceeding to the next S code instruction. The $\texttt{dispatch}(t, h, a)$ instruction begins or resumes the execution of a task t until the timeout h expires. There are two possible outcomes: (1) the S machine proceeds to the next instruction when t completes, in case this happens before the timeout h expires, or (2) the S machine proceeds to the instruction at address a when the timeout h expires, in case this happens before t completes. Case (1) applies also if

```
RM: dispatch(lieu, +4)        EDF0/60: dispatch(lieu, +4)        EDF40/80: dispatch(control, +4)
    dispatch(control, +3)              dispatch(control, +3)               dispatch(lieu, +3)
    dispatch(pilot, +2)                dispatch(pilot, +2)                 dispatch(pilot, +2)
    idle()                             idle()                              idle()
    fork(RM)                           fork(EDF40/80)                      fork(EDF0/60)
    return                             return                              return
```

Fig. 3. Rate-monotonic (RM) and earliest-deadline-first (EDF) S programs for the *hover* mode of the Giotto program from Figure 2

t has not been released or has already completed when the `dispatch` instruction is encountered. The `idle`(h) instruction makes the S machine idle until the timeout h expires even when there are released tasks that have not been completed. The S machine proceeds to the subsequent instruction when the timeout expires. The `fork`(a) instruction marks the S code at address a for execution in parallel to the S code that follows the instruction. The S code at a is a new *thread* of execution. In order to handle multiple threads, the S machine maintains a set of thread instances. S code may also have control-flow instructions, but we do not consider them here. The `call`, `fork`, and control-flow instructions of S code are *transient instructions*, as they execute, like E code instructions, in logical zero time. In contrast, the `dispatch` and `idle` instructions are *timed instructions*, as they cause a passage of time.

Algorithm 2 summarizes the S code interpreter, which maintains the set *ThreadSet* of thread instances. A *thread instance* has the form (t, b, h, a, s), where either t is the idle task and $b = \perp$, or t is a regular task and b is the address at which the S machine continues executing when t completes before the timeout h expires. When h expires before t completes, the S machine continues executing, instead, the S code at address a. The *reference time* s is the time when the thread instance was created by a `fork` instruction, which is required for evaluating clock timeouts. The S machine is woken up by an input event: a hardware port may signal the completion of a task or the expiration of a clock timeout, or a release port may signal the release of a task. If a task t completes, then the thread instances of the form $(t, b, \cdot, \cdot, s)$ become *enabled*, with *ProgramCounter* set to b, and *ReferenceTime* set to s. If a timeout h expires, then the thread instances of the form $(\cdot, \cdot, h, a, s)$ become *enabled*, with *ProgramCounter* $= a$ and *ReferenceTime* $= s$. With each input event, every enabled thread instance is removed from *ThreadSet* and executed until a timed instruction is encountered (at that time, *ProgramCounter* is set to $\perp$ by the interpreter). The execution order for the enabled thread instances in *ThreadSet* is chosen nondeterministically. If control ends at `dispatch` instructions in more than one thread, then a *time-sharing exception* occurs, because only one task can be dispatched on the CPU; otherwise the execution is *time-sharing.*

Examples. The left column of Figure 3 shows an S program with the initial address `RM` which implements rate-monotonic (RM) scheduling of the tasks in the *hover* mode of the Giotto program from Figure 2. The S program only works if no mode switching occurs; S code that supports mode switching will be discussed below. We use `idle`() to abbreviate `idle`(Θ) and `dispatch`($t, +n$) to abbreviate

```
NP0: dispatch(move)      NP30: dispatch(move)       NP60: dispatch(pilot)  NP90: dispatch(control)
     dispatch(control)         dispatch(pilot, NP60)      dispatch(move)         dispatch(move)
     idle()                    idle()                     idle()                 idle()
     fork(NP30)                fork(NP60)                 fork(NP90)             fork(NP0)
     return                    return                     return                 return
```

Fig. 4. Nonpreemptive (NP) S program for the *cruise* mode

`dispatch`$(t, \Theta, a+n)$, where a is the address of the instruction itself and n is a relative offset. In the example, the offsets in `dispatch` instructions always point to `fork` instructions. The S program dispatches the tasks in a fixed, RM order starting with the task that has the highest frequency. Suppose that the tasks in the *hover* mode have been released by the E machine. Now, the S machine starts executing the `dispatch`$(lieu, +4)$ instruction. The *lieu* task executes until it either completes or is preempted by the release of some other task. If the *lieu* task completes first, then the S machine proceeds to the `dispatch`$(control, +3)$ instruction and executes the *control* task. Otherwise, if some task is released before *lieu* completes, the S machine proceeds to the `fork(RM)` instruction and forks a new thread starting again at address `RM`. The following `return` instruction terminates the current thread. If all tasks complete during the execution of a thread, then the `idle()` instruction is reached and the S machine waits until some task is released to start a new thread. Note that the execution of the S code is time sharing, and if there exists a feasible RM schedule for the tasks of the *hover* mode, then the S code guarantees the time-safe execution of the E code for the *hover* mode.

As an alternative to RM scheduling, the S code in the middle and right column implements earliest-deadline-first (EDF) scheduling of the tasks in the *hover* mode. The initial address of the EDF S program is `EDF0/60`. The thread at `EDF0/60` initially dispatches the *lieu* task followed by the *control* and *pilot* tasks. However, unlike the RM S program, as soon as the *lieu* task is released again, the S machine forks a new thread at address `EDF40/80` and terminates the current thread. Now, at the 40ms instant, the *control* task is dispatched before the *lieu* task, because the *control* task has an earlier absolute deadline than the *lieu* task. If the *control* task already completed before the 40ms instant, then the S machine immediately proceeds to the next instruction and dispatches the *lieu* task. At the 60ms instant, the situation is the same as at 0ms. So, we fork again a thread at `EDF0/60`. At the 80ms instant, the absolute deadlines of all tasks are the same. Thus we can use the thread at `EDF40/80` again.

The S code of Figure 4 implements nonpreemptive (NP) scheduling of the tasks in the *cruise* mode. The initial address of the NP S program is `NP0`. We use `dispatch`(t) to abbreviate `dispatch`(t, false, a), where a is the address of the next instruction. Thus a `dispatch`(t) instruction executes the task t until completion. Given the worst-case execution time $w(t)$ of each task t, the S program guarantees time safety if $w(move) + w(control) \leq 30ms$ and $2 \cdot w(move) + w(pilot) \leq 60ms$. Time sharing is ensured because the S program dispatches all tasks nonpreemptively, i.e., each task completes before another task is dispatched. Tasks may still be preempted by E code and S code, but not by other tasks.

```
h0/h60:                    h40/h80:                   c0/c30/c60/c90:
dispatch(lieu, +3)         dispatch(control, +3)      dispatch(move, +3)
dispatch(control, +2)      dispatch(lieu, +2)         dispatch(control, +2)
dispatch(pilot, +1)        dispatch(pilot, +1)        dispatch(pilot, +1)
return                     return                     return
```

Fig. 5. Earliest-deadline-first (EDF) S program for both modes

Figure 5 shows an S program that EDF schedules the tasks of both modes. In order to facilitate the interaction of E and S code in the presence of conditional-branch instructions (mode switching) in E code, we allow the E code to fork a new thread instance of S code through E code annotations. Recall the E code of Figure 2. The `return`[h0] instruction of the code block at address `H0` terminates the execution of the code block and, in addition, forks a new thread of S code at address `h0`. We start running the E program and S program by executing the E code at the initial address `H0`. Now the initial address of the S program is not required and thus $\perp$. The E code releases all three tasks of the *hover* mode and then creates a new S code thread starting at `h0`. The thread dispatches the three tasks in EDF order. Suppose that the *lieu* task completes before the 40ms instant, but not the *control* task. Thus, at the 40ms instant, the E code at `H40a` preempts the *control* task. Now, suppose that we switch from the *hover* to the *cruise* mode. We branch to the E code at `H40b`, which does not release any tasks but only jumps to the E code at `C60` after another 20ms elapse. Therefore, after executing the E code at `H40b`, the current S code thread continues where it was preempted by resuming the execution of the *control* task. No new thread is created. Now, suppose that the *control* task completes before the 60ms instant, but not the *pilot* task. Now, at the 60ms instant, the E code at `C60` preempts the *pilot* task. Unlike before, the *control* task is released now, no matter if we switch mode or not. Since the current S code thread becomes enabled upon the release of a task, the thread is terminated by jumping to the `return` instruction of the code block at `h0`. Suppose that we stay in the *cruise* mode, i.e., the code block at `C60` is executed. In addition to the *control* task, the *move* task is released and a new S code thread at `c60` is created, which dispatches now the tasks of the *cruise* mode in EDF order. Note that the S code for the *cruise* mode is equivalent to RM S code, because the task frequencies in the *cruise* mode are harmonic. The execution of the remaining E and S code works in a similar way. The S program is time sharing, and since EDF is optimal for Giotto programs with multiple modes [6], if there exists a feasible, preemptive schedule for the tasks of the *hover* and *cruise* modes, then the EDF S program guarantees the time-safe execution of the E program regardless of any mode switching.

2.3 Schedule-Carrying Code

An *SCC program* is a pair $(\mathcal{E}, \mathcal{S})$ consisting of an E program $\mathcal{E}$ that shares a set of tasks with an S program $\mathcal{S}$. If $\mathcal{E}$ contains conditional-branch instructions, then its `return` instructions may be annotated with S code addresses as shown in the example above. The runtime system for SCC is the E machine interacting with the S machine through release and completion ports. The machines are invoked as follows: (1) if there is an enabled thread instance that contains a completed task, then the S machine must handle that thread instance before the E machine handles any enabled triggers; (2) if there is an enabled trigger binding, then the E machine must handle that trigger binding before the S machine handles any expired timeouts. The reason for (1) is that, when a task completes, the dispatching thread must be processed before any E code is executed in order to enable the prompt handling of output data upon task completion, say through driver calls in S code. The reason for (2) is that the E machine must be invoked before the S machine when an E code trigger is enabled at the same time when an S code timeout expires, because the E code may release tasks that require scheduling service from the S code. This will be formalized in the next section.

Implementation. With the help of Marco Sanvido, we have developed a microkernel on a StrongARM SA-1110 processor with 206MHz which executes SCC programs using an integrated implementation of the E and S machine [7]. The microkernel has a footprint of 8kB. We have tested the microkernel with SCC programs that implement four periodic, nonharmonic task sets with 4, 10, 50, and 100 tasks. Each set consists of four equally large task groups with 16.66Hz, 33.33Hz, 50Hz, and 100Hz tasks. The microkernel is invoked every 1ms by a timer. The periodic release of the tasks is described by E code. We have compared the performance of an EDF scheduler with S code that specifies EDF scheduling. The average time spent in the EDF scheduler is 1.4μs to schedule 4 tasks and 35μs to schedule 100 tasks. The average time to execute, instead, the EDF S code grows from 2.2μs for scheduling 4 tasks to only 3.9μs for scheduling 100 tasks. With 100 tasks, EDF S code performs 35% better than the EDF scheduler (51% vs. 78% CPU utilization). For details, the reader is referred to [7].

3 Generating SCC vs. Checking SCC

We show that checking the time safety of given S code can be exponentially simpler than generating time-safe S code. Checking the time safety of SCC is a program-analysis problem. Following the tradition of path-insensitive program analysis, we define the *abstract* semantics of SCC, which ignores all port values and assumes that both branches of conditionals can be taken. On this abstract semantics, we present an efficient algorithm that provides a sufficient check for the time safety of SCC generated from single-mode Giotto, which specifies a set of periodic tasks.

3.1 Abstract Semantics of SCC

An *abstract E program* $\mathcal{E} = (V, E, \kappa, \lambda, \hat{v})$ over a set T of tasks consists of a control-flow graph (V, E) which is a binary[1] digraph, two edge-labeling functions κ and λ, and an initial node $\hat{v} \in V$. Each edge $e \in E$ is labeled with an instruction $\kappa(e)$ and an argument $\lambda(e)$ as follows:

[1] Each node has at most two successors.

– $\kappa(e) = \texttt{call}$ and $\lambda(e) \subseteq T$. The execution of e calls a driver that accesses ports of the tasks in $\lambda(e)$.
– $\kappa(e) = \texttt{schedule}$ and $\lambda(e) \in T$. The execution of e releases the task $\lambda(e)$.
– $\kappa(e) = \texttt{future}$ and $\lambda(e) \in \mathbb{N}_{>0} \times V$. The execution of e activates the trigger with the binding $\lambda(e) = (\delta, u)$, which means that after δ time units, E code will be executed starting from control location u.

We require that the initial node $\hat{v}$ has a single successor $\hat{v}' \in V$ such that $\kappa(\hat{v}, \hat{v}') = \texttt{future}$ and $\hat{v}'$ is a leaf. We assume that a driver can access the ports of at most two (or any fixed number) of tasks, and that all integers δ can be stored in constant space; hence the size of $\mathcal{E}$ is $|V|$. A *state* $q = (v, r, s, \tau)$ of $\mathcal{E}$ consists of a program counter $v \in V$, a status $s\colon T \to \mathbb{N} \cup \{\top, \bot\}$ for each task, and a queue $\tau \subseteq (\mathbb{N} \times V)^*$ of trigger bindings. For each task $t \in T$, the status $s(t) = \top$ indicates that t has been released at the current time instant and not yet executed; the status $s(t) \in \mathbb{N}$ indicates that t has been released at a previous time instant and executed for $s(t) \geq 0$ time units; the status $s(t) = \bot$ indicates that t has been completed (or not yet released). Note that the number of different status functions, and hence the number of states, is exponential in $|T|$.

An *abstract S program* $\mathcal{S} = (V, E, \mu, \nu, \kappa, \lambda)$ over a set T of tasks consists of a control-flow graph (V, E) which is a binary digraph, two node-labeling functions μ and ν, and two edge-labeling functions κ and λ. Each control location $v \in V$ is labeled by one of the following:

– $\mu(v) = \texttt{dispatch}$ and $\nu(v) \in T$ and v has a successor v_1 such that $\lambda(v, v_1) = \bot$, and possibly a second successor v_2 such that either $\lambda(v, v_2) = *$ or $\lambda(v, v_2) \in \mathbb{N}$. The execution of v dispatches the task $\nu(v)$. If $\lambda(v, v_2) = *$ and a task is released before $\nu(v)$ completes, control proceeds to v_2; if $\lambda(v, v_2) \in \mathbb{N}$ and $\nu(v)$ does not complete within the first $\lambda(v, v_2)$ time units from the time at which the current thread was created, control proceeds to v_2; otherwise, control proceeds to v_1 when $\nu(v)$ completes.
– $\mu(v) = \texttt{idle}$ and v has a single successor v' such that either $\lambda(v, v') = *$ or $\lambda(v, v') \in \mathbb{N}$. The execution of v idles the processor until a task is released (if $\lambda(v, v') = *$), or until $\lambda(v, v') \in \mathbb{N}$ time units pass from the time at which the current thread was created.
– $\mu(v) = \nabla$. This indicates that control is at a transient instruction.

If $e = (v, v')$ and $\mu(v) = \nabla$, then the edge $e \in E$ is labeled by one of the following:

– $\kappa(e) = \texttt{call}$ and $\lambda(e) \subseteq T$. The execution of e calls a driver that accesses ports of the tasks in $\lambda(e)$.
– $\kappa(e) = \texttt{fork}$ and $\lambda(e) \in V$. If $\lambda(e) = u$, then the execution of e creates a new thread, which starts at control location u.

By our assumptions, the size of $\mathcal{S}$ is $|V|$. A *state* $q = (s, \theta)$ of $\mathcal{S}$ consists of the status $s\colon T \to \mathbb{N} \cup \{\top, \bot\}$ for each task, and a set θ of threads. Each thread (u, δ) consists of a program counter $u \in V$ and a number $\delta \in \mathbb{N}$ of time units for which the thread has been executed. If u is a leaf, then the thread (u, δ) has terminated and may be removed from θ.

An *abstract single-processor SCC program* $\Pi = (\mathcal{E}, \mathcal{S}, \Phi)$ over a set T of tasks consists of an abstract E program $\mathcal{E}$ over T, an abstract S program $\mathcal{S}$ over T, and an *annotation function* Φ that maps each leaf in the control graph of $\mathcal{E}$ to a node in the control graph of $\mathcal{S}$. When the E code execution arrives at a leaf v, this creates a new thread of S code which starts at control location $\Phi(v)$. A *state* of Π is a tuple $q = (v, s, \tau, \theta)$ such that (v, s, τ) is a state of $\mathcal{E}$, and (s, θ) is a state of $\mathcal{S}$. The state q violates *nonpreemption* if there exist two different tasks $t_1, t_2 \in T$ such that $s(t_1), s(t_2) \notin \{\top, 0, \bot\}$. The state q violates *time sharing* if there exist two different threads $(u_1, \cdot), (u_2, \cdot) \in \theta$ such that $\mu(u_1) = \mu(u_2) = \texttt{dispatch}$ and $s(\nu(u_1)) \neq \bot$ and $s(\nu(u_2)) \neq \bot$. The state q violates *time safety* if there exists a task $t \in T$ with $s(t) \neq \bot$ and one of the following: (1) v has a successor v' in $\mathcal{E}$ with either $\kappa(v, v') = \texttt{call}$ and $t \in \lambda(v, v')$, or $\kappa(v, v') = \texttt{schedule}$ and $t = \lambda(v, v')$; or (2) there exists a thread $(u, \cdot) \in \theta$ such that $\mu(u) = \nabla$ and u has a successor u' in $\mathcal{S}$ with $\kappa(u, u') = \texttt{call}$ and $t \in \lambda(u, u')$. The state q has a *transition* to the state $q' = (v', s', \tau', \theta')$ if one of the following:

Completion S transition The state q is *completion enabling*, that is, there exist a thread $(u, \delta) \in \theta$ and a successor u' of u in $\mathcal{S}$ such that $\mu(u) = \texttt{dispatch}$ and $s(\nu(u)) = \bot$ and $\lambda(u, u') = \bot$. Then $(v', s', \tau') = (v, s, \tau)$ and $\theta' = (\theta \backslash \{(u, \delta)\}) \cup \{(u', \delta)\}$.

Transient S transition The state q is *transient enabling*, that is, there exist a thread $(u, \delta) \in \theta$ and a successor u' of u in $\mathcal{S}$ such that $\mu(u) = \nabla$. Then $(v', s', \tau') = (v, s, \tau)$ and one of the following:
- $\kappa(u, u') = \texttt{call}$ and $\theta' = (\theta \backslash \{(u, \delta)\}) \cup \{(u', \delta)\}$.
- $\kappa(u, u') = \texttt{fork}$ and $\lambda(u, u') = \hat{u}$ and $\theta' = (\theta \backslash \{(u, \delta)\}) \cup \{(u', \delta), (\hat{u}, 0)\}$.

E transition q is neither completion nor transient enabling but *E enabling*, that is, either (1) v has no successor and $(0, u) \in \tau$ for some u, or (2) v has a successor v' in $\mathcal{E}$. If (1) let $(0, u')$ be the first such pair in τ. Then $v' = u'$ and $s' = s$ and $\tau' = \tau \setminus \{(0, u')\}$ and $\theta' = \theta$. If (2) then one of the following:
- $\kappa(v, v') = \texttt{call}$ and $s' = s$ and $\tau' = \tau$.
- $\kappa(v, v') = \texttt{schedule}$ and $\lambda(v, v') = t$ and $s'(t) = \top$ and $s'(t') = s(t')$ for all tasks $t' \in T \backslash \{t\}$, and $\tau' = \tau$.
- $\kappa(v, v') = \texttt{future}$ and $s' = s$ and $\tau' = \tau \circ \{\lambda(v, v')\}$.

In all three cases, if v' is a leaf, then $\theta' = \theta \cup \{(\Phi(v'), 0)\}$; otherwise $\theta' = \theta$.

Timeout S transition The state q is neither completion nor transient nor E enabling but *timeout enabling*, that is, there exist a thread $(u, \delta) \in \theta$ and a successor u' of u in $\mathcal{S}$ such that $\mu(u) \in \{\texttt{dispatch}, \texttt{idle}\}$ and either (1) $\lambda(u, u') \in \mathbb{N}$ and $\lambda(u, u') \leq \delta$, or (2) $\lambda(u, u') = *$ and $s(t) = \top$ for some task $t \in T$. Then $(v', s', \tau') = (v, s, \tau)$ and $\theta' = (\theta \backslash \{(u, \delta)\}) \cup \{(u', \delta)\}$.

Time transition The state q is neither completion nor transient nor timeout nor E enabling. Then $v' = v$. For all tasks $t \in T$, if there exists a thread $(u, \delta) \in \theta$ with $\mu(u) = \texttt{dispatch}$ and $\nu(u) = t$, then either $s(t) = \top$ and $s'(t) = 1$, or $s(t) \in \mathbb{N}$ and $s'(t) = s(t) + 1$, or $s'(t) = \bot$; if no such thread exists, then either $s(t) = \top$ and $s'(t) = 0$, or $s'(t) \neq \top$ and $s'(t) = s(t)$. In case $s'(t) = \bot$, we say that on the transition (q, q'), task t *completes* after execution time $s(t) + 1$. The queue τ' results from τ by replacing each trigger binding (δ, u) by $(\delta - 1, u)$. The set θ' results from θ by replacing each thread (u, δ) by $(u, \delta + 1)$.

Note the priorities implied by this definition: transient S code that is enabled by the completion of tasks has priority over E code, which has priority over all remaining S code. A *trace* of the abstract single-processor SCC program Π is a sequence $\gamma = q_0, q_1, \ldots, q_n$ of states of Π such that (1) $q_0 = (\hat{v}, \hat{s}, \emptyset, \emptyset)$, where $\hat{v}$ is the initial node of $\mathcal{E}$, and $\hat{s}(t) = \bot$ for all tasks $t \in T$, and (2) for all $i \geq 0$, there is a transition from q_i to q_{i+1}. The trace γ is *time-safe* (resp. *time-sharing*; *nonpreemptive*) if no state of γ violates time safety (time sharing; nonpreemption).

An *abstract multiprocessor SCC program Π* over a set P of processors and a set T of tasks is a function that assigns to each processor $p \in P$ a pair (T_p, Π_p), where $T_p \subseteq T$ such that $\{T_p \mid p \in P\}$ is a partition of the task set T, and Π_p is an abstract single-processor SCC program over the set T_p of tasks. A *trace* γ of Π is a function that assigns to each processor $p \in P$ a trace γ_p of Π_p such that all single-processor traces γ_p contain the same number of time transitions. The trace γ is time-safe (resp. time-sharing; nonpreemptive) if γ_p is time-safe (time-sharing; nonpreemptive) for all $p \in P$. A *wcet map* w: $P \times T \to \mathbb{N}$ assigns to every processor p and task t a worst-case execution time $w(p, t) > 0$. There are a number of techniques for obtaining wcet maps, e.g. [2]. If P is a set of *identical* processors, then $w(p_1, t) = w(p_2, t)$ for all processors $p_1, p_2 \in P$ and tasks $t \in T$. The trace γ of Π is an *w-trace* if for all processors $p \in P$, tasks $t \in T$, and $i \geq 0$, if t completes on the transition (q_i, q_{i+1}) of γ_p, then it completes with execution time at most $w(p, t)$. The abstract (single- or multiprocessor) SCC program Π is *time-safe* (resp. *time-sharing*; *nonpreemptive*) for wcet map w if all w-traces of Π are time-safe (time-sharing; nonpreemptive). The time safety (time sharing; nonpreemption) of an abstract SCC program can be checked by searching the state space, but the number of states is exponential in the number of tasks. We will see that the check is simpler for SCC programs of a special form.

3.2 Giotto-Generated SCC

The E programs generated from Giotto programs have a special form [6]. Let G be a Giotto program [4] with task set T and a single mode m, let π_m be the period of m, let f_t be the frequency in mode m of task t, and let f_m be the least common multiple of all task and actuator frequencies in m. Then $d_m = \pi_m / f_m$ denotes the time interval between consecutive input events. The abstract E program $\mathcal{E} = (V, E, \kappa, \lambda, \hat{v})$ over a set $T' \subseteq T$ of tasks is *G-generated* if the control graph (V, E) consists, in addition to the initial location $\hat{v}$ and its successor $\hat{v}'$, of a set of f_m acyclic digraphs $\mathcal{E}_i$, for $0 \leq i < f_m$, such that every node in $\mathcal{E}_i$ has at most one successor, and $\mathcal{E}_i$ consists of a source node v_i followed by (1) a sequence of $O(|G|)$ edges (v, v') with $\kappa(v, v') = \texttt{call}$, followed by (2) a sequence of edges (v, v') with $\kappa(v, v') = \texttt{schedule}$ and $\lambda(v, v') = t$ for each task $t \in T'$ for which $(i \cdot f_t / f_m) \in \mathbb{N}$, followed by (3) a single edge (v, v') with $\kappa(v, v') = \texttt{future}$ and $\lambda(v, v') = (d_m, v_{(i+1) \texttt{mod} f_m})$. Moreover, $\lambda(\hat{v}, \hat{v}') = (0, v_0)$. If all numbers in G

(mode period as well as task and actuator frequencies) are bounded by n, we have $|V| = O(|G| \cdot n)$. An abstract single-processor SCC program $\Pi = (\mathcal{E}, \cdot, \cdot)$ is *G-generated* if $\mathcal{E}$ is a G-generated abstract E program over the set T of all Giotto tasks. An abstract multiprocessor SCC program Π is *G-generated* if for each processor $p \in P$, if $\Pi_p = (\mathcal{E}_p, \cdot, \cdot)$, then $\mathcal{E}_p$ is a G-generated abstract E program over the set T_p of tasks that are assigned to processor p.

Proposition 1. *Let P be a set of identical processors, let G be a single-mode Giotto program with task set T, and let w be a wcet map for P and T. It is NP-hard in the strong sense to decide if there is a G-generated abstract multiprocessor SCC program over P and T which is time-safe and time-sharing for w.*

Proof. Reduction from BIN PACKING [3]. Given an instance of BIN PACKING, we choose as many processors as there are bins, and construct a single-mode Giotto program whose tasks have periods equal to the bin capacity. □

Proposition 2. *Let G be a single-mode Giotto program with task set T, and let w be a wcet map for T. It is NP-hard in the strong sense to decide if there is a G-generated abstract single-processor SCC program over T which is nonpreemptive, time-safe, and time-sharing for w.*

Proof. Reduction from NSPT (nonpreemptive scheduling of periodic tasks) [1]. □

Checking time safety becomes simpler if we restrict also the shape of S code. Let G be a Giotto program with task set T and numbers bounded by n, as above. The abstract S program $\mathcal{S} = (V, E, \mu, \nu, \kappa, \lambda)$ is *simple* if the control graph (V, E) is acyclic and for every node $v \in V$, (1) if $\mu(v) = \triangledown$, then v has at most one successor, and (2) if $\mu(v) = \triangledown$ and v' is a successor of v with $\kappa(v, v') = \texttt{fork}$, then v' is a leaf. Condition (1) ensures that the S code does not contain conditional branching. Condition (2) ensures that the S code is single-threaded, i.e., during execution, there is always at most one thread in the thread set. Single-threadedness, in turn, implies time sharing. An abstract single-processor SCC program $\Pi = (\mathcal{E}, \mathcal{S}, \Phi)$ is *simple G-generated* if (1) $\mathcal{E}$ is a G-generated abstract E program over T, (2) $\mathcal{S}$ is a simple abstract S program of size $O(|G| \cdot n)$ which does not contain numbers (clock timeouts) larger than the time step d_m, and (3) for each leaf v of $\mathcal{E}$, if v is not the successor of the initial node of $\mathcal{E}$, then $\Phi(v)$ is a leaf of $\mathcal{S}$. An abstract multiprocessor SCC program Π is *simple G-generated* if for each processor $p \in P$, the single-processor program Π_p over T_p satisfies conditions (1)–(3). Note that for the Giotto program G from Figure 2, the E code from Figure 2 together with the S code samples from Figures 3 and 4 yields simple G-generated SCC programs.

Proposition 3. *Let P be a set of (nonidentical) processors, let G be a single-mode Giotto program with task set T and numbers bounded by n, and let w be a wcet map for P and T. It can be checked in time $O(|G| \cdot n)$ if a given simple G-generated abstract multiprocessor SCC program is time-safe (resp. nonpreemptive) for w.*

```
h0:                        h40:                      h60:                      h80:

dispatch(lieu, 10, +2)     dispatch(control, +2)     dispatch(lieu, +2)        dispatch(lieu, 30, +4)
idle(10)                   dispatch(lieu, +1)        dispatch(control, +1)     dispatch(control, 30, +2)
call(ctrl_in)              return                    return                    idle(30)
dispatch(lieu, +2)                                                             call(lieu_out)
dispatch(control, +1)                                                          dispatch(control, +1)
return                                                                         return

c0:                         c30/c60:                 c90:                       p0:

dispatch(move, 10, +2)      dispatch(move, +2)       dispatch(move, 20, +4)     call(pilot_in)
idle(10)                    dispatch(control, +1)    dispatch(control, 20, +2)  dispatch(pilot, +3)
call(ctrl_in)               return                   idle(20)                   idle(120)
dispatch(move, +2)                                   call(move_out)             call(pilot_out)
dispatch(control, +1)                                dispatch(control, +1)      return
return                                               return
```

Fig. 6. Distributed, TDMA-based S code for a two-processor TTA

Proof. For unconditional single-threaded S code, if all traces in which each task completes with an execution time equal to the time given by the wcet map w is time-safe, than all w-traces are time-safe. Therefore all transitions are deterministic, and it suffices to check the time-safety of a trace whose duration corresponds to one mode period. The number of states thus explored is $O(|G| \cdot n)$. □

For multimode Giotto programs, we know that if each mode in isolation is time-safe under EDF scheduling, then the whole program is time-safe under EDF [6]. Furthermore, we can check the EDF schedulability of a single mode by solving a utilization equation [6]. Hence, for SCC, it remains to be checked if a given abstract SCC program represents an EDF schedule, such as the example from Figure 5. This can be done in time linear in the size of the Giotto program.

4 SCC for Time-Triggered Networks

We illustrate how to generate distributed SCC programs that run on a *time-triggered architecture* (TTA) [9], whose nodes are connected by a bus on which all communication is scheduled according to a collision-free TDMA protocol. Each time slot assigns exclusive network access to one of the nodes to send data. Each node has a host and a network processor connected by a send and a receive buffer. Thus the host processor can execute programs while data is being sent or received. To send data, the host processor loads the send buffer before its time slot arrives. Similarly, to receive data, the host reads the receive buffer after a time slot ends. There are an E machine and an S machine running on each host processor. The E code portion of a distributed SCC program may be generated from Giotto. The S code portion specifies the execution order of released tasks, and also calls drivers that transport data between message buffers and tasks before and after the appropriate time slots. Thus the TDMA protocol imposes additional timing constraints on the tasks.

Example. Figure 6 shows distributed, TDMA-based S code for two host processors, p_0 and p_1, which execute the Giotto program from Figure 2. Suppose that

p_0 executes the *pilot* task, while p_1 takes care of the other tasks by executing the E code in Figure 2 with the `schedule`(*pilot*) instructions removed. For p_1, we use the E code `P0`: `schedule`(*pilot*); `future`(120, `P0`); `return`[`p0`]. Suppose that the output of the *pilot* task is read by the *control* task, while the outputs of the *lieu* and *move* tasks are read by the *pilot* task. For this purpose, we use a TDMA-schedule with two time slots, l_0 and l_1, where l_0 is from 0ms to 10ms, and l_1 is from 110ms to 120ms. Processor p_0 sends during l_0, and p_1 sends during l_1. The *pilot_out* driver writes the output of the *pilot* task into the send buffer of p_0. Similarly, the *pilot_in* driver reads the input for the *pilot* task from the receive buffer of p_0. On processor p_1, the *ctrl_in* driver reads the input for the *control* task from the receive buffer. The *lieu_out* and *move_out* drivers write the outputs of the *lieu* and *move* tasks, respectively, to the send buffer. Note that tasks may be executed during the time slots l_0 and l_1, because in a TTA each host has a separate network processor that handles the network traffic. For example, the *pilot* task may run for its full period of 120ms. The *control* task, on the other hand, cannot start before 10ms elapse from the beginning of the mode period, because its inputs depend on values received during the time slot l_0. Likewise, the *lieu* and *move* tasks must complete before 110ms. The S code represents an EDF schedule *under the constraints* imposed by the TDMA protocol. As the constraints are the same for all modes of the Giotto program, it can be shown that this schedule is optimal.

References

1. Y. Cai and M.C. Kong. Nonpreemptive scheduling of periodic tasks in uni- and multiprocessor systems. *Algorithmica* 15:572–599, 1996.
2. C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, R. Wilhelm. Reliable and precise WCET determination for a real-life processor. In *Embedded Software*, LNCS 2211, pp. 469–485. Springer, 2001.
3. M.R. Garey and D.S.Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman, 1979.
4. T.A. Henzinger, B. Horowitz, C.M. Kirsch. Giotto: a time-triggered language for embedded programming. *Proc. IEEE* 91:84–99, 2003.
5. T.A. Henzinger and C.M. Kirsch. The Embedded Machine: predictable, portable real-time code. In *Proc. Programming Language Design and Implementation*, pp. 315–326. ACM, 2002.
6. T.A. Henzinger, C.M. Kirsch, R. Majumdar, S. Matic. Time-safety checking for embedded programs. In *Embedded Software*, LNCS 2491, pp. 76–92. Springer, 2002.
7. C.M. Kirsch, T.A. Henzinger, M.A.A. Sanvido. *A Programmable Microkernel for Real-Time Systems.* Technical Report CSD-03-1250, UC Berkeley, 2003.
8. C.M. Kirsch, M.A.A. Sanvido, T.A. Henzinger, W. Pree. A Giotto-based helicopter control system. In *Embedded Software*, LNCS 2491, pp. 46–60. Springer, 2002.
9. H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications.* Kluwer, 1997.
10. G.C. Necula. Proof-carrying code. In *Proc. Principles of Programming Languages*, pp. 106–119. ACM, 1997.

Energy-Efficient Multi-processor Implementation of Embedded Software

Shaoxiong Hua, Gang Qu, and Shuvra S. Bhattacharyya

Electrical and Computer Engineering Department and Institute for Advanced Computer Studies University of Maryland, College Park, MD 20742, USA
{shua,gangqu,ssb}@eng.umd.edu

Abstract. This paper develops energy-driven completion ratio guaranteed scheduling techniques for the implementation of embedded software on multi-processor system with multiple supply voltages. We leverage application's performance requirements, uncertainties in execution time, and tolerance for reasonable execution failures to scale processors' supply voltages at run-time to reduce energy consumption. Specifically, we study how to trade the difference between the highest achievable completion ratio $\mathcal{Q}^{max}$ and the required completion ratio $\mathcal{Q}_0$ for energy saving. We propose several on-line scheduling policies, which are all capable of providing $\mathcal{Q}_0$, based on the knowledge about application's execution time. We show that significant energy saving is achievable when only the worst/best case execution time are known and further energy reduction is possible with the probabilistic distribution of execution time. The proposed algorithms have been implemented and their energy-efficiency have been verified by simulations over real-life DSP applications and the TGFF random benchmark suite.

1 Introduction

Performance guarantee and energy efficiency are becoming increasingly important for the implementation of embedded software. Traditionally, the worst case execution time (WCET) is considered to provide performance guarantee, however, this often leads to over-designing the system (e.g., more hardware and more energy consumed than necessary). We discuss the problem of how to implement multi-processor embedded systems to deliver performance guarantee with reduced energy consumption.

Many applications, such as multimedia and digital signal processing (DSP) applications, are characterized by repetitive processing on periodically arriving inputs (e.g., voice samples or video frames). Their processing deadlines, which are determined by the throughput of the input data streams, may occasionally be missed without being noticeable or annoying to the end user. For example, in packet audio applications, loss rates between 1% - 10% can be tolerated [2], while tolerance for losses in low bit-rate voice applications may be significantly lower [13]. Such tolerance gives rise to slacks that can be exploited when streamlining the embedded processing associated with such applications. Specifically, when the embedded processing does not interact with a lossy communication channel, or when the channel quality is high compared to the tolerable rate of missed deadlines, we are presented with slacks in the application that can be used to reduce cost or power consumption.

Typically, slacks arise from the run-time task execution time variation and can be exploited to improve real-time application's response time or reduce power. For example, Bambha and Bhattacharyya examined voltage scaling for multi-processor with known computation time and hard deadline constraints [1]. Luo and Jha presented a power-conscious algorithm [14] and static battery-aware scheduling algorithms for distributed real-time battery-powered systems [15]. Zhu et al. introduced the concept of slack sharing on multi-processor systems to reduce energy consumption [26]. The essence of these works is to exploit the slacks by using voltage scaling to reduce energy consumption without suffering any performance degradation (execution failures).

The slack we consider in this paper comes from the tolerance of execution failures or deadline missings. In particular, since the end user will not notice a small percentage of execution failure, we can *intentionally* drop some tasks to create slack for voltage scaling as long as we keep the loss rates to be tolerable. Furthermore, much richer information than task's WCET is available for many DSP applications. Examples include the best case execution time (BCET), execution time with cache miss, when interrupt occurs, when pipeline stalls or when different conditional branch happens. More important, most of these events are predictable and we will be able to obtain the probabilities that they may happen by knowing (e.g. by sampling technique) detailed timing information about the system or by simulation on the target hardware [24]. This gives another degree of freedom to explore on-line and offline voltage scaling for energy reduction.

Dynamic voltage scaling(DVS), which can vary the supply voltage and clock frequency according to the workload at run-time, can exploit the slack time generated by the workload variation and achieve the highest possible energy efficiency for time-varying computational loads [3,19]. It is arguably the most effective technique to reduce the dynamic energy, which is still the dominate part of system's energy dissipation despite the fast increase of leakage power on modern systems. The most relevant works on DVS, to this paper, are on the energy minimization of dependent tasks on multiple voltage processors. Schmitz and Al-Hashimi investigated DVS processing elements power variations based on the executed tasks, during the synthesis of distributed embedded systems, and its impact on the energy saving [21]. Gruian and Kuchcinski introduced a new scheduling approach, LEneS, that uses list-scheduling and a special priority function to derive static schedules with low energy consumption. The assignment of tasks to multiple processors is assumed to be given [8]. Luo and Jha presented static scheduling algorithm based on critical path analysis and task execution order refinement. An on-line scheduling algorithm is also developed to reduce the energy consumption for real-time heterogeneous distributed embedded systems while providing the best-effort services to soft aperiodic tasks. The deadlines and precedence relationships of hard real-time periodic tasks are guaranteed [16]. In

[18], Mishra et al. proposed static and dynamic power management schemes for distributed hard real-time systems, where the communication time is significant and tasks may have precedence constraints. However, these algorithms use the slacks to reduce energy but they do not drop tasks to create more slacks. Different from them, some energy reduction techniques on single processor have been proposed by Hua et al. for multimedia applications with tolerance to deadline misses while providing a statistical completion ratio guarantee [10].

Finally, we mention that early efforts on multi-processor design range from the design space exploration algorithm [12] to the implementation of such systems [7,23]. And scalable architectures and co-design approaches have been developed for the design of multi-processor DSP systems (e.g., see [11,20]). These approaches, however, do not provide systematic techniques to handle voltage scaling, non-deterministic computation time, or completion ratio tolerance. Performance-driven static scheduling algorithms that allocate task graphs to multi-processors [22] can be used in conjunction with best- or average-case task computation time to generate an initial schedule for our proposed methods. It can then interleave performance monitoring and voltage adjustment functionality into the schedule to streamline its performance.

A Motivational Example

We consider a simple case when a multiple-voltage processor executes three tasks $\mathcal{A}, \mathcal{B}, \mathcal{C}$ in that order repetitively. Table 1(a) gives each task's only two possible execution time and the probabilities that they occur. Table 1(b) shows the normalized power consumption and processing speed of the processor at three different voltages.

Table 1. Characteristics of the tasks and the processor. (a): each entry shows the best/worst case execution time at v_1 and the probability this execution time occurs at run time. (b): *power* is normalized to the power at v_1 and *delay* column gives the normalized processing time to execute the same task at different voltages.

task	BCET	WCET
$\mathcal{A}$	(1, 80%)	(6, 20%)
$\mathcal{B}$	(2, 90%)	(7, 10%)
$\mathcal{C}$	(2, 75%)	(5, 25%)

(a) Three tasks.

voltage	power	delay
$v_1 = 3.3V$	1	1
$v_2 = 2.4V$	0.30	1.8
$v_3 = 1.8V$	0.09	3.4

(b) Processor parameters.

Suppose that each iteration of "$\mathcal{A} \to \mathcal{B} \to \mathcal{C}$" must be completed in 10 CPU units and we can tolerate 40% of the 10,000 iterations to miss their deadlines. We now compare the following three different algorithms:

(I) For each iteration, run at the highest voltage v_1 to the completion or the deadline whichever happens first.

(II) Assign deadline pairs (0,6), (5,8), and (10,10) to $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$ respectively. For each task, terminate the current iteration if the task cannot be completed by its second and longer deadline at v_1; otherwise, run at the lowest voltage without violating its first and shorter deadline or run at v_1 to its completion.

(III) In each iteration, assign 1, 7, and 2 (a total of 10) CPU units to $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$ respectively. Each task can only be executed within its assigned slot: if it cannot be finished at v_1, terminate; otherwise run at the lowest voltage to completion.

Assuming that the execution time of each task follows the above probability, for each algorithm, we obtain the completion ratio $\mathcal{Q}$, each iteration's average processing time (at different voltages) and power consumption (Table 2). We mention that 1) algorithm I gives the highest possible completion ratio; 2) algorithm II achieves the same ratio with less energy consumption; and 3) algorithm III trades unnecessary completion for further energy reduction. Although algorithm I is a straightforward best-effort approach, the settings for algorithms II and III are not trivial: *Why the deadline pairs are determined for $\mathcal{A}$ and $\mathcal{B}$? Is it a coincidence that such setting achieves the same completion ratio as algorithm I? How to set execution slot for each task in algorithm III to guarantee the 60% completion ratio, in particular if we cannot find 80% and 75% whose product gives the desired completion ratio?*

Table 2. Expected completion ratio and energy consumption for the three algorithms. $t@v_1$, $t@v_2$, and $t@v_3$ are the average time that the processor operates at three voltages for each iteration; E is the average energy consumption to complete one iteration; and the last column, obtained by $E \cdot 60\%/\mathcal{Q}$, corresponds to the case of shutting the system down once 6,000 iterations are completed.

	$\mathcal{Q}$	$t@v_1$	$t@v_2$	$t@v_3$	E	E@($\mathcal{Q}=60\%$)
I	91.5%	6.94	0	0	6.94	4.55
II	91.5%	4.21	4.54	0	5.57	3.65
III	60%	2.56	0	4.90	3.00	3.00

In this paper, i) we first formulate the energy minimization problem with deadline miss tolerance on multi-processor (DSP) systems; ii) we then develop on-line scheduling techniques to convert deadline miss tolerances into energy reduction via DVS; iii) this departs us from the conservative view of over-implementing the embedded software in order to meet deadlines under WCET; iv) our result is an algorithmic framework that integrates considerations of iterative multiprocessor scheduling, voltage scaling, non-deterministic computation time, and completion ratio requirement, and provides robust, energy-efficient multi-processor implementation of embedded software for embedded DSP applications.

2 Problem Formulation

We consider the *task graph* $G = (V, E)$ for a given application. Each vertex in the graph represents one computation and directed edges represent the data dependencies between vertices. For each vertex v_i, we associate it with a finite set of possible execution time $\{t_{i,1} < t_{i,2} < \cdots < t_{i,k_i}\}$ and the corresponding set of probabilities $\{p_{i,1}, p_{i,2}, \cdots, p_{i,k_i} | \sum_{l=1}^{k_i} p_{i,l} = 1\}$ that such execution time may occur. That is, with probability $p_{i,j}$, vertex v_i requires an execution time in the amount of $t_{i,j}$. Note that t_{i,k_i} is the WCET and $t_{i,1}$ is the BCET for task v_i. We then define the prefix sum of the occurrence probability

$$P_{i,l} = \sum_{j=1}^{l} p_{i,j} \tag{1}$$

Clearly, $P_{i,l}$ measures the probability that the computation at vertex v_i can be completed within time $t_{i,l}$ and we have $P_{i,k_i} = 1$ which means that a completion is guaranteed if we allocate CPU to vertex v_i based on its WCET t_{i,k_i}.

A directed edge $(v_i, v_j) \in E$ shows that the computation at vertex v_j requires data from vertex v_i. For each edge (v_i, v_j), there is a cost for *inter-processor communication* (IPC) w_{v_i,v_j}, which is the time to transfer data from the processor that executes v_i to a different processor that will execute v_j. There is no IPC cost, i.e. $w_{v_i,v_j} = 0$, if vertices v_i and v_j are mapped to the same processor by the task scheduler. For a given datapath $< v_1 v_2 \cdots v_n >$, its *completion time* is the sum of the execution time at run-time, of each vertex, e_i, and all the IPC costs. That is,

$$C(< v_1 v_2 \cdots v_n >) = e_1 + \sum_{i=2}^{n} (w_{v_{i-1},v_i} + e_i) \tag{2}$$

The *completion time* for the entire task graph G (or equivalently the given application), denoted by $C(G)$, is equal to the completion time of its *critical path*, which has the longest completion time among all its datapaths.

We are also given a *deadline* constraint $\mathcal{M}$, which specifies the maximum time allowed to complete the application. The application (or its task graph) will be executed on a multi-processor system periodically with its deadline $\mathcal{M}$ as the period. We say that an iteration is *successfully completed* if its completion time, which depends on the run-time behavior, $C(G) \leq \mathcal{M}$. Closely related to $\mathcal{M}$ is a real-valued *completion ratio* constraint(or requirement) $\mathcal{Q}_0 \in [0, 1]$, which gives the minimum acceptable completion ratio over a sufficiently large number of iterations. Alternatively, $\mathcal{Q}_0$ can be interpreted as a guarantee on the probability with which an arbitrary iteration can be successfully completed.

Finally, we assume that there are multiple supply voltage levels available at the same time for each processor in the multi-processor system. This type of system can be implemented by using a set of voltage regulators each of which regulates a specific voltage for a given clock frequency. In this way, the operating system can control the clock frequency at run-time by writing to a register in the system control state exactly the way as in [3] except that the system does not need to wait for the voltage converter to generate the desired operating voltage. In sum we can assume that each processor can switch its operating voltage from one level to another instantaneously and independently with the power dissipation $P \propto CV_{dd}^2 f$ and gate delay $d \propto \frac{V_{dd}}{(V_{dd}-V_{th})^\alpha}$ at supply voltage V_{dd} and threshold voltage V_{th}, where $1 < \alpha \leq 2$ is a constant depends on the technology [4]. Furthermore, on a multiple voltage system, for a task under any time constraint, the voltage scheduling with at most two voltages minimizes the energy consumption and the task is finished just at its deadline [19].

In this paper, we consider the following problem:

For a given task graph G with its deadline $\mathcal{M}$ and completion ratio constraint $\mathcal{Q}_0$, find a scheduling strategy for a multi-processor multi-voltage system (a means of (1) assigning vertices to processors, (2) determining the execution order of vertices on the same processor, and (3) selecting the supply voltage for each processor) such that the energy consumption to satisfy the completion ratio constraint $\mathcal{Q}_0$ is minimized.

It is well-known that the variable voltage task scheduling for low power is in general NP-hard [9,19]. On the other hand, there exist intensive studies on multi-processor task scheduling problem with other optimization objectives such as completion time or IPC cost [17,22]. In this paper, We focus on developing on-line algorithms for voltage scaling (and voltage selection in particular) on a scheduled task graph. That is, we assume that tasks have already been assigned to processors and our goal is to determine *when* and *at which voltage* each task should be executed in order to minimize the total energy consumption while meeting the completion ratio constraint $\mathcal{Q}_0$.

3 Energy-Driven Voltage Scaling Techniques with Completion Ratio Constraint

In this section, we first obtain, with a simple algorithm, the best completion ratio on multi-processor system for a given task assignment. We then give a lower bound on the energy consumption to achieve the best completion ratio. Our focus will be on the development of on-line energy reduction algorithms that leverage the required completion ratio, which is lower than the best achievable.

3.1 $\mathcal{Q}^{max}$: The Highest Achievable Completion Ratio

Even when there is only one supply voltage, which results in a fixed processing speed, and each task has its own fixed execution time, the problem of determining whether a set of tasks can be scheduled on a multi-processor system to be completed by a specific deadline remains NP-complete (this is the *multiprocessor scheduling* problem [SS8], which is NP-complete for two processors [6].). However, for a given task assignment, the highest possible completion ratio can be trivially achieved by simply applying the highest supply voltage on all the processors. That is, each processor keeps on executing whenever there exist tasks assigned to this processor ready for execution;and stops when it completes all its assigned tasks in the current iteration or when the deadline $\mathcal{M}$ is reached. In the latter, if any processor has not finished its execution, we say the current iteration is *failed*; otherwise, we have a *successful completion* or simply *completion*. Clearly this naïve method is a best-effort approach in that it tries to complete as many iterations as possible. Since it operates all the processors at the highest voltage, the naïve approach will provide the highest possible completion ratio, denoted by $\mathcal{Q}^{max}$. In another word, if a completion ratio requirement cannot be achieved by this naïve approach within the given deadline $\mathcal{M}$, then no other algorithms can achieve it either.

When the application-specified completion ratio requirement $\mathcal{Q}_0 < \mathcal{Q}^{max}$, a simple counting mechanism can be used to reduce energy consumption. Specifically we cut the N iterations into smaller groups and shut the system down once sufficient iterations have been completed in each group. For example, if an MPEG application requires a 90% completion ratio, we can slow down the system (or switch the CPU to other applications) whenever the system has correctly decoded 90 out of 100 consecutive frames. This counting mechanism saves total energy by preventing the system from over-serving the application.

For system with multiple operating voltages, we mention that energy could have been saved over the above naïve approach in the following scenario: i) if we knew that an iteration would be completed earlier than the deadline $\mathcal{M}$, we could have processed with a lower voltage; and ii) if we knew that an iteration cannot be completed and have stopped the execution earlier. To save the maximal amount of energy, we want to *determine the lowest voltage levels to lead us to completions right at the deadline* $\mathcal{M}$ and *find the earliest time to terminate an incompletable iteration.* However, additional information about the task's execution time (e.g. WCET, BCET, and/or the probabilistic distribution) is required to answer these questions. In the rest of this section, we propose on-line voltage scaling techniques to reduce energy with the help of such information.

3.2 BEEM: Achieving $\mathcal{Q}^{max}$ with the Minimum Energy

The best-effort energy minimization (BEEM) technique proposed by Hua et al. gives the minimum energy consumption on a single processor system to provide the highest achievable completion ratio [10]. We extend this approach and propose algorithm BEEM1 for the multi-processor system and BEEM2 that does not assume tasks' execution time are known as a priori.

We define the *latest completion time* T_l^v and the *earliest completion time* T_e^v for a vertex v using the following recursive formulas:

$$T_e^v = T_l^v = \mathcal{M} \qquad \text{(if } v \text{ is a sink node)} \tag{3}$$

$$T_e^{v_i} = \min\{T_e^{v_j} - t_{j,k_j} - w_{v_i,v_j} | (v_i, v_j) \in E\} \tag{4}$$

$$T_l^{v_i} = \min\{T_l^{v_j} - t_{j,1} - w_{v_i,v_j} | (v_i, v_j) \in E\} \tag{5}$$

where $t_{j,1}$ and t_{j,k_j} are the BCET and WCET of vertex v_j, w_{v_i,v_j} is the cost of IPC from vertices v_i to v_j which is 0 if the two vertices are assigned to the same processor.

Lemma 1. If an algorithm minimizes energy consumption, then vertex v_i's completion time cannot be earlier than $T_e^{v_i}$.

[Proof]: Clearly such algorithm will complete each iteration at deadline $\mathcal{M}$. Otherwise, one can always reduce the operating voltage and processing speed

(or adjust the combination of two operating voltages) for the last task to save more energy.

Let t be vertex v_i's completion time at run time. If $t < T_e^{v_i}$, for any path from v_i to a sink node v, $u_0 = v_i, u_1, \cdots, u_k = v$, let $WCET_{u_j}$ be the worst case execution time of vertex u_j, then the completion time of this path will be

$$\begin{aligned}
T &\le t + \sum_{j=0}^{k-1}(w_{u_j,u_{j+1}} + WCET_{u_{j+1}}) < T_e^{v_i} + \sum_{j=0}^{k-1}(w_{u_j,u_{j+1}} + WCET_{u_{j+1}}) \\
&= T_e^{u_0} + w_{u_0,u_1} + WCET_{u_1} + \sum_{j=1}^{k-1}(w_{u_j,u_{j+1}} + WCET_{u_{j+1}}) \\
&\le T_e^{u_1} + \sum_{j=1}^{k-1}(w_{u_j,u_{j+1}} + WCET_{u_{j+1}}) \le \cdots \le T_e^{v} = \mathcal{M}
\end{aligned}$$

This implies that even when the WCET happens for all the successor vertices of v_i on this path, the completion of this path occurs before the deadline $\mathcal{M}$. Note that this is true for all the path, therefore the iteration finishes earlier and this cannot be the most energy efficient. Contradiction. ▯

Lemma 2. If vertex v_i's completion time $t > T_l^{v_i}$, then the current iteration is not completable by deadline $\mathcal{M}$.

[Proof]: Assuming that best case execution time occur for all the rest vertices at time t when v_i is completed, this gives us the earliest time that we can complete the current iteration and there exists at least one path from v_i to one sink node v ($u_0 = v_i, u_1, \cdots, u_k = v$), and for each pair (u_j, u_{j+1}) $T_l^{u_j} = T_l^{u_{j+1}} - BCET_{u_{j+1}} - w_{u_j,u_{j+1}}$. The completion of this path happens at time

$$\begin{aligned}
T &= t + \sum_{j=0}^{k-1}(w_{u_j,u_{j+1}} + BCET_{u_{j+1}}) > T_l^{v_i} + \sum_{j=0}^{k-1}(w_{u_j,u_{j+1}} + BCET_{u_{j+1}}) \\
&= T_l^{u_1} + \sum_{j=1}^{k-1}(w_{u_j,u_{j+1}} + BCET_{u_{j+1}}) = \cdots = T_l^{v} = \mathcal{M}
\end{aligned}$$

▯

Combining these two lemmas and the naïve approach that achieves the highest possible completion ratio $\mathcal{Q}^{max}$, we have:

Theorem 3. (BEEM1) If we know the execution time t_e^v of vertex v, the following algorithm achieves $\mathcal{Q}^{max}$ with the minimum energy consumption.

Let t be the current time that v is going to be processed and t_e^v be v's real execution time,

– if $t + t_e^v > T_l^v$, terminate the current iteration;
– if $t + t_e^v < T_e^v$, scale voltage such that v will be completed at T_e^v;
– otherwise, process at the highest voltage as in the naïve approach;

However, it is unrealistic to have each task's real execution time known as a priori, we hereby propose algorithm BEEM2, another version of BEEM that does not require tasks' real-time execution time to make decisions, yet still achieves the highest completion ratio $\mathcal{Q}^{max}$:

Algorithm BEEM2

Let t be the current time that v is going to be processed,
– if $t + BCET_v > T_l^v$, terminate the current iteration;
– if $t + WCET_v < T_e^v$, scale voltage such that $WCET_v$ will be completed at T_e^v;
– otherwise, process at the highest voltage;

Without knowing task's real execution time, BEEM2 conservatively i) terminates an iteration if it is incompletable even in vertex v's best case execution time $BCET_v$; and ii) slows down to save energy while still guaranteeing that vertex v's worst case execution time $WCET_v$ can be completed at its earliest completion time T_e^v. We mention that the pair $\{T_e^v, T_l^v\}$ can be computed offline only once and both BEEM1 and BEEM2 algorithms require at most two additions and two comparisons. Therefore, the on-line decision making takes constant time and will not increase the run time complexity. Finally, similar to our discussion for the naïve approach, further energy reduction is possible if the required completion ratio $\mathcal{Q}_0 < \mathcal{Q}^{max}$.

3.3 QGEM: Completion Ratio Guaranteed Energy Minimization

Both the naïve approach and BEEM algorithms achieve the highest completion ratio. Although they can also be adopted to provide exactly the required completion ratio $\mathcal{Q}_0$ for energy reduction, they may not be the most energy efficient way to do so when $\mathcal{Q}_0 < \mathcal{Q}^{max}$. In this section, we propose a hybrid offline on-line completion ratio $\mathcal{Q}$ guaranteed energy minimization (QGEM) algorithm, which consists of three steps:

In Step 1, we seek to find the minimum effort (that is, the least amount of computation t_s^i we have to process on each vertex v_i) to provide the required completion ratio $\mathcal{Q}_0$ (Fig. 1). Starting with the full commitment to serve every task's WCET (*line 2*), we use a greedy heuristic to lower our commitment the vertices along critical paths (*lines 6-13*). Vertex v_j is selected first if the reduction from its WCET t_{j,k_j} to t_{j,k_j-1} (or from the current $t_{j,l}$ to $t_{j,l-1}$) maximally shortens the critical paths and minimally degrades the completion ratio, measuring by their product (*line 10*).

The goal in Step 2 is to allocate the maximum execution time t_q^i for each task v_i to process the minimum computation t_s^i and to have the completion time L close to deadline $\mathcal{M}$ (Fig. 2). *Lines 3-9* repetitively scale t_q^i for all the tasks.

```
/* Step 1: Minimium effort for completion ratio guarantee. */
1. find a topological order of the vertices: $v_1, \cdots, v_n$;
2. $t_s^i = t_{i,k_i}$;                               /* assign WCET to each vertex */
3. $\mathcal{Q} = 1$;        /* completion ratio must be 1 if each vertex gets its WCET */
4. determine the completion time $L$;
5. while ($\mathcal{Q} > \mathcal{Q}_0$)
6. { for each vertex $v_j$ along critical paths;
7.    { determine the completion time $L'$ when reduces $t_s^j$ from its current $t_{j,l}$ to $t_{j,l-1}$;
8.      compute the completion ratio $\mathcal{Q}'_j = \mathcal{Q} \cdot \frac{P_{j,l-1}}{P_{j,l}}$;
9.    }
10.   pick the vertex $v_j$ that achieves the maximum gain $(L - L') \cdot \frac{P_{j,l-1}}{P_{j,l}}$;
11.   $\mathcal{Q} = \mathcal{Q} \cdot \frac{P_{j,l-1}}{P_{j,l}}$;
12.   if ($\mathcal{Q} > \mathcal{Q}_0$)    $t_s^j = t_{j,l-1}$ ;
13. }
```

Fig. 1. QGEM's offline part to determine the minimum commitment to provide $\mathcal{Q}_0$.

Because the IPCs are not scaled, maximally extending the allocated execution time to each task by a factor of $\mathcal{M}/L$ (*line 6*) may not stretch the completion time from L to $\mathcal{M}$. Furthermore, this unevenly extends each path and we re-evaluate the completion time (and critical path) at *line 7*. To prevent an endless repetition, we stop when the scale factor r is less than a small number ϵ (*line 5*), which is set as 10^{-6} in our simulation. *Lines 11-22* continue to scale t_q^i for vertices off critical paths in a similar way.

Now for vertex v_i, we have the pair (t_s^i, t_q^i) which represent the minimum amount of work and maximal execution time we commit to v_i. Define, recursively, the expected drop-time for v_i to be

$$D_i = t_q^i + \max\{D_k + w_{v_k,v_i} | (v_k, v_i) \in E\} \tag{6}$$

Step 3 defines the on-line voltage scheduling policy for the QGEM approach in Fig. 3, where we scale voltage to complete a task v_i by its expected drop-time D_i assuming that the real-time execution time requirement equals to the minimum workload t_s^i we have committed to v_i (*line 2*). If v_i demands more, it will be finished after D_i and we will drop the current iteration (*line 4*).

Note that if every task v_i has real execution time less than t_s^i in an iteration, QGEM's on-line scheduler will be able to complete this iteration. On the other hand, if longer execution time occurs at run-time, QGEM will terminate the iteration right after the execution of this task. From the way we determine t_s^i (in Fig. 1), we know that the required completion ratio $\mathcal{Q}_0$ will be guaranteed. Energy saving comes from two mechanisms: the early termination of *unnecessary* iterations (*line 5* in Fig. 3) and the use of low voltage to fully utilize the time from now to a task's expected drop-time (*line 2* in Fig. 3). We will confirm our claim on QGEM's completion ratio guarantee and demonstrate its energy efficiency by simulation in the next section.

```
/* Step 2: Maximum execution time allocation with deadline constraint.*/
1. for each vertex $v_i$
2.     $done(v_i) = 0$;   $t_q^i = t_s^i$;                /* allocate time $t_s^i$ to each vertex */
3. determine the completion time $L$;
4. $r = \frac{\mathcal{M}}{L} - 1$;
5. while ( $r \geq \epsilon$ )                          /* to prevent an endless loop */
6. { $t_q^i = t_q^i \cdot (1 + r)$;              /* scale the time allocated to each vertex */
7.     determine the completion time $L$;
8.     $r = \frac{\mathcal{M}}{L} - 1$;
9. }
10. for each vertex $v_i$ on critical paths     $done(v_i) = 1$;
11. while ($done(v_i) = 0$ for some vertex $v_i$)
12. { determine the completion time $L$;
13.   while ($L < \mathcal{M}$)
14.     { for each vertex $v_i$ with $done(v_i) = 0$
15.         $t_q^i = t_q^i \cdot (1 + \delta)$;               /* $\delta$ is a small positive number */
16.       determine the completion time $L$;
17.     }                  /* $L$ may exceed deadline $\mathcal{M}$, so we have to scale back $t_q^i$. */
18.   for each vertex $v_i$ with $done(v_i) = 0$
19.     { $t_q^i = t_q^i/(1 + \delta)$;
20.       if $v_i$ is on the critical path     $done(v_i) = 1$;
21.     }                        /* it is still possible to scale vertices off critical paths. */
22. }
```

Fig. 2. QGEM's offline part to allocate execution time for each task.

```
/* Step 3: On-line voltage scheduling. */
1. $t$ = current time when vertex $v_i$ is ready for processing;
2. scale voltage such that the fixed workload $t_s^i$ can be completed by time $D_i$;
3. execute task $v_i$ to its completion;
4. if the completion occurs later than $D_i$
5.     report *failure*; break and wait for the next iteration;
```

Fig. 3. On-line scheduling policy for algorithm QGEM.

4 Simulation Results

In this section we present the simulation results to verify the efficacy of our proposed approaches. We have implemented the proposed algorithms and simulated them over a variety of real-life and random benchmark graphs. Some task graphs, such as FFT(Fast Fourier Transform), Laplace(Laplace transform) and karp10 (Karplus-Strong music synthesis algorithm with 10 voices), are extracted from popular DSP applications. The others are generated by using *TGFF* [5], which is a randomized task graph generator. We assume that there are a set of homogeneous processors available. However, our approaches are general enough to be applied to embedded systems with heterogeneous multi-processors.

Before we apply our approaches to the benchmark graphs, we need to schedule all of tasks to available processors based on the performance such as latency. Here we use the dynamic level scheduling (DLS) [22] method to schedule the tasks, however, our techniques could be used with any alternative static scheduling strategy. The DLS method accounts for interprocessor communication overhead when mapping precedence graphs onto multiple processors in order to achieve the latency from the source to the sink as small as possible. We apply this method to the benchmarks and obtain the scheduling results which include the task execution order in each processor and interprocessor communication links and costs. Furthermore, we assume that the interprocessor communication is full-duplex and the intraprocessor data communication cost can be neglected.

After we obtain the results from DLS, we apply the proposed algorithms to them. There are several objectives for our experiments. First, we want to compare the energy consumption by using different algorithms under same deadline and completion ratio requirements. Secondly, we want to investigate the impact of completion ratio requirement and deadline requirement to the energy consumption of the proposed approaches. Finally, we want to study the energy efficiency of our algorithms with different number of processors.

We set up our experiments in the following way. For each task, there are three possible execution time, $e_0 < e_1 < e_2$, that occur at the following corresponding probabilities $p_0 >> p_1 > p_2$ respectively. All processors support real time voltage scheduling and power management (such as shut down) mechanism. Four different voltage levels, 3.3V, 2.6V, 1.9V, and 1.2V are available with threshold voltage 0.5V. For each pair of deadline $\mathcal{M}$ and completion ratio $\mathcal{Q}_0$, we simulate 1,000,000 iterations for each benchmark by using each algorithm. Because naïve, BEEM1 and BEEM2 all provide the highest possible completion ratio that is higher than the required $\mathcal{Q}_0$, in order to reduce the energy, we take 100 iterations as a group and stop execution once $100\mathcal{Q}_0$ iterations in the same group have been completed.

Table 3 reports the average energy consumption per iteration by different algorithms on each benchmark with deadline constraint $\mathcal{M}$ and completion ratio constraint $\mathcal{Q}_0(0.900)$. From the table we can see that both BEEM1 and BEEM2 provide the same completion ratio with an average of nearly **29%** and **26%** energy saving over naïve. Compared with BEEM2, BEEM1 saves more energy because it assumes that the actual execution time can be known a priori. However, without this assumption the QGEM approach can still save more energy than BEEM2 in most benchmarks. Specifically, it provides **36%** and **12%** energy saving over naïve and BEEM2 and achieves 0.9111 average completion ratio which is higher than the required completion ratio 0.9000. It is mentioned that for FFT2 benchmark, QGEM has negative energy saving compared to BEEM2 because the deadline $\mathcal{M}$ is so long that BEEM2 can scale down the voltage to execute most of the tasks and save energy.

Fig. 4 depicts the completion ratio requirement's impact to energy efficiency of different algorithms with same deadline $\mathcal{M}(9705)$. We can see that with the decrement of $\mathcal{Q}_0$, the energy consumption of each algorithm is decreased. However, different from naïve, BEEM1 and BEEM2, the energy consumption of QGEM doesn't change dramatically. Therefore, although under high completion

Table 3. Average energy consumption per iteration by naïve, BEEM1, BEEM2 and QGEM to achieve $\mathcal{Q}_0 = 0.900$ with deadline constraints $\mathcal{M}$. (n: number of vertices in the benchmark task graph; m: number of processors; $\mathcal{Q}$: the actual completion ratio achieved by QGEM without forcing the processors stop at $\mathcal{Q}_0$; energy is in the unit of the dissipation in one CPU unit at the reference voltage 3.3V.)

Benchmark	n	m	No. IPCs	$\mathcal{M}$	naïve	BEEM1	BEEM2	QGEM		
					energy	saving vs. naïve	saving vs. naïve	saving vs. naïve	saving vs. BEEM2	$\mathcal{Q}$
FFT1	28	2	15	1275	1040.4	6.78%	6.07%	35.71%	31.56%	0.9118
FFT2	28	2	16	2445	2122.4	18.15%	18.15%	15.96%	-2.67%	0.9104
Laplace	16	2	13	2550	1799.7	42.75%	32.63%	45.12%	18.53%	0.9232
karp10	21	2	12	993	592.8	23.44%	15.84%	50.54%	41.23%	0.9392
TGFF1	39	2	20	4956	4437.7	33.98%	30.75%	38.94%	11.82%	0.9090
TGFF2	51	3	36	4449	6102.8	34.20%	31.27%	34.36%	4.49%	0.9185
TGFF3	60	3	51	5487	8541.2	29.73%	27.01%	33.13%	8.39%	0.9034
TGFF4	74	2	49	9216	8838.9	32.08%	30.68%	38.67%	11.53%	0.9109
TGFF5	84	3	74	6990	11137.6	29.38%	27.85%	34.56%	9.31%	0.9065
TGFF6	91	2	59	11631	10799.3	33.23%	32.25%	41.16%	13.16%	0.9057
TGFF7	107	3	89	9129	13608.3	31.15%	29.71%	36.23%	9.28%	0.9027
TGFF8	117	3	111	9705	15674.0	28.30%	27.07%	34.51%	10.21%	0.9074
TGFF9	131	2	85	15225	15165.7	31.00%	30.31%	37.81%	10.77%	0.9084
TGFF10	147	4	163	10124	21925.8	30.09%	29.04%	31.69%	3.71%	0.9029
TGFF11	163	3	159	13068	22984.4	25.61%	24.95%	31.76%	9.08%	0.9100
TGFF12	174	4	169	12183	25220.2	29.89%	29.08%	33.35%	6.02%	0.9074
average						28.73%	26.42%	35.84%	12.28%	0.9111

ratio requirement ($\mathcal{Q}_0$ >0.75 in Fig. 4), using QGEM consumes the least energy, it may consume more energy than BEEM1, BEEM2 even naïve when $\mathcal{Q}_0$ is low.

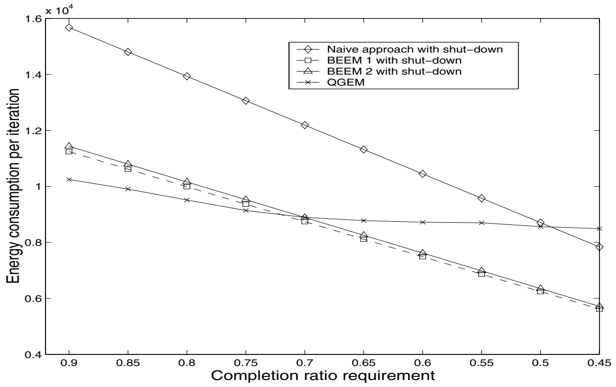


Fig. 4. Different completion ratio requirement's impact to the average energy consumption per iteration on benchmark TGFF8 with 3 processors.

The deadline requirement's impact to the energy consumption is shown in Fig. 5 with the same $\mathcal{Q}_0(0.900)$. Because the naïve approach operates at the highest voltage till the required $\mathcal{Q}_0$ is reached, when the highest possible completion ratio of the system is close to 1, its energy consumption keeps constant regardless of the change of the deadline $\mathcal{M}$. However in BEEM1 and BEEM2, the latest completion time T_l^v and the earliest completion time T_e^v for each vertex v depend on $\mathcal{M}$(see (3)-(5)), and the energy consumption will be reduced dramatically with the increment of $\mathcal{M}$. For QGEM, the increment of deadline also has positive effect on the energy saving while it is not as dramatic as it does to BEEM1 and BEEM2. Similar to the completion ratio requirement's impact, we conclude that QGEM consumes less energy than BEEM1 and BEEM2 in the short deadline (with the condition that $\mathcal{Q}_0$ is achievable), while consuming more energy when the deadline is long.

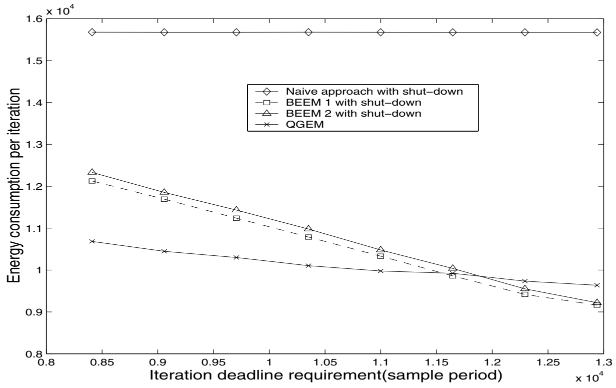


Fig. 5. Different deadline requirement's impact to the average energy consumption per iteration on benchmark TGFF8 with 3 processors.

From Table 3 and Fig. 4-5, we can conclude that QGEM save more energy than BEEM1 and BEEM2 when $\mathcal{Q}_0$ is high and $\mathcal{M}$ is not too long. Actually this conclusion is valid regardless of the number of multiple processors. Fig. 6 shows the energy consumption of different algorithms under different deadlines and different number of processors. With the increment of the number of processors, its latency will be reduced. So for the same deadline(e.g., 7275), it is not relatively long and QGEM saves more energy than BEEM1 and BEEM2 for the system with small number of processors (e.g., 4 processors), however, for the system with large number of processors(e.g., >5 processors), QGEM will consume more energy than BEEM1 and BEEM2.

5 Conclusions

Many embedded applications, such as multimedia and DSP applications, have high performance requirement yet are able to tolerate certain level of execution

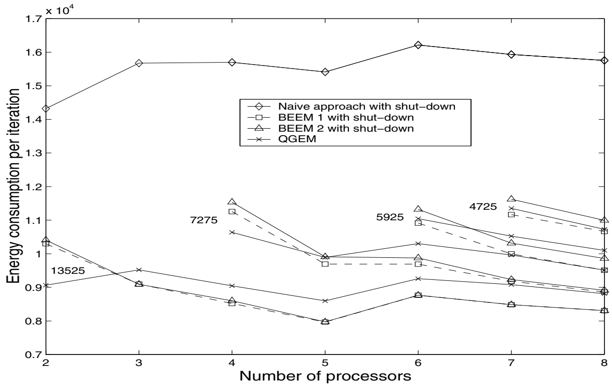


Fig. 6. The average energy consumption per iteration on benchmark TGFF8 with different number of processors and different deadlines(13525, 7275, 5925 and 4725).

failures. We investigate how to trade this tolerance for energy efficiency, another increasingly important concern in the implementation of embedded software. In particular, we consider systems with multiple supply voltages that enable dynamic voltage scaling, arguably the most effective energy reduction technique. We present several on-line scheduling algorithms that scale operating voltage based on some parameters pre-determined offline. All the algorithms have low run-time complexity yet achieve significant energy saving while providing the required performance, measured by the completion ratio.

References

1. N. K. Bambha and S. S. Bhattacharyya. "A Joint Power/Performance Optimization Technique for Multiprocessor Systems Using a Period Graph Construct", *Proceedings of the International Symposium on System Synthesis*, pp. 91–97, 2000.
2. J. Bolot and A. Vega-Garcia. "Control Mechanisms for Packet Audio in the Internet", *Proceedings of IEEE Infocom*, pp. 232–239, March 1996.
3. T.D. Burd, T. Pering, A. Stratakos, and R. Brodersen. "A Dynamic Voltage Scaled Microprocessor System", *IEEE J. Solid-State Circuits*, Vol. 35, No.11, pp. 1571–1580, November 2000.
4. A.P.Chandrakasan, S.Sheng, and R.W.Broderson. "Low-Power CMOS Digital Design", *IEEE Journal of Solid-State Circuits*, Vol. 27, No. 4, pp. 473–484, 1992.
5. R. P. Dick, D. L. Rhodes, and W. Wolf. "TGFF: Task Graphs for Free", *Proc. Int. Workshop Hardware/Software Codesign*, pp. 97–101, Mar. 1998.
6. M.R.Garey and D.S.Johnson. *Computer and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman and Company, New York, NY, 1979.
7. A. Grbic, S. Brown, S. Caranci, et al. "Design and Implementation of the NUMAchine Multiprocessor", *35th ACM/IEEE Design Automation Conference*, pp. 65–69, June 1998.
8. F. Gruian and K. Kuchcinski. "LEneS: Task Scheduling for Low-Energy Systems Using Variable Supply Voltage Processors" *Proc. of Asia and South Pacific Design Automation Conference*, pp. 449–455, 2001

9. I. Hong, M. Potkonjak, and M.B. Srivastava. “On-Line Scheduling of Hard Real-Time Tasks on Variable Voltage Processor”, *IEEE/ACM International Conference on Computer Aided Design*, pp. 653–656, 1998.
10. S. Hua, G. Qu, and S. S. Bhattacharyya. “Energy Reduction Techniques for Multimedia Applications with Tolerance to Deadline Misses”, *40th ACM/IEEE Design Automation Conference*, pp. 131–136, June 2003.
11. R. S. Janka and L. M. Wills. “A Novel Codesign Methodology for Real-Time Embedded COTS Multiprocessor-Based Signal Processing Systems”, *Proceedings of the International Workshop on Hardware/Software Co-Design*, pp. 157–161, 2000.
12. I. Karkowski and H. Corporaal. “Design Space Exploration Algorithm For Heterogeneous Multi-processor Embedded System Design”, *35th ACM/IEEE Design Automation Conference*, pp. 82–87, June 1998.
13. M. J. Karam and F. A. Tobagi. “Analysis of the Delay and Jitter of Voice Traffic Over the Internet”, *Proceedings of IEEE Infocom*, pp. 824–833, April 2001.
14. J. Luo and N. K. Jha. “Power-Conscious Joint Scheduling of Periodic Task Graphs and Aperiodic Tasks in Distributed Real-Time Embedded Systems”, *IEEE/ACM International Conference on Computer-Aided Design*, pp. 357–364, November 2000.
15. J. Luo and N. K. Jha. “Battery-Aware Static Scheduling for Distributed Real-Time Embedded Systems”, *38th ACM/IEEE Design Automation Conference*, pp. 444–449, June 2001.
16. J. Luo and N. K. Jha. “Static and Dynamic Variable Voltage Scheduling Algorithms for Real-Time Heterogeneous Distributed Embedded Systems”, *Proc. of Asia and South Pacific Design Automation Conference*, pp. 719–726, Jan. 2002.
17. C.L. McCreary, A.A. Khan, J.J. Thompson, and M.E. McArdle. “A Comparison of Heuristics for Scheduling DAGs on Multiprocessors”, *Proceedings of the International Parallel Processing Symposium*, pp. 446–451, April 1994.
18. R. Mishra, N. Rastogi, D. Zhu, D. Mosse, and R. Melhem. “Energy Aware Scheduling for Distributed Real-Time Systems”, *International Parallel and Distributed Processing Symposium*, Apr. 2003.
19. G. Qu. “What is the Limit of Energy Saving by Dynamic Voltage Scaling?” *IEEE/ACM International Conference on Computer-Aided Design*, pp. 560–563, November 2001.
20. D. Scherrer and H. Eberle. “A Scalable Real-time Signal Processor for Object-oriented Data Flow Applications”, *Proceedings of the International Conference on Parallel and Distributed Computing Systems*, pp. 183–189, September 1998.
21. M. T. Schmitz and B. M. Al-Hashimi. “Considering Power Variations of DVS processing elements for energy minimisation in distributed systems”, *Proceedings of 14th International Symposium on System Synthesis*, pp. 250–255, 2001.
22. G.C. Sih and E.A. Lee. “A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures”, *IEEE Tran. on Parallel and Distributed Systems*, Vol. 4, No. 2, February 1993.
23. R.A. Sutton, V.P. Srini, and J.M. Rabey. “A Multiprocessor DSP System Using PADDI-2”, *35th ACM/IEEE Design Automation Conference*, pp. 62–65, June 1998.
24. T.S. Tia, Z. Deng, M. Shankar, M. Storch, J. Sun, L.-C. Wu, and J.W.-S. Liu. “Probabilistic Performance Guarantee for Real-Time Tasks with Varying Computation Times”, *Proc. Real-Time Technology and Applications Symposium*, pp. 164–173, 1995
25. V. Tiwari, D. Singh, S. Rajgopal, G. Mehta, R. Patel, and F. Baez. “Reduced Power in High-Performance Microprocessors”, *35th ACM/IEEE Design Automation Conference* pp. 732–737, June 1998

26. D. Zhu, R. Melhem and B. Childers. “Scheduling with Dynamic Voltage/Speed Adjustment Using Slack Reclamation in Multi-Processor Real-Time Systems”, *IEEE 22nd Real-Time Systems Symposium* pp. 84–94, 2001

A Methodology and Tool Support for Generating Scheduled Native Code for Real-Time Java Applications[⋆]

Christos Kloukinas, Chaker Nakhli, and Sergio Yovine

VERIMAG Centre Equation, 2 Ave. Vignate, 38610 Gières, France
{Christos.Kloukinas,Chaker.Nakhli,Sergio.Yovine}@imag.fr

1 Introduction

Current trends in industry are leading towards the use of Java [5] as a programming language for implementing embedded and real-time applications. From the software engineering perspective, the Java environment is indeed a very attractive development framework. Object-oriented programming provides encapsulation of abstractions into objects that communicate through clearly defined interfaces. Dynamic loading eases the maintenance and improvement of complex applications with evolving requirements and functionality. Besides, Java provides built-in support for multi-threading.

However, the semantics of Java do not guarantee a predictable run-time behavior, which is an essential issue in embedded real-time software. To overcome this problem, work has been done to extend the language and the platform to accommodate to the requirements of real-time systems by focusing on current practices. Among such work, we should mention the Real-Time Specification for Java [11], and the Real-Time Core Extension for the Java Platform [12], that provide support for real-time programming (timers, clocks, handlers, priorities, ...). Still, these extensions leave some important issues unspecified, like the scheduling algorithm to be used, allowing an implementation to resolve them at will.

In order to obtain more precise semantics, domain-specific profiles have also been defined, such as the Ravenscar-Java [8] high-integrity profile for safety-critical systems. This profile settles an execution model based on a two/three-phase program execution, comprising an initialization phase and a mission phase (possibly followed by a termination phase), and multi-threading semantics relying on fixed priority preemptive scheduling and priority ceiling inheritance. Though designed to ease analysis and programming, this profile still has some drawbacks. For instance, it does not directly support threads which synchronize and communicate using the `wait` and `notify` methods of Java. Besides, the underlying schedulability analysis is pessimistic by nature and not well adapted for systems with heterogeneous tasks and constraints.

The other important issue is performance. Though there are efforts to produce efficient implementations of Java Virtual Machines (*e.g.*, [1,15]), the slowdown due to the VM remains an argument against adopting Java for real-time applications. Real-time systems can afford neither the overhead nor the non-determinism of using a Just-In-Time (JIT) compiler (*e.g.*, [3,14]). An alternative approach consists in using an Ahead-of-Time (AOT) compiler (*e.g.*, [10,17]) to generate executable code for a run-time system and to

[⋆] Partially supported by the RNTL project Expresso (`http://www.irisa.fr/rntl-expresso`).

provide a native implementation of the real-time primitives. A major advantage of AOT compilation is that it allows performing sophisticated analysis techniques to produce highly optimized code.

In this paper we present an approach that takes into account the demands of both precise semantics and performance. Our work is based on the two/three-phase execution model and API of the Expresso High-Integrity Profile [4], which itself inherits concepts from the Ravenscar-Java profile and the RTSJ API. However, the semantics proposed by the profile do not fit the needs of applications that would demand alternative scheduling and synchronization paradigms, and handling quality of service requirements. To accommodate to such demands, our approach focuses primarily on the application.

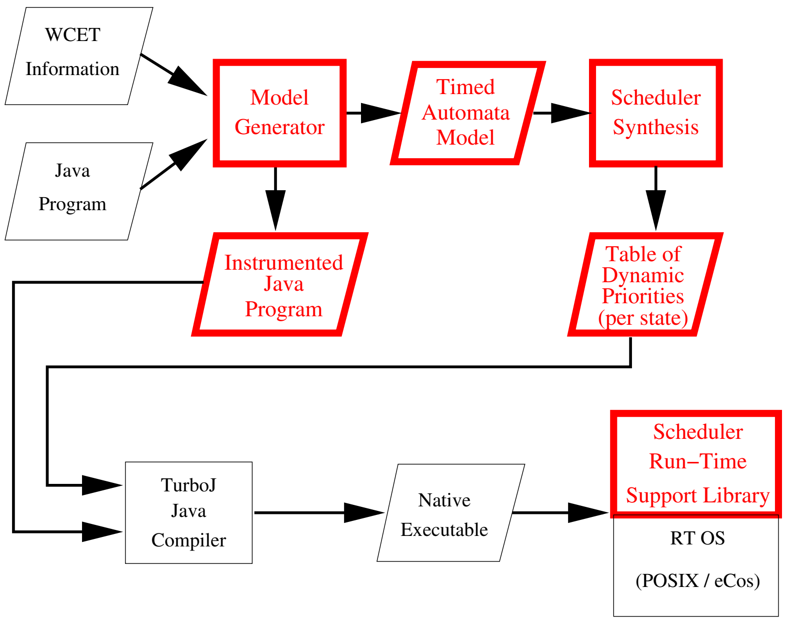


Fig. 1. Code analysis & generation chain

Following [13], we first extract a formal model of the behavior of the application program as an extended automaton (see Fig. 1 & section 2). Then, we synthesize an application-dependent scheduler (see sections 3–4) which is *safe* (*i.e.*, it is deadlock free and meets all timing constraints) and *QoS extendible* (*i.e.*, it can be extended to handle QoS requirements, such as reducing response jitter, power consumption or context switches). This synthesized scheduler is meant to be used with an appropriate native run-time support we have developed, which itself uses the underlying *R-T OS*'s primitives (see section 5). Our scheduler also needs an instrumented version of the original Java code (also produced by our model extracting tool), so as to be able to follow the changes of thread states (*i.e.*, the instrumentation implements an abstract program counter). The test-bed we have developed has been integrated together with the Expresso High-Integrity API and the TurboJ [17] Java-to-native compiler.

This framework provides a complete analysis and compilation chain for embedded real-time systems based on Java, allowing one to substitute RMA/EDF & PCP with a scheduler which is still safe but not as pessimistic. In this article, we describe the model generator, the scheduler architecture and synthesis methodology, and the prototype test-bed implemented using POSIX [6] primitives.

We will illustrate our approach throughout the paper with a case study inspired by the robotic arm system described in [9] (see Fig. 2). The arm is programmed to take objects from a conveyor belt, to store them in a buffer shelf, and to put them into a basket. The arm is controlled by threads running on a single processor. The `TrajectoryControl` thread reads commands from a shared buffer and issues set-points to the low-level arm controller `Controler`. If there are no commands to process, it holds, otherwise reads sensor values and computes the new set-point. Its execution time is between 5ms and 6ms. The `Lifter` thread is activated periodically every 40ms. Its role is to command the arm to pick objects from the conveyor belt. Upon termination, it issues a command to the `TrajectoryControl` and activates the `Putter`. Its execution time is between 4ms to 8ms. The `Putter` thread sends commands to take the object from the buffer shelf and put it into the basket. Its execution time is between 4ms to 8ms. The `SensorReader` thread reads sensors every 24ms. Its execution time is 1ms. The results of the sensors are used by `TrajectoryControl`. `Controler` is a periodic thread with period 16ms. Notice in Fig. 2 how, according to the Expresso HIP-API, **waitForPeriod** returns a boolean value which is **false** if the thread misses its period. In such a case, the application ends the mission phase and goes into a termination phase which is omitted here. In this paper we only consider the problem of synthesizing a scheduler for the mission phase.

The paper is organized as follows. Section 2 presents the technique we use to generate models from Java source code. Section 3 explains the scheduler architecture and execution semantics, while section 4 describes our methodology for synthesizing an application-dependent scheduler. Section 5 discusses our test-bed implementation.

2 Model Generation

We consider real-time Java applications made up of a fixed number of threads that synchronize and communicate through a fixed set of global shared objects. There is a distinguished thread, namely `Main`, which is the first thread to wake up at application startup and the unique entry-point of the application. We assume that `Main` is programmed according to the Expresso HIP, that is, all the shared objects and threads are created during the initialization phase.

The model of a Java program is a transition system which abstracts program actions and states. Each state in the model is an abstraction of a program state at run-time. Transitions are associated with source code and capture the change that its execution makes to the program state. More formally, let $\Theta = \{th_i\}$ be a finite set of threads, $\Omega = \{O_i\}$ be a finite set of shared objects, and A be a set of labels. Labels may correspond to large blocks of source code or to specific statements, such as: **lock** (corresponding to the Java-bytecode **monitorEnter**), **unlock** (**monitorExit**), **wait**, **waitTimed** (the Java

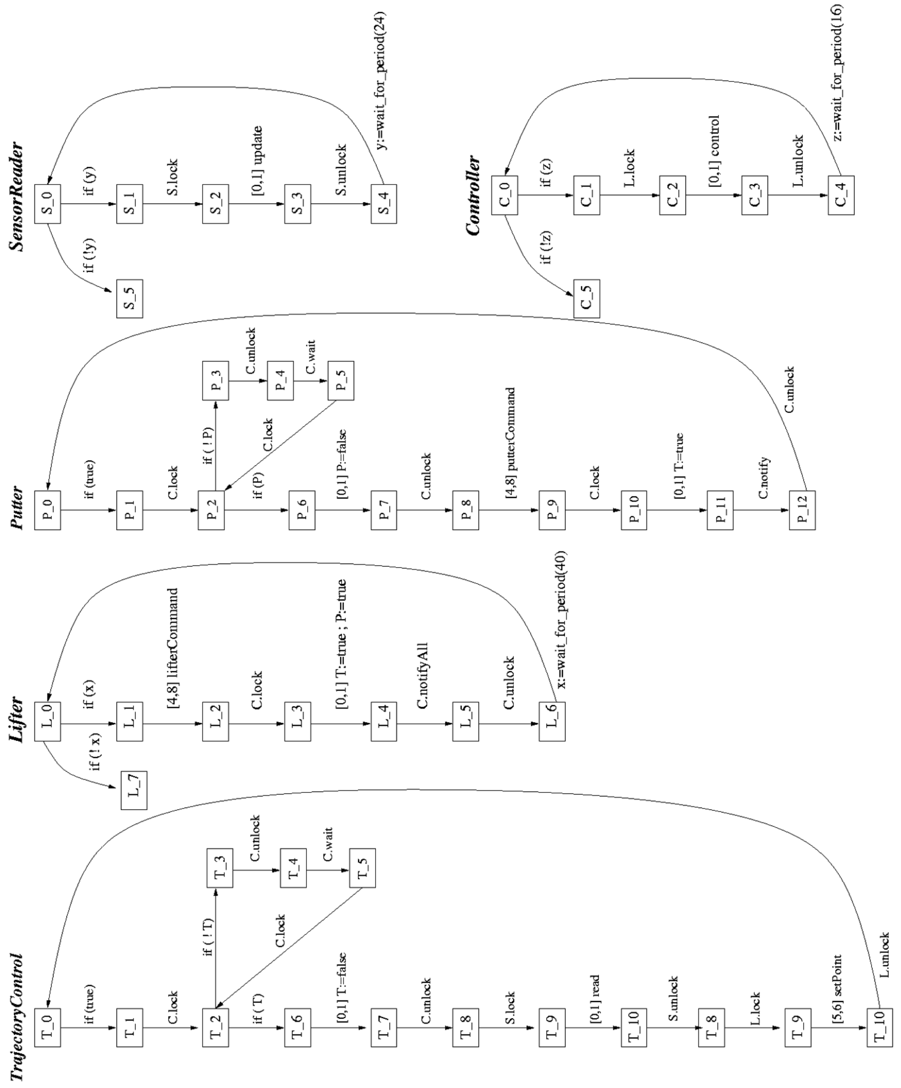


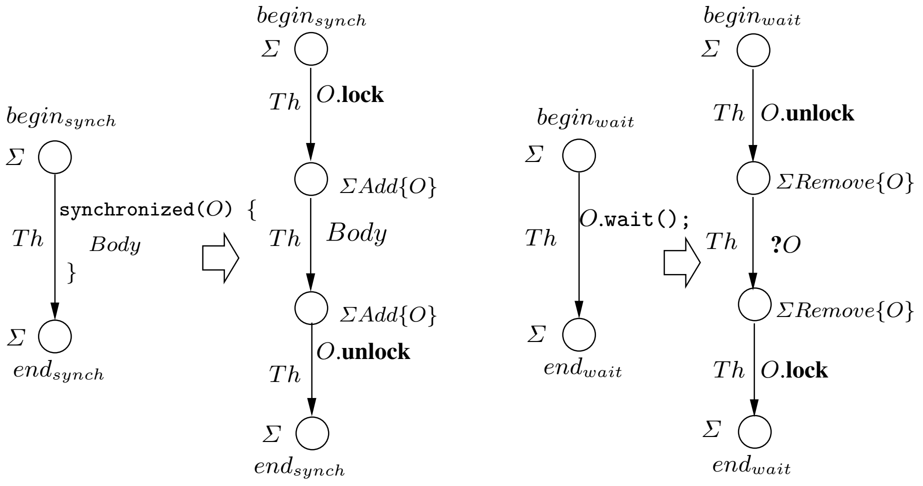
Fig. 2. Robotic arm system architecture
Intervals preceding computations give their minimum & maximum execution duration. Transitions `T_9` → `T_10` (`setPoint`), `L_1` → `L_2`, `P_8` → `P_9`, `S_2` → `S_3` and `C_2` → `C_3` correspond to code which has been sliced away.

wait method with a timeout parameter), and **waitForPeriod** (the method of the class `PeriodicThread` in the Expresso profile, which blocks a thread until its next period). The model is a tuple $P = (S, A, T)$ where: S is a finite set of states, and $T \subseteq S \times A \times S$ is a transition relation. We define $TH : T \longrightarrow \Theta$ to be the function mapping each transition to the corresponding executing thread.

For each thread th we define $\Sigma_{th} : S \longrightarrow 2^{\Omega} \times 2^{\Omega}$, where $\Sigma_{th}(s) = (\Sigma^{+}, \Sigma^{-})$ is called the *synchronization context* of th at the state s. Σ^{+} contains the set of objects which are locked by th, when th is at state s. Σ^{-} is either the empty set or a singleton containing the object that cannot be synchronized by th at s (and thus cannot be added to Σ^{+}) which corresponds to the lock released when **wait** or **waitTimed** are invoked on that object. This is done to keep the synchronization context consistent during the model extraction process. Let $\Sigma = (\Sigma^{+}, \Sigma^{-})$ be a synchronization context, and $\omega \subset \Omega$ a set of global objects. We define:

$$\Sigma \; Add \, \omega = (\Sigma^{+} \cup (\omega \setminus \Sigma^{-}), \Sigma^{-})$$
$$\Sigma \; Remove \, \omega = (\Sigma^{+} \setminus \omega, \Sigma^{-} \cup \omega)$$

The Add operation appends objects to the synchronization context by adding them to the Σ^{+} set, as long as these objects are not in the Σ^{-} set. The $Remove$ operation removes objects from the set of locked ones, and records them in the Σ^{-} set.



(a) `synchronized` block (b) `wait()` statement

Fig. 3. Graph rewrite rules for the generation of models

Our model generator constructs a model from the source code by applying graph rewriting rules on the control flow graph. A segment P of sequential source code is modeled by a state labelled $begin_P$ denoting the control state preceding the execution of P, a transition labelled by P, and a state labelled end_P denoting the control state following P. The translation could be kept at this abstraction level or may be refined recursively. Therefore, the granularity of the model can be controlled by the designer. The translation of control-flow statements (*e.g.*, ;, **if-then-else**, **while**, ...) is done according to standard rewriting rules. Synchronization statements are treated specially though. The `synchronized` statement (characterized by the requested object O) is translated as shown in Fig. 3(a). Entry in the `synchronized` block is modeled by a transition labeled with a **lock** on object O, while the exit is modeled by a transition labeled with an **unlock**. The fact that the thread Th holds the lock of O is recorded by adding $\{O\}$ to the Σ of the `synchronized` statement. An invocation of **wait** on O is translated by three transitions (Fig. 3(b)): one modeling the lock release and leading to a waiting state, another labelled by a *reception-of-notification* action, and a final transition modeling the lock request. These graph-rewrite rules allow us to obtain an extended automaton at the desired level of abstraction for each application thread. The information encoded in the synchronization context Σ of each state of these automata is used for informing the scheduler synthesis program about the resources which are used at the states of a thread. They are also used for constructing a *resource allocation graph* which is subsequently used for deriving a set of initial constraints against the deadlocks of the system. These constraints can be used as an initial scheduler, so as to decrease the possible behaviors of the system.

3 Scheduler Architecture and Semantics

3.1 Architecture

The architecture of the schedulers we synthesize consists of two three-layered stacks [7], as shown in Fig. 4. The left stack selects a thread for execution. The right stack selects a thread for the reception of a notification. Being able to control which thread will be notified for a particular event is something that other scheduling policies do not offer, since they concentrate only on the selection of threads for execution. After one of the scheduler stacks is finished, it passes control to an underlying *R-T OS* which provides low-level kernel mechanisms such as thread creation, suspension and resumption, and timeout enabling/disabling.

Controlling the Executing Thread. The left stack takes control of the system when the application calls one of **lock**, **unlock**, **waitForPeriod**, **wait** and **waitTimed**. In these cases, it must choose one of the available threads as the thread which should be executed next. It does this in three steps, each one performed at a different layer. In the first layer, referred to as the *Ready-Exec* scheduler, it calculates the set of threads $\mathcal{R}_{\text{exec}}$ which are ready to execute without directly blocking due to mutual exclusion. The *Safe-Exec* scheduler layer is responsible for calculating the subset $\mathcal{S}_{\text{exec}}$ of $\mathcal{R}_{\text{exec}}$, consisting of those threads which can *safely* execute, that is, their execution will not cause a deadlock nor a deadline miss. The third layer *QoS-Exec* calculates the set $\mathcal{Q}_{\text{exec}} \subseteq \mathcal{S}_{\text{exec}}$, consisting of those *safe* threads which also respect the QoS requirements of the application.

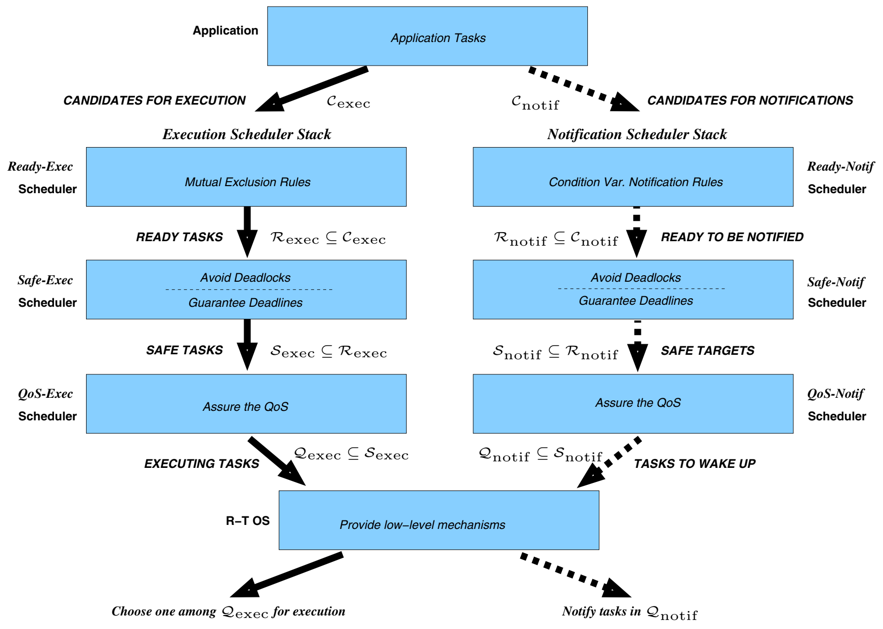

Application
Application Tasks
CANDIDATES FOR EXECUTION
$\mathcal{C}_{\mathrm{exec}}$
$\mathcal{C}_{\mathrm{notif}}$
CANDIDATES FOR NOTIFICATIONS
Execution Scheduler Stack
Notification Scheduler Stack
Ready-Exec Scheduler
Mutual Exclusion Rules
Condition Var. Notification Rules
Ready-Notif Scheduler
READY TASKS
$\mathcal{R}_{\mathrm{exec}} \subseteq \mathcal{C}_{\mathrm{exec}}$
$\mathcal{R}_{\mathrm{notif}} \subseteq \mathcal{C}_{\mathrm{notif}}$
READY TO BE NOTIFIED
Safe-Exec Scheduler
Avoid Deadlocks
Guarantee Deadlines
Avoid Deadlocks
Guarantee Deadlines
Safe-Notif Scheduler
SAFE TASKS
$\mathcal{S}_{\mathrm{exec}} \subseteq \mathcal{R}_{\mathrm{exec}}$
$\mathcal{S}_{\mathrm{notif}} \subseteq \mathcal{R}_{\mathrm{notif}}$
SAFE TARGETS
QoS-Exec Scheduler
Assure the QoS
Assure the QoS
QoS-Notif Scheduler
EXECUTING TASKS
$\mathcal{Q}_{\mathrm{exec}} \subseteq \mathcal{S}_{\mathrm{exec}}$
$\mathcal{Q}_{\mathrm{notif}} \subseteq \mathcal{S}_{\mathrm{notif}}$
TASKS TO WAKE UP
R–T OS
Provide low–level mechanisms
Choose one among $\mathcal{Q}_{\mathrm{exec}}$ for execution
Notify tasks in $\mathcal{Q}_{\mathrm{notif}}$

Fig. 4. Scheduler architecture

Controlling the Notified Thread. The right stack is passed control when the application calls **notify** or **notifyAll**. The reason for this is that the threads which will be notified (if any) cannot ever be selected for execution. This is because they will immediately try to re-enter the monitor after being notified and thus get blocked by the notifier (which is already in the monitor). Nevertheless, we can control which among the threads waiting for the notification should receive the notification, if there are more than one threads waiting and we are not performing a **notifyAll**. Thus, the top layer *Ready-Notif* calculates the set $\mathcal{R}_{\mathrm{notif}}$ of threads waiting on the condition variable being notified. The middle layer *Safe-Notif*, calculates the subset $\mathcal{S}_{\mathrm{notif}}$ of $\mathcal{R}_{\mathrm{notif}}$ consisting of those threads which, if notified, will not cause the system to enter into a deadlock state or to miss a deadline. The *Safe-Notif* layer passes the $\mathcal{S}_{\mathrm{notif}}$ set to the bottom *QoS-Notif* layer, which calculates the subset $\mathcal{Q}_{\mathrm{notif}}$ of $\mathcal{S}_{\mathrm{notif}}$, consisting of the threads which can be safely notified and also respect the QoS requirements.

QoS Policies. The complexity of the QoS layer is controlled by the application designer. In choosing a QoS policy (or policies, since these are composable) the designer can balance between the execution time and extra memory space needed by the policy and the gains to the overall system quality the particular policy can offer. A QoS policy is, for example, the *minimization of the response jitter of some thread* (*e.g.*, if it controls a physical device), or the *local minimization of context switches* (LMCS) in order to speed-up the execution and (hopefully) minimize cache misses/flushes and, thus, also energy consumption. This latter policy can be implemented quite easily, since all one needs to examine is whether the currently executing thread T_{Exec} is in the set $\mathcal{S}_{\text{exec}}$ of threads which are safe to execute next. If this is the case, then we can let it continue its execution, by setting the set $\mathcal{Q}_{\text{exec}}$ equal to the singleton $\{T_{\text{Exec}}\}$.

Other, more complex policies may take their decisions by examining application variables and/or (parts of) the execution history.

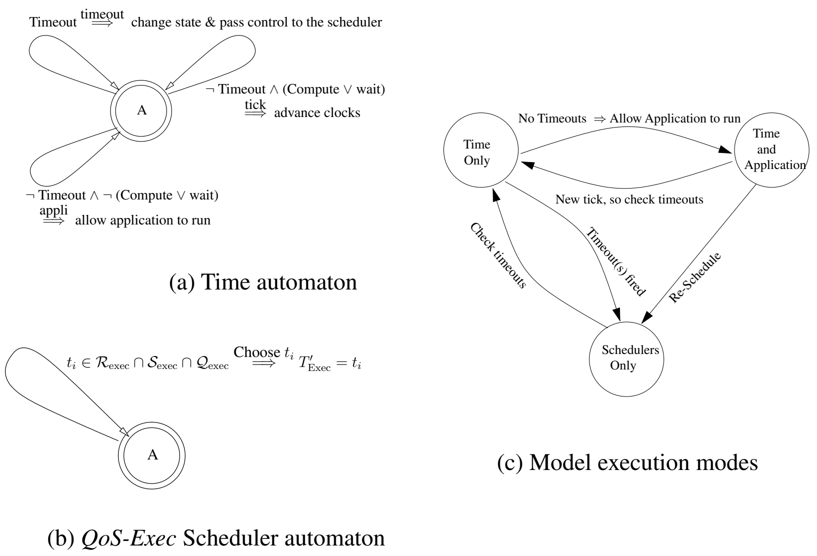
3.2 Semantics

The model of the system we construct is the parallel composition of: *(i)* an automaton which is responsible for advancing time and firing timeouts, *(ii)* one automaton for each of the application threads, and *(iii)* two more automata, for the *QoS-Exec* and the *QoS-Notif* scheduler layers respectively. The application automata are those obtained from the Java source code, appropriately annotated with the timing constraints modeling the execution times of the code that has been abstracted away.

The state of the system model comprises of:

– a program counter (PC_i) for each of the application threads,
– a local clock (C_i) for each thread, which is used for their computations and the timeouts if they execute a **waitTimed**,
– a global clock (C_{System_i}) for modeling the periods of each periodic thread,
– a variable (T_{Exec}) holding the currently executing thread,
– two boolean variables (*Exec_Sched_Enabled* & *Notif_Sched_Enabled*) for controlling whether it is one of the scheduler automata (and which one of them), or the time (when they are both **true**) or the time and application automata (when they are both **false**) which should execute, and
– the *boolean* variables of the application threads used in conditionals associated with waiting statements.

The system goes through three different modes of execution, as shown in Fig. 5(c). In the "Time Only" mode (where *Exec_Sched_Enabled* $=$ *Notif_Sched_Enabled* $=$ **true**) the automaton responsible for advancing time and firing timeouts (shown in Fig. 5(a)) is the sole automaton enabled in the system and it can fire one or more timeouts, if any is enabled, corresponding to the expiration of a **waitTimed** or **waitForPeriod**. If a timeout is fired then the execution mode changes to "Schedulers Only" (where *Exec_Sched_Enabled* $= \neg$*Notif_Sched_Enabled*), so that our scheduler can handle it. If there is no timeout to be fired then the execution mode changes to "Time and Application" (where *Exec_Sched_Enabled* $=$ *Notif_Sched_Enabled* $=$ **false**). At this mode, both the time automaton and the automata of the application are enabled. If the application automaton needs to execute a time guarded action (*i.e.*, a computation), then it blocks, allowing time to advance. The time automaton then performs a tick (*i.e.*, a time step) and we pass back to the "Time Only" mode, so as to check if a timeout has now been enabled. If, however, an application automaton needs to perform an action which causes re-scheduling, then it passes control back to the schedulers (*i.e.*, the mode now becomes "Schedulers Only").


Timeout $\overset{\text{timeout}}{\Longrightarrow}$ change state & pass control to the scheduler
A
¬ Timeout ∧ (Compute ∨ wait) $\overset{\text{tick}}{\Longrightarrow}$ advance clocks
¬ Timeout ∧ ¬ (Compute ∨ wait) $\overset{\text{appli}}{\Longrightarrow}$ allow application to run
$t_i \in \mathcal{R}_{\text{exec}} \cap \mathcal{S}_{\text{exec}} \cap \mathcal{Q}_{\text{exec}} \overset{\text{Choose } t_i}{\Longrightarrow} T'_{\text{Exec}} = t_i$
A
Time Only
Time and Application
Schedulers Only
No Timeouts ⇒ Allow Application to run
New tick, so check timeouts
Check timeouts
Timeout(s) fired
Re-Schedule

(a) Time automaton

(b) *QoS-Exec* Scheduler automaton

(c) Model execution modes

Fig. 5. Time & Scheduler automata and system execution modes

4 Scheduler Synthesis

In order to synthesize the *Safe-Exec* and *Safe-Notif* scheduler layers, we first construct the set of reachable states and, thus, identify the deadlocks. These are the states where the application threads are deadlocked, or the states where some thread has missed its deadline or period (since in that case we block the system explicitly). The existence of these states indicates that the predicate we are currently using to describe the set $\mathcal{S}_{\text{exec}}$ (resp. $\mathcal{S}_{\text{notif}}$) needs to be constrained even further. This predicate starts with the value of **true**, thus accepting initially all threads in the set $\mathcal{R}_{\text{exec}}$ (resp. $\mathcal{R}_{\text{notif}}$) as safe, where $\mathcal{R}_{\text{exec}}$ has been calculated during the extraction of the model from the source code. Having obtained the deadlocked states, we do a backwards traversal of the whole state space starting from the deadlocked states, until we reach a state which corresponds to a choice of one of the scheduler automata. There, we identify the choice $T_{\text{Exec}} = t_i$ which allowed the path leading to a deadlock state(s) and create a new constraint for the layer *Safe-Exec* (resp. *Safe-Notif*). This constraint is constructed by changing the set $\mathcal{S}_{\text{exec}}$ (resp. $\mathcal{S}_{\text{notif}}$) to be:

$$\mathcal{S}'_{\text{exec}}(\overrightarrow{state}) = \mathcal{S}_{\text{exec}}(\overrightarrow{state}) \setminus \{t_i\}$$

If at some point we find that $\mathcal{S}'_{\text{exec}}(\overrightarrow{state})$ is equal to the empty set, then we add the current state to the set of deadlocks and continue the synthesis procedure.

4.1 State-Space Reduction and Application Analysis

Even though the basic idea of synthesizing the *Safe-Exec* and *Safe-Notif* scheduler layers is simple, it is evident that in practice it suffers from the state explosion problem.

Therefore, it is imperative that we use techniques to minimize the size of the state space. Our method consists of synthesizing schedulers for successively more detailed models, adding thus complexity to a model only when we have already calculated how we can constrain the more abstract one. The scheduler synthesis is performed in five major steps.

Compositional Synthesis. First, we decompose the system and synthesize constraints independently for each of the components. We then apply the synthesis algorithm again on the parallel composition of the already constrained models. In the case study, we decided to decompose the application in two sub-systems, one consisting of 4 threads, namely, `Lifter`, `Putter`, `SensorReader`, and `TrajectoryControl`, and another one comprising the `Controler` thread. The decision was mainly taken because of the size of the corresponding models (see Table 1). This approach allows us to start the synthesis with a model about 84% smaller than the original one.

Table 1. Model abstractions and optimizations

In the "original" models the `IDLE` *task is allowed to execute only when no other task is safe. Otherwise, the state-spaces explode - the 5-thread system has more than 404 M states.*

Model kind	**States**	**Red.**	**Trans.**	**Red.**	**Dead.**
Model Abstractions & Optimizations for 4 threads					
T *original (***i.e.,** ***P)***	92382	0.00%	103658	0.00%	61
U	3320	96.41%	4680	95.49%	0
T ***NP***	74650	19.19%	83018	19.91%	61
T ***P, bbe***	20866	77.41%	25020	75.86%	1
T ***NP, bbe***	18304	80.19%	21738	79.03%	1
Model Abstractions & Optimizations for 5 threads					
T *original (***i.e.,** ***P)***	600086	0.00%	695653	0.00%	1814
U	5080	99.15%	8232	98.82%	0
T ***NP***	445979	25.68%	511809	26.43%	1579
T ***P, bbe***	221750	63.05%	271429	60.98%	1
T ***NP, bbe***	171238	71.46%	206655	70.29%	1

Abstraction of Time. Second, we consider the issue of *time*. We examine the *untimed* model (U) of the system and search for constraints which can guarantee the *absence of deadlocks*. Searching for deadlocks in the untimed model allows us to examine a much smaller search space. In the case study we observed a reduction of 96% for the 4-thread subsystem and 99% for the whole system (see Table 1). More importantly, finding and removing *all* deadlocks in the untimed model means that the application is *logically* correct. An initial set of deadlocks can actually be obtained during the generation of the model from the source code by applying standard analysis techniques for deadlock detection (typically a search for loops in a dependency graph). Our model generator implements such an analysis as well.

Having found all the potential deadlocks in the untimed system, we add the synthesized $\mathcal{S}_{\text{exec}}$ and $\mathcal{S}_{\text{notif}}$ scheduler sets obtained so far to the *timed* model (T), in order to search for the *timeliness* constraints, which can guarantee that all threads will meet their deadlines. In order to make the problem more tractable, we reduced the timed model modulo the *branching bisimulation equivalence (bbe) reduction* [16], which eliminates "unobservable" actions (in our case the Tick action) but only when doing so preserves the branching structure of processes. Given a set of equivalent states under the *bbe* reduction, we elect as a representative of this set the state which has the maximum global clock value.

Execution Model. Third, we analyze the behavior of the system for two different execution models, namely *preemptive* and *non-preemptive*. We first consider that the application threads cannot be preempted while they are computing. The non-preemptive execution model hypothesis reduces the state space, since it removes all the cases where the execution of a thread is suspended so as to handle an interrupt. Once we can indeed safely schedule the system under the hypothesis that threads are never preempted, then we can use the constraints obtained during this step to *reduce even further* the state space that we have to construct and analyze when we do allow threads to be preempted. The non-preemption of threads is easily added to our models *through the use of a QoS policy* that forbids the schedulers from choosing a thread for execution, when another thread is already in a state where it is performing a computation:

$$\mathcal{Q}_{\text{exec}}(\overrightarrow{state}) = \{t\ .\ t \in \mathcal{S}_{\text{exec}}(\overrightarrow{state}) \wedge \neg\exists t' \neq t\ .\ \text{computes}(t')\}$$

In the case study, the combination of the non-preemptive execution model (NP) with the *bbe* leads to a reduction of about 80% for the 4-thread subsystem and 70% for the 5 threads. However, we cannot safely schedule all systems when we do not allow threads to be preempted. This means that for these systems we will not obtain any scheduling constraints and, therefore, will be obliged to examine the larger, unconstrained state space of the timed model, which corresponds to a preemptive execution model.

Observability of Clocks. Having synthesized a safe scheduler for an application does not necessarily mean that we can implement it easily with an existing *R-T OS* though. The difficulty of implementing it as is, arises from the fact that the constraints we produce during the synthesis use the state of the system to decide what are the safe choices at each point during the execution and, therefore, also make reference to the values of the local clocks of the threads. However, these clocks do not really exist in the application but were only introduced as a way to model the computations of the threads. Introducing them in the final code means that we will have to add for each thread an additional timer object and reset and activate (resp. deactivate) the timer before (resp. after) each computation and read its value when making a scheduling decision. As using timers may substantially increase the execution time of the scheduler, we investigate the possibility of synthesizing a clock-free one, which only examines the *PC*s of the threads. This will make the scheduler itself faster to execute, since in order to make a scheduling decision it now only needs to examine the n values of the different *PC*s and not the $2n + 1$ values of the *PC*s, the local thread clocks and the global clock.

On the other hand, removing the clocks from the constraints can introduce states where the scheduler will take the wrong decision and cause a thread to miss its deadline. These states are those where a scheduler gets called at the same configuration of thread *PC*s but at different time instances. Since the time instances (and therefore the clock values) are different, the safe sets $\mathcal{S}_{\text{exec}}$ of these states can be different themselves as well. When we decide to not observe the clock values while scheduling, we are effectively unable to differentiate among these different sets and all these states become *equivalent*, as far as our scheduler is concerned. Therefore, if we wish our scheduler to always make a decision which is *safe*, then the $\mathcal{S}_{\text{exec}}$ set of this *equivalence class* of states should be the *intersection* of the $\mathcal{S}_{\text{exec}}$ sets of the states which belong to the same equivalence class.

$$\mathcal{S}_{\text{exec}}(\overrightarrow{class_j}) = \bigcap_{state_i \in class_j} \mathcal{S}_{\text{exec}}(\overrightarrow{state_i})$$

Sometimes, the $\mathcal{S}_{\text{exec}}(\overrightarrow{class_j})$ set will be empty, if the scheduler decisions at the members of this equivalence class were conflicting. When encountering such an equivalence class whose $\mathcal{S}_{\text{exec}}$ set is the empty set, we need to add its members to the set of deadlocked states and continue the synthesis algorithm, until we find a set of constraints which helps us to avoid the whole class, if any.

Other QoS Policies. Once we have synthesized a safe scheduler, we can compose it with other QoS policies to choose among the safe threads those which better realize the QoS requirements of the system. Actually, as Altisen *et al.* [2] showed, a QoS policy can also be used from the start of the synthesis as an initial *scheduling policy*, so as to reduce the size of the model we want to analyze. In this case, the QoS policy used should give preference to tasks of the system which have a higher probability to miss their deadlines.

Table 2. Synthesis steps

Model kind	**States**	**Red.**	**Trans.**	**Red.**	**Dead.**	**Constr.**
Synthesis Steps for 4 threads						
U	3320	96.41%	4680	95.49%	0	0
U, No Deadlocks	3320	96.41%	4680	95.49%	0	0
T ***NP, bbe*** , No Deadlocks	18304	80.19%	21738	79.03%	1	0
T ***NP, bbe*** , Safe	13830	85.03%	16196	84.38%	0	15
T ***P, bbe*** , No Clocks	16807	81.81%	20003	80.70%	1	15
T ***P, bbe*** , No Clocks, Safe	16249	82.41%	19245	81.43%	0	23
Synthesis Steps for 5 threads						
T ***LMCS, bbe*** , 4 threads Safe	23302	96.12%	26515	96.19%	1	23
T ***LMCS, bbe***	15742	97.38%	17584	97.47%	0	38
T ***LMCS, bbe*** , No Clocks (Safe)	15742	97.38%	17584	97.47%	0	38

Table 2 shows the results obtained when applying our methodology on the case study. First we synthesized a safe scheduler consisting of 15 constraints for the 4-thread subsystem, considering that the execution mode is non-preemptive. Then we used these constraints to render the state space smaller once we choose a preemptive execution mode and we no longer observe clocks. This lead to another 8 constraints, with which the 4-thread subsystem is indeed always safe. Then, we used the 23 constraints we synthesized for the 4-thread subsystem, to minimize the 5-thread one. In parallel, we also used a QoS policy which locally minimizes context switches (LMCS), so as to render the state space even smaller. This produced another 15 constraints, under which the 5-thread system is safe, when it is running with the LMCS QoS policy.

5 Scheduler Implementation

Once the scheduler has been synthesized using the model, it has to be implemented and integrated with the code generated by the TurboJ compiler. The structure of the executable code is depicted in Fig. 6. The generated code consists of two parts, namely, the application code and the synthesized scheduling predicates. The application code is instrumented to call the application-level scheduler called `J_Scheduler` when an application thread executes the code corresponding to the Java-bytecode statements **monitorEnter** or **monitorExit**, and the methods **notify**, **notifyAll**, **wait**, **waitTimed** and **waitForPeriod**. `J_Scheduler` calls the function `Synthesized_Predicates` which evaluates the synthesized predicates corresponding to the different scheduler layers. The application-level scheduler is implemented on top of an accompanying run-time library, which offers an implementation in native code of the aforementioned statements and methods.

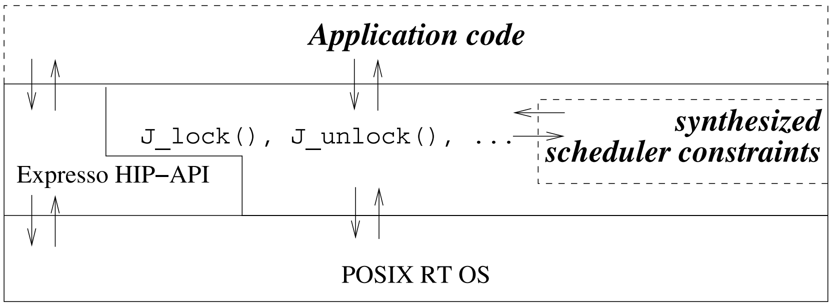


Fig. 6. System decomposition

More precisely, our implementation uses a subset of Posix, *i.e.*, it uses mutexes (without any priority inheritance protocol), condition variables & priorities. Indeed, it does not use real-time timers, since one of the underlying OS's we needed to support (in the context of the Expresso project) does not provide any. More specifically, we use a single mutex (`sched_mx`) and provide to each application thread a unique condition variable. These condition variables are all associated with the aforementioned mutex (a capability which exists in Posix but not in Java). Finally, we use three different

Table 3. Pseudo-code of the application scheduler

```
void J_Scheduler(int tc, bool in_notify, timespec &deadline) {
  bool finished = true, level_super = false;
  int  tn;
  /* tc is the current thread ($t_c$) & tn the next one ($t_n$) */
  do {
    tn = Synthesized_Predicates(THREADS_TABLE); // calculate Ready, Safe & QoS sets
    if (tn != tc) {
      if (in_notify) {
        if (-1 != tn) { // -1 means no thread is waiting
          J_Set_Priority(tn, BLOCKED); // Don't allow $t_n$ to preempt you
          notify(THREADS_TABLE[tn].cv);
        }
      } else { // ! in_notify
        J_Set_Priority(tn, EXEC);
        THREADS_TABLE[tn].position = THREADS_TABLE[tn].pos_after_notif;
        notify(THREADS_TABLE[tn].cv);

        if (NULL == deadline) { // Not a waitTimed or waitForPeriod
          // Release CPU here
          wait(THREADS_TABLE[tc].cv, sched_mx);
          /* Here I have been signaled  */
        } else {            // NULL != deadline
          J_Set_Priority(tc, SUPER); // Release CPU here
          timed_wait(THREADS_TABLE[tc].cv, sched_mx, deadline);
          /* Here I have been signaled or I have timed-out.
             Must re-schedule to be safe, if I timed-out. */
          if (THREADS_TABLE[tc].position != THREADS_TABLE[tc].pos_after_notif) {
            finished = false;
            THREADS_TABLE[tc].position = THREADS_TABLE[tc].pos_after_timeout;
          }
          level_super = true ; deadline = NULL;
        }
      }
    }
  } while (! finished);
  if (level_super) J_Set_Priority(tc, EXEC);
}

void J_lock(int tc, int curr_pos) { // J_unlock is exactly the same
  lock(sched_mx);
  THREADS_TABLE[tc].position = curr_pos;
  J_Scheduler(tc, false, NULL);
  unlock(sched_mx);
}
```

priority levels, namely, **BLOCKED**, **EXEC** & **SUPER** (from lowest to highest) and the `SCHED_FIFO` Posix scheduling policy (*i.e.*, priority based, FIFO, non-preemptive execution of tasks having the same priority).

The pseudo-code of this implementation is shown in Table 3. Before calling `J_Scheduler`, the running application thread locks `sched_mx` and changes the label used for marking its position in the model. `J_Scheduler` calls the function `Synthesized_Predicates` (generated by the synthesis tool) passing it the current labels of all the application threads. If the thread chosen to be executed next (t_n) is different from the current one (t_c) and t_c is not doing a notification, we set the priority of t_n to **EXEC**, we notify the condition variable of t_n (cv_{t_n}) and finish by having t_c wait on its own condition variable (cv_{t_c}). This final action releases `sched_mx` just before blocking, thus allowing the notified thread t_n to resume execution. If t_n happens to be the same as t_c, then the application scheduler returns normally and t_c unlocks `sched_mx`.

The algorithm changes somewhat when calling the application scheduler because of a **waitTimed** or a **waitForPeriod**. In this case, we also pass to our scheduler the time that the current thread should wait. The scheduler then performs a timed wait on cv_{t_c} using as timeout the absolute deadline passed as an argument, instead of doing a simple wait. In addition, it also increases the priority of t_c to **SUPER** just before performing the timed wait, so that t_c has the chance to get the CPU as soon as it timeouts. If t_c does indeed timeout, it re-calculates the scheduler predicates so as to find out if it is indeed safe to continue execution. Functions `J_timed_wait` and `J_wait` set the field `THREADS_TABLE[].pos_after_notif` right before calling function `J_Scheduler` to be the label corresponding to the internal state of the wait where the thread has been notified but has not yet reacquired the mutex of the object on which it was waiting. In a similar manner, the field `pos_after_timeout` is set by the functions `J_timed_wait`, and `J_wait_for_period` to the label corresponding to the internal state of the **waitTimed**, or the label corresponding to the first statement following the beginning of a new period.

We have successfully executed our implementation over two different combinations of hardware architecture and embedded OS's, namely an Intel Pentium II (333MHz) running `eCos` over Linux and a PowerPC simulator (PSIM) using the proprietary OS of an industrial partner of the Expresso project. The experiments we performed with `eCos` showed that the execution time of the synthesized predicates (*i.e.*, the execution time of function `Synthesized_Predicates` called by `J_Scheduler`) is comparable to the execution time of locking an (unlocked) mutex, having a WCET of the order of 4μs.

Besides, we have a non-POSIX prototype implementation over `eCos` that uses alarms and alarm handlers, thus allowing us to support deadline and period miss handlers as proposed by the RTSJ, but disallowed by the Expresso and the Ravenscar-Java profiles.

6 Conclusions

We have presented a complete application-driven scheduler synthesis chain that allows to automatically generate native code for embedded real-time systems based on Java. In addition, it allows one to substitute RMA/EDF & PCP with a scheduler which is still safe but not as pessimistic and which can be easily extended with QoS scheduling policies. We are not aware of any other similar chain. The full integration of the model-generation and scheduler-synthesis tools with the compiler is under test as it requires a close collaboration with the TurboJ development team.

References

1. AICAS. http://www.aicas.com/jamaica.html. JamaicaVM.
2. K. Altisen, G. Göessler, and J. Sifakis. Scheduler modeling based on the controller synthesis paradigm. *Real-Time Systems*, 23(1), 55–84, July 2002.
3. B. Delsart, V. Joloboff, and E. Paire. JCOD: A Leightweight Modular Compilation Technology for Embedded Java. In *EMSOFT'02*, Grenoble, France, October , 2002.

4. L. Gauthier and M. Richard-Foy. Expresso RNTL Project - High Integrity Profile, 2002. Available from urlhttp://www.irisa.fr/rntl-expresso/docs/hip-api.pdf
5. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, Reading, MA, USA, 1996.
6. IEEE. POSIX.1. IEEE Std 1003.1:2001. Standard for Information Technology - Portable Operating System Interface (POSIX). The Institute of Electrical and Electronic Engineers, 2001.
7. Ch. Kloukinas and S. Yovine. Synthesis of Safe, QoS Extendible, Application Specific Schedulers for Heterogeneous Real-Time Systems. In *Proceedings of 5th Euromicro Conference on Real-Time Systems (ECRTS'03)*, Porto, Portugal, July 2003.
8. J. Kwon, A. J. Wellings, and S. King. Ravenscar-Java: A High-Integrity Profile for Real-Time Java. In *Java Grande*, pp. 131–140, 2002.
9. M. Lusini and E. Vicario. Static analysis and dynamic steering of time-dependent systems using Petri Nets. Technical Report # 28.98, University of Florence, 1998.
10. G. Muller, B. Moura, F. Bellard, and Ch. Consel. Harissa: A flexible and efficient Java environment mixing bytecode and compiled code. In *Proc. of Usenix COOTS'97*, Berkeley, 1997.
11. Real-Time for Java Expert Group. The Real-Time Specification for Java, 2001. Available from urlhttp://www.rtj.org
12. Real-Time Java Working Group. Real-Time Core Extensions, revision 1.0.14, 2001. Available from urlhttp://www.j-consortium.org/rtjwg.
13. J. Sifakis, S. Tripakis, and S. Yovine. Building models of real-time systems from application software. In *Proceedings of the IEEE, Special issue on modeling and design of embedded software*, 91(1):100–111, January 2003. IEEE.
14. Sun Microsystems. The Java HotSpot Performance Engine Architecture. urlhttp://java.sun.com/products/hotspot/whitepaper.html, April 1999.
15. TimeSys. urlhttp://www.timesys.com. JTime.
16. R. J. van Glabbeek and W. P. Weijland. Branching time and abstraction in bisimulation semantics. *JACM*, 43(3), 1996.
17. M. Weiss, F. de Ferrière, B. Delsart, Ch. Fabre, F. Hirsch, E. A. Johnson, V. Joloboff, F. Roy, F. Siebert, and X. Spengler. TurboJ, a Java bytecode-to-native compiler. In *Proc. of LCTES'98*, volume 1474 of *LNCS*, 1998.

Constraint-Based Design-Space Exploration and Model Synthesis

Sandeep Neema[1], Janos Sztipanovits[1], Gabor Karsai[1], and Ken Butts[2]

[1]Institute for Software Integrated Systems, Vanderbilt University, P.O. Box 1829 Sta. B. Nashville, TN 37235, USA
{sandeep.neema,janos.sztipanovits,gabor.karsai@vanderbilt.edu}
[2]Ford Motor Company, MD 7, 20600 Rotunda Dr, Dearborn, MI, 48124
{kbutts1@ford.com}

Abstract. An important bottleneck in model-based design of embedded systems is the cost of constructing models. This cost can be significantly decreased by increasing the reuse of existing model components in the design process. This paper describes a tool suite, which has been developed for component-based model synthesis. The DESERT tool suite can be interfaced to existing modeling and analysis environments and can be inserted in various, domain specific design flows. The modeling component of DESERT supports the modeling of design spaces and the automated search for designs that meet structural requirements. DESERT has been introduced in automotive applications and proved to be useful in increasing design productivity.

1 Introduction

Extensive modeling and model analysis is a crucial element of design flows for embedded systems. The cost of the design process is largely influenced by the cost of constructing models for detailed analysis and code generation. There is a significant pressure toward tool manufacturers to decrease this cost by increasing the automation of the modeling process. A promising approach to achieve this goal is to take advantage of reusable modeling components and automated model composition. An exposition of this problem was detailed by Ken Butts and others [8] describing the need for a model compiler in automotive applications.

Automotive controller modeling is typically done by means of model-based tool suites, such as MathWorks Simulink®[1] and Stateflow® [2]. According to the authors [1], building models for powertrain controller assemblies requires selection of 75-100 component models from thousands for each of the 130 vehicle types. The model components include 1 to 3 sampling contexts (crankshaft position or time) and on the average 20 inputs and 14 outputs. This means that the complete model has over 100 scheduling connections and 2000 data connections. Components cannot be selected

* Support for this research was provided by NSF ITR Grant CCR-0225610 and DARPA MoBIES grant F30602-00-1-0580

[1] MATLAB, Simulink and Stateflow are registered trademarks of The MathWorks, Inc., Natick, MA.

independently from each other. There are complex compatibility relationships among model components. Selection of components may require selection of others and the overall component structure influences or determines basic performance characteristics of the design.

The goal of our research has been the design of a component-based model synthesis tool, which given a set of models for subcomponents, composes a system model *m* automatically such that a set of design constraints are satisfied. The design requirements for the tools have been inspired by the practical needs of automotive control engineers:

1. Model components are MATLAB®, Simulink®, Stateflow® models.
2. Model components are characterized by a set of component attributes (component type like "continuous" and "discrete", input/output (I/O) definitions, essential parameters, solver used in simulation, sampling time, etc.) that influence composability and capture performance characteristics.
3. Model architectures are described by an abstract, hierarchical, high-level modeling language, whose leaf nodes refer to model components defined above.
4. There is a rich set of compatibility relations among model components. Structural constraints focus on I/O signal types and simulation properties. Component compatibility relates components via high-level design goals, such as "fun-to-drive" or "green".

The challenge is to synthesize integrated models that meet set design goals and performance targets using the available model components. While the original model compiler challenge [8] is defined in the context of Simulink® models, we generalized the method and tools developed and made them independent from the actual domain specific modeling languages and modeling tools used in a particular design flow.

The primary contribution of the described work is the DEsign Space ExploRation Tool (DESERT), which starts with carefully constructed design spaces representing design templates and synthesizes fully specified models that meet selected design constraints. DESERT is a domain independent tool chain for defining design spaces and executing constraint-based design space exploration. The DESERT tool chain can be linked to different domain specific modeling environments via model transformations. These domain specific modeling environments may include hardware architecture or software architecture models.

The paper first gives a short summary of relevant concepts. This will be followed by a discussion on constructing, shaping and aggregating design spaces. The paper concludes with a description of our constraint-based model synthesis technique, which is currently based on a symbolic representation of the design space and symbolic pruning of the design alternatives.

2 Background and Terminology

In model-based design, systems are described by models expressed in domain specific modeling languages (DSML). Formally, a DSML is a five-tuple of concrete syntax (*C*), abstract syntax *(A)*, semantic domain *(S)* and semantic and syntactic mappings (M_S and M_C) [9]:

$$L = < C, A, S, M_S, M_C >$$

The *C concrete syntax* defines the specific (textual or graphical) notation used to express models, which may be graphical, textual or mixed. The *A abstract syntax* defines the *concepts, relationships,* and *integrity constraints* available in the language. Thus, the abstract syntax determines all the (syntactically) correct "sentences" (in our case: models) that can be built. (It is important to note that the abstract syntax includes semantic elements as well. The integrity constraints, which define well-formedness rules for the models, are frequently called "static semantics".) The *S semantic domain* is usually defined by means of some mathematical formalism in terms of which the meaning of the models is explained. The $M_C : A \rightarrow C$ mapping assigns syntactic constructs (graphical, textual or both) to the elements of the abstract syntax. The M_S: $A \rightarrow S$ semantic mapping relates syntactic concepts to those of the semantic domain.

Any DSML, which is to be used in the development process of embedded systems, requires the precise specification (or modeling) of all five components of the language definition. The languages, which are used for defining components of DSMLs are called *meta-languages* and the concrete, formal specifications of DSMLs are called *metamodels* [1].

The specification of the abstract syntax of DSMLs requires a meta-language that can express concepts, relationships, and integrity constraints. In our work in Model-Integrated Computing (MIC) [4], we adopted UML class diagrams and the Object Constraint Language (OCL) as meta-language. This selection is consistent with UML's four layer meta-modeling architecture [16], which uses UML class diagrams and OCL as meta-language for the abstract syntax specification of UML.

The semantic domain and semantic mapping defines semantics for a DSML. The role of semantics is to give a precise interpretation for the meaning of models that we can create using the modeling language. Naturally, models might have different interesting properties; therefore a DSML might have a multitude of semantic domains and semantic mappings associated with it. For example, *structural* and *behavioral* semantics are frequently associated with DSMLs. The *structural semantics* of a modeling language describes the meaning of the models in terms of the structure of model instances: all of the possible sets of components and their relationships, which are consistent with the well-formedness rules in defined by the abstract syntax (structural semantics is frequently called *instance semantics* [16]). Accordingly, the semantic domain for structural semantics is defined by some form of set-relational mathematics. The *behavioral semantics* describes the evolution of the state of the modeled artifact along some time model. Hence, behavioral semantics is formally modeled by mathematical structures representing some form of dynamics, such as discrete transition system [2] or Hybrid Systems [3].

Although specification of semantics is commonly done informally using English text (see e.g. the specification of UML 1.3 [16]), the desirable solution is explicit, formal specification. There are two frequently used methods for specifying semantics: the *metamodeling approach* and the *translational approach* (see Figure 1).

- In the *meta-modeling approach* (see e.g. [5]), the semantics is defined by a meta-language that already has a well-defined semantics. For example, the UML/OCL meta-language that we use for defining the abstract syntax of a DSML has a structural semantics: it describes the possible components and structure of valid, syntactically correct domain models. The semantics of this meta-language can be represented by using a formal language, which supports the precise definition of sets and relations on sets. By providing the formal semantics for UML class diagrams and OCL - say, in Z [6]- the UML/OCL meta-model of the abstracts syntax of a DSMLs specifies not only its abstract syntax, but its structural semantics as well.

- The *translationa*l approach defines semantics via specifying the mapping between a DSML and another modeling language will well-defined semantics. As it can be seen on the right side of Figure 1, a model translator is a function $T : A_1 \rightarrow A_2$ whose domain and co-domain are the abstract syntax specifications. By choosing UML class diagrams and OCL as meta-language for the specification of abstract syntax, the specification of the model translator can be facilitated using graph transformations between (instance) graphs generatively specified by the class diagram [7].

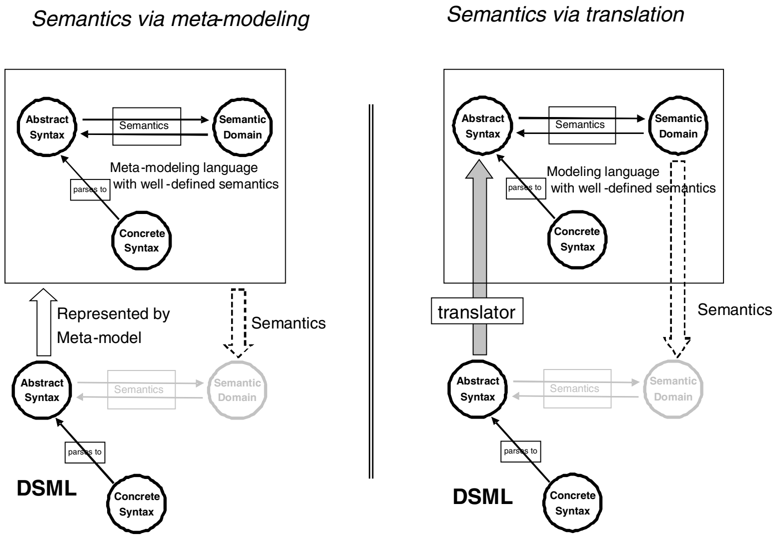

Semantics via meta-modeling
Semantics via translation
Abstract Syntax
Semantics
Semantic Domain
Meta-modeling language with well-defined semantics
parses to
Concrete Syntax
Represented by Meta-model
Semantics
DSML
Modeling language with well-defined semantics
translator

Fig. 1. Specification of semantics

As it will be shown later, formally specified structural semantics, meta-modeling and model transformations play essential role in solving the model synthesis problem in a domain independent manner. Although in this paper we focus on meta-models representing the abstract syntax of DSMLs (because of their significance in defining structural semantics and because of the central role of abstract syntax in specifying model translations), we should mention that meta-models are also used for defining behavioral semantics. For example, Lee and Sangiovanni-Vincentelli [19] developed the Tagged Signal Model to compare models of computations. Burch, Passerone and Sangiovanni-Vincentelli introduced trace algebras as semantic domain for modeling behaviors in the Metropolis project [20]. These, and other meta-modeling approaches have important roles in defining different behavioral semantics for DSMLs.

3 Construction of the Design Space

Model structures for automotive control are specified as refinements of a generic architecture [8] shown in Figure 2. Each top-level model component, such as transmission, transmission control T/M, etc., has many alternative realizations with different parameters and internal structure - arranged in a refinement hierarchy. Accordingly, components at any level in the refinement hierarchy may have alternative implementations. We call components with alternative implementations *templates.* At the leaf nodes of the refinement hierarchy are the *primitive* model components, which may also be templates with alternative (primitive) implementations. In our specific application context, the primitive model components are Simulink® models. (The "Bus" concepts on the diagram represent a set of variables shared among the model components.)

The *template* concept has a significant impact on the *structural semantics* of models. Without templates, each m model is unique, i.e. represents a point design. By introducing *templates* in the specification of the modeling language, each model including templates defines a set of models, M_D, which we call *design space*. An $m_j \in M_D$ model instance in the design space is defined by binding the component templates to one of the alternative implementations and by the binding the parameter values of the parameterized components to a specific value.

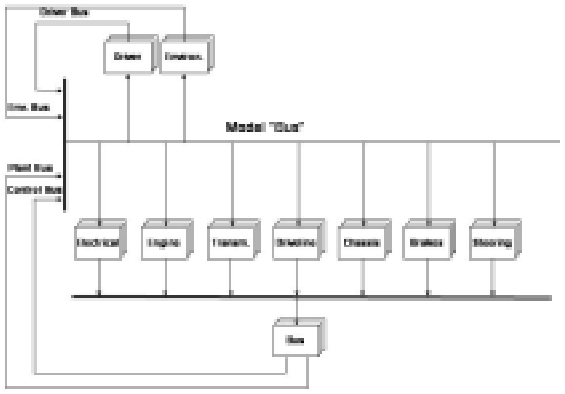


Fig. 2. Top-level Model Architecture in Automotive Systems

Remark 1: The scalability of hierarchically structured alternatives in capturing large design space can be judged from the following example: With a alternative implementations per Template, and n Template blocks on each level of an m-level deep refinement hierarchy, this representation can define: a^{k_m} design configurations, where $k_m = (k_{m-1} + 1) \times n$, and $k_1 = n$, using just $(a \times n)^m$ primitives. As an example, with $n = 4$, $a = 3$, and $m = 3$, a total of 1728 primitives can represent 3^{84} design configurations!

Remark 2: The design space captures all possible $m_j \in M_D$ model instances defined by the instantiation of templates and model parameters. Therefore, the essential semantics of a DSML, which is extended with the template and parameter concepts is *structural semantics*.

The goal of model synthesis is to find model instances in the design space, which satisfy design constraints regarding the feasible composition of model components (see Compatibility Constraints in section 5.3) and satisfy selected structural characteristics. *Structural characteristics* can be directly derived from structural features of the model. For example, let C be the set of all primitive control components and let $C_j=\{c|c\in C,c\in m_j\}$ is the subset of primitive controller components, which are elements of model $m_j\in M_D$. Furthermore, let $cost(c_i)$ is a function, which assigns and estimated implementation cost for each primitive controller components. Since the overall cost of the controller implementation for m_j is (approximately) the sum of the implementation cost of all controller components: $cost(C_j)=\sum_i cost(c_i|c_i\in C,c_i\in m_j)$, we can further restrict the design space with a cost constraint L:

$$M_{D,<L}=\{m_j\in M_D|cost(C_j)<L\}$$

where $M_{D,<L}$ is the set of all models in the design space, whose controller cost is less then L. The significance of *structural constraints* is that constraint-based design space exploration tools can efficiently restrict the design space using computationally inexpensive techniques.

Remark 3: As opposed structural constraints, *behavioral constraints*, such as controller dynamic performance, require behavioral analysis of designs using simulation or other analysis tools. Since simulation for large models is expensive, it is a good strategy to restrict the design space first by means of *structural constraints* and explore only the structurally correct designs using simulation.

It is interesting to relate our concept of *design space* with the concept of *platform* in Sangiovanni-Vincentelli's platform-based design [21]. Applying the definition of platforms in [22] we can conclude that the set of all designs that can arise from composing the C set of primitive model components form a *platform* (we will refer to this platform as *Model Platform, MP*). Clearly, the set of all designs, which can be generated by combining the primitive model components (i.e. the Model Platform) is not practical for automated model synthesis. The size of this set is unbounded and includes practically only meaningless configurations, therefore any automated search in that space would be hopeless. The top-down refinement strategy followed in our design space construction strategy starts with an application (or a set of applications, such as automotive controllers) and incrementally refines it to *MP*. Using the terminology of platform-based design [21][22], we start by selecting a point or a set of points in an *Application Platform* (formed by the interesting set of automotive controllers) and capture the feasible refinement paths to *MP* via hierarchically layered alternative refinements and parameterization. The resulting set of designs $M_D \subset MP$ restricts tremendously the number of meaningless configurations. The remaining, meaningless configurations in the top-down refinement strategy are the result of compatibility violations and neglected interdependences among design decisions. However, we can express these formally by means of compatibility and interdependency *constraints,* which than can be used for further restricting the design space.

Creation of extensive well-structured design spaces for different categories of applications requires significant effort. However, this effort is not required in the form of an initial investment, which is to be done before design space exploration can start. We envision design space construction as byproduct of product development process; the creation of a structured repository where design experience is accumulated.

4 Overview of the DESERT Design Flow

Our goal with DESERT has been to provide a model synthesis tool suite, which can represent large design spaces and can manipulate them by means of structural constraints. The place of these steps in the overall design flow is shown in Figure 3. The inputs to the DESERT tool suite are models and model components used for behavioral modeling and analysis (in our case Simulink® models). DESERT does not require all details of Simulink models, only those, which are required to specify the basic structure of designs.

1. The *Component Abstraction* tool maps Simulink® model components into Abstract Components by executing the T_A model transformation. The Abstract Components preserve the structure of I/O interfaces (using interface types) and configurable parameters.
2. The *Design Space Modeling* tool supports the construction of M_D design spaces using Abstract Components. Templates and parameters are the added language constructs that enable modelers define design spaces instead of point designs.
3. The *Design Space Encoding* tool maps the M_D design space into a representation form, which is suitable for manipulating/changing the space by restricting it with various design constraints.
4. The *Design Space Pruning* tool performs the design space manipulation and enables the user to select a design, which meets all structural design constraints.
5. The *Design Decoding* tool reconstructs the selected design from its encoded version.
6. The *Component Reconstruction* tool takes the design with abstracted components and reconstructs the detailed design suitable for behavioral analysis (i.e. it synthesizes the detailed Simulink model in our example).

In the following sections we describe the key principles we used in implementing the system.

5 Specification of DSMLs in the DESERT Design Flow

As we discussed earlier, the domain and codomain of model transformations in the design flow are represented through meta-models specifying the abstract syntax of the modeling languages. The DESERT design flow includes several transformations on the models. These transformations decouple the complex design space exploration tools (Design Space Encoding and Design Space Pruning) from the domain (and tool) specific detailed Simulink® component models and from the Design Space Modeling tool, which still uses a modeling language with domain specific flavor (although it uses abstracted components). In this section we show some of the important components of the design flow focusing more on the approach rather than the technical details.

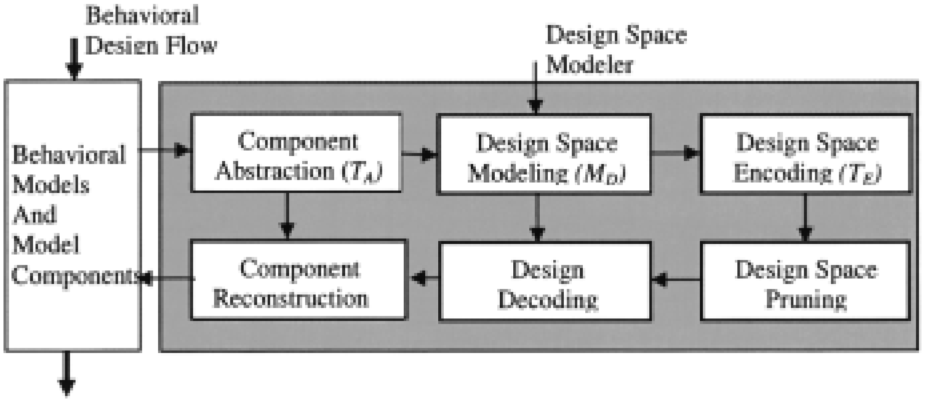

Behavioral Design Flow
Design Space Modeler
Behavioral Models And Model Components
Component Abstraction (T_A)
Design Space Modeling (M_D)
Design Space Encoding (T_E)
Component Reconstruction
Design Decoding
Design Space Pruning

Fig. 3. DESERT Design flow

5.1 Component Abstraction

The T_A model transformation receives Simulink® model components and generates abstract components for Design Space Modeling. The domain of T_A is represented by the meta-model of the Simulink® models, which is shown in Figure 4. (Detailed description of the Simulink® meta-model can be found in [10].) In short, the *System* class encapsulates a system model in Simulink. It serves as a container for the block diagram that models a dynamical system. It contains *Block* objects and connections (*Line* objects) between the Block objects. Some of these Block objects may be *Subsystem* objects. A Subsystem can be composed as a block diagram. The composition is indirect through a System object i.e. every Subsystem object contains a single System object. This enables hierarchical representation of a complex system. Every Model/Subsystem object contains a single System object. A System object may also contain *Annotation* objects that are used to add documentary information/user comments to the system models. The System class has a small set of attributes that capture various visual preferences, printing preferences, and system information.

The codomain of the T_A model transformation is represented by the meta-model of the Design Space Modeling tool. The meta-model is shown in Figure 5.

Similarly to the Simulink meta-model, the Design Space meta-model specifies also a hierarchical block diagram language with object types *Compound, Template* and *SimulinkSystem.* The *SimulinkSystem* objects refer to the abstracted components imported from the Simulink environment. The Simulink and Design Space meta-models in Figures 4 and 5 clearly show the role of the T_A (Component Abstraction) transformation: internal details of the Simulink model objects are suppressed, only the name (*Name* attribute in the *System* object) I/O interfaces (*InputPort* and *OutputPort*) and parameters (*Parameter* object) are preserved in the Design Space Modeling environment. The specific new constructs added to the meta-model for modeling Design

Spaces are the *Template*-s (see our earlier discussion on templates) and *Constraint*-s. In DESERT's Design Space Modeling environment the constraints are described as OCL expressions [13].

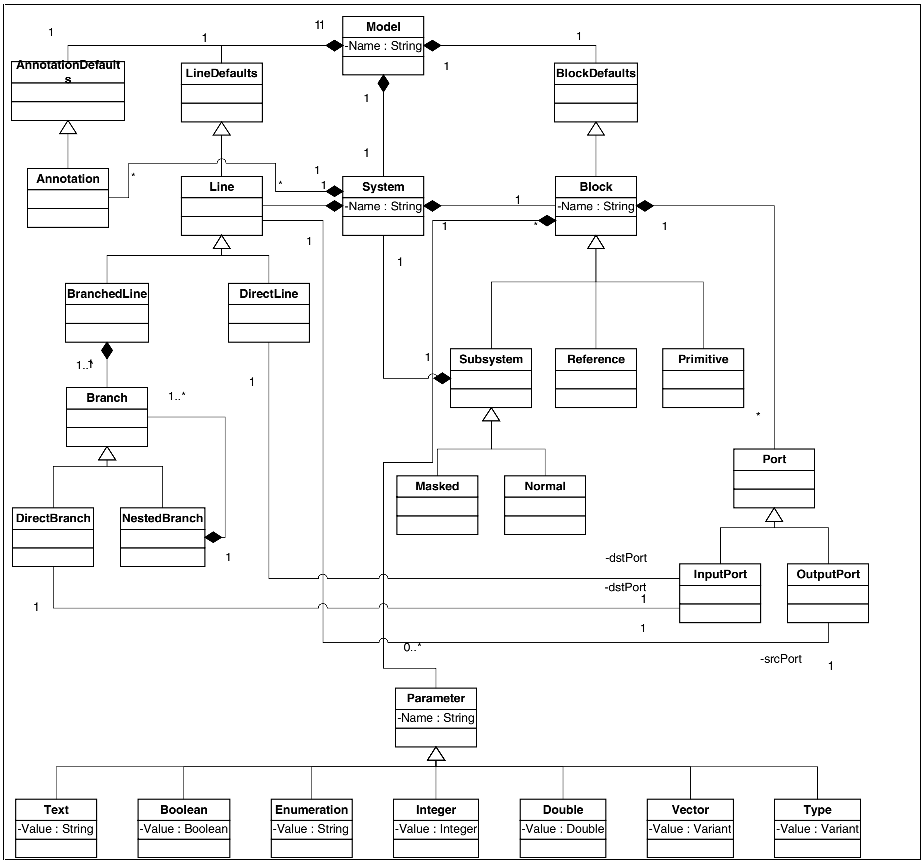


Fig. 4. Meta-model of Simulink®

5.2 Design Space Modeling

As we described earlier, the M_D design space is constructed manually by Design Space Modelers (see Figure 3). The DESERT tool suit uses Vanderbilt/ISIS Graphical Modeling Environment (GME) as Design Space Modeling tool. GME is a metaprogrammable graphical model builder [11], which can be customized to a DSML via abstract syntax and concrete syntax specification in terms of meta-models. The latest public release of GME3 is downloadable from the ISIS web site [12].

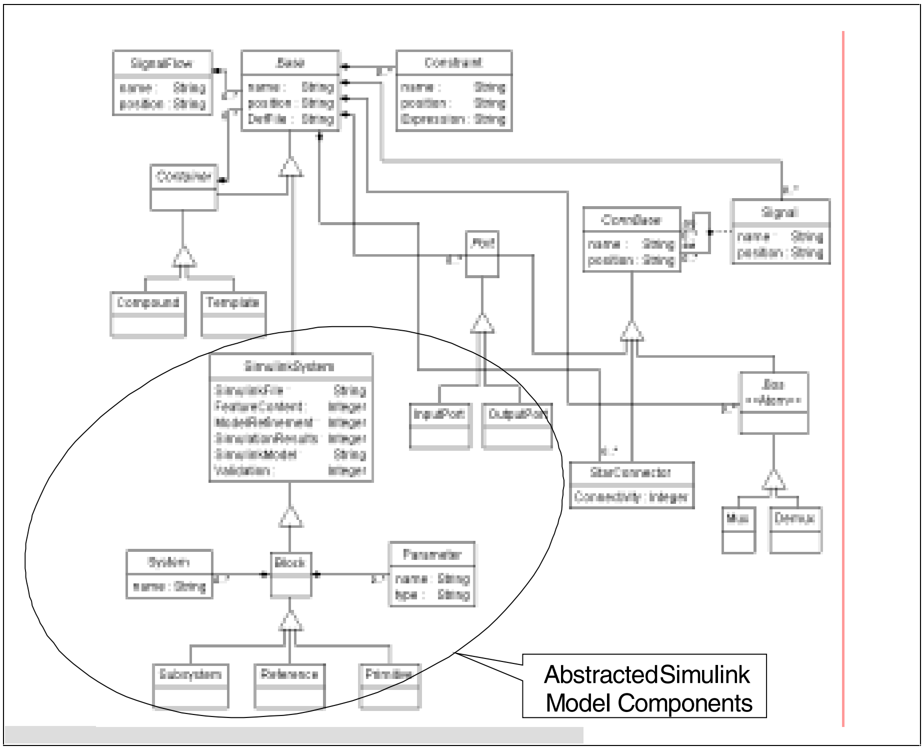
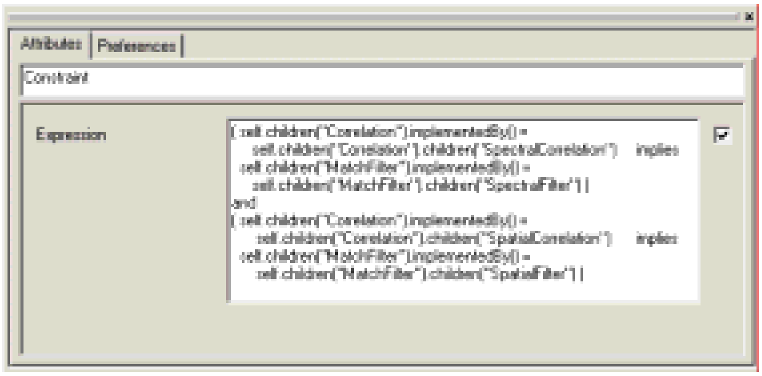


Fig. 5. Design Space meta-model

Design space modeling is essentially creating product line model architectures: modelers incrementally build, expand the M_D space by adding new primitive model components (abstracted from Simulink® models), and compose them into new versions of plant and controller models. *Note that this process is not the enumeration of all possible designs; the designer merely specifies alternatives on various levels of the hierarchy.* As a consequence of this, we are *not* considering explicit traversal of the design space during the pruning process. While adding new implementation alternatives to plant and controller models on various levels of the refinement hierarchy, the rapidly expanding design space can be restricted by new compatibility and other structural constraints. To remain consistent with the selected meta-modeling language (UML class diagrams and OCL), we use OCL-based constraints [13] to "shape" the design space. While the meaning of these constraints is domain-specific, there are typical constraint categories that are suitable to demonstrate the method.

1. *Compatibility constraints* – Matching interfaces are not sufficient conditions for composability. In fact, in many situations selection of implementation alternatives are not orthogonal due to the lack of semantic compatibility. A simple example for a semantic compatibility constraint for a design space defined in a signal processing domain is shown in Figure 6. The meaning of the constraint is that "Spectral domain correlation composes only with Spectral domain filters and Spatial domain correlation composes only with Spatial domain filters".

2. *Inter-aspect constraints* – Inter-aspect constraints express interdependencies among design spaces defined for capturing different aspects of designs. For example, plant models and controller models can be represented in two separate design spaces. Naturally, a large number of inter-dependencies exist between the plant model configurations and controller configurations. Composing an end-to-end system requires evaluating these inter-aspect constraints. Inter-aspect constraints can be used to explicate these dependencies and relations as a constraint network, which can then be subsequently utilized in the design space exploration to systematically synthesize a point-design for the aggregate system.



```
( self.children("Correlation").implementedBy() =
    self.children("Correlation").children("SpectralCorrelation")    implies
 self.children("MatchFilter").implementedBy() =
    self.children("MatchFilter").children("SpectralFilter") )
and
( self.children("Correlation").implementedBy() =
    self.children("Correlation").children("SpatialCorrelation")    implies
 self.children("MatchFilter").implementedBy() =
    self.children("MatchFilter").children("SpatialFilter") )
```

Fig. 6. Example for semantic composability constraints

5.3 Design Space Encoding and Pruning

Up to this point, we identified constructs and methods for defining, aggregating and constraining design spaces. The roles of the next two steps in the DESERT design flow (see Figure 3) Design Space Encoding and Design Space Pruning are the followings:

1. understand whether or not we have created inconsistency during design space modeling (meaning that the design space is 'empty'), and
2. find models that meet the required structural constraints.

Since we are focusing on structural semantics of the design space and intend to compute with structural constraints, manipulation of design-spaces can be reduced to set operations: calculating product spaces (composition of design spaces) and finding subspaces that satisfy various (structural) constraints. Since the size of design-spaces is frequently huge, execution of these set manipulation operations with enumerating all elements is hopeless. Therefore, we choose to perform the manipulation operations *symbolically*. Two problems had to be solved: 1) symbolic representation of design-spaces, and 2) symbolic representation of constraints.

If we restrict the parameters of model objects to finite domains, the design space will be also finite. By introducing a binary encoding of the elements in a finite set, all operations involving the set and its subsets can be represented as Boolean functions

[15]. These can then be symbolically manipulated with Ordered Binary Decision Diagrams (OBDD-s), a powerful tool for representing, and performing operations involving Boolean function. The choice of encoding scheme has a strong impact on the scalability of the symbolic manipulation algorithms, as it determines the number of binary variables required for representing the sets. The details of our encoding scheme has been described in [14], here we only demonstrate the approach.

Figure 7 shows the encoding of a simple design space formed by hierarchically structured alternatives. In this example, *s*, the top-level node in the design space is a template, which has three alternative implementations: *s1 or s2 or s3*. *s1* is also a template with three alternative implementations: *s11 or s12 or s13*. *s2*'s implementation requires three components, *s21 and s22 and s23*. Out of these components, *s21* and *s23* are templates with two-two alternative implementations, *s211 or s212* and *s231 or s232*, respectively. The prefix-based binary encoding scheme [14] assigns binary code to each element such that each configuration receives a unique encoding value. In the example, four Boolean variables, [v_0, v_1, v_2, v_3] are required for the encoding of the structure. Figure 8 shows the symbolic Boolean representation of this design space, given the encoding. For example, the binary code of node *s2* in the design space is $S2=\neg v_0 v_1$. As it can be seen, *s2* represents a subspace in the design space with four alternative configurations.

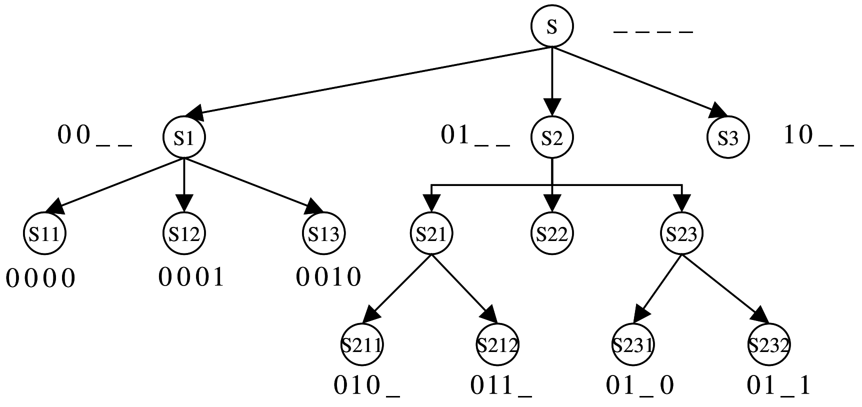


Fig. 7. Encoding abstracted design-spaces

$S=S_1 \vee S_2 \vee S_3$
$S_1=S_{11} \vee S_{12} \vee S_{13}$
$S_2=S_{21} \wedge S_{22} \wedge S_{23}$
$S_{21}=S_{211} \vee S_{212}$
$S_{23}=S_{231} \vee S_{232}$

$S_{11}=\neg v_0 \neg v_1 \neg v_2 \neg v_3$
$S_{12}=\neg v_0 \neg v_1 \neg v_2 v_3$
$S_{13}=\neg v_0 \neg v_1 v_2 \neg v_3$
$S_{22}=\neg v_0 v_1$
$S_3=v_0 \neg v_1$

$S_{211}=\neg v_0 v_1 \neg v_2$
$S_{212}=\neg v_0 v_1 v_2$
$S_{231}=\neg v_0 v_1 \neg v_3$
$S_{232}=\neg v_0 v_1 v_3$

Fig. 8. Symbolic Boolean representation of abstracted design-spaces

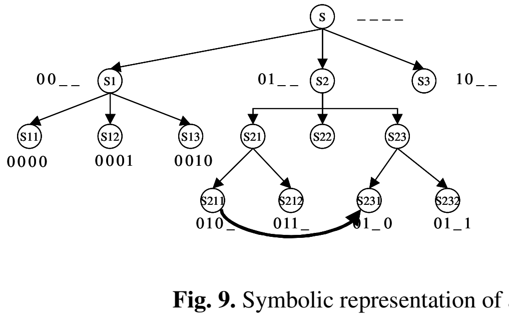
In addition to encoding the structure of the design-space, the encoding scheme has to encode the parameters of the parameterized model components. Subsequent to encoding, and deciding the variable ordering, the symbolic Boolean representation is mapped to an OBDD representation in a straightforward manner [14].

Earlier we listed some basic categories of structural constraints. Symbolic representation of each of these categories of constraints is summarized below.

1. *Compatibility and Inter-aspect constraints* – These constraints specify relations among subspaces in the overall design space. Symbolically, the constraints can be represented as a Boolean expression over the Boolean representation of the design-space. Figure 9 shows an example of a compatibility constraint, and its symbolic Boolean representation. The compatibility constraint in the example expresses interdependency between implementation decisions. Specifically, selection of implementation *s211* for template *s21* implies that *s231* must be the selected implementation for template *s23*. In the Boolean space, this constraint is expressed by the following Boolean expression:

$$cc = S211 \Rightarrow S231 = S211 \wedge S231 \vee \neg S211$$

2. *Resource constraints* – Resource constraints specify bounds on the resource needs of a composed system. These may be in the form of size, weight, energy, cost, etc. As we described earlier, the important limitations for the resource constraints, which DESERT is able to manage is that they are derived from structural characteristics of designs. In general, resource constraints are more challenging to represent symbolically, than composability or inter-aspect constraints. Different resource attributes compose differently, e.g. cost can be composed additively, reliability can be composed multiplicatively, latency can be composed as additively for pipelined components, and as maximum for parallel components, etc. DESERT provides some built-in composition functions (addition, maximum, minimum, etc.), and has a well-defined interface for creating custom composition functions.



$$cc = S_{211} \Rightarrow S_{231}$$
$$cc = S_{211} \wedge S_{231} \vee \neg S_{211}$$
$$cc = \neg v_0 v_1 \neg v_2 \neg v_3 \vee v_0 \vee \neg v_1 \vee v_2$$

Fig. 9. Symbolic representation of a compatibility constraint

In addition to these basic categories of constraints, complex constraints may be expressed by combining one or more of these constraints with first order logic connectives. The symbolic representation of the complex constraints can be accomplished simply by composing the symbolic representation of the basic constraints.

The symbolic pruning of the design-space, as observed earlier, in essence, is a set manipulation problem. Constraint-based pruning is a restriction of the aggregate space with the constraints. Symbolic pruning is simply the logical conjunction of the symbolic representation of the space with the symbolic representation of the constraints. It is worth emphasizing that during the pruning process all of the (potentially very large) design spaces are evaluated simultaneously. Figure 9 illustrates the process of symbolic design-space pruning. As the figure shows, the Design Space Pruning tool is prepared for applying constraints on the Design Space. According to our experience, OBDD based representations scale well for representing the structure of the design space (nested AND/OR expressions). The critical challenge in scalability occurs during the design-space pruning step. Automatic application of complex constraints to large spaces may result in explosion of the OBDD-s therefore DESERT has an interactive user interface to influence this process. Users can control the importance of constraints and select the sequence order of their application. We are experimenting with re-encoding the design-space after each pruning step, which usually results in a drastic decrease in the number of binary variables (see Figure 10).

The primary advantage of the symbolic design space pruning approach is that it is exhaustive: the pruned space includes all of the designs, which meet the applied design constraints. As experience with scalability in real-life industrial applications at Ford accumulate, we will be able to see the limitations of the method.

A significantly simpler, but still useful alternative approach to design space pruning is to find a single design configuration (not all), which satisfies the selected design constraints. We currently experiment with various constraint solvers and languages, such as Oz [17] to develop solution for this approach.

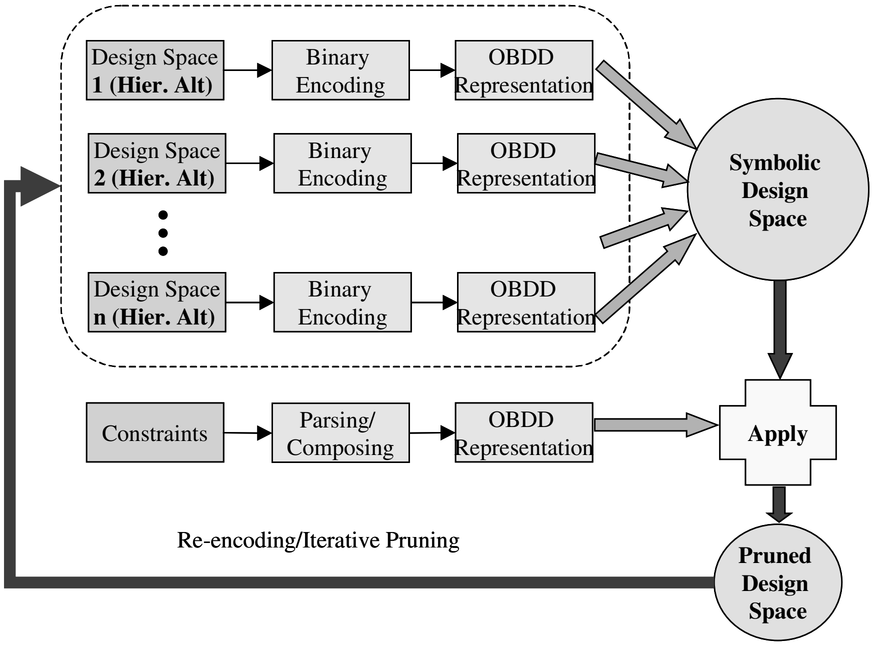


Fig. 10. Symbolic design space pruning

5.4 Reconstructing Detailed Design

The concluding steps of the DESERT design flow (see Figure 3) are the decoding of the selected abstract design from its binary code and reconstructing the detailed models from the abstract design. The decoding process involves reconstructing a data structure from the encoded form, where the data structure is an abstracted representation of the actual design. The design space encoder and decoder components operate on a shared "dictionary", which captures what specific design space model elements correspond to what codes. The resulting data structure is a specific, point design (i.e. a single model *m*), which can be looked at in the modeling tool. Naturally, it does not have any templates (i.e. no alternatives).

The final step is the creation of the actual design model in Simulink®. The design model created in the design decoding step is follows the Design Space metamodel (from Figure 5), and from it a model is constructed that follows the Simulink® metamodel (from Figure 4). This construction is mostly straightforward, the only complication arises because of the connections: the blocks in the final design have to be "wired-up" as required. However, this wiring can be algorithmically generated from the (higher-level) buses and wires captured in the design space models.

6 Conclusions and Future Work

Design space modeling and model synthesis are important part of the design flow for embedded systems. Automated tools can offer major productivity advantages for practicing engineers. During the development of the DESERT tool suite the Vanderbilt/ISIS team has been in close interaction with Ford, which was essential in identifying the crucial challenges engineers face. At this point, the prototype tool is field tested in real-life vehicle development programs.

The method described has been based on the structural semantics of models. The role of model synthesis based on structural constraints is to prune the design space before the computationally more expensive behavioral analysis methods are used for finding optimal design solutions. Since structural semantics is defined for all domain specific modeling languages, the tool suite can be interfaced to many design flows (independently from the behavioral semantics of the languages). We have been using this opportunity extensively in other domains, such as modeling environment for power-aware computing [18].

References

[1] Karsai, G., Sztipanovits, J., Ledeczi, A., Bapty, T.: "Model-Integrated Development of Embedded Software", *Proceedings of the IEEE*, Vol. 91, No.1., pp.145–164, January, 2003

[2] D. Harel and M. Politi, Modeling Reactive Systems with Statecharts: The STATEMATE Approach, McGraw-Hill, 1998

[3] Thomas A. Henzinger. The theory of hybrid automata. Proceedings of the 11th Annual Symposium on Logic in Computer Science (LICS), IEEE Computer Society Press, 1996, pp. 278–292

[4] J. Sztipanovits and G. Karsai: “Model-Integrated Computing,” *IEEE Computer,* April, 1997 (1997) 110–112

[5] Clark, T., Evans, A., Kent, S.: “Engineering Modeling Languages: A Precise Metamodeling Approach,” R.-D. Kutsche and H. Weber (Eds.): FASE 2000, LNCS 2306, pp. 159–173, 2002

[6] Evans, A., France, R., Lano, K., Rumpe, B. : « Developing UML as a Formal Modeling Notation,”LNCS 1357, pp. 145–150, Springer Verlag Berlin, 1997

[7] Levendovszky, T., Karsai G.: “Model reuse with metamodel based-transformations,” ICSR, LNCS, Austin, TX, April 18, 2002.

[8] Butts, K., Bostic, D., Chutinan, A., Cook, J., Milam, B., Wand, Y.: “Usage Scenarios for an Automated Model Compiler,” EMSOFT 2001, LNCS 2211, Springer. (2001) 66–79

[9] T. Clark, A. Evans, S. Kent, P. Sammut: “The MMF Approach to Engineering Object-Oriented Design Languages,” Workshop on Language Descriptions, Tools and Applications (LDTA2001), April, 2001

[10] Neema, S.: “Simulink and Stateflow Data Model”, see www.isis.vanderbilt.edu.

[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] Object Constraint Language Specification, ver. 1.1, Rational Software Corporation, et al., Sept. 1997. (1997)

[14] Neema, S., “Design Space Representation and Management for Embedded Systems Synthesis,” Technical Report, ISIS-01-203, February 2001.
http://www.isis.vanderbilt.edu/publications/archive/Neema_S_2_0_2003_Design_Spa.pdf

[15] Bryant R., “Symbolic Manipulation with Ordered Binary Decision Diagrams,” School of Computer Science, Carnegie Mellon University, Technical Report CMU-CS-92-160, July 1992

[16] UML Semantics, Ver. 1.1., Rational Software Corporation, 1997.

[17] http://www.mozart-oz.org/

[18] Ledeczi A., Davis J., Neema S., Eames B., Nordstrom G., Prasanna V., Raghavendra, C., Bakshi A., Mohanty S., Mathur V., Singh M.: Overview of the Model-based Integrated Simulation Framework, Tech. Report, ISIS-01-201, January 30, 2001.

[19] Lee, E.A., Sangiobanni-Vincentelli, A.L.: “A framework for comparing models of computations,” *IEEE Transactions on Computer Aided Design Integrated Circuits*, 17(12):1217–1229, Dec. 1998.

[20] Burch, J.R., Passerone, R., Sangiovanni-Vincentelli, A.L.: “Modeling Techniques in Design-by-Refinement Methodologies, Proc. of IDPT-2002, June, 2002

[21] Sangiovanni-Vincentelli, A.L.: “ Defining platform-based design,” *EEDesign,* February, 2002

[22] Lee, E.A., Neuendorffer, S., Wirthlin, M.J.: “Actor-Oriented Design of Hardware and Software Systems,“ (to be published in *Journal of Circuits, Systems and Computers*) *Technical Memorandum UCB/ERL M02/13*, University of California, Berkeley, CA 94720, May 1, 2002

Eliminating Stack Overflow by Abstract Interpretation

John Regehr, Alastair Reid, and Kirk Webb

School of Computing, University of Utah

Abstract. An important correctness criterion for software running on embedded microcontrollers is *stack safety*: a guarantee that the call stack does not overflow. We address two aspects of the problem of creating stack-safe embedded software that also makes efficient use of memory: statically bounding worst-case stack depth, and automatically reducing stack memory requirements. Our first contribution is a method for statically guaranteeing stack safety by performing whole-program analysis, using an approach based on context-sensitive abstract interpretation of machine code. Abstract interpretation permits our analysis to accurately model when interrupts are enabled and disabled, which is essential for accurately bounding the stack depth of typical embedded systems. We have implemented a stack analysis tool that targets Atmel AVR microcontrollers, and tested it on embedded applications compiled from up to 30,000 lines of C. We experimentally validate the accuracy of the tool, which runs in a few seconds on the largest programs that we tested. The second contribution of this paper is a novel framework for automatically reducing stack memory requirements. We show that goal-directed global function inlining can be used to reduce the stack memory requirements of component-based embedded software, on average, to 40% of the requirement of a system compiled without inlining, and to 68% of the requirement of a system compiled with aggressive whole-program inlining that is not directed towards reducing stack usage.

1 Introduction

Inexpensive microcontrollers are used in a wide variety of embedded applications such as vehicle control, consumer electronics, medical automation, and sensor networks. Static analysis of the behavior of software running on these processors is important for two main reasons:

– Embedded systems are often used in safety critical applications and can be hard to upgrade once deployed. Since undetected bugs can be very costly, it is useful to attempt to find software defects early.
– Severe constraints on cost, size, and power make it undesirable to overprovision resources as a hedge against unforeseen demand. Rather, worst-case resource requirements should be determined statically and accurately, even for resources like memory that are convenient to allocate in a dynamic style.

In this paper we describe the results of an experiment in applying static analysis techniques to binary programs in order to bound and reduce their stack memory requirements. We check embedded programs for *stack safety*: the property that they will not

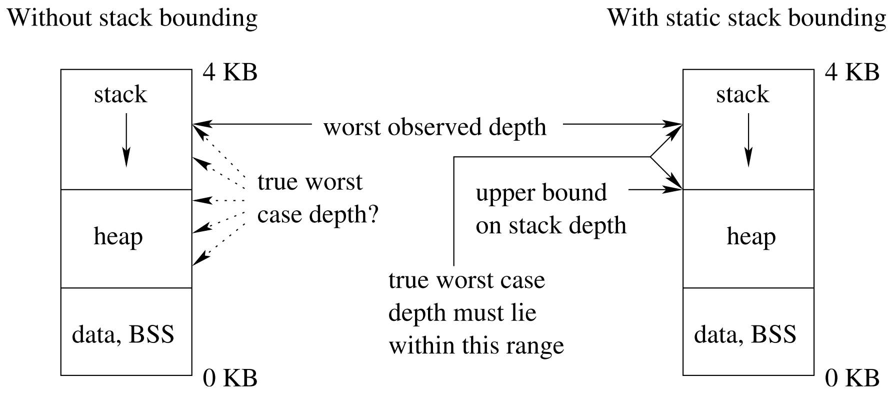


Fig. 1. Typical RAM layout for an embedded program with and without stack bounding. Without a bound, developers must rely on guesswork to determine the amount of storage to allocate to the stack.

run out of stack memory at run time. Stack safety, which is not guaranteed by traditional type-safe languages like Java, is particularly important for embedded software because stack overflows can easily crash a system. The transparent dynamic stack expansion that is performed by general-purpose operating systems is infeasible on small embedded systems due to lack of virtual memory hardware and limited availability of physical memory. For example, 8-bit microcontrollers typically have between a few tens of bytes and a few tens of kilobytes of RAM. Bounds on stack depth can also be usefully incorporated into executable programs, for example to assign appropriate stack sizes to threads or to provide a heap allocator with as much storage as possible without compromising stack safety.

The alternative to static stack depth analysis that is currently used in industry is to ensure that memory allocated to the stack exceeds the largest stack size ever observed during testing by some safety margin. A large safety margin would provide good insurance against stack overflow, but for embedded processors used in products such as sensor network nodes and consumer electronics, the degree of overprovisioning must be kept small in order to minimize per-unit product cost. Figure 1 illustrates the relationship between the testing- and analysis-based approaches to allocating memory for the stack.

Testing-based approaches to software validation are inherently unreliable, and testing embedded software for maximum stack depth is particularly unreliable because its behavior is timing dependent: the worst observed stack depth depends on what code is executing when an interrupt is triggered and on whether further interrupts trigger before the first returns. For example, consider a hypothetical embedded system where the maximum stack depth occurs when the following events occur at almost the same time: 1) the main program summarizes data once a second spending 100 microseconds at maximum stack depth; 2) a timer interrupt fires 100 times a second spending 100 microseconds at maximum stack depth; and 3) a packet arrives on a network interface up to 10 times a second; the handler spends 100 microseconds at maximum stack depth. If these events occur independently of each other, then the worst case will occur roughly once every 10 years. This means that the worst case will probably not be discovered during testing, but will probably occur in the real world where there may be many instances of the embedded system. In practice, the events are not all independent and the timing of some events can be controlled by the test environment. However, we would expect a real system to spend less time at the worst-case stack depth and to involve more events.

Another drawback of the testing-based approach to determining stack depth is that it treats the system as a black box, providing developers with little or no feedback about how to best optimize memory usage. Static stack analysis, on the other hand, identifies the critical path through the system and also the maximum stack consumption of each function; this usually exposes obvious candidates for optimization.

Using our method for statically bounding stack depth as a starting point, we have developed a novel way to automatically reduce the stack memory requirement of an embedded system. The optimization proceeds by evaluating the effect of a large number of potential program transformations in a feedback loop, applying only transformations that reduce the worst-case depth of the stack. Static analysis makes this kind of optimization feasible by rapidly providing accurate information about a program. Testing-based approaches to learning about system behavior, on the other hand, are slower and typically only explore a fraction of the possible state space.

Our work is preceded by a stack depth analysis by Brylow et al. [3] that also performs whole-program analysis of executable programs for embedded systems. However, while they focused on relatively small programs written by hand in assembly language, we focus on programs that are up to 30 times larger, and that are compiled from C to a RISC architecture. The added difficulties in analyzing larger, compiled programs necessitated a more powerful approach based on context-sensitive abstract interpretation of machine code; we motivate and describe this approach in Section 2. Section 3 discusses the problems in experimentally validating the abstract interpretation and stack depth analysis, and presents evidence that the analysis provides accurate results. In Section 4 we describe the use of a stack bounding tool to support automatically reducing the stack memory consumption of an embedded system. Finally, we compare our research to previous efforts in Section 5 and conclude in Section 6.

2 Bounding Stack Depth

Embedded system designers typically try to statically allocate resources needed by the system. This makes systems more predictable and reliable by providing a priori bounds on resource consumption. However, an almost universal exception to this rule is that memory is dynamically allocated on the call stack. Stacks provide a useful model of storage, with constant-time allocation and deallocation and without fragmentation. Furthermore, the notion of a stack is designed into microcontrollers at a fundamental level. For example, hardware support for interrupts typically pushes the machine state onto the stack before calling a user-defined interrupt handler, and pops the machine state upon termination of the handler. For developers of embedded systems, it is important not only to know that the stack depth is bounded, but also to have a tight bound — one that is not much greater than the true worst-case stack depth. This section describes the whole-program analysis that we use to obtain tight bounds on stack depth.

Our prototype stack analysis tool targets programs for the Atmel AVR, a popular family of microcontrollers. We chose to analyze binary program images, rather than source code, for a number of reasons:

- There is no need to predict compiler behavior. Many compiler decisions, such as those regarding function inlining and register allocation, have a strong effect on stack depth.
- Inlined assembly language is common in embedded systems, and a safe analysis must account for its effects.
- The source code for libraries and real-time operating systems are commonly not available for analysis.
- Since the analysis is independent of the compiler, developers are free to change compilers or compiler versions. In addition, the analysis is not fragile with respect to non-standard language extensions that embedded compilers commonly use to provide developers with fine-grained control over processor-specific features.
- Adding a post-compilation analysis step to the development process presents developers with a clean usage model.

2.1 Analysis Overview and Motivation

The first challenge in bounding stack depth is to measure the contributions to the stack of each interrupt handler and of the main program. Since indirect function calls and recursion are uncommon in embedded systems [4], a callgraph for each entry point into the program can be constructed using standard analysis techniques. Given a callgraph it is usually straightforward to compute its stack requirement.

The second, more difficult, challenge in embedded systems is accurately estimating interactions between interrupt handlers and the main program to compute a maximum stack depth for the whole system. If interrupts are disabled while running interrupt handlers, one can safely estimate the stack bound of a system containing n interrupt handlers using this formula:

$$\text{stack bound} = \text{depth(main)} + \max_{i=1..n} \text{depth(interrupt}_i\text{)}$$

However, interrupt handlers are often run with interrupts enabled to ensure that other interrupt handlers are able to meet real-time deadlines. If a system permits at most one concurrent instance of each interrupt handler, the worst-case stack depth of a system can be computed using this formula:

$$\text{stack bound} = \text{depth(main)} + \sum_{i=1..n} \text{depth(interrupt}_i\text{)}$$

Unfortunately, as we show in Section 3, this simple formula often provides unnecessarily pessimistic answers when used to analyze real systems where only some parts of some interrupt handlers run with interrupts enabled.

```
in      r24, 0x3f    ; r24 <- CPU status register
cli                  ; disable interrupts
adc     r24, r24     ; carry bit <- prev interrupt status
eor     r24, r24     ; r24 <- 0
adc     r24, r24     ; r24 <- carry bit
mov     r18, r24     ; r18 <- r24

... critical section ...

and     r18, r18     ; test r18 for zero
breq    .+2          ; if zero, skip next instruction
sei                  ; enable interrupts
ret                  ; return from function
```

Fig. 2. This fragment of assembly language for Atmel AVR microcontrollers motivates our approach to program analysis and illustrates a common idiom in embedded software: disable interrupts, execute a critical section, and then reenable interrupts only if they had previously been enabled

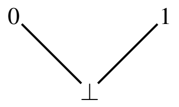
To obtain a safe, tight stack bound for realistic embedded systems, we developed a two-part analysis. The first must generate an accurate estimate of the state of the processor's interrupt mask at each point in the program, and also the effect of each instruction on the stack depth. The second part of the analysis — unlike the first — accounts for potential preemptions between interrupts handlers and can accurately bound the global stack requirement for a system.

Figure 2 presents a fragment of machine code that motivates our approach to program analysis. Analogous code can be found in almost any embedded system: its purpose is to disable interrupts, execute a critical section that must run atomically with respect to interrupt handlers, and then reenable interrupts only if they had previously been enabled. There are a number of challenges in analyzing such code.

First, effects of arithmetic and logical operations must be modeled with enough accuracy to track data movement through general-purpose and special-purpose registers. In addition, partially unknown data must be modeled. For example, analysis of the code fragment must succeed even when only a single bit of the CPU status register — the master interrupt control bit — is initially known.

Second, dead edges in the control-flow graph must be detected and avoided. For example, when the example code fragment is called in a context where interrupts are disabled, it is important that the analysis conclude that the `sei` instruction is not executed since this would pollute the estimate of the processor state at subsequent addresses.

Finally, to prevent procedural aliasing from degrading the estimate of the machine state, a context sensitive analysis must be used. For example, in some systems the code in Figure 2 is called with interrupts disabled by some parts of the system and is called with interrupts enabled by other parts of the system. With a context-insensitive approach, the analysis concludes that since the initial state of the interrupt flag can vary, the final state



(a) Lattice for each bit in the machine state

and

	1	0	$\perp$
1	1	0	$\perp$
0	0	0	0
$\perp$	$\perp$	0	$\perp$

or

	1	0	$\perp$
1	1	1	1
0	1	0	$\perp$
$\perp$	1	$\perp$	$\perp$

xor

	1	0	$\perp$
1	0	1	$\perp$
0	1	0	$\perp$
$\perp$	$\perp$	$\perp$	$\perp$

merge

	1	0	$\perp$
1	1	$\perp$	$\perp$
0	$\perp$	0	$\perp$
$\perp$	$\perp$	$\perp$	$\perp$

(b) Logical operations on abstract bits and combining machine states at merge points

Fig. 3. Modeling machine states and operations in the abstract interpretation

of the interrupt flag can also vary and so analysis of both callers of the function would proceed with the interrupt flag unknown. This can lead to large overestimates in stack bounds since unknown values are propagated to any code that could execute after the call. With a context-sensitive analysis the two calls are analyzed separately, resulting in an accurate estimate of the interrupt state.

The next section describes the abstract interpretation we have developed to meet these challenges.

2.2 Abstracting the Processor State

The purpose of our abstract interpretation is to generate a safe, precise estimate of the state of the processor at each point in the program; this is a requirement for finding a tight bound on stack depth. Designing the abstract interpretation boils down to two main design decisions.

First, how much of the machine state should the analysis model? For programs that we have analyzed, it is sufficient to model the program counter, general-purpose registers, and several I/O registers. Atmel AVR chips contain 32 general-purpose registers and 64 I/O registers; each register stores eight bits. From the I/O space we model the registers that contain interrupt masks and the processor status register. We do not model main memory or most I/O registers, such as those that implement timers, analog-to-digital conversion, and serial communication.

Second, what is the abstract model for each element of machine state? We chose to model the machine at the bit level to capture the effect of bitwise operations on the interrupt mask and condition code register — we had initially attempted to model the machine at word granularity and this turned out to lose too much information through conservative approximation. Each bit of machine state is modeled using the lattice depicted in Figure 3(a). The lattice contains the values 0 and 1 as well as a bottom element, $\perp$, that corresponds to a bit that cannot be proven to have value 0 or 1 at a particular program point.

Figure 3(b) shows abstractions of some common logical operators. Abstractions of operators should always return a result that is as accurate as possible. For example, when all bits of the input to an instruction have the value 0 or 1, the execution of the instruction should have the same result that it would have on a real processor. In this respect our abstract interpreter implements most of the functionality of a standard CPU simulator.

For example, when executing the `and` instruction with $\{1,1,0,0,1,1,0,0\}$ as one argument and $\{\perp,\perp,\perp,\perp,1,1,1,1\}$ as the other argument, the result register will contain the value $\{\perp,\perp,0,0,1,1,0,0\}$. Arithmetic operators are treated similarly, but require more care because bits in the result typically depend on multiple bits in the input. Furthermore, the abstract interpretation must take into account the effect of instructions on processor condition codes, since subsequent branching decisions are made using these values.

The example in Figure 2 illustrates two special cases that must be accounted for in the abstract interpretation. First, the add-with-carry instruction `adc`, when both of its arguments are the same register, acts as rotate-left-through-carry. In other words, it shifts each bit in its input one position to the left, with the leftmost bit going into the CPU's carry flag and the previous carry flag going into the rightmost bit. Second, the exclusive-or instruction `eor`, when both of its arguments are the same register, acts like a clear instruction — after its execution the register is known to contain all zero bits regardless of its previous contents.

2.3 Managing Abstract Processor States

An important decision in designing the analysis was when to create a copy of the abstract machine state at a particular program point, as opposed to merging two abstract states. The merge operator, shown in Figure 3(b), is lossy since a conservative approximation must always be made. We have chosen to implement a context-sensitive analysis, which means that we fork the machine state each time a function call is made, and at no other points in the program. This has several consequences. First, and most important, it means that the abstract interpretation is not forced to make a conservative approximation when a function is called from different points in the program where the processor is in different states. In particular, when a function is called both with interrupts enabled and disabled, the analysis is not forced to conclude that the status of the interrupt bit is unknown inside the function and upon return from it. Second, it means that we cannot show termination of a loop implemented within a function. This is not a problem at present since loops are irrelevant to the stack depth analysis as long as there is no net change in stack depth across the loop. However, it will become a problem if we decide to push our analysis forward to bound heap allocation or execution time. Third, it means that we can, in principle, detect termination of recursion. However, our current implementation rarely does so in practice because most recursion is bounded by values that are stored on the stack — which our analysis does not model. Finally, forking the state at function calls means that the state space of the stack analyzer might become large. This has not been a problem in practice; the largest programs that we have analyzed cause the analyzer to allocate about 140 MB. If memory requirements become a problem for the analysis, a relatively simple solution would be to merge program states that are identical or that are similar enough that a conservative merging will result in minimal loss of precision.

2.4 Abstract Interpretation and Stack Analysis Algorithms

The program analysis begins by initializing a worklist with all entry points into the program; entry points are found by examining the vector of interrupt handlers that is stored at the bottom of a program image, which includes the address of a startup routine that eventually jumps to `main()`. For each item in the worklist, the analyzer abstractly interprets a single instruction. If the interpretation changes the state of the processor at that program point, items are added to the worklist corresponding to each live control flow edge leaving the instruction. Termination is assured because the state space for a program is finite and because we never revisit states more than once.

The abstract interpretation detects control-flow edges that are dead in a particular context, and also control-flow edges that are dead in all contexts. In many systems we have analyzed, the abstract interpretation finds up to a dozen branches that are provably not taken. This illustrates the increased precision of our analysis relative to the dataflow analysis that an optimizing compiler has previously performed on the embedded program as part of a dead code elimination pass.

In the second phase, the analysis considers there to be a control flow edge from every instruction in the program to the first instruction of every interrupt handler that cannot be proven to be disabled at that program point. An interrupt is disabled if either the master interrupt bit is zero or the enable bit for the particular interrupt is zero. Once these edges are known, the worst-case stack depth for a program can be found using the method developed by Brylow et al. [3]: perform a depth-first search over control flow edges, explicit and implicit, keeping track of the effect of each instruction on the stack depth, and also keeping track of the largest stack depth seen so far.

A complication that we have encountered in many real programs is that interrupt handlers commonly run with all interrupts enabled, admitting the possibility that a new instance of an interrupt handler will be signaled before the previous instance terminates. From an analysis viewpoint reentrant interrupt handlers are a serious problem: systems containing them cannot be proven to be stack-safe without also reasoning about time. In effect, the stack bounding problem becomes predicated on the results of a real-time analysis that is well beyond the current capabilities of our tool.

In real systems that we have looked at reentrant interrupt handlers are so common that we have provided a facility for working around the problem by permitting a developer to manually assert that a particular interrupt handler can preempt itself only up to a certain number of times. Programmers appear to commonly rely on ad hoc real-time reasoning, e.g., “this interrupt only arrives 10 times per second and so it cannot possibly interrupt itself.” In practice, most instances of this kind of reasoning should be considered to be design flaws — few interrupt handlers are written in a reentrant fashion so it is usually better to design systems where concurrent instances of a single handler are not permitted. Furthermore, stack depth requirements and the potential for race conditions will be kept to a minimum if there are no cycles in the interrupt preemption graph, and if preemption of interrupt handlers is only permitted when necessary to meet a real-time deadline.

2.5 Other Challenges

In this section we address other challenges faced by the stack analysis tool: loads into the stack pointer, self-modifying code, indirect branches, indirect stores, and recursive function calls. These features can complicate or defeat static analysis. However, embedded developers tend to make very limited use of them, and in our experience static analysis of real programs is still possible and, moreover, effective.

We support code that increments or decrements the stack pointer by constants, for example to allocate or deallocate function-scoped data structures. Code that adds non-constants to the stack pointer (e.g., to allocate variable sized arrays on the stack) would require some extra work to bound the amount of space added to the stack. We also do not support code that changes the stack pointer to new values in a more general way, as is done in the context switch routine of a preemptive operating system.

The AVR has a Harvard architecture, making it possible to prove the absence of self-modifying code simply by ensuring that a program cannot reach a "store program memory" instruction. However, by reduction to the halting problem, self-modifying code cannot be reliably detected in the general case. Fortunately, use of self-modifying code is rare and discouraged — it is notoriously difficult to understand and also precludes reducing the cost of an embedded system by putting the program into ROM.

Our analysis must build a conservative approximation of the program's control flow graph. Indirect branches cause problems for program analysis because it can be difficult to tightly bound the set of potential branch targets. Our approach to dealing with indirect branches is based on the observation that they are usually used in a structured way, and the structure can be exploited to learn the set of targets. For example, when analyzing TinyOS [6] programs, the argument to the function `TOS_post` is usually a literal constant representing the address of a function that will be called by an event scheduling loop. The value of the argument is identified by abstract interpretation, but the connection between posting an event and making the indirect call must be established manually. Making it easier to express such connections is an area of future work.

The stack analysis cannot deal with the form of indirect branch found in the context switch routine of a preemptive real-time operating system — the set of potential targets is too large. However, these branches need not be analyzed: since switching context to a new thread involves a change to a completely separate stack, it suffices to learn the worst-case stack usage of the operating system code and add it to the worst-case stack usage for each thread.

Memory writes can compromise our static analysis in two ways. First, an out-of-bounds store may overwrite a return address on the call stack, causing the program to return to an unforeseen location. Second, since the AVR maps its registers into low memory, a stray write could change a register in a way invisible to the analysis. We deal with direct stores by ensuring that they reference an appropriate range of RAM that is not occupied by the registers or call stack. Indirect stores are simply assumed not to overwrite a register or return address; our rationale is that a program containing this behavior is so flawed that its stack safety is irrelevant. In the long run a better solution would be to construct embedded software in a type-safe language.

Recursive code is uncommon in embedded software. For example, Engblom [4] studied a collection of embedded systems containing over 300,000 lines of C code, and it contained only 14 recursive loops. Our approach to dealing with recursion, therefore, is blunt: we require that developers explicitly specify a maximum iteration count for each recursive loop in a system. The analysis returns an unbounded stack depth if the developers neglect to specify a limit for a particular loop.

It would be straightforward to port our stack analyzer to other processors: the analysis algorithms, such as the whole-program analysis for worst-case stack depth, operate on an abstract representation of the program that is not processor dependent. However, the analysis would return pessimistic results for register-poor architectures such as the Motorola 68HC11, since code for those processors makes significant use of the stack, and stack values are not currently modeled by our tool. In particular, we would probably not obtain precise results for code equivalent to the code in Figure 2 that we used to motivate our approach. To handle register-poor architectures we are developing an approach to modeling the stack that is based on a simple type system for registers that are used as pointers into stack frames.

2.6 Using the Stack Tool

We have a prototype tool that implements our stack depth analysis. In its simplest mode of usage, the stack tool returns a single number: an upper bound on the stack depth for a system. For example:

```
$ ./stacktool -w flybywire.elf
total stack requirement from global analysis = 55
```

To make the tool more useful we provide a number of extra features, including switching between context-sensitive and context-insensitive program analysis, creating a graphical callgraph for a system, listing branches that can be proven to be dead in all contexts, finding the shortest path through a program that reaches the maximum stack depth, and printing a disassembled version of the embedded program with annotations indicating interrupt status and worst-case stack depth at each instruction. These are all useful in helping developers understand and manually reduce stack memory consumption in their programs.

There are other obvious ways to use the stack tool that we have not yet implemented. For example, using stack bounds to compute the maximum size of the heap for a system so that it stops just short of compromising stack safety, or computing a minimum safe stack size for individual threads in a multi-threaded embedded system. Ideally, the analysis would become part of the build process and values from the analysis would be used directly in the code being generated.

3 Validating the Analysis

We used several approaches to increase our confidence in the validity of our analysis techniques and their implementations.

3.1 Validating the Abstract Interpretation

To test the abstract interpretation, we modified a simulator for AVR processors to dump the state of the machine after executing each instruction. Then, we created a separate program to ensure that this concrete state was "within" the conservative approximation of the machine state produced by abstract interpretation at that address, and that the simulator did not execute any instructions that had been marked as dead code by the static analysis. During early development of the analysis this was helpful in finding bugs and in providing a much more thorough check on the abstract interpretation than manual inspection of analysis results — our next-best validation technique. We have tested the current version of the stack analysis tool by executing at least 100,000 instructions of about a dozen programs, including several that were written specifically to stress-test the analysis, and did not find any discrepancies.

3.2 Validating Stack Bounds

There are two important metrics for validating the bounds returned by the stack tool. The first is qualitative: Does the tool ever return an unsafe result? Testing the stack tool against actual execution of about a dozen embedded applications has not turned up any examples where it has returned a bound that is less than an observed stack depth. This justifies some confidence that our algorithms are sound.

Our second metric is quantitative: Is the tool capable of returning results that are close to the true worst-case stack depth for a system? The maximum observed stack depth, the worst-case stack depth estimate from the stack tool, and the (non-computable) true worst-case stack depth are related in this way:

$$\text{worst observed} \leq \text{true worst} \leq \text{estimated worst}$$

One might hope that the precision of the analysis could be validated straightforwardly by instrumenting some embedded systems to make them report their worst observed stack depth and comparing these values to the bounds on stack depth. For several reasons, this approach produces maximum observed stack depths that are significantly smaller than the estimated worst case and, we believe, the true worst case. First, the timing issues that we discussed in Section 1 come into play, making it very hard to observe interrupt handlers preempting each other even when it is clearly possible that they may do so. Second, even within the main function and individual interrupt handlers, it can be very difficult to force an embedded system to execute the code path that produces the worst-case stack depth. Embedded systems often present a narrower external interface than do traditional applications, and it is correspondingly harder to force them to execute certain code paths using test inputs. While the difficulty of thorough testing is frustrating, it does support our thesis that static program analysis is particularly important in this domain.

The 71 embedded applications that we used to test our analysis come from three families. The first is Autopilot, a simple cyclic-executive style control program for an autonomous helicopter [10]. The second is a collection of application programs that are distributed with TinyOS version 0.6.1, a small operating system for networked sensor nodes. The third is a collection of application programs that are distributed with TinyOS 1.0 [6]. Version 1.0 is a complete rewrite of TinyOS using nesC [5], a programming language very similar to C that is compiled by translating into C. All programs were compiled from C using gcc version 3.0.2 or 3.1.1, and all target the ATmega103 chip, a member of the Atmel AVR family that contains 4 KB of RAM and 128 KB of flash memory.

3.3 Validating Analysis of Individual Interrupts

To quantitatively evaluate the stack tool, we wrote a program that modifies the assembly language version of an AVR program in such a way that each interrupt is handled on its own stack. This makes stack measurement timing-independent, but still leaves the difficult problem of making the main function and each interrupt handler execute the path to the worst-case stack depth.

We found that a perfect match between predicted and actual stack depth could only be obtained for slightly modified versions of simple embedded applications such as the `BlinkTask` TinyOS kernel whose function is to flash an LED. Even for this example, we were forced to comment out a call to a function supporting one-shot timers in a timer module: it contributed to the worst-case stack depth, but could never be called in our system. After making this small modification and adding serial-line driver code to enable reporting of stack depths to a separate computer, the `BlinkTask` application contained about 4000 lines of C code, even after a dead-code elimination pass performed by nesC. Running the stack analysis on this modified kernel produced the following results:

```
Stack depths:
   vector  0 main               = 33, at d30
   vector 15 _output_compare0_  = 32, at 50a
   vector 18 _uart_recv_        = 27, at 3e8
   vector 20 _uart_trans_       = 23, at a90
```

This shows the estimated worst-case stack depth of each entry point into the program and also an address at which this depth is reached. We then ran this kernel on an AVR processor and queried it to learn the worst observed stack depth; it reported the same stack depths as the analysis reported.

3.4 Evaluating the Global Analysis

Of the 71 applications used to test our analysis, there are seven that defeat our analysis tool, for example because they make an indirect jump based on a value that is not a literal constant, or they load an indeterminate value into the stack pointer. We believe that some of these applications could be successfully analyzed by a slightly improved version of our stack tool, but for now we disregard them.

The stack analysis results from the remaining 64 kernels are too large to display in a figure so we have chosen, at random, 15 TinyOS 0.6.1 applications and 15 TinyOS 1.0 applications and displayed them in Figure 4. Analysis of the results for all 64 applications shows that on average, the context insensitive global analysis returns a stack bound that is 15% lower than the bound determined by summing the requirements of interrupt handlers and the main function, and that on average the context sensitive analysis returns a bound that is 35% lower than the bound computed by summing interrupts.

Another metric that can be used to compare the context sensitive and insensitive analyses is the number of instructions for which each method fails to identify a concrete interrupt mask. The context insensitive analysis was unable to identify a concrete interrupt mask for 41% of the instructions in a given kernel, on average. The context sensitive analysis, on the other hand, failed to identify a concrete value for the interrupt

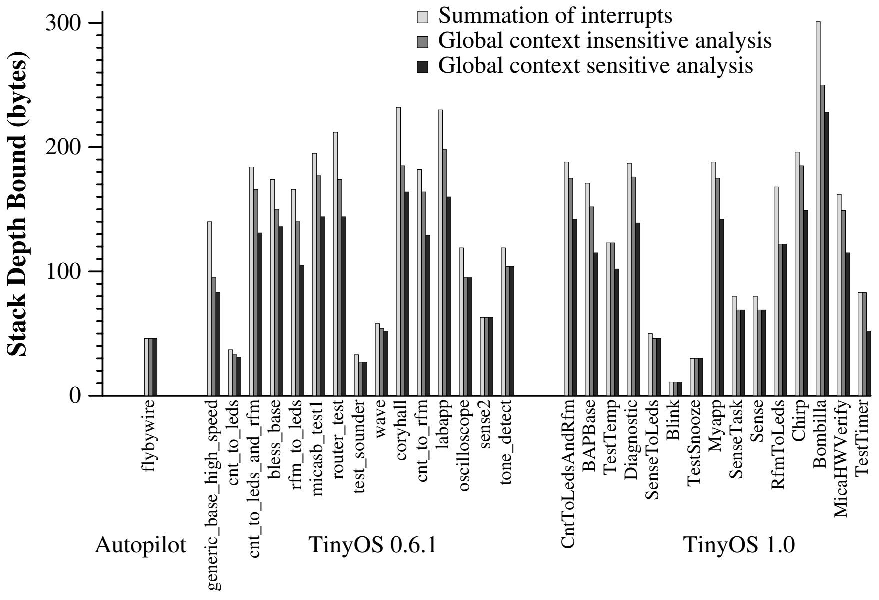


Fig. 4. Comparing stack bounds for summation of interrupts, global context insensitive analysis, and context sensitive analysis

mask for only 2.2% of instructions, which mostly fall into two categories. First, instructions whose interrupt mask would become analyzable if the call stack were modeled. Second, instructions whose interrupt mask is genuinely ambiguous — within a single context they can be shown to execute both with interrupts disabled and enabled.

Since increased precision in the analysis translates directly into memory savings for embedded developers, we believe that the added complexity of the context-sensitive analysis is justified. In most cases where the more powerful analysis did not decrease the stack bound — for example, the Autopilot application — there was simply nothing that the tool could do: these applications run all interrupt handlers with interrupts enabled, precluding tight bounds on stack depth. Finally, the stack depth analysis requires less than four seconds of CPU time on a 1.4 GHz Athlon for all of our example programs, and for many applications it requires well under one second.

4 Reducing Stack Depth

Previous sections described and evaluated a tool for bounding stack depth. In this section we go a step further by exploring the use of the stack bounding tool as an essential part of a method for automatically reducing the stack memory requirements of embedded software. Reducing stack depth is useful because it potentially frees up more storage for the heap, permits more threads to be run on a given processor, or permits a product to be based on a less expensive CPU with fewer on-chip resources.

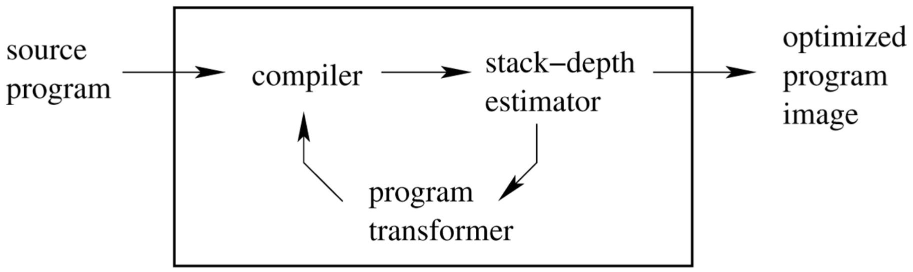


Fig. 5. Overview of stack depth reduction

The basic observation that makes stack minimization possible is that given a way to quickly and accurately bound the stack depth of a program, it becomes possible for a compiler or similar tool to rapidly evaluate the effect of a large number of program transformations on the stack requirements of a system. We then choose to apply only the transformations that improve stack memory usage.

Figure 5 illustrates our approach to automatic stack depth reduction. Although this technique is generic and would admit a variety of program transformations, so far the only transformation we have experience with is global function inlining. Inlining is a common optimization that replaces a function call with a copy of the function body. The immediate effect of function inlining on stack usage is to avoid the need to push a return address and function arguments onto the stack. More significantly, inlining allows intraprocedural optimizations to apply which may simplify the code to the extent that fewer temporary variables are required, which may further reduce stack usage. Inlining also allows better register allocation since the compiler considers the caller and the callee simultaneously instead of separately. In general, inlining needs to be used sparingly. If a function is inlined many times, the size of the compiled binary can increase. Furthermore, aggressive inlining can actually increase stack memory requirements by overloading the compiler's register allocator. In previous work [11], we developed a global function inliner for C programs that can perform inlining on complete programs instead of within individual C files. To support the work reported in this paper, we modified this inliner to accept an explicit list of callgraph edges to inline.

To reduce the stack depth requirements of an embedded system we perform a heuristic search that attempts to minimize the *cost* of a program, where cost is a user-supplied function of stack depth and code size. For example, one obvious cost function minimizes stack depth without regard to code size. Another useful cost function is willing to trade one byte of stack memory for 32 bytes of code memory, since the processors we currently use have 32 times more code memory than data memory.

Systems that we have analyzed contain between 80 and 670 callgraph edges that could be inlined, leading in the worst case to 2^{670} possible inlining decisions. Since this space obviously cannot be searched exhaustively, we use a heuristic search. We have found that an effective approach is to bound the degree of inlining "from above" and "from below" and then perform a random search of the space in between. Minimizing

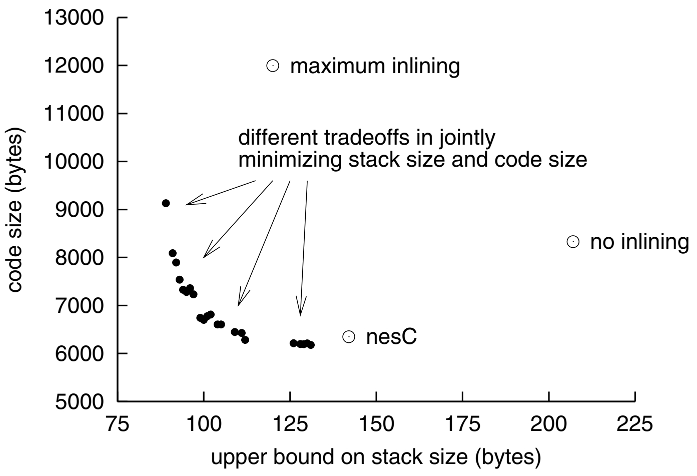


Fig. 6. Comparing stack reduction tradeoffs with default compilation methods for an example TinyOS kernel. Note that the origin of the graph is not at 0, 0.

code size is often best accomplished by starting with no functions inlined and then repeatedly picking an uninlined function and inlining it only if this improves the cost metric. Minimizing stack depth, on the other hand, is often best accomplished by starting with all functions inlined and then repeatedly picking an inlined function and dropping the inlining if this improves the cost metric. To see why this is more effective at reducing stack depth, consider a system where there are two paths to the maximum stack depth. Separately considering inlining decisions on either path will not improve the stack depth: it is only by considering both inlinings at once that their benefit is seen.

Having found upper and lower bounds on the inlining decisions, we search the space between the bounds by accepting inlinings where the previous solutions agreed, and then repeatedly test inlinings that they disagreed on. In practice this step often finds solutions missed by the previous two steps.

Figure 6 shows the results of applying the stack depth / code size reduction algorithm to the TinyOS kernel `CntToLedsAndRfm`. There are three data points corresponding respectively to a system compiled without any function inlining, to a system compiled with as much inlining as possible (subject to limitations on inlining recursive functions and indirect calls), and to a system compiled by the nesC compiler [5], which performs fairly aggressive global function inlining on its own. The remaining data points were collected by running our stack reduction algorithm with a variety of cost functions ranging from those that gave high priority to reducing stack depth to those that gave high priority to reducing code size. These results are typical: we applied stack depth reduction to a number of TinyOS kernels and found that we could usually use about 40% of the stack required by a kernel without any inlining, and about 68% of the stack required by kernels compiled using nesC.

5 Related Work

The previous research most closely related to our work is the stack depth analysis by Brylow et al. [3]. Their analysis was designed to handle programs written by hand that are on the order of 1000 lines of assembly language; the programs we analyze, on the other hand, are compiled and are up to 30 times larger. Their contribution was to model interrupt-driven embedded systems, but their method could only handle immediate values loaded into the interrupt mask register — an ineffective technique when applied to software where all data, including interrupt masks, moves through registers. Our work goes well beyond theirs through its use of an aggressive abstract interpretation of ALU operations, conditional branches, etc. to track the status of the interrupt mask.

Palsberg and Ma [9] provide a calculus for reasoning about interrupt-driven systems and a type system for checking stack boundedness. Like us, they provide a degree of context sensitivity (in their type system this is encoded using intersection types). Unlike us, they model just the interrupt mask register, which would prevent them from accurately modeling our motivating example in Figure 2. The other major difference is that their focus is on the calculus and its formal properties and so they restrict their attention to small examples (10–15 instructions) that can be studied in detail, and they restrict themselves to a greatly simplified language that lacks pointers and function calls.

AbsInt makes a commercial product called StackAnalyzer [1]; its goal is to estimate stack depth in embedded software. We were not able to find much information about this tool. In particular, there is no indication that it is attempting to model the status of the interrupt mask, the most important feature of our analysis.

Our abstract interpretation is largely a combination of standard techniques. Java virtual machines perform an intraprocedural stack depth analysis [8], and program modeling at the bit-level has been done before (see, for example, the discussion of possible lattices for MIT's Bitwise project [12]). Our contribution here lies in determining which combination of techniques obtains good results for our particular problem.

Function inlining has traditionally been viewed as a performance optimization [2] at the cost of a potentially large increase in code size. More recent work [7] has examined the use of inlining as a technique to reduce both code size and runtime. We are not aware of any previous work that uses function inlining specifically to reduce stack size or, in fact, of any previous work on automatically reducing the stack memory requirements of embedded software.

6 Conclusion

The potential for stack overflow in embedded systems is hard to detect by testing. We have developed a static analysis that can prove that an embedded system will not overflow its stack, and demonstrated that the analysis provides accurate results. Experiments show that modeling the enabling and disabling of interrupt handlers using context sensitive abstract interpretation produces estimates that are an average of 35% lower than estimates produced using the simpler approach of summing the stack requirements of interrupt handlers and the main function. We have also demonstrated a novel use of this analysis to drive a search for function inlining decisions that reduce stack depth. Experiments on a number of component-based embedded applications show that this approach reduces stack memory requirements by an average of 32% compared with aggressive global inlining without the aid of a stack depth analysis.

Availability. Source code for the stack analyzer is available under `http://www.cs.utah.edu/flux/stacktool`, and the global inliner can be downloaded from `http://www.cs.utah.edu/flux/alchemy/software.html`.

Acknowledgments. The authors would like to thank Dennis Brylow, Eric Eide, Matthew Flatt, Wilson Hsieh, Jay Lepreau, and Michael Nahas for providing helpful feedback on drafts of this paper.

References

1. AbsInt. StackAnalyzer. `http://www.absint.com/stackanalyzer`.
2. Andrew Ayers, Robert Gottlieb, and Richard Schooler. Aggressive inlining. In *Proc. of Programming Language Design and Implementation*, pages 134–145, Las Vegas, NV, June 1997.
3. Dennis Brylow, Niels Damgaard, and Jens Palsberg. Static checking of interrupt-driven software. In *Proc. of the 23rd Intl. Conf. on Software Engineering*, pages 47–56, Toronto, Canada, May 2001.
4. Jakob Engblom. Static properties of commercial embedded real-time programs, and their implication for worst-case execution time analysis. In *Proc. of the 5th IEEE Real-Time Technology and Applications Symp.*, Vancouver, Canada, June 1999.
5. David Gay, Phil Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesC language: A holistic approach to networked embedded systems. In *Proc. of Programming Language Design and Implementation*, pages 1–11, San Diego, CA, June 2003.
6. Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for networked sensors. In *Proc. of the 9th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 93–104, Cambridge, MA, November 2000.
7. Rainer Leupers and Peter Marwedel. Function inlining under code size constraints for embedded processors. In *Proc. of the International Conference on Computer-Aided Design*, pages 253–256, San Jose, CA, November 1999.
8. Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* The Java Series. Addison-Wesley, January 1997.
9. Jens Palsberg and Di Ma. A typed interrupt calculus. In *Proc. of Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 2469 of *Lecture Notes in Computer Science*, pages 291–310. Springer Verlag, 2002.
10. The Autopilot Project. `http://autopilot.sourceforge.net`.
11. Alastair Reid, Matthew Flatt, Leigh Stoller, Jay Lepreau, and Eric Eide. Knit: Component composition for systems software. In *Proc. of the 4th Symp. on Operating Systems Design and Implementation*, pages 347–360, San Diego, CA, October 2000.
12. Mark Stephenson, Jonathan Babb, and Saman Amarasinghe. Bitwidth analysis with application to silicon compilation. In *Proc. of Programming Language Design and Implementation*, pages 108–120, Vancouver, Canada, June 2000.

Event Correlation: Language and Semantics

César Sánchez, Sriram Sankaranarayanan, Henny Sipma,
Ting Zhang, David Dill, and Zohar Manna [⋆]

Computer Science Department
Stanford University
{cesar,srirams,sipma,tingz,dill,zm}@CS.Stanford.EDU

Abstract. Event correlation is a service provided by middleware platforms that allows components in a publish/subscribe architecture to subscribe to patterns of events rather than individual events. Event correlation improves the scalability and performance of distributed systems, increases their analyzability, while reducing their complexity by moving functionality to the middleware. To ensure that event correlation is provided as a standard and reliable service, it must possess well-defined and unambiguous semantics.

In this paper we present a language and formal model for event correlation with operational semantics defined in terms of transducers. The language has been motivated by an avionics application and includes constructs for modes in addition to the more common constructs such as selection, accumulation and sequential composition. Prototype event processing engines for this language have been implemented in both C++ and Java and are now being integrated with third-party event channels.

1 Introduction

Publish-subscribe is an event-based model of distributed systems that decouples publishers from consumers. Consumers register their interests with the middleware by means of a subscription policy. Events published to the middleware are delivered only to those consumers that expressed an interest. The publish-subscribe paradigm is useful in practice for building large systems in which not all components are known at design time, or components may be dynamically added at runtime (such as in mobile systems). These systems may even extend world wide. Crowcroft et al [5] predict a world in which pervasive computing devices generate 10,000,000,000 events per second, with billions of subscribers from all over the planet. Clearly, extensive filtering and correlation are required at multiple levels and locations to manage these volumes.

Several middleware platforms provide event notification services. Examples include Gryphon [1], Ace-Tao [19], Siena [3], Elvin [2]. An overview and comparison of such systems is in [15]. In some middleware platforms, such as the

[⋆] This research was supported in part by NSF grants CCR-01-21403, CCR-02-20134 and CCR-02-09237, by ARO grant DAAD19-01-1-0723, and by ARPA/AF contracts F33615-00-C-1693 and F33615-99-C-3014.

real-time event channel (RTEC) of ACE-TAO [19], consumers subscribe with the middleware with a list of event types or sources they wish to receive. Suppliers publish their events to the RTEC, which then distributes them to the subscribed consumers. This service is called event filtering.

Event correlation extends filtering with more complex subscriptions, including combinations of events and temporal patterns. Providing event correlation as a standard middleware service further enhances the performance of embedded applications. However, more importantly, it enables the transfer of functionality from application components to the middleware, which

– reduces software development and maintenance cost by decreasing the number of special-purpose components to be developed, as complexity is factored out of the component into the middleware;
– increases reliability, as this service can be verified and tested once as part of the platform and reused many times;
– increases the analyzability of the system, and accuracy of schedulability. Where event dependencies were before largely hidden inside application components, with event correlation provided, these dependencies are available explicitly to analysis tools in the form of event correlation expressions with well-defined semantics;
– increases flexibility in configuration: event correlation expressions can be changed on a shorter notice than components.

To enable event correlation to be provided as a standard, reliable middleware service, it must come with a well-defined, unambiguous semantics. In addition, it is desirable that the event-processing code be generated automatically from the (declarative) event correlation expressions. This ensures adherence to the semantics by construction.

In this paper, we present a language and a computational model for event correlation that provides an unambiguous semantics for event correlation expressions. The model is based on automata/transducers, a well-studied domain with a large body of analysis methods and tools. Another attractive property of transducers is that it is an operational model: they can be used directly in the RTEC to process events interpretatively, or used to automatically generate the code to process the events.

The development of this language was initiated to enable the use of event correlation in the Boeing Open Experimental Platform for the DARPA Information Exploitation Office Program Composition for Embedded Systems (PCES) program, based on Boeing's Bold Stroke avionics architecture [20]. Some of the constructs included in the language were directly motivated by Boeing product scenarios which illustrate important component level interactions in associated avionics applications.

The paper is organized as follows. In section 2 we give an intuitive overview of the features of a simple version of our event correlation expression language. In section 3 we present the new model of *correlation machines* and its extension *correlation modules* that are used to define the operational semantics of the constructs in the correlation language. This translation is described in section 4.

In section 5 we briefly present some related work. Finally, section 6 contains the conclusions and outlines some directions for future research.

2 Event Correlation Language

Syntax: An ECL expression is constructed out of *predicates* over events, to which we apply the combination operators shown below.

A predicate formula is itself a correlation expression. If $\phi, \phi_1 \dots \phi_n$ are correlation expressions then so are

$\mathbf{accumulate}\{\phi_1, \dots, \phi_n\}$	$\mathbf{fail}\{\phi\}$
$\mathbf{select}\{\phi_1, \dots, \phi_n\}$	$\mathbf{push}(x)\{\phi\}$
$\mathbf{sequential}\{\phi_1, \dots, \phi_n\}$	$\mathbf{persist}\{\phi\}$
$\mathbf{do}\{\phi_1\}\mathbf{unless}\{\phi_2\}$	$\mathbf{repeat}\{\phi\}$
$\mathbf{parallel}\{\phi_1, \dots, \phi_n\}$	$\mathbf{loop}\{\phi\}$

An informal description of each of the constructs follows.

In general, an expression is evaluated over an input stream of events. On any event the evaluation may complete successfully, complete in failure, or may not complete. processed.

Basic Constructs: The lowest-level construct is a *simple event predicate* on a single event. This may range from being an enumeration on a finite alphabet to first-order predicates on the source, timestamp, type, and data content of the event. In the context of the language described here we are only concerned with whether or not an event satisfies the predicate. In general we assume we have an effective way of deciding this. To simplify this presentation we assume a finite alphabet of input events.

The evaluation of a predicate expression completes successfully when the event received satisfies the predicate. It does not complete on any other event. In particular, the expression never fails: events that do not satisfy the predicate are simply ignored.

Simple predicate expressions may be combined using the standard boolean operators with the usual semantics into *compound predicate* expressions. An example of a compound predicate expression is $\langle source = GPS \rangle \wedge \langle type = DATA \rangle$ which is satisfied by any event that meets both conditions.

The boolean constants **true** and **false** are available as trivial predicate expressions. The predicate **true** completes successfully upon the reception of any event; **false** is never satisfied, and hence never completes. Note that it does not complete in failure either (it simply blocks waiting for an event satisfying "false").

Simple Correlator Expressions: Predicate expressions can be combined into *correlator expressions* that may need to consume multiple events before they complete.

The most commonly seen event correlation constructs are the accumulation and selection operators, with various semantics. In our semantics the evaluation of an accumulation expression evaluates all subexpressions in parallel. It completes successfully when all subexpressions have completed successfully, and it completes with failure when one of the subexpressions results in failure. The selection operator is the dual of the accumulation operator. Here the subexpressions are also evaluated in parallel, but the evaluation completes if one of the subexpressions completes successfully and fails if all subexpressions complete in failure. An alternative semantics of selection, denoted by **select*** is sometimes useful, in which the evaluation completes in failure when one of the subexpressions ends in failure.

A sequential operator is less commonly seen in the popular middle-ware platforms. However it is useful in forming more complex patterns. The evaluation of a sequential expression proceeds sequentially, where the evaluation of ϕ_{i+1} is started after successful completion of ϕ_i. The expression completes successfully upon successful completion of ϕ_n. It ends in failure when any of the subexpressions ends in failure.

None of the constructs presented so far can lead to completion in failure. (Note that the accumulation and selection operator can propagate failure, but not cause it.) The do-unless operator is the first operator that can cause an expression to end in failure. It was inspired by the, less general, sequential unless operator in GEM [14]. The evaluation of the do-unless expression evaluates the two subexpressions in parallel. It completes successfully when ϕ_1 completes successfully, it completes with failure when either ϕ_1 completes in failure or ϕ_2 completes successfully. Note that completion of ϕ_2 in failure does not affect the evaluation of ϕ_1. The do-unless expression is useful to preempt other expressions, especially those that do not complete by themselves.

The *fail* operator "inverts" its argument. That is, a fail expression completes successfully when ϕ completes in failure and vice-versa.

Repetition: The language provides several constructs for repetition, which may be parameterized by the maximum of number of repetitions. The constructs differ in their handling of failure of the subexpressions. The evaluation of a repeat expression repeats ϕ, irrespective of its failure or success. With a persist expression, ϕ is repeated until success, while a loop expression repeats ϕ until failure.

Generating Output: The expressions presented so far do not generate any output. The purpose of the language is to provide a means to notify the consumer that certain patterns of events have occurred, and therefore specific operators are introduced to generate output. The simplified version of the language presented in this paper does not support the forwarding of events. The only output that can be generated are tokens from an alphabet of constants. The push expression $\mathbf{push}(x)\{\phi\}$ outputs character x upon the successful completion of ϕ, after which it completes successfully. If ϕ completes with failure no output is generated and the push expression itself completes with failure.

Parallel Expressions: Correlator expressions may be combined into parallel expressions. In the previous subsection, several subexpressions were said to be evaluated in parallel. However, their evaluations were linked in the sense that completion of one of them affected the completion or termination of the others.

On the other hand, in a parallel expression, the subexpressions are evaluated independently of each other. Thus, the completion of any subexpression does not affect the evaluation of any other. A parallel expression never completes, but it can be preempted by, for example, an unless expression. Typically, a parallel expression would contain multiple **repeat** subexpressions.

Mode Expressions: Mode expressions were directly motivated by the avionics applications that initiated this work. Mode expressions provide a convenient way to support different modes of operation. They allow simultaneous mode switching of multiple components in a system without any direct coordination between these components.

A mode expression has multiple modes, each consisting of a predicate expression called the *mode guard*, and a correlator expression. The mode guards are expected to be mutually exclusive, such that at any time exactly one mode is active.

$$\begin{array}{l}\textbf{in}\ (p_1)\ \textbf{do}\ \{\phi_1\}\\ \dots\\ \textbf{in}\ (p_n)\ \textbf{do}\ \{\phi_n\}\end{array}$$

A mode i is activated when its mode-guard p_i completes successfully. Upon activation of a new mode, evaluation of the expression in the current mode is terminated, and the evaluation of the expression associated with the new mode is started. Like the parallel expression, the mode expression does not complete by itself, but it can be preempted by other expressions.

3 Correlation Machines

ECL expressions can be viewed as temporal filters, that is, for a given input sequence of events they specify "what" must be transmitted to the consumer, and "when".

Definition 1 (Input-Output pair). *Let Σ_{in} be a finite alphabet of input events and Σ_{out} be a finite alphabet of output constants. Let $\sigma : e_1, e_2, \dots$ be a sequence of events with $e_i \in \Sigma_{in}$, and $\kappa : o_1, o_2, \dots$ be a sequence of output constants with $o_i \in \Sigma_{out}$. An* input-output pair *is a pair $(\sigma, \langle\kappa, f\rangle)$ with $f : N^+ \mapsto N$ a weakly increasing function that specifies the position of the outputs relative to the input sequence. That is, for each $i > 0$, o_i is produced while or after the event $e_{f(i)}$ is consumed, and strictly before $e_{f(i)+1}$ (if present) is consumed.*

Example 1. The input-output pair $(\sigma, \langle \kappa, f \rangle)$ with

$$\begin{array}{l} \sigma : e_1, e_2, e_3, e_4, e_5 \\ \kappa : o_1, o_2 \end{array} \qquad f(1) = 2, \quad f(2) = 4$$

specifies that output o_1 should be produced between the consumption of events e_2 and e_3, and output o_2 should be produced after consumption of event e_4.

Although it may be interesting to define the semantics of ECL expressions directly in terms of input-output pairs, we have found it more practical to do it operationally in terms of transducers. The main advantage of this approach is that it immediately provides a standard implementation for each correlator expression.

To facilitate a compositional definition of the semantics of ECL expressions we introduce the *correlation machine*, a finite-state transducer extended with facilities for concurrency and reset in a way similar to Petri Nets [18], that supports a concise representation of simultaneous evaluation and preemption. Concurrency is provided by having transitions that map sets of sets of states into sets of states, such that multiple states may have to be active for the transition to be enabled. Reset is provided by explicitly including a *clear set* in the transition, which may be a superset of the enabling condition. A similar way of reset was also proposed in [21].

The addition of concurrency makes the transducer potentially nondeterministic, which is undesirable for an operational model. In the compositional construction of the correlators we have found it convenient to eliminate this nondeterminism by means of a partial order on transitions that specifies a priority ordering among enabled transitions.

Correlation machines may have internal transitions, that is, transitions that do not consume any input events. To enable uniform treatment of all transitions we define expanded input and output sequences that are padded with empty input events and empty output constants in such a way that the constraints on the relative positions of input and output are preserved.

We will use $\overline{\Sigma_{in}}$ for $\Sigma_{in} \cup \{\epsilon\}$, and $\overline{\Sigma_{out}}$ for $\Sigma_{out} \cup \{\epsilon\}$.

Definition 2 (Expanded input-output pair). *The pair (σ', κ') is an expansion of $(\sigma, \langle \kappa, f \rangle)$ if σ' (resp. κ') is equal to σ (resp. κ) interleaved with finite sequences of the empty input (resp. output) event ϵ. Let g, h be the (weakly increasing) functions that map the indices of the elements in σ, κ into the indices of the corresponding elements in σ', κ'. We say that the expansion respects f if the outputs in the expansion are produced "at the right time", that is, if for all $i > 0$*

$$g(f(i)) \leq h(i) < g(f(i) + 1)$$

Example 2. For the sequences σ, κ in example 1 the expansion

$$\begin{array}{llllllllllllll} \sigma' : & e_1 & \epsilon & \epsilon & e_2 & \epsilon & \epsilon & e_3 & e_4 & e_5 & \epsilon & \epsilon \\ \kappa' : & \epsilon & \epsilon & \epsilon & \epsilon & o_1 & \epsilon & \epsilon & o_2 & \epsilon & \epsilon & \epsilon \end{array}$$

respects f, as $g(f(1)) = g(2) = 4$, $g(f(1)+1) = g(3) = 7$, and $h(1) = 5$ for the first output, and $g(f(2)) = g(4) = 8$, $g(f(2)+1) = g(5) = 9$, and $h(2) = 8$ for the second output.

Definition 3 (Correlation Machine). *A* correlation machine $\Psi : \langle Q, I, \mathcal{T}, \prec\rangle$ *has the following components*

- *Q: a finite set of states,*
- *$I \subseteq Q$: the set of initial states,*
- *$\mathcal{T}$: a finite set of transitions $\tau = (En, a, Clr, Tgt, o) \in \mathcal{T}$, with $En \subseteq 2^Q$, the enabling states, $Clr \subseteq Q$, the clear states, $Tgt \subseteq Q$, the target states, and $a \in \overline{\Sigma_{in}}$, the input event, and $o \in \overline{\Sigma_{out}}$, the output character, and*
- *$\prec \subseteq \mathcal{T} \times \mathcal{T}$: a partial order on transitions.*

Definition 4 (Behavior). *An input-output pair $(\sigma, \langle\kappa, f\rangle)$ is a* behavior *of a correlation machine $\Psi : \langle Q, I, \mathcal{T}, \prec\rangle$ if there exists an expansion (σ', κ'),*

$$\begin{array}{l} \sigma' : y_0 \; y_1 \; y_2 \; y_3 \; \ldots \\ \kappa' : w_0 \; w_1 \; w_2 \; w_3 \; \ldots \end{array}$$

that respects f, and if there exists a sequence of sets of states and sets of transitions $S_0, T_0, S_1, T_1, \ldots$ such that

Initiation *(I): $S_0 = I$*
Consecution *: for each $j \geq 0$*

(C0) all transitions in T_j are enabled, that is, for all $\tau = (En_\tau, \ldots) \in T_j$, for all sets $s \in En_\tau$ at least one state q is in the current set of states:

$$\forall s \in En_\tau \quad \exists q \in s \;\; . \;\; q \in S_j$$

(C1) all transitions $\tau \in T_j$ are taken:

$$S_{j+1} = (S_j - \bigcup_{\tau \in T_j} Clr_\tau) \;\cup\; \bigcup_{\tau \in T_j} Tgt_\tau$$

(C2) all transitions $\tau \in T_j$ are minimal in the partial order with respect to all transitions that are enabled on S_j, that is, for all $\tau' \in \mathcal{T}$, $\tau' \prec \tau$

$$\exists s \in En_{\tau'} \quad \forall q \in s \;\; . \;\; q \notin S_j$$

and T_j is maximal, that is, it contains all *transitions that are enabled and minimal with respect to the partial order.*

(C3) for all transitions $\tau = (\ldots, a, \ldots, o) \in T_j$, the input event a agrees with the input event in the input sequence in σ', and the output o agrees with the output constant in κ', that is, $a = y_j$ and $o = w_j$.

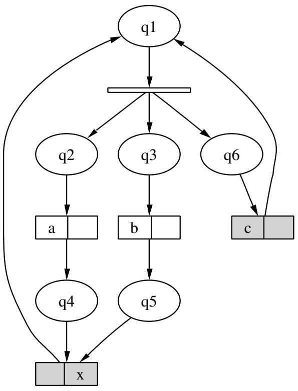


Fig. 1. Correlation automaton

Several transitions can be run in parallel. Condition *C0* says that all of them have to be enabled, while condition *C3* establishes that they have to be consistent with the input and output. Conditions *C1* and *C2* establish that the set of fired transitions is the greatest set of enabled transitions that are minimal with respect to the partial order.

Note, in particular, that if in state S_j no transition is enabled, or no enabling transition is interested in input event y_j then $T_j = \emptyset$ and $S_{j+1} = S_j$.

Example 3. Consider the correlation machine $\mathcal{A} = \langle Q, I, \mathcal{T}, \prec \rangle$ with components

- $Q = \{q_1, q_2, q_3, q_4, q_5, q_6\}$, with $I = \{q_1\}$,
- $\mathcal{T} = \{\tau_1, \tau_2, \tau_3, \tau_4, \tau_5\}$ with

$$\begin{aligned}
\tau_1 &= (\{\{q_1\}\}, \epsilon, \{q_1\}, \{q_2, q_3, q_6\}, \epsilon) \\
\tau_2 &= (\{\{q_2\}\}, a, \{q_2\}, \{q_4\}, \epsilon) \\
\tau_3 &= (\{\{q_3\}\}, b, \{q_3\}, \{q_5\}, \epsilon) \\
\tau_4 &= (\{\{q_4\}, \{q_5\}\}, \epsilon, Q, \{q_1\}, x) \\
\tau_5 &= (\{\{q_6\}\}, c, Q, \{q_1\}, \epsilon)
\end{aligned}$$

- $\prec = \emptyset$.

Remark: In the rest of the paper, when the enabling condition of a transition consists of a single set we write just the set rather than the set of that set, to avoid clutter in notation.

A graphical representation of the machine is shown in figure 1. In the figure transitions are shown by rectangles. The automaton produces an output x after every a and b in any order, without an intervening c. For example, the event sequence *aabbcba* produces the run and output sequence shown in figure 2. Notice the reset effect of transitions τ_4 and τ_5. Both clear all currently active states and start afresh.

σ :				a		a		b						b		c				b		a				
π :	q_1	τ_1	q_2 q_3 q_6	τ_2	q_4 q_3 q_6	$\emptyset$	q_4 q_3 q_6	τ_3	q_4 q_5 q_6	τ_4	q_1	τ_1	q_2 q_3 q_6	τ_3	q_2 q_5 q_6	τ_5	q_1	τ_1	q_2 q_3 q_6	τ_3	q_2 q_5 q_6	τ_2	q_4 q_5 q_6	τ_4	q_1	τ_1	q_2 q_3 q_6
κ :										x														x			

Fig. 2. Run of $\mathcal{A}$ on event sequence *aabbcba*

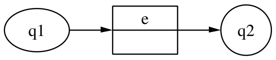


Fig. 3. Correlation module for a single event e

Correlation Module: Correlation machines for correlation expressions are constructed in a bottom-up fashion from the subexpressions. The building block, called a *correlation module* is a correlation machine extended with two final states: s to indicate successful completion and f to indicate failure of a subexpression. Final states only have significance in the modular construction; they become regular states in the final machine. In the following section we introduce the correlation module for some of the constructs defined in section 2.

4 Translation

Single Input Event: The correlation module for a single input event $e \in \Sigma_{in}$ is shown in figure 3. It has three states, $\{q_1, q_2, q_3\}$, of which q_1 is initial, and one transition, $\langle\{q_1\}, e, \{q_1\}, \{q_2\}, \epsilon\rangle\rangle$, that takes the given event as input and moves to the success state without producing any output. The success state is q_2 and the failure state is q_3. Notice that q_3 is not reachable, reflecting the fact that a predicate expression cannot complete in failure.

Compound Expressions: The description of the construction of correlation modules for compound expressions assumes the correlation modules $\mathcal{M}_1, \ldots \mathcal{M}_n$ for the subexpressions have already been constructed. We describe the construction for the accumulation expression in some detail to illustrate the construction method and mostly rely on the figures for the other constructs.

The correlation module $\mathcal{M} = \langle Q, I, \mathcal{T}, \prec, s, f\rangle$ for an accumulation expression **accumulate**$\{\phi_1, \phi_2\}$ with correlation modules $\mathcal{M}_1$ and $\mathcal{M}_2$ for ϕ_1 and ϕ_2 respectively is shown in figure 4(a). It consists of the following components:

- The set of states is the union of the set of states for the subexpressions extended with two new states (not appearing in Q_1 or Q_2) to be the new success and failure states. The set of initial states is set to reflect that both subexpressions should be evaluated in parallel:

$$Q = Q_1 \cup Q_2 \cup \{q_1, q_2\} \quad \text{with} \quad I = I_1 \cup I_2$$

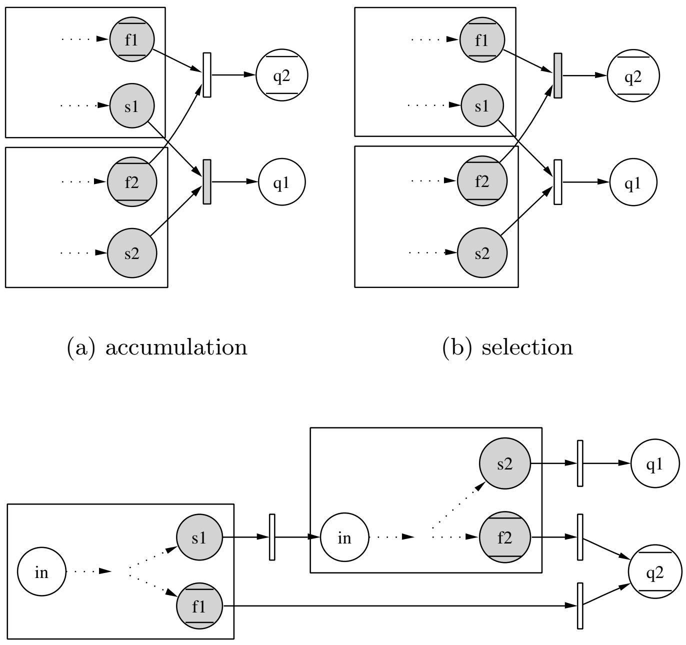


Fig. 4. Correlation modules for accumulation, selection and sequential. The shaded circles denote the success and failure nodes of the modules $\mathcal{M}_{\{1,2\}}$, and the unshaded circles denote the success and failure nodes of the resulting module. Transitions with conjunctive enabling condition are shown shaded and disjunctive transitions are unshaded.

– The set of transitions is the union of the sets of transitions of the subexpressions, extended with two new internal transitions that complete the evaluation of the accumulation expression:

$$\mathcal{T} = \mathcal{T}_1 \cup \mathcal{T}_2 \cup \{\tau_1, \tau_2\}$$

with

$$\tau_1 = (\{\{s_1\}, \{s_2\}\}, \epsilon, Q_1 \cup Q_2, \{q_1\}, \epsilon)$$
$$\tau_2 = (\{f_1, f_2\}, \epsilon, Q_1 \cup Q_2, \{q_2\}, \epsilon)$$

Notice that the enabling condition of τ_1 is conjunctive: both states must be present, while the enabling condition of τ_2 is disjunctive: the transition is enabled if one of the states is present.

– The partial orders of the two subexpressions are combined and the new transitions are given lower priority

$$\prec = \prec_1 \cup \prec_2 \cup (\mathcal{T}_1 \cup \mathcal{T}_2, \{\tau_1, \tau_2, \tau_3\}) \cup \{(\tau_1, \tau_2)\}$$

to reflect that internal transitions of the subexpressions must always be taken before the internal transitions of this module, to make sure the subexpressions have finished their "cleaning up". The pair (τ_1, τ_2) is added to eliminate the potential nondeterminism if these transitions become enabled simultaneously: it gives priority to success.

– The final states are the two newly added states $s = q_1$ and $f = q_2$.

The correlation module for the selection expression **select**$\{\phi_1, \phi_2\}$ is illustrated in figure 4(b). As mentioned before, the selection expression is the dual of the accumulation expression, which is reflected in the dualization of the transitions τ_1 and τ_2:

$$\begin{aligned}\tau_1 &= (\{s_1, s_2\}, \epsilon, Q_1 \cup Q_2, \{q_1\}, \epsilon)\\ \tau_2 &= (\{\{f_1\}, \{f_2\}\}, \epsilon, Q_1 \cup Q_2, \{q_2\}, \epsilon).\end{aligned}$$

The correlation module for sequential composition **sequential**$\{\phi_1, \phi_2\}$ is shown in figure 4(c). It adds two new states, the success state q_1 and the failure state q_2, and four internal transitions

$$\begin{aligned}\tau_1 &= (\{s_1\}, \epsilon, Q_1, I_2, \epsilon)\\ \tau_2 &= (\{f_1\}, \epsilon, Q_1, \{q_2\}, \epsilon)\\ \tau_3 &= (\{s_2\}, \epsilon, Q_2, \{q_1\}, \epsilon)\\ \tau_4 &= (\{f_2\}, \epsilon, Q_2, \{q_2\}, \epsilon)\end{aligned}$$

where τ_1 links the success state of the first module to the initial states of the second module.

The correlation module for an unless expression, **do**$\{\phi_1\}$ **unless**$\{\phi_2\}$ is shown in figure 5(a). The two new transitions have transition relation

$$\begin{aligned}\tau_1 &= (\{s_1\}, \epsilon, Q_1 \cup Q_2, \{q_1\}, \epsilon)\\ \tau_2 &= (\{f_1, s_2\}, \epsilon, Q_1 \cup Q_2, \{q_2\}, \epsilon)\end{aligned}$$

where transition τ_1 is the success transition, while τ_2 leads to failure. As mentioned in section 2, the unless expression can cause an expression to fail, as witnessed by τ_2, which leads from s_2 (a success state) to q_2 (a failure state).

Output Expressions: The correlation module for the push expression **push** (x) $\{\phi\}$, shown in figure 5(b), adds two transitions

$$\begin{aligned}\tau_1 &= (\{s_1\}, \epsilon, Q_1, \{q_1\}, x)\\ \tau_2 &= (\{f_1\}, \epsilon, Q_1, \{q_2\}, \epsilon)\end{aligned}$$

the first of which outputs constant x upon successful completion of the module for ϕ, while τ_2 simply propagates the failure.

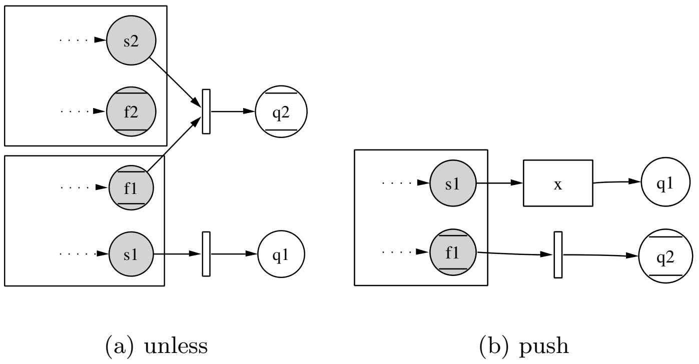


Fig. 5. Correlation modules for unless and push.

Mode Expressions: The correlation module $\mathcal{M}$ for a mode expression with correlation modules $\mathcal{M}_i$ for ϕ_i,$i = 1, \ldots n$, and $\mathcal{M}_{gi}$ for p_i, $i = 1 \ldots n$ consists of the following components:

- The set of states, Q, is the union of the states of the guards and the expressions and two additional states for the new success and failure state. The initial states are those of the first expression, ϕ_1, combined with the initial states of the guards of the other expressions.
- The set of transitions includes the transitions of the expressions and the guards, and is extended with one *mode entry transition* τ_i for each mode ϕ_i, $i = 1 \ldots n$ with transition relation $\tau_i = (\{s_{gi}\}, \epsilon, Q, I_i \cup G_i, \epsilon)$ where G_i is the union of the set of initial states of the guards other than p_i. Both the success and failure state of this module are unreachable.
- The partial orders of all modules are combined, and the transitions belonging to the guard expressions are given higher priority than those belonging to the expressions, to reflect that the preemption of a mode by another mode has higher priority than the consumption of the same event within a mode.

Example 4. Figure 7 shows an example of a small avionics scenario inspired by a real-life system. It consists of seven components which are triggered either periodically by time-out, manually by the pilot, or by notification of a correlation expression. The purpose of the system is to control the data displayed on the navigation display. The displayed data can be navigational steering data or tactical steering data, as selected by the pilot. Only components that contribute to the selected data should be activated. Figures 8 and 9 show the result of the bottom-up construction of the correlation module, and the pruned correlation machine, respectively, for the navigation display component. These were generated automatically from the correlation expression.

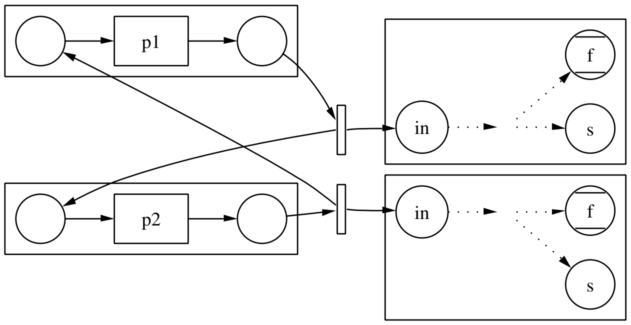


Fig. 6. Correlation module for mode expression

component	trigger	output
global positioning	time-out	gps
navigator	time-out	nav
pilot control	manual	tm,nm
airframe	**repeat**{**notify**{gps}}	af

tactical steering	**modes**{
	in{tm}**do**{**repeat**{**notify**{af}}}
	in{nm}**do**{false}}
		tacst

navigational steering	**modes**{
	in{tm}**do**{false}
	in{nm}**do**{**repeat**{
	notify{**select**{nav,af}}}}}
		navst

navigation display	**modes**{
	in{tm}**do**{**repeat**{
	notify{ **accumulate**{af,tacst}}}}
	in{nm}**do**{**repeat**{
	notify{ **accumulate**{af,navst}}}}}	

Fig. 7. Scenario for controlling a navigation display

5 Related Work

Several proposals for specification languages for event correlation appear in the literature. In [4], a language based on typed λ-calculus is used, where composite events are represented by *curried functional expressions* with a formal semantics defined in terms of reduction rules.

Zhu and Sethi [22] propose a language that includes a negation operator. Expressions are evaluated relative to some fixed or sliding time window; the negation operator precludes the occurrence of its argument during that window.

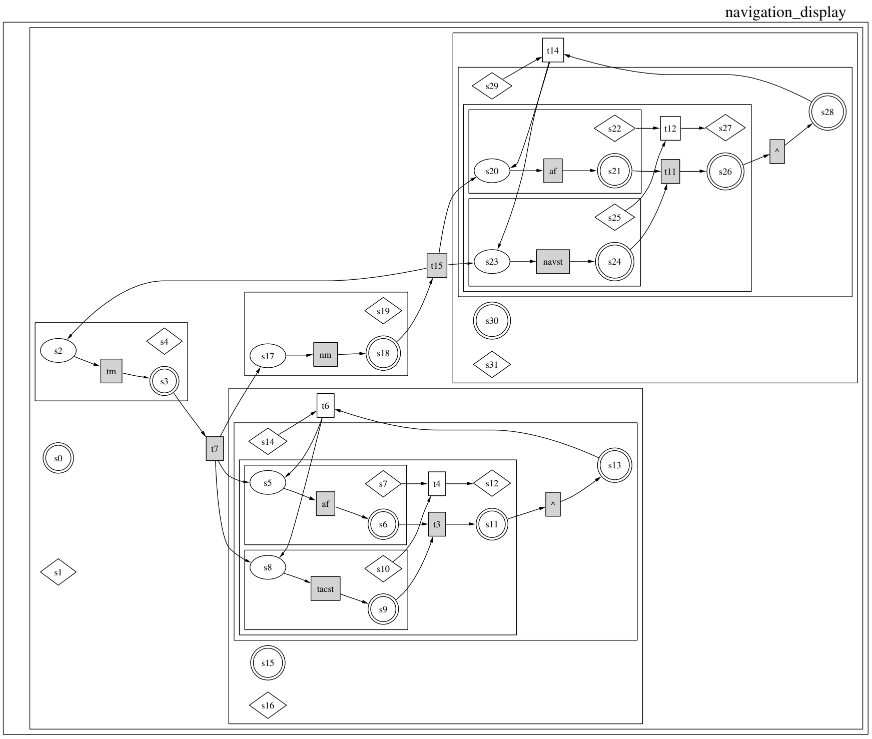


Fig. 8. Correlation module for the navigation display

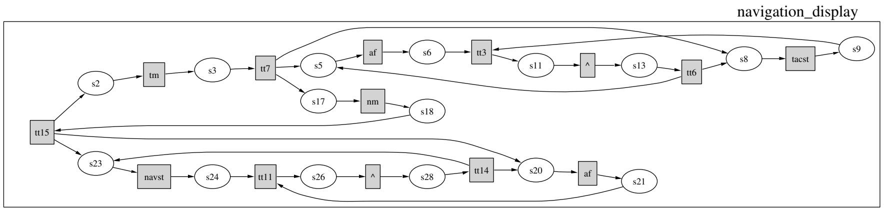


Fig. 9. Correlation machine for the navigation display

COBEA [13] is an event-based architecture that includes an evaluation engine for composite events specified in the Cambridge Composite Event Language [9, 10]. The semantics is defined in terms of push-down machines.

The use of composite temporal events has received much attention in the active database community. Gehani et al. [7] propose a language for specifying composite events with semantics in terms of event history maps, which is expressively equivalent to regular expressions and can be translated into NFA. Coordination of subexpressions is done through correlation variables to allow parameterization. The method is implemented in COMPOSE [6]. In [16,17] Motakis and Zaniolo propose a specification language based on Datalog. Their pattern language allows parameterization, with parameter instantiations propagated through the expression.

6 Conclusion and Future Directions for Research

We have presented a declarative language to express event correlation expressions for publish-subscribe systems. The semantics of this language was defined in terms of correlation machines, an operational model that can be directly used as an event processing engine in an event channel.

Applications and Implementation: The language presented here has been applied in Boeing's Open Experimental Platform (OEP). It was found that the use of event correlation expressions reduced the need for special-purpose components by moving functionality to the event channel. A prototype event processor based on correlation machines has been implemented in C++ and is being integrated in the OEP. A separate event processor has been implemented in Java and has been integrated with FACET [12], a real-time event channel developed at Washington University at St Louis.

Analysis: Publish-subscribe systems are used in mission-critical avionic applications [19] and may potentially be used in other safety-critical systems. The availability of analysis tools for event correlation expressions will contribute to the acceptance of this technology as a reliable addition to simple event filtering. Since these machines are essentially finite state, we believe that many of the analysis problems are tractable. Some of these include:

– checking expressions for *triviality*, (i.e. whether the expression filters in everything), or *vacuity* (where the expression rejects everything),
– checking *liveness*, that is, at any point it is possible for some event-sequence to lead the machine to acceptance,
– checking containment among correlators,
– checking correlation expressions against event-loops for equivalence,
– checking event dependencies.

Optimization: The correlation machines that are generated by the construction method described are obviously rather inefficient; they contain unreachable states and redundant internal transitions. Other transformations may be envisaged for time/space trade-offs. Our current model favors a concise representation. However this comes at the price of increased event processing time. In time-critical applications, for example, one may want to eliminate the concurrency and fully determinize the transducer.

Evaluation Strategies: A large system may have many subscriptions. Hence, the middleware is faced with the task of evaluating each of these expressions for each incoming event. This can cause a severe overhead and lead to performance degradation. There are two complementary approaches to alleviate this problem.

The first tactic is to *compose* correlation machines. Consider a number of consumers with different subscriptions. A naive implementation would run all the correlators in parallel. A more efficient version should try to *reuse* the work of the evaluation of different machines. In [1] a first approach to this problem has been considered. This problem resembles that of performing multiple parallel searches in a string [8].

The second tactic is to *decompose* a machine into many machines and distribute these along the network. Thus, if while routing an event, a determination can be made that it is not of interest, it can be discarded. This can save network bandwidth and yield more processing time. This has been called the *quenching problem* in [11].

References

1. M. Aguilera, R. Storm, D. Sturman, M. Astley, and T. Chandra. Matching events in a content based subscription system. In *PODC*, 1999.
2. B.Segall and S. Arnold. Elvin has left the building: A publish/subscribe notification service with quenching. In *Queensland AUUG Summer Technical Conference, Brisbane, Australia*, 1997.
3. A. Carzaniga, D. S. Rosenblum, and A. L Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, August 2001.
4. S. Courtenage. Specifying and detecting composite events in content-based publish/subscribe systems. In *Proc. DEBS'02*, 2002.
5. J. Crowcroft, J. Bacon, P. Pietzuch, G. Coulouris, and H. Naguib. Channel islands in a reflective ocean: Large-scale event distribution in heterogeneous networks. *IEEE Communications Magazine*, (9):112–115, September 2002.
6. N.H. Gehani, H.V. Jagadish, and Oded Shmueli. COMPOSE. a system for composite event specification and detection. Technical report, AT&T Bell Laboratories, 1992.
7. N.H. Gehani, H.V. Jagadish, and Oded Shmueli. Composite event specification in active databases: Model and implementation. In *Proceeding VLDB'92*, pages 327–338, 1992.
8. D. Gusfield. *Algorithms on strings, trees and sequences.* Cambridge Univ. Press, 1997.
9. R. Hayton, J. Bacon, J. Bates, and K. Moody. Using events to build large scale distributed applications. In *Proc. ACM SIGOPS European Workshop*, pages 9–16. ACM, September 1996.
10. Richard Hayton. *OASIS. An Open Architecture for Secure Internetworking Services.* PhD thesis, Fitzwilliam College, University of Cambridge, 1996.
11. Y. Huang and H. Garcia-Molina. Publish/subscribe in a mobile environment. In *Proc.* MobiDE, 2001.
12. Frank Hunleth, Ron Cytron, and Christopher Gill. Building customizable middleware using aspect oriented programming. In *Workshop on Advanced Separation of Concerns (OOPSLA'01)*, 2001.
13. Chaoying Ma and Jean Bacon. COBEA: A CORBA-based event architecture. In *Proc. USENIX COOTS'98*, pages 117–131, April 1998.
14. M. Mansouri-Samani and M. Sloman. GEM, a generalised event monitoring language for distributed systems. In *Proceedings of ICODP/ICDP'97*, 1995.

15. R. Meier. State of the art review of distributed event models. Technical report, University of Dublin, Trinity College, 2000.
16. I. Motakis and C. Zaniolo. Composite temporal events in active database rules: A logic-oriented approach. In *Proceedings of DOOD'95*, volume 1013 of *LNCS*, pages 19–37. Springer-Verlag, 1995.
17. I. Motakis and C. Zaniolo. Formal semantics for composite temporal events in active database rules. *Journal of Systems Integration*, 7(3–4):291–325, 1997.
18. W. Reisig. *Petri Nets: An Introduction*, volume 4 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Berlin, 1985.
19. D. Schmidt, D. Levine, and T. Harrison. The design and performance of a real-time CORBA object event service. In *Proceedings of OOPSLA '97*, 1997.
20. David Sharp. Reducing avionics software cost through component based product line development. In *Proceedings of the Software Technology Conference*, 1998.
21. T. Vesper and M. Weber. Structuring with distributed algorithms. In *Proceeding of CS&P98*, September 1998.
22. D. Zhu and A. S. Sethi. SEL, a new event pattern specification language for event correlation. In *Proc. ICCCN-2001,*, pages 586–589, October 2001.

Generating Heap-Bounded Programs in a Functional Setting*

Walid Taha[1], Stephan Ellner[1], and Hongwei Xi[2]

[1] Rice University, Houston, TX, USA
{taha,besan}@cs.rice.edu
[2] Boston University, Boston, MA, USA
hwxi@cs.bu.edu

Abstract. High-level programming languages offer significant expressivity but provide little or no guarantees about resource utilization. Resource-bounded languages provide strong guarantees about the runtime behavior of programs but often lack mechanisms that allow programmers to write more structured, modular, and reusable programs. To overcome this basic tension in language design, this paper advocates taking into account the natural distinction between the development platform and the deployment platform for resource-sensitive software.
To illustrate this approach, we develop the meta-theory for GeHB, a two-level language in which first stage computations can involve arbitrary resource consumption, but the second stage can only involve functional programs that do not require new heap allocations. As an example of a such a second-stage language we use the recently proposed first-order functional language LFPL. LFPL can be compiled directly to malloc-free, imperative C code. We show that all generated programs in GeHB can be transformed into well-typed LFPL programs, thus ensuring that the results established for LFPL are directly applicable to GeHB.

1 Introduction

Designers of embedded and real-time software attend not only to functional specifications, but also to a wider range of concerns, including resource consumption and integration with the physical world. Because of the need for strong a-priori guarantees about the runtime behavior of embedded software, resource-bounded languages (c.f. [19,8,9,28,29]) generally trade expressivity for guarantees about runtime behavior. Depending on the kind of guarantees required, a resource-bounded language may have to deprive the programmer from useful abstraction mechanisms such as higher-order functions, dynamic data structures, and general recursion. We argue that this trade-off can be avoided if the language itself can express the distinction between computation on the development platform and computation on the deployment platform. Such a language could provide a bridge between traditional software engineering techniques on one side, and the specific demands of the embedded software domain on the other.

* Supported by NSF grants ITR-0113569, CCR-0224244 and CCR-0229480.

LFPL: Bounded Space and Functional In-Place Update. Recently, Hofmann proposed the resource-bounded programming language LFPL [8,10], a first-order functional language with constructors and destructors for dynamic data structures. The type system of LFPL ensures that all well-typed programs can be compiled into malloc-free, imperative C code. [8]. Without any special optimizations, the performance of resulting programs is competitive with the performance of programs generated by the OCaml native code compiler.

The essential idea behind LFPL is the use of a linear type system that ensures that references to dynamically allocated structures are not duplicated at runtime. The key mechanism to achieving this is ensuring that certain variables in the source program are used at most once (linearity). Constructors for dynamic structures carry an extra field that can be informally interpreted as a capability. The following is an implementation of insertion sort over lists in LFPL [10]: [1]

```
let rec insert(d,a,l) =
    case l of
      nil -> cons(a, nil) at d
    | cons(b,t) at d’ -> if a < b
                             then cons(a, cons(b, t) at d’) at d
                             else cons(b, insert(d’,a, t)) at d
let rec sort(l) =
  case l of
    nil -> nil
  | cons(a,t) at d -> insert(d, a, sort(t))
```

The main function, `sort`, takes a list `l` and returns a sorted list. Using pattern matching (the `case`-statement), the function checks if the list is empty. If so, the empty list is returned. If the list is non-empty, the constructor `cons` for the list carries three values: the head `a`, the tail `t`, and a capability `d` for the location at which this list node is stored. Note that the capability `d` is passed to `insert`, along with the head of the list and the result of recursively calling `sort` on the tail. The type system ensures that well-typed programs do not duplicate such capabilities. This is achevied by ensuring that a variable like `d` is not used more than once.

It has been shown that any LFPL program can be compiled to a C program that requires no heap allocation, and therefore no dynamic heap management [8,10]. Instead of allocating new space on the heap, constructor applications are implemented by modifying heap locations of previously allocated data structures.

Expressivity. The example above points out a common limitation of resource-bounded languages. In particular, we often want to parameterize sorting functions by a comparison operation that allows us to reuse the same function to sort different kinds of data (c.f. [14]). This is especially problematic with larger and more sophisticated sorting algorithms, as it would require duplicating the source code for such functions for every new data structure. In practice, this can also cause code duplication, thus increasing maintenance cost.

[1] We use an OCaml-like concrete syntax for LFPL.

Such parameterization requires higher-order functions, which are usually absent from resource-bounded languages: implementing higher-order functions requires runtime allocation of closures on the heap. Furthermore, higher-order functions are expressive enough to encode dynamically allocated data structures such as lists.

Contributions. A key observation behind our work is that even when targeting resource-bounded platforms, earlier computations that lead to the *construction* of such programs are often not run on resource-bounded development platforms. Commercial products such as National Instrument's LabVIEW Real-Time [20] and LabVIEW FPGA [21] already take advantage of such stage distinctions. The question this paper addresses is how to reflect this reality into the design of a language that gives feedback to the programmer as soon as possible as to whether the "final product" is resource-bounded or not.

We demonstrate that ideas from two-level and multi-stage languages can be used to develop natural, two-stage extensions of a resource-bounded language. The focus of this paper is on developing the meta-theory for GeHB, a statically-typed two-stage language that extends LFPL. The resulting language is more expressive and takes advantage of the realistic assumption of having two distinct stages of computation. In the first stage, general higher-order functions can be used. In the second (or "LFPL") stage, no dynamic heap allocation is allowed. Type-checking a GeHB program (which happens before the first stage) statically guarantees that all second-stage computations are heap-bounded. This strong notion of static typing requires that we type check not only first-stage computations, but also templates of second-stage programs. Compared to traditional statically-typed multi-stage languages, the technical challenge lies in ensuring that generated programs satisfy the linearity conditions that Hofmann defines for LFPL. While a direct combination of a multi-stage language and LFPL does not work, we show how this problem can be addressed using the recently proposed notion of closed types [4,7]. Finally, we show that all generated programs can be transformed into well-typed LFPL programs, thus ensuring that the results established for LFPL are directly applicable to GeHB.

The results presented here are a step toward our long-term goal of developing an expressive statically-typed functional language that can be used in the challenging context of embedded systems [27].

Organization of This Paper. Section 2 introduces the basic concepts of general-purpose multi-stage programming, and points out a technical problem in the direct combination of a multi-stage type system with the type system of LFPL. Section 3 formalizes the type system and semantics of GeHB, and presents a type preservation result for first-stage evaluation. The section also shows that second-stage values determine LFPL programs and that results about LFPL apply to GeHB. Section 4 illustrates programming in GeHB and the impact of typing rules from the programmer's perspective. Sections 5 and 6 discuss various aspects of the work and conclude.

2 Multi-stage Programming

Multi-stage languages [22,13,24] provide light-weight, high-level *annotations* that allow the programmer to break down computations into distinct *stages*. This facility supports a natural and algorithmic approach to program generation, where generation occurs in a first stage, and the synthesized program is executed during a second stage. The annotations are a small set of constructs for the construction, combination, and execution of delayed computations. Underlying program generation problems, such as avoiding accidental variable capture and the representation of programs, are completely hidden from the programmer (c.f. [24]). The following is a simple program written in the multi-stage language MetaOCaml [17]:

```
let rec power n x = if n=0 then <1> else <~x * ~(power (n-1) x)>
let power3 = <fun x -> ~(power 3 <x>)>
```

Ignoring the staging annotations (brackets `<`*e*`>` and escapes `~`*e*) [2], the above code is a standard definition of a function that computes x^n, which is then used to define the specialized function x^3. Without staging, the last step just produces a closure that invokes the power function every time it gets a value for x. The effect of staging is best understood by starting at the end of the example. Whereas a term `fun x ->` *e* `x` is a value, an annotated term `<fun x -> ~(`*e* `<x>)>` is not. Brackets indicate that we are constructing a future stage computation, and an escape indicates that we must perform an immediate computation *while* building the bracketed computation. The application *e* `<x>` has to be performed even though `x` is still an uninstantiated symbol. In the `power` example, `power 3 <x>` is performed immediately, once and for all, and not repeated every time we have a new value for `x`. In the body of the definition of the function `power`, the recursive application of `power` is escaped to ensure its immediate execution in the first stage. Evaluating the definition of `power3` produces `<fun x -> x*x*x*1>`.

General-purpose multi-stage languages provide strong safety guarantees. For example, a program generator written in such a language is not only type-safe in the traditional sense, but the type system also guarantees that *any generated program will be type safe*. Traditionally, multi-stage languages were used for quantitative benefits (speed-up). In this work, the benefit is more qualitative, in that it allows programmers to write programs that previously could not be expressed in a resource-bounded language.

A Naive Approach to Generating LFPL Programs. Combining the type systems for multi-stage languages and LFPL is not trivial. In particular, it cannot be achieved by simply treating level 0 variables as non-linear variables (that can be used multiple times). The level of a term *e* is 0 if it is executed in the first stage, or 1 if it is executed in the second. A term is at level 0 by default, unless it occurs (unescaped) inside brackets. For example, consider the expression `<fun x -> ~(power 3 <x>)>`. The use of `power` is surrounded by

[2] In the implementation of MetaOCaml, dots are used around brackets and before escapes to disambiguate the syntax.

one set of brackets and one escape, therefore `power` is used at level 0. The use of `x` is surrounded by two sets of brackets and one escape, and therefore occurs at level 1. Similarly, the binding of `x` occurs at level 1. Treating level 0 variables as non-linear and level 1 variables as linear is problematic for two reasons:

1. *Some level 0 variables must be treated linearly:* Variables bound at level 0 can refer to computations that contain free level 0 variables. A simple example is the following:

   ```
   <fun x -> ~((fun y -> <(~y,~y)>) <x>)>
   ```

   In this term, `x` is bound at level 1, while `y` is bound at level 0. Note that `x` is used only once in the body of the term, and is therefore linear. However, if we evaluate the term (performing the first stage computation) we get the term

   ```
   <fun x -> (x,x)>
   ```

   where `x` is no longer used linearly. This occurs because `y` was not used linearly in the original term. Thus, one must also pay attention to linearity even for level 0 variables such as `y`.
2. *Not all level 1 variables need to be linear:* Hofmann's type system is parameterized by a signature of function names. Such functions can be used many times in an LFPL program. A natural way to integrate such function names into our formulation is to treat them as non-linear variables. [3]

Ignoring the first point leads to an unsound static type system: it fails to ensure that generated programs require no heap allocation in the second stage. Ignoring the second leads to an overly restrictive type system.

3 GeHB: Generating Heap Bounded Programs

The syntax of the subset of GeHB that we study is defined as follows:

$$\begin{aligned} e ::= {} & i \mid x \mid \lambda x.e \mid e(e) \mid \mathsf{let}\ x = e\ \mathsf{in}\ e \mid (e,e) \mid \pi_1\ e \mid \pi_2\ e \mid \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \mid \\ & \mathsf{let\ rec}\ f(x) = e\ \mathsf{in}\ e \mid [e] \mid \mathsf{let}\ [x] = e\ \mathsf{in}\ e \mid \langle e \rangle \mid \tilde{}\,e \mid \mathsf{nil} \mid \mathsf{cons}(e,e)\mathsf{at}\ e \mid \\ & \mathsf{case}\ e\ \mathsf{of\ nil} \to e\ '|'\ \mathsf{cons}(x,y)\mathsf{at}\ z \to e \end{aligned}$$

The first line introduces integers, variables, lambda-abstractions, applications, local variable bindings, pairs, and conditionals. The second line introduces recursive definitions, as well as the closedness annotations $[e]$ and $\mathsf{let}\ [x] = e_0\ \mathsf{in}\ e_1$. (We read $[e]$ as "closed of e".)

Intuitively, closedness annotations are assertions reflected in the type and checked by the type system. The first annotation instructs the type checker

[3] While Hofmann's type system for LFPL does not include an explicit typing rule for function definitions, it is essential that functions not introduce new resources when they are applied. A natural way to achieve this, as we show here, is for the body of a function definition to be typed in an environment where only non-linear variables are visible.

$$\frac{}{\Gamma;\Delta \overset{n}{\vdash} i : \mathsf{int}}\,(\mathrm{INT}) \qquad \frac{\Gamma(x) = (t,0)}{\Gamma;\Delta \overset{0}{\vdash} x : t}\,(\mathrm{VARN})$$

$$\frac{\Delta(x) = (t,n)}{\Gamma;\Delta \overset{n}{\vdash} x : t}\,(\mathrm{VARL}) \qquad \frac{\overset{0}{\vdash} t_0 \quad \Gamma;\Delta, x : (t_0,0) \overset{0}{\vdash} e : t_1}{\Gamma;\Delta \overset{0}{\vdash} \lambda x.e : t_0 \to t_1}\,(\mathrm{LAM})$$

$$\frac{\Gamma;\Delta_0 \overset{0}{\vdash} e_0 : t_1 \to t_2 \quad \Gamma;\Delta_1 \overset{0}{\vdash} e_1 : t_1}{\Gamma;\Delta_0,\Delta_1 \overset{0}{\vdash} e_0(e_1) : t_2}\,(\mathrm{APP}) \qquad \frac{\Gamma(f) = (t_0 \to t_1, 1) \quad \Gamma;\Delta \overset{1}{\vdash} e : t_0}{\Gamma;\Delta \overset{1}{\vdash} f(e) : t_1}\,(\mathrm{APPF})$$

$$\frac{\Gamma;\Delta_0 \overset{n}{\vdash} e_0 : t_0 \quad \Gamma;\Delta_1, x : (t_0,n) \overset{n}{\vdash} e_1 : t_1}{\Gamma;\Delta_0,\Delta_1 \overset{n}{\vdash} \mathsf{let}\ x = e_0\ \mathsf{in}\ e_1 : t_1}\,(\mathrm{LET}) \qquad \frac{\Gamma;\Delta_0 \overset{n}{\vdash} e_0 : \mathsf{int} \quad \Gamma;\Delta_1 \overset{n}{\vdash} e_1 : t \quad \Gamma;\Delta_1 \overset{n}{\vdash} e_2 : t}{\Gamma;\Delta_0,\Delta_1 \overset{n}{\vdash} \mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : t}\,(\mathrm{IF})$$

$$\frac{\overset{n}{\vdash} t_0, t_1 \quad \Gamma, f : (t_0 \to t_1, n); x : (t_0, n) \overset{n}{\vdash} e_0 : t_1 \quad \Gamma, f : (t_0 \to t_1, n); \Delta \overset{n}{\vdash} e_1 : t_2}{\Gamma;\Delta \overset{n}{\vdash} \mathsf{let\ rec}\ f(x) = e_0\ \mathsf{in}\ e_1 : t_2}\,(\mathrm{REC})$$

$$\frac{\Gamma;\emptyset \overset{0}{\vdash} e : t}{\Gamma;\Delta \overset{0}{\vdash} [e] : [t]}\,(\mathrm{CLOS}) \qquad \frac{\Gamma;\Delta_0 \overset{0}{\vdash} e_0 : [t_0] \quad \Gamma, x : (t_0,0);\Delta_1 \overset{0}{\vdash} e_1 : t_1}{\Gamma;\Delta_0,\Delta_1 \overset{0}{\vdash} \mathsf{let}\ [x] = e_0\ \mathsf{in}\ e_1 : t_1}\,(\mathrm{LETC})$$

$$\frac{\Gamma;\Delta \overset{n}{\vdash} e : t_1 * t_2}{\Gamma;\Delta \overset{n}{\vdash} \pi_i\ e : t_i}\,(\mathrm{PI}) \qquad \frac{\Gamma;\Delta_0 \overset{n}{\vdash} e_0 : t_0 \quad \Gamma;\Delta_1 \overset{n}{\vdash} e_1 : t_1}{\Gamma;\Delta_0,\Delta_1 \overset{n}{\vdash} (e_0, e_1) : t_0 * t_1}\,(\mathrm{PAIR}) \qquad \frac{\Gamma;\Delta \overset{1}{\vdash} e : t}{\Gamma;\Delta \overset{0}{\vdash} \langle e \rangle : \langle t \rangle}\,(\mathrm{BRAC})$$

$$\frac{\Gamma;\Delta \overset{0}{\vdash} e : \langle t \rangle}{\Gamma;\Delta \overset{1}{\vdash} \tilde{}e : t}\,(\mathrm{ESC}) \qquad \frac{\Gamma;\Delta, x : (t_0,1), y : (t_0,1) \overset{1}{\vdash} e : t_1 \quad t_0 \in \{u ::= \mathsf{int} \mid u * u\}}{\Gamma;\Delta, x : (t_0,1) \overset{1}{\vdash} e[y := x] : t_1}\,(\mathrm{CONTR})$$

$$\frac{\overset{1}{\vdash} t}{\Gamma;\Delta \overset{1}{\vdash} \mathsf{nil} : \mathsf{list}(t)}\,(\mathrm{NIL}) \qquad \frac{\Gamma;\Delta_0 \overset{1}{\vdash} e_0 : t \quad \Gamma;\Delta_1 \overset{1}{\vdash} e_1 : \mathsf{list}(t) \quad \Gamma;\Delta_2 \overset{1}{\vdash} e_2 : \diamond}{\Gamma;\Delta_0,\Delta_1,\Delta_2 \overset{1}{\vdash} \mathsf{cons}(e_0, e_1)\mathsf{at}\ e_2 : \mathsf{list}(t)}\,(\mathrm{CONS})$$

$$\frac{\Gamma;\Delta_0 \overset{1}{\vdash} e_0 : \mathsf{list}(t_0) \quad \Gamma;\Delta_1 \overset{1}{\vdash} e_1 : t_1 \quad \Gamma;\Delta_1, x : (t_0,1), y : (\mathsf{list}(t_0),1), l : (\diamond,1) \overset{1}{\vdash} e_2 : t_1}{\Gamma;\Delta_0,\Delta_1 \overset{1}{\vdash} \mathsf{case}\ e_0\ \mathsf{of\ nil} \to e_1 \mid \mathsf{cons}(x,y)\mathsf{at}\ l \to e_2 : t_1}\,(\mathrm{CASE})$$

Fig. 1. Type System for GeHB

to ensure that the value resulting from evaluating e can be safely duplicated, without leading to any runtime heap allocation. If e can be type-checked as such, then the term $[e]$ is assigned a special type. By default, variables will be considered linear. The second annotation allows the programmer to promote the value resulting from evaluating e_0 to a special type environment for non-linear variables, if the type type of e_0 indicates that it can be safely duplicated. The construct then binds this value to x, and makes x available for an arbitrary number of uses in e_1. The syntax for these constructs was inspired by the syntax for S4 modal logic used by Davies and Pfenning [7].

Next in the definition of the syntax are brackets and escapes, which have been described in the previous section. The remaining constructs are constructors and pattern matching over lists. We avoid artificial distinctions between the syntax of terms that will be executed at level 0 and terms executed at level 1. As much as possible, we delegate this distinction to the type system. This simplifies the meta-theory, and allows us to focus on typing issues.

Type System. The syntax of types for GeHB is defined as follows:

$$t ::= \mathsf{int} \mid t \to t \mid t * t \mid \diamond \mid [t] \mid \langle t \rangle \mid \mathsf{list}(t)$$

The first three types are standard. The type $\diamond$ (read "diamond") can be intuitively interpreted as the type of the capability for each heap-allocated value. The type $[t]$ (read "closed of t") indicates that a value of that type can be safely duplicated. The semantics for this type constructor was inspired by both the linear type system of LFPL and the closed type constructor used for type-safe reflective and imperative multi-stage programming [24,18,4]. The type $\langle \mathrm{t} \rangle$ (read "code of t") is the type of second-stage computations and is associated with all generated LFPL programs. The type $\mathsf{list}(t)$ is an example of a dynamic data structure available in the second stage.

Different types are valid at each level:

$$\frac{}{\overset{n}{\vdash} \mathsf{int}} \quad \frac{\overset{0}{\vdash} t_0 \quad \overset{0}{\vdash} t_1}{\overset{0}{\vdash} t_0 \to t_1} \quad \frac{\overset{n}{\vdash} t_0 \quad \overset{n}{\vdash} t_1}{\overset{n}{\vdash} t_0 * t_1} \quad \frac{}{\overset{1}{\vdash} \diamond} \quad \frac{\overset{0}{\vdash} t}{\overset{0}{\vdash} [t]} \quad \frac{\overset{1}{\vdash} t}{\overset{0}{\vdash} \langle t \rangle} \quad \frac{\overset{1}{\vdash} t}{\overset{1}{\vdash} \mathsf{list}(t)}$$

We will write write t^n for a type t when $\overset{n}{\vdash} t$ is derivable.

Two kinds of typing environments will be used. The first holds values that can be safely duplicated, whereas the second holds linear values. The environments will have the following form:

$$\Gamma ::= \emptyset \mid \Gamma, x : (t^0, 0) \mid \Gamma, x : (t^1 \to t^1, 1) \quad \text{and} \quad \Delta ::= \emptyset \mid \Delta, x : (t^n, n)$$

Thus, all environments carry mappings from variable names to *pairs* of types and binding levels. In Γ, for any given variable binding, if the binding level is 0 then the type can be any type valid at level 0. If the binding level is 1, however, only functions bound at level 1 are allowed. In particular, the only use of Γ at level 1 is to express that functions defined at level 1 (using let rec) can be safely used multiple times in the body of a program. For Δ, the situation is straightforward. A valid combined environment $\Gamma; \Delta$ is a pair Γ and Δ such that any variable x either occurs exactly once in one of them or occurs in neither.

We will write Γ^n (and similarly Δ^n) for a Γ where all bindings are at level n.

The judgment $\Gamma; \Delta \overset{n}{\vdash} e : t$ presented in Figure 1 defines the type system for GeHB. The environment Γ is standard whereas the environment Δ is linear. This distinction manifests itself in the type system as a difference in the way these environments are propagated to subterms. As a simple example, consider the rule for application $e_0(e_1)$. The environment Γ is used in type checking both terms. In contrast, the linear environment Δ is split into two parts by pattern matching it with Δ_0 and Δ_1, and exactly one of these two parts is used in type checking each of the subterms e_0 and e_1.

Proposition 1. *1. If $\Gamma; \Delta \overset{n}{\vdash} e : t$ is derivable, then so is $\overset{n}{\vdash} t$.*

2. If $\Gamma; \Delta, x : (t^0, 0) \overset{m}{\vdash} e : t'$ is derivable, then x occurs at most once in e.

The first eight rules are essentially standard. Integers are available at both levels. Two rules are needed for variables, one for the non-linear environment (VARN) and one for the linear environment (VARL). Note that the variable rules require that the binding level be the same as the level at which the variable is used. Additionally, the (VARN) rule only applies at level 0. We can only lookup a level 1 variable bound in Γ if it is used in an application at level 1 (APPF). Lambda abstractions are mostly standard, but they are only allowed at level 0.

The rule for function definitions (REC) makes the newly defined function available as a non-linear variable. To make sure that this is sound, however, we have to require that the body e_0 of this definition be typable using no linear variables except for the function's parameter.

The rule for close (CLOS) uses a similar condition: a closedness annotation around a term e is well typed only if e can be typed without reference to any linear variables introduced by the context. The rule for let-close (LETC) is the elimination rule for the closed type constructor $[t]$. The rule allows us to take any value of closed type and add it to the non-linear environment Γ.

The next two rules for projection and pairing are essentially standard. The rules for brackets and escapes are standard in multi-stage languages. The remaining rules define conditions for terms that are valid only at level 1, and come directly from LFPL.

Lemma 1 (Weakening). *If* $\Gamma \overset{n}{\vdash} e : t$ *then*

1. $\Gamma, x : (t', n'); \Delta \overset{n}{\vdash} e : t$, *and*
2. $\Gamma; \Delta, x : (t', n') \overset{n}{\vdash} e : t$

Lemma 2 (Substitution).

1. *If* $\Gamma; \Delta_0 \overset{0}{\vdash} e_1 : t_1$ *and* $\Gamma; \Delta_1, x : (t_1, 0) \overset{n}{\vdash} e_2 : t_2$ *then* $\Gamma; \Delta_0, \Delta_1 \overset{n}{\vdash} e_2[x := e_1] : t_2$.
2. *If* $\Gamma; \emptyset \overset{0}{\vdash} e_1 : t_1$ *and* $\Gamma, x : (t_1, 0); \Delta_1 \overset{n}{\vdash} e_2 : t_2$ *then* $\Gamma; \Delta_1 \overset{n}{\vdash} e_2[x := e_1] : t_2$.

Operational Semantics for First Stage. The judgment $e_0 \overset{n}{\hookrightarrow} e_1$ presented in Figure 2 defines the operational semantics of GeHB programs. Even though the evaluation function used in this paper uses an index n, it is *not* defining how evaluation occurs during both stages. Rather, it defines how the mix of both level 0 and level 1 terms are evaluated *during the first stage*. When the index n is 0 we say we are evaluating e_0, and when the index is 1 we say we are rebuilding this term. Note that this judgment defines only what gets done during the first stage of execution, namely, the generation stage. Execution in the second stage is defined in terms of Hofmann's semantics for LFPL terms (Section 3).

Lemma 3 (Type Preservation). *If* $\Gamma^1, \Delta^1 \overset{n}{\vdash} e : t$ *and* $e \overset{n}{\hookrightarrow} e'$ *then* $\Gamma^1, \Delta^1 \overset{n}{\vdash} e' : t$.

$$\frac{}{i \overset{n}{\hookrightarrow} i} \qquad \frac{}{x \overset{1}{\hookrightarrow} x} \qquad \frac{}{\lambda x.e \overset{0}{\hookrightarrow} \lambda x.e} \qquad \frac{e_0 \overset{0}{\hookrightarrow} \lambda x.e_2 \quad e_1 \overset{0}{\hookrightarrow} e_3 \quad e_2[x := e_3] \overset{0}{\hookrightarrow} e_4}{e_0(e_1) \overset{0}{\hookrightarrow} e_4}$$

$$\frac{e_0 \overset{1}{\hookrightarrow} e_2 \quad e_1 \overset{1}{\hookrightarrow} e_3}{e_0(e_1) \overset{1}{\hookrightarrow} e_2(e_3)} \qquad \frac{e_0 \overset{n}{\hookrightarrow} e_2 \quad e_1 \overset{n}{\hookrightarrow} e_3}{(e_0, e_1) \overset{n}{\hookrightarrow} (e_2, e_3)} \qquad \frac{e \overset{0}{\hookrightarrow} (e_1, e_2)}{\pi_i(e) \overset{0}{\hookrightarrow} e_i} \qquad \frac{e_0 \overset{1}{\hookrightarrow} e_1}{\pi_i(e_0) \overset{1}{\hookrightarrow} \pi_i(e_1)}$$

$$\frac{e_0 \overset{0}{\hookrightarrow} e_2 \quad e_1[x := e_2] \overset{0}{\hookrightarrow} e_3}{\mathsf{let}\ x = e_0\ \mathsf{in}\ e_1 \overset{0}{\hookrightarrow} e_3} \qquad \frac{e_0 \overset{1}{\hookrightarrow} e_2 \quad e_1 \overset{1}{\hookrightarrow} e_3}{\mathsf{let}\ x = e_0\ \mathsf{in}\ e_1 \overset{1}{\hookrightarrow} \mathsf{let}\ x = e_2\ \mathsf{in}\ e_3}$$

$$\frac{e_0 \overset{0}{\hookrightarrow} i \quad i \neq 0 \quad e_1 \overset{0}{\hookrightarrow} e_3}{\mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \overset{0}{\hookrightarrow} e_3} \qquad \frac{e_0 \overset{0}{\hookrightarrow} 0 \quad e_2 \overset{0}{\hookrightarrow} e_3}{\mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \overset{0}{\hookrightarrow} e_3}$$

$$\frac{e_0 \overset{1}{\hookrightarrow} e_3 \quad e_1 \overset{1}{\hookrightarrow} e_4 \quad e_2 \overset{1}{\hookrightarrow} e_5}{\mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \overset{1}{\hookrightarrow} \mathsf{if}\ e_3\ \mathsf{then}\ e_4\ \mathsf{else}\ e_5} \qquad \frac{e_1[f := \mathsf{let\ rec}\ f(x) = e_0\ \mathsf{in}\ \lambda x.e_0] \overset{0}{\hookrightarrow} e_2}{\mathsf{let\ rec}\ f(x) = e_0\ \mathsf{in}\ e_1 \overset{0}{\hookrightarrow} e_2}$$

$$\frac{e_0 \overset{1}{\hookrightarrow} e_2 \quad e_1 \overset{1}{\hookrightarrow} e_3}{\mathsf{let\ rec}\ f(x) = e_0\ \mathsf{in}\ e_1 \overset{1}{\hookrightarrow} \mathsf{let\ rec}\ f(x) = e_2\ \mathsf{in}\ e_3} \qquad \frac{e_0 \overset{n}{\hookrightarrow} e_1}{[e_0] \overset{n}{\hookrightarrow} [e_1]}$$

$$\frac{e_0 \overset{0}{\hookrightarrow} [e_3] \quad e_1[x := e_3] \overset{0}{\hookrightarrow} e_2}{\mathsf{let}\ [x] = e_0\ \mathsf{in}\ e_1 \overset{0}{\hookrightarrow} e_2} \qquad \frac{e_0 \overset{1}{\hookrightarrow} e_2 \quad e_1 \overset{1}{\hookrightarrow} e_3}{\mathsf{let}\ [x] = e_0\ \mathsf{in}\ e_1 \overset{1}{\hookrightarrow} \mathsf{let}\ [x] = e_2\ \mathsf{in}\ e_3}$$

$$\frac{e_0 \overset{1}{\hookrightarrow} e_1}{\langle e_0 \rangle \overset{0}{\hookrightarrow} \langle e_1 \rangle} \qquad \frac{e_0 \overset{0}{\hookrightarrow} \langle e_1 \rangle}{\tilde{}\, e_0 \overset{1}{\hookrightarrow} e_1} \qquad \frac{}{\mathsf{nil} \overset{1}{\hookrightarrow} \mathsf{nil}} \qquad \frac{e_0 \overset{1}{\hookrightarrow} e_3 \quad e_1 \overset{1}{\hookrightarrow} e_4 \quad e_2 \overset{1}{\hookrightarrow} e_5}{\mathsf{cons}(e_0, e_1)\mathsf{at}\ e_2 \overset{1}{\hookrightarrow} \mathsf{cons}(e_3, e_4)\mathsf{at}\ e_5}$$

$$\frac{e_0 \overset{1}{\hookrightarrow} e_3 \quad e_1 \overset{1}{\hookrightarrow} e_4 \quad e_2 \overset{1}{\hookrightarrow} e_5}{\mathsf{case}\ e_0\ \mathsf{of} \left\{ \begin{array}{l} \mathsf{nil} \to e_1 \\ '|'\ \mathsf{cons}(x, y)\mathsf{at}\ z \to e_2 \end{array} \right. \overset{1}{\hookrightarrow} \mathsf{case}\ e_3\ \mathsf{of} \left\{ \begin{array}{l} \mathsf{nil} \to e_4 \\ '|'\ \mathsf{cons}(x, y)\mathsf{at}\ z \to e_5 \end{array} \right.}$$

Fig. 2. Operational Semantics of First Stage in GeHB

Generated Programs as LFPL Programs. Hofmann's type system does not have an explicit rule for recursive definitions. Instead, the system assumes a signature of top-level functions under which programs are typable. It remains for us to show that programs generated by GeHB can be converted into LFPL programs. The key observations needed to make this connection are as follows:

1. A code value $\langle e \rangle$ generated by evaluating a GeHB program is free of escapes and level 0 terms.
2. The typing of let rec ensures that all such declarations can be lifted to top-level.
3. The result of lifting is a sequence of function declarations ending with a proper LFPL term.

In what follows we justify these claims.

Code Values Are Escape Free. To establish this property it is simpler to work with a superset of typed terms called expression families [24,26]. Expression families classify terms based on appropriateness for appearing at a certain level:

$$\begin{aligned}
e^n \in \mathbb{E}^n &::= i \mid x \mid e^n(e^n) \mid \mathsf{let}\ x = e^n\ \mathsf{in}\ e^n \mid (e^n, e^n) \mid \pi_1\ e^n \mid \pi_2\ e^n \mid \\
&\quad\ \mathsf{if}\ e^n\ \mathsf{then}\ e^n\ \mathsf{else}\ e^n \mid \mathsf{let\ rec}\ f(x) = e^n\ \mathsf{in}\ e^n \\
e^0 \in \mathbb{E}^0 &\mathrel{+}= \lambda x.e^0 \mid [e^0] \mid \mathsf{let}\ [x] = e^0\ \mathsf{in}\ e^0 \mid \langle e^1 \rangle \\
e^1 \in \mathbb{E}^1 &\mathrel{+}= \tilde{\ }e^0 \mid \mathsf{nil} \mid \mathsf{cons}(e^1, e^1)\mathsf{at}\ e^1 \mid \mathsf{case}\ e^1 \mathsf{of\ nil} \to e^1\ '|'\ \mathsf{cons}(x, y)\mathsf{at}\ z \to e^1
\end{aligned}$$

The first line defines expressions that can be associated with either level. The second and third definitions extend the set of terms that can be associated with levels 0 and 1, respectively. Values are defined similarly:

$$\begin{aligned}
v \in \mathbb{V}^0 &::= i \mid \lambda x.e^0 \mid (v, v) \mid [v] \mid \langle g \rangle \\
g \in \mathbb{V}^1 &::= i \mid x \mid g(g) \mid \mathsf{let}\ x = g\ \mathsf{in}\ g \mid (g, g) \mid \pi_1\ g \mid \pi_2\ g \mid \\
&\quad\ \mathsf{if}\ g\ \mathsf{then}\ g\ \mathsf{else}\ g \mid \mathsf{let\ rec}\ f(x) = g\ \mathsf{in}\ g \\
&\quad\ \mathsf{nil} \mid \mathsf{cons}(g, g)\mathsf{at}\ g \mid \mathsf{case}\ g\ \mathsf{of\ nil} \to g\ '|'\ \mathsf{cons}(x, y)\mathsf{at}\ z \to g
\end{aligned}$$

For level 0 values, the first three cases are standard. Values with the closed code constructor at the head must carry a level 0 value (thus $[t]$ is a strict constructor). Code values must carry a subterm that is a level 1 value. Level 1 values are level 1 expressions that do not contain escapes.

Proposition 2. *1.* $\mathbb{V}^n \subseteq \mathbb{E}^n$, *2.* $\Gamma; \Delta \overset{n}{\vdash} e : t$ *implies* $e \in \mathbb{E}^n$, *and 3.* $e^n \overset{n}{\hookrightarrow} e'$ *implies* $e' \in \mathbb{V}^n$.

Function Declarations Can Be Lifted. The subset of GeHB terms corresponding to LFPL terms is easily defined as the following subset of $\mathbb{V}^1$: [4]

$$\begin{aligned}
h \in \mathbb{H} &::= i \mid x \mid h(h) \mid \mathsf{let}\ x = h\ \mathsf{in}\ h \mid (h, h) \mid \pi_1\ h \mid \pi_2\ h \mid \mathsf{if}\ h\ \mathsf{then}\ h\ \mathsf{else}\ h \\
&\quad\ \mathsf{nil} \mid \mathsf{cons}(h, h)\mathsf{at}\ h \mid \mathsf{case}\ h\ \mathsf{of\ nil} \to h\ '|'\ \mathsf{cons}(x, y)\mathsf{at}\ z \to h
\end{aligned}$$

A term h is typable in our system as $f_i : (t_i^1 \to t'^1_i, 1); \emptyset \overset{1}{\vdash} h : t$ if and only if it is typable as $\emptyset \vdash h : t$ in LFPL's type system under a function signature $f_i : t_i^1 \to t'^1_i$.

Lifting recursive functions can be performed by a sequence of reductions carried out in the appropriate context. Declaration contexts are defined as follows:

$$D \in \mathbb{D} ::= [] \mid \mathsf{let\ rec}\ f(x) = D[h]\ \mathsf{in}\ D$$

[4] The LFPL calculus has no explicit let construct, but let has no effect on heap usage.

We will show that for every term g there is a unique term of the form $D[h]$. The essence of the argument is the following partial function from $\mathbb{V}^1$ to $\mathbb{V}^1$:

$$\frac{}{i \mapsto i} \quad \frac{}{x \mapsto x} \quad \frac{g_1 \mapsto D_1[h_1] \quad g_2 \mapsto D_2[h_2]}{g_1(g_2) \mapsto D_1[D_2[h_1(h_2)]]}$$

$$\frac{g_1 \mapsto D_1[h_1] \quad g_2 \mapsto D_2[h_2] \quad x \notin FV(D_2)}{\mathsf{let}\ x = g_1\ \mathsf{in}\ g_2 \mapsto D_1[D_2[\mathsf{let}\ x = h_1\ \mathsf{in}\ h_2]]} \quad \frac{g_1 \mapsto D_1[h_1] \quad g_2 \mapsto D_2[h_2]}{(g_1, g_2) \mapsto D_1[D_2[(h_1, h_2)]]}$$

$$\frac{g \mapsto D[h]}{\pi_i\ g \mapsto D[\pi_i\ h]} \quad \frac{g_1 \mapsto D_1[h_1] \quad g_2 \mapsto D_2[h_2] \quad g_3 \mapsto D_3[h_3]}{\mathsf{if}\ g_1\ \mathsf{then}\ g_2\ \mathsf{else}\ g_3 \mapsto D_1[D_2[D_3[\mathsf{if}\ h_1\ \mathsf{then}\ h_2\ \mathsf{else}\ h_3]]]}$$

$$\frac{g_1 \mapsto D_1[h_1] \quad g_2 \mapsto D_2[h_2]}{\mathsf{let\ rec}\ f(x) = g_1\ \mathsf{in}\ g_2 \mapsto \mathsf{let\ rec}\ f(x) = D_1[h_1]\ \mathsf{in}\ D_2[h_2]}$$

$$\frac{}{\mathsf{nil} \mapsto \mathsf{nil}} \quad \frac{g_1 \mapsto D_1[h_1] \quad g_2 \mapsto D_2[h_2] \quad g_3 \mapsto D_3[h_3]}{\mathsf{cons}(g_1, g_2)\mathsf{at}\ g_3 \mapsto D_1[D_2[D_3[\mathsf{cons}(h_1, h_2)\mathsf{at}\ h_3]]]}$$

$$\frac{g_1 \mapsto D_1[h_1] \quad g_2 \mapsto D_2[h_2] \quad g_3 \mapsto D_3[h_3] \quad x, y, z \notin FV(D_2)}{\begin{array}{l}\mathsf{case}\ g_1\ \mathsf{of\ nil} \to g_2\ '|'\ \mathsf{cons}(x, y)\mathsf{at}\ z \to g_3 \\ \mapsto D_1[D_2[D_3[\mathsf{case}\ h_1\ \mathsf{of\ nil} \to h_2\ '|'\ \mathsf{cons}(x, y)\mathsf{at}\ z \to h_3]]]\end{array}}$$

The definition above uses the Barendregt convention [1] of assuming that all bound variables are distinct in terms such as $D_1[h_1]$ and $D_2[h_2]$. This allows us to construct terms such as $D_1[D_2[h_1(h_2)]]$ without introducing accidental variable capture.

Proposition 3. *The following properties capture the basic features of lifting and declaration contexts:*

1. $g \mapsto e$ *implies that there is a unique pair* D *and* h *such that* $D[h] = e$.
2. $\Gamma; \Delta \overset{1}{\vdash} D[e] : t$ *implies that there exists a* $\Gamma' = f_i : (t_i^1 \to t'^1_i, 1)$ *such that* $\Gamma, \Gamma'; \Delta \overset{1}{\vdash} e : t$. *Furthermore, if* $\Gamma, \Gamma'; \Delta \overset{1}{\vdash} e' : t$, *then* $\Gamma; \Delta \overset{1}{\vdash} D[e'] : t$. *Finally,* $x \in dom(\Delta)$ *implies* $x \notin FV(D)$.
3. $\Gamma; \Delta \overset{1}{\vdash} g : t$ *and* $g \mapsto e$ *implies* $\Gamma; \Delta \overset{1}{\vdash} e : t$. *(Type-preservation for lifting.)*
4. $\Gamma; \Delta \overset{1}{\vdash} g : t$ *implies there is an* e *such that* $g \mapsto e$. *(Totality for lifting.)*

Proof. Part 1 is by a simple induction over the lifting derivation. Part 2 is by induction over the structure of D, and noting that let recs only extend Γ. Part 3 is by induction over the height of the lifting derivation, using the first fact in Part 2 as well as weakening. Part 4 is by induction over the typing derivation, and noting that the only risk to totality is the side conditions on free variables. These side conditions only arise with binding constructs, namely, let and case. For both constructs, the last observation in Part 2 implies that these side conditions hold.

Theorem 1 (Generated Programs are LFPL Programs).
Whenever $\emptyset; x_j : (\diamond, 1) \overset{0}{\vdash} e : \langle t \rangle$ *and* $e \overset{0}{\hookrightarrow} e'$, *then*

- *there exists a term* g *such that* $\langle g \rangle = e'$,
- *there exists* D *and* h *such that* $g \mapsto D[h]$,
- *there exists* $\Gamma' = f_i : (t_i^1 \to t'^1_i, 1)$ *such that* $\Gamma'; x_j : (\diamond, 1) \overset{1}{\vdash} h : t$, *and*
- *the LFPL judgment* $x_j : \diamond \vdash h : t$ *is derivable under signature* $f_i : t_i^1 \to t'^1_i$.

4 Parameterized Insertion Sort in GeHB

GeHB allows us to parameterize the insertion sort function presented in the introduction with a comparison operator. This generalization takes advantage of both staging and closedness annotations. Intuitively, this function will take as argument a staged comparison function, a second-stage list computation, and returns a second-stage list computation. Formally, the type of such a function would be:

```
(<A>*<A> -> <int>) * <list(A)> -> <list(A)>
```

for any given type `A`. But because this function will itself be a generator of function declarations, and the body of a function declaration cannot refer to any linear variables except for its arguments, we will need to use closedness annotations to express the fact that we have more information about the comparison operator. In particular, we *require* that the comparison operation not be allowed to allocate second-stage heap resources. The closed type constructor `[...]` allows us to express this requirement in the type of the generalized sort function as follows:

```
[<A>*<A> -> <int>] * <list(A)> -> <list(A)>
```

The staged parameterized sort function itself can be defined as follows:

```
let rec sort_gen(f,ll) =
  let [f'] = f in
  <let rec insert(d,a,l) =
     case l of
       nil -> cons(a, nil) at d
     | cons(b,t) at d' -> if ~(f'(<a>, <b>))
                          then cons(a, cons(b, t) at d') at d
                          else cons(b, insert(a, d', t)) at d
   in let rec sort(l) =
        case l of
          nil -> nil
        | cons(a,t) at d -> insert(d, a, sort(t))
      in sort(~ll)>

in sort_gen([fun (x,y) -> <fst(~x) > fst(~y)>],
            <cons((3,33), cons((100,12), cons((44,100),
               cons((5,13),nil) at d4) at d3) at d2) at d1>)
```

Without the staging and the closedness annotations, this definition is standard. We assume that we are given four free heap locations `d1` to `d4` to work with. As illustrated in Section 2, the staging annotations separate first stage computations from second stage ones. Closedness annotations appear in two places in the program. Once on the second line of the program, and once around the first argument to the `sort_gen` function at the end of the program. The closedness annotation at the end of the program asserts that the program fragment `fun (x,y) -> <fst(~x) > fst(~y)>` is free of operations that allocate resources on the heap during the second stage. Because function parameters are linear, and function definitions can only use their formal parameters, the variable `f` cannot be used in the body of the (inner) declaration of `insert`. Knowing that this variable has closed type, however, allows us to use the construction `let [f']=f` to copy the value of `f` to the non-linear variable `f'`, which can thereafter be used in the body of `insert`.

Evaluating the GeHB program above generates the following code fragment:

```
<let rec insert(d,a,l) =
  case l of nil -> cons(a, nil) at d
          | cons(b,t) at d' -> if fst(a) > fst(b)))
                               then cons(a, cons(b, t) at d') at d
                               else cons(b, insert(a, d', t)) at d
 in let rec sort(l) =
     case l of nil -> nil
             | cons(a,t) at d -> insert(d, a, sort(t))

    in sort(cons((3,33), cons((100,12), cons((44,100),
              cons((5,13),nil) at d4) at d3) at d2) at d1)>
```

The generated program is a valid LFPL program, and in this particular case, no lifting of function declarations is required. Note also that because higher-order functions are only present in the first stage, there is no runtime cost for parameterizing `insert` with respect to the comparison operator `f`. In particular, the body of the comparison operation will always be inlined in the generated sorting function.

5 Discussion

An unexpected finding from the study of GeHB is that generating LFPL programs requires the use of a primarily linear type system in the first stage. This is achieved using a single type constructor $[t]$ that captures both linearity [3] *and* closedness [4,24]. Both notions can separately be interpreted as comonads (c.f. [3] and [2]). The type constructor $[t]$ seems novel in that it compacts two comonads into one constructor. From a language design perspective, the possibility of combining two such comonads into one is attractive, because it reduces annotations associated with typing programs that use these comonads. A natural and important question to ask is whether the compactness of $[t]$ is an accident or an instance of a more general pattern. It will be interesting to see whether the useful composability of monads translates into composability of comonads.

GeHB provides new insights about multi-stage languages. In particular, we find that the closed type constructor is useful even when the language does not support reflection (the `run` construct). Because the notion of closedness is self-motivated in GeHB, we are hopeful that it will be easy to introduce a construct like `run` to this setting. With such a construct, not only will the programmer be able to construct computations for an embedded platform, but also to execute and receive results of such computations. An important practical problem for general multi-stage programming is code duplication. In the language presented here, no open code fragments can be duplicated. It will be interesting to see if, in practice, a linear typing approach could also be useful to address code duplication in multi-stage languages.

Czarnecki et al. [6] describe a template-based system for generating embedded software for satellite systems. In this approach programs are generated automatically from XML specifications and domain-specific libraries written in languages like Ada and Fortran. Their system was designed with the explicit goal of producing human-readable code and relies on the programmer to verify its correctness. In contrast, our goal is to design languages that let programmers reason about generated programs while they are reading and writing *the generator*. If enough guarantees can be made just by type checking the generator, then we know that the full *family* of programs produced by this generator is well-behaved. The contributions of GeHB and Czarnecki et al. are orthogonal and compatible.

6 Conclusions and Future Work

We have presented the two-level language GeHB that offers more expressivity than LFPL and at the same time offers more resource guarantees than general-purpose multi-stage languages. The technical challenge in designing GeHB lies in finding the appropriate type system. The development is largely modular, in the sense that many results on LFPL can be reused. While we find that a naive combination of two-level languages and LFPL is not satisfactory, we are able to show that recently proposed techniques for type-safe, reflective, and imperative multi-stage programming can be used to overcome these problems.

Future work will focus on studying the applicability of two-level languages for other kinds of resource bounds. For example, languages such as RT-FRP [28] and E-FRP [29] have complete bounds on all runtime space allocation. This is achieved by interpreting recursive definitions as difference equations indexed by the sequence of events or stimuli that drive the system. An important direction for future work is to define a useful class of terminating recursive definitions with a standard interpretation. Three interesting approaches appear in the literature, which we plan to investigate in future work: The use of a type system that 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 [12,11], the use of special iterators that are always guaranteed to terminate [5], enriching all types with information about space needed to store values of the respective types, and the use of the principle of non-size-increasing parameters [16,15].

Acknowledgements. Eric Allen and Jonathan Bannet read and commented on early drafts. We would also like to thank Paul Hudak and Valery Trifonov for discussions when these ideas were in their formative stages.

References

1. Barendregt, H. P. Lambda calculi with types. In *Handbook of Logic in Computer Science*, S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, Eds. Oxford University Press, Oxford, 1991.
2. Benaissa, Z. E.-A., Moggi, E., Taha, W., and Sheard, T. Logical modalities and multi-stage programming. In *Federated Logic Conference (FLoC) Satellite Workshop on Intuitionistic Modal Logics and Applications (IMLA)* (1999).
3. Benton, N., and Wadler, P. Linear logic, monads and the lambda calculus. In *the Symposium on Logic in Computer Science (LICS '96)* (New Brunswick, 1996), IEEE Computer Society Press.
4. Calcagno, C., Moggi, E., and Taha, W. Closed types as a simple approach to safe imperative multi-stage programming. In *the International Colloquium on Automata, Languages, and Programming (ICALP '00)* (Geneva, 2000), vol. 1853 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 25–36.
5. Crary, K., and Weirich, S. Resource bound certification. In *the Symposium on Principles of Programming Languages (POPL '00)* (N.Y., Jan. 19–21 2000), ACM Press, pp. 184–198.
6. Czarnecki, K., Bednasch, T., Unger, P., and Eisenecker, U. Generative programming for embedded software: An industrial experience report. In *Generative Programming and Component Engineer SIGPLAN/SIGSOFT Conference, GPCE 2002* (Oct. 2002), D. Batory, C. Consel, and W. Taha, Eds., vol. 2487 of *Lecture Notes in Computer Science*, ACM, Springer, pp. 156–172.
7. Davies, R., and Pfenning, F. A modal analysis of staged computation. *Journal of the ACM 48*, 3 (2001), 555–604.
8. Hofmann, M. Linear types and non-size-increasing polynomial time computation. In *the Symposium on Logic in Computer Science (LICS '99)* (July 1999), IEEE, pp. 464–473.
9. Hofmann, M. A type system for bounded space and functional in-place update. *Nordic Journal of Computing 7*, 4 (Winter 2000).
10. Hofmann, M. A type system for bounded space and functional in-place update. In *European Symposium on Programming (ESOP)* (2000), Lecture Notes in Computer Science, Springer-Verlag.
11. Hughes, R., and Pareto, L. 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)* (N.Y., Sept. 27–29 1999), vol. 34.9 of *ACM Sigplan Notices*, ACM Press, pp. 70–81.
12. Hughes, R., Pareto, L., and Sabry, A. Proving the correctness of reactive systems using sized types. In *In proceedings of the ACM Symposium on Principles of Programming Languages (POPL)* (St Petersburg, Florida, 1996), G. L. S. Jr, Ed., vol. 23, ACM Press.

13. Jones, N.D., Gomard, C.K., and Sestoft, P. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, 1993.
14. Kamin, S., Callahan, M., and Clausen, L. Lightweight and generative components II: Binary-level components. In *[25]* (2000), pp. 28–50.
15. Lee, C. S. Program termination analysis in polynomial time. In *Generative Programming and Component Engineering: ACM SIGPLAN/SIGSOFT Conference, GPCE 2002* (Oct. 2002), D. Batory, C. Consel, and W. Taha, Eds., vol. 2487 of *Lecture Notes in Computer Science*, ACM, Springer, pp. 218–235.
16. Lee, C. S., Jones, N. D., and Ben-Amram, A. M. The size-change principle for program termination. In *ACM Symposium on Principles of Programming Languages* (january 2001), vol. 28, ACM press, pp. 81–92.
17. MetaOCaml: A compiled, type-safe multi-stage programming language. Available online from http://www.cs.rice.edu/ taha/MetaOCaml/, 2001.
18. Moggi, E., Taha, W., Benaissa, Z. E.-A., and Sheard, T. An idealized MetaML: Simpler, and more expressive. In *European Symposium on Programming (ESOP)* (1999), vol. 1576 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 193–207.
19. Mycroft, A., and Sharp, R. A statically allocated parallel functional language. In *Automata, Languages and Programming* (2000), pp. 37–48.
20. National Instruments. Introduction to LabVIEW Real-Time. Available online from http://volt.ni.com/niwc/labviewrt/lvrt_intro.jsp?node=2381&node=2381, 2003.
21. National Instruments. LabVIEW FPGA Module. Available online from http://sine.ni.com/apps/we/nioc.vp?cid=11784&lang=US, 2003.
22. Nielson, F., and Nielson, H. R. *Two-Level Functional Languages.* No. 34 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, 1992.
23. 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.
24. Taha, W. *Multi-Stage Programming: Its Theory and Applications.* PhD thesis, Oregon Graduate Institute of Science and Technology, 1999. Available from [23].
25. Taha, W., Ed. *Semantics, Applications, and Implementation of Program Generation* (Montréal, 2000), vol. 1924 of *Lecture Notes in Computer Science*, Springer-Verlag.
26. Taha, W. A sound reduction semantics for untyped CBN multi-stage computation. Or, the theory of MetaML is non-trivial. In *Proceedings of the Workshop on Partial Evaluation and Semantics-Based Program Maniplation (PEPM)* (Boston, 2000), ACM Press.
27. Taha, W., Hudak, P., and Wan, Z. Directions in functional programming for real(-time) applications. In *the International Workshop on Embedded Software (EMSOFT '01)* (Lake Tahoe, 2001), vol. 221 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 185–203.
28. Wan, Z., Taha, W., and Hudak, P. Real-time FRP. In *the International Conference on Functional Programming (ICFP '01)* (Florence, Italy, September 2001), ACM.
29. Wan, Z., Taha, W., and Hudak, P. Event-driven FRP. In *Proceedings of Fourth International Symposium on Practical Aspects of Declarative Languages* (Jan 2002), ACM.

Pre-Scheduling: Integrating Offline and Online Scheduling Techniques

Weirong Wang[1], Aloysius K. Mok[1], and Gerhard Fohler[2]

[1] Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712-1188
{weirongw, mok}@cs.utexas.edu
http://www.cs.utexas.edu/{~weirong,~mok}

[2] Department of Computer Engineering
Malardalen University, Sweden
gerhard.fohler@mdh.se
http://www.idt.mdh.se/~gfr/

Abstract. The issue of integrating event-driven workload into existing static schedules has been addressed by Fohler's Slot Shifting method [5] [6]. Slot Shifting takes a static schedule for a time-driven workload as input, analyzes its slacks off-line, and makes use of the slacks to accommodate an event-driven workload on-line. The Slot Shifting method does not address how to produce a static schedule which it assumes as given. We propose an integrated approach with an off-line pre-scheduler and an on-line scheduler. The pre-scheduler produces a pre-schedule of the time-driven workload with sufficient embedded slacks to accommodate the event-driven workload. The on-line scheduler schedules all workloads by EDF with one extra constraint: the order of execution of the time-driven workload must follow the pre-schedule. Our pre-scheduler produces a valid pre-schedule if and only if one exists and is therefore optimal. We shall show that the choice of the pre-schedule cannot be considered independent of the event-driven workload.

1 Introduction

An embedded computer system may be required to process time-driven as well as event-driven tasks. Consider a node in a wireless sensor network. The wireless node may collect information from its sensors and performs signal processing transformations on the data at fixed time intervals, and these time-driven tasks may be characterized by periodic tasks [8]. The node may also perform mode changes and relay control signals among the nodes, and these event-driven tasks may be characterized as sporadic tasks [9]. This paper addresses the problem of scheduling a combination of periodic tasks and sporadic tasks by introducing the concept of "pre-scheduling" which integrates off-line and on-line scheduling techniques.

The Cyclic Executive (CE) [1] is a well accepted scheduling technique for periodic tasks. A CE schedule is produced off-line to cover the length of a hyper-period, i.e., the least common multiple of the periods of all the tasks. The CE schedule is represented as a list of "executives", where each executive defines an interval of time in a hyper-period to be allocated to a specific task. During on-line execution, a CE scheduler partitions the time line into an infinite number of consecutive hyper intervals, each the length of a hyper-period, and repeats the CE schedule within each hyper interval. The significant advantages of CE include the following: (1) the on-line overhead of CE is very low, $O(1)$ and can usually be bounded by a small constant. (2) a variety of timing constraints, such as mutual exclusion and distance constraints can be solved by off-line computation [12], which might otherwise be difficult to handle by typical on-line schedulers such as the Earliest Deadline First (EDF) scheduler. However, the drawback of CE is that it does not provide sufficient flexibility for handling the unpredictable arrival times of sporadic tasks. Even though sporadic tasks can be modeled as pseudo periodic tasks [9] and can therefore be scheduled by CE, this method may reserve excessive capacity whenever the deadline of a sporadic task is short compared with its mean time of arrival, as is typically the case of mode changes in state machines.

The Earliest Deadline First scheduler (EDF) is known to be optimal for scheduling periodic and sporadic tasks [8] [9]. EDF requires that at any moment during on-line scheduling, among all arrived but unfinished jobs, the job with the nearest deadline be selected for execution. However, the on-line complexity of EDF is $O(lgn)$, which is higher than that of CE ($O(1)$). Other than the potential problem with over capacity as mentioned above, the EDF scheduler does not provide strong predictability as the CE scheduler in terms of guaranteeing the ordering of jobs, which is unknown off-line in general. This ordering may be important if programmers exploit this ordering to minimize the use of data locks at run time, as is common practice in avionics software systems.

Seeking a balanced solution, Fohler has investigated the issue of integrating event-driven workload, modeled as aperiodic tasks into pre-computed static schedules by the Slot Shifting [5] method. Isovic and Fohler further integrated sporadic tasks into this approach [6]. In the Slot Shifting approach, the "slacks" left within the static schedule are computed off-line and then applied on-line to accommodate event-driven workload. While the Slot Shifting approach tests if a given set of aperiodic or sporadic tasks can be scheduled within a given static schedule, this method does not address how to generate a static schedule for periodic tasks to fit with the aperiodic and sporadic tasks if both the periodic and sporadic workload are given as input.

In this paper we propose an integrated approach consisting of an off-line component and an on-line component, as shown in Figure 1. The off-line component is called Slack-Reserving Pre-scheduler (SRP), and it produces a flexible pre-schedule with slacks embedded in it. A pre-schedule consists of a list of job fragments, and it defines a fixed order by which the fragments of periodic jobs are to be scheduled. A pre-schedule is *not* a schedule, because it does not define the exact time intervals in which each job fragment is scheduled. Instead, a job fragment in a pre-schedule may be preempted and/or delayed on-line to accommodate the unpredictable arrivals of sporadic tasks. The on-line component schedules the pre-schedule together with the jobs of the sporadic tasks by a Constrained Earliest Deadline First (CEDF) scheduler.

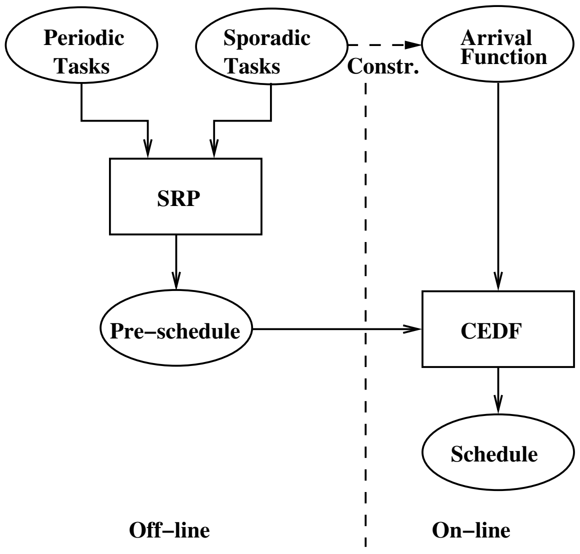


Fig. 1. Framework of the Integrated Scheduling Approach

The primary contribution of this paper is an optimal pre-scheduler – SRP, which takes as input a set of periodic tasks and a set of sporadic tasks and produces a valid pre-schedule for the given set of periodic tasks if and only if one exists. The second contribution of this paper is the demonstration of the non-existence of universally valid pre-schedule in general: Given a fixed set of periodic tasks, there might exist a set Φ of sporadic task sets, such that there exists a valid pre-schedule for each specific sporadic task set in Φ, but there might not exist a single common pre-schedule valid for all the sporadic task sets in Φ. This fact implies that an optimal pre-scheduler needs the information of the sporadic task set as input.

The remainder of the paper is organized as follows. Section 2 defines the task models and the definition of schedule. Section 3 defines pre-schedule, the on-line component, and the validity of a pre-schedule. Section 4 describes and analyzes the off-line component SRP. Section 5 shows the non-existence of universally valid pre-schedule in general. Section 6 measures the success rates of SRP. Section 7 addresses related work. Section 8 summarizes this paper and proposes future work.

2 Task Model

We shall adopt the usual model of real-time tasks and assume that a task can be modeled either as a periodic task or as a sporadic task with preperiod deadlines. All workloads are modeled as being made up of a periodic task set $\mathbf{T_P}$ and a sporadic task set $\mathbf{T_S}$. Each task T in either $\mathbf{T_P}$ or $\mathbf{T_S}$ is an infinite sequence of jobs. A job is defined by a triple: ready time, deadline and execution time and is written as (r, d, c); the job must receive a total execution time of c between time r and time d at run-time. A periodic task is defined by a 4-tuple: initial offset, period, relative deadline, and execution time and is written as (o, p, l, c). The first job of a periodic task is ready at time o, and the subsequent jobs are ready at the beginning of each period exactly p time units apart. We shall adopt as a convention in this paper the notation $X.a$ which denotes the attribute a of entity X. The j^{th} job of a periodic task T may be written as $(T.o + j \cdot T.p$, $T.o + j \cdot T.p + T.l$, $T.c)$, starting with job 0. Similarly, a sporadic task is defined by a tuple: (p, l, c), with its attributes defined the same way as a periodic task, except that the period of a sporadic task is the minimal length of the time interval between two consecutive jobs, and the actual ready time of any job of a sporadic task cannot be known until it arrives at run-time. We use an arrival function A to represent the arrival times of sporadic jobs in a particular run. A valid arrival function must satisfy the minimal arrival interval constraint: for any two consecutive jobs J_i and J_{i+1} of a sporadic task T, it must be the case that $A(J_{i+1}) - A(J_j) \geq T.p$. A job J of sporadic task T will be written as $(A(J)$, $A(J) + T.l$, $T.c)$.

Let the hyper-period P be the least common multiple of the periods of all periodic tasks in $\mathbf{T_P}$ and all sporadic tasks in $\mathbf{T_S}$. For any natural number n, the time interval $(n \cdot P$, $(n+1) \cdot P)$ is called a hyper interval. Our scheduler will generate a pre-schedule for the length of one hyper-period and repeat the same sequence of jobs of the periodic tasks every hyper interval at run-time, i.e., the on-line component of our scheduler restarts the pre-schedule at the beginning of every hyper interval. For ease of discussion, let $\mathbf{J_P}$ represent the list of the jobs of the periodic tasks within one hyper interval. Every job in $\mathbf{J_P}$ occurs exactly once in every hyper interval, and will be called a *periodic job* in the remainder of this paper. Notice that in this terminology, there are as many periodic jobs in $\mathbf{J_P}$ from the same periodic task as the number of periods of the task in a hyper-period. For each periodic task T in $\mathbf{T_P}$, for any natural number j less than $\frac{P}{T.p}$, there is a periodic job in $\mathbf{J_P}$ which is defined by $(r, d, c) = (T.o + j \cdot T.p$, $T.o + j \cdot T.p + T.l$, $T.c)$.

Job J is *before* job J' and job J' is *after* job J if and only if either (1) $J.r < J'.r$ and $J.d \leq J'.d$; or (2) $J.r \leq J'.r$ and $J.d < J'.d$. Job J *contains* job J' or job J' *is contained by* job J if and only if $J.r < J'.r$ and $J'.d < J.d$. Job J is *parallel with* job J' if and only if $J.r = J'.r$ and $J.d = J'.d$. For scheduling purposes, parallel periodic jobs can be transformed into a single periodic job with their aggregate execution time. Therefore, we shall not consider parallel jobs in the remainder of the paper. We assume that $\mathbf{J_P}$ is sorted such that a job with lower index is either before or contained by a job with higher index.

Example 1. The task sets $\mathbf{T_P}$ and $\mathbf{T_S}$ defined below are used in later examples in this paper.

$$\mathbf{T_P} = \{(90, 225, 30, 5), (0, 75, 75, 15), (0, 225, 225, 150)\}; \quad \mathbf{T_S} = \{(225, 25, 25)\}$$

The hyper-period of the task sets P is 225. The set of periodic jobs $\mathbf{J_P}$ is defined as follows.

$$\mathbf{J_P} = [(0, 75, 15), (90, 120, 5), (75, 150, 15), (0, 225, 150), (150, 225, 15)]$$

Fig. 2 illustrates the periodic tasks and the periodic jobs. The vertical bars indicate the ready times and deadlines of periodic jobs, and the length of the box inside the scope of a periodic job indicates its execution time. ■

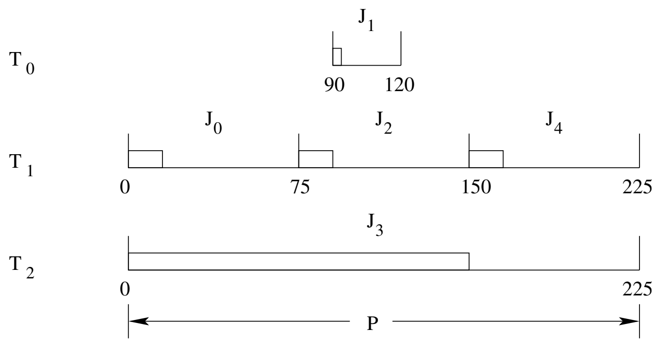


Fig. 2. Definition of $\mathbf{T_P}$ and $\mathbf{J_P}$

We shall assume that all tasks are preemptable and are scheduled on a single processor. A *schedule* is represented by a function S which maps each job to a set of time intervals. For instance, $S(J) = \{(b_0, e_0), (b_1, e_1)\}$ means that job J is scheduled to two time intervals (b_0, e_0) and (b_1, e_1). A schedule must satisfy the following two constraints. Firstly, there is at most one job to be scheduled at any point in time, i.e., at any time t, there exists at most one job J, by which $(b, e) \in S(J)$ and $b < t < e$. Secondly, the accumulated time allocated to each job between its ready time and deadline must be no less than its specified execution time, i.e., for any job J, $\sum_{(b,e)\in S(J)}(min(J.d, e) - max(J.r, b)) \geq J.c$.

3 Definitions of Pre-Schedule and the Online Component

A *pre-schedule* $\mathbf{F}$ is a list of fragments, where a *fragment* F is defined by a 5-tuple, (j, k, r, d, c) with the following meaning. Suppose J is the j^{th} periodic job in job list $\mathbf{J_P}$. F is the k^{th} (starting from 0) fragment in the pre-schedule of job

J, and job J is scheduled for at least a total execution time of c between times r and time d in every hyper interval.

The CEDF scheduler is the EDF scheduler plus one additional constraint: the sequencing of the periodic jobs must agree with the pre-schedule exactly. This may be implemented as follows. At the beginning of each hyper interval, let the first fragment in the pre-schedule be marked as "current". Define **R** as the set of sporadic jobs waiting to be scheduled. The set **R** is initialized at time 0 as an empty set. When a sporadic job becomes ready, it is added into **R**; when it is completely scheduled, it is removed from **R**. At any time, if the deadline d of the current fragment is earlier than the deadline of any job in **R**, the current fragment is scheduled; otherwise, the sporadic job with the earliest deadline in **R** is scheduled. When the execution time of the current fragment is completely scheduled, mark the next fragment in the pre-schedule as "current", and so on.

Example 2. Pre-schedule **F** is a pre-schedule for $\mathbf{J_P}$ and $\mathbf{T_S}$ defined in Example 1.

$$\mathbf{F} = [(0,0,0,75,15),(3,0,0,120,60),(2,0,75,120,15),\\ (1,0,90,120,5),(3,1,90,225,90),(4,0,150,225,15)]$$

Suppose that the first job of a sporadic task T_3, written as J_S arrives at different times in two different runs of the system, denoted by the arrival functions A and A', such that $A(J_S) = 0$ and $A'(J_S) = 30$. The on-line scheduler will produce two different schedules S and S', as illustrated in Fig. 3. Each box in the schedule represents an interval of time scheduled to a job. The fragments corresponding to periodic jobs being scheduled are on the top of the boxes in the figure. Notice that the start times and the finish times of periodic jobs may vary to accommodate the arrivals of sporadic jobs, but the order defined by the pre-schedule is always followed. ■

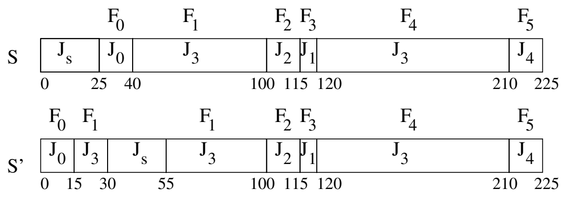


Fig. 3. Schedules Accommodating Different Arrival Functions

A pre-schedule is *valid* if and only if the following three conditions are satisfied. Firstly, under any valid arrival function A of $\mathbf{T_S}$, CEDF always produces a valid schedule for $\mathbf{T_S}$ and $\mathbf{J_P}$. Secondly, for any fragment F, suppose J is its corresponding periodic job. Then the ready time and the deadline of F will be within the effective scheduling scope of J, i.e., $J.r \leq F.r \leq F.d \leq J.d$. Third, for any fragment F, the execution time $F.c$ is greater than or equal to 0.

4 The Offline Component

The off-line component SRP produces a pre-schedule in three steps. The first step establishes $\mathbf{F^{(j,k)}}$, a list of partially defined fragments, in which only the attributes j and k of each fragment are defined. Essentially, the first step defines a total order of fragments in the pre-schedule. The following two steps will not change the order of the fragments. They only define the other attributes for each fragment. The second step defines the ready time r and deadline d of each fragment and produces $\mathbf{F^{(j,k,r,d)}}$. The third step defines the execution time c of each fragment to produce $\mathbf{F^{(j,k,r,d,c)}}$, the completed pre-schedule which is also represented as $\mathbf{F}$. We shall describe SRP in detail step by step and show that SRP produces a valid pre-schedule if and only if one exists.

4.1 Step 1: Generating $\mathbf{F^{(j,k)}}$

In this step, a partially defined pre-schedule $\mathbf{F^{(j,k)}}$ is created according to the following constraint-based definition.

A periodic job in $\mathbf{J_P}$ is a *top* periodic job if and only if it does not contain any other periodic job in $\mathbf{J_P}$. Constraint 1 and 2 are about the fragments of top periodic jobs.

Constraint 1 Each top periodic job has one and only one fragment.

Constraint 2 Let J and J' be any pair of top periodic jobs, and F and F' be their corresponding fragments. Fragment F is before fragment F' if and only if periodic job J is before periodic job J'.

Constraint 3 and Constraint 4 are about the fragments of non-top periodic jobs. For convenience, we introduce the concept of "gap". A gap is a sub-sequence of non-top fragments in $\mathbf{F^{(j,k)}}$. Let F_{T_0} and $F_{T_{t-1}}$ be the first and the last top job fragments. All fragments before F_{T_0} are in a gap denoted as $(\perp, F_{T_0})$, and all fragments after $F_{T_{t-1}}$ are in a gap denoted as $(F_{T_{t-1}}, \perp)$. Let F_{T_x} and $F_{T_{x+1}}$ be two consecutive fragments of top jobs, and all fragments of non-top jobs between them are in a gap denoted as $(F_{T_x}, F_{T_{x+1}})$. Fragments F_{T_0}, $F_{T_{t-1}}$ F_{T_x} and $F_{T_{x+1}}$ are "boundary fragments" of gaps.

Constraint 3 If a periodic job J contains at least one of the top jobs corresponding to the boundary fragments of a gap, then there exists one and only one fragment of J in the gap; otherwise, there exists no fragment of J in that gap.

Constraint 4 Suppose that both fragment F of periodic job J and fragment F' of periodic job J' are in the same gap. Case 1: J is before J'. F is before F'. Case 2: J contains J'. If the gap is $(\perp, F_{T_0})$, F is before F'; if the gap is $(F_{T_{t-1}}, \perp)$, F' is before F; if the gap is $(F_{T_x}, F_{T_{x+1}})$, and J' contains the corresponding job of $F_{T_{x+1}}$, F is before F'; if the gap is $(F_{T_x}, F_{T_{x+1}})$, and J' does not contain the corresponding job of $F_{T_{x+1}}$, F' is before F.

Example 3. $\mathbf{J_P}$ is defined in Example 1. Jobs J_0, J_1, and J_4 are top jobs, while job J_2 and J_3 are not. Partially defined pre-schedule $\mathbf{F^{(j,k)}}$ is shown below.

$$\mathbf{F^{(j,k)}} = [(0,0),(3,0),(2,0),(1,0),(2,1),(3,1),(4,0)]$$

4.2 Step 2: Generating $\mathbf{F^{(j,k,r,d)}}$

This step augments $\mathbf{F^{(j,k)}}$ to $\mathbf{F^{(j,k,r,d)}}$; in other words, it defines the ready time r and deadline d for each fragment. The ready times of fragments are defined as the earliest times satisfying the following constraints: (1) the ready time of each fragment is not earlier than the ready time of its corresponding job; (2) the ready times of fragments in $\mathbf{F^{(j,k,r,d)}}$ are non-decreasing. Similarly, the deadlines of fragments are defined as the latest times satisfying the following constraints: (1) the deadline of each fragment is not later than the deadline of its corresponding job; (2) the deadlines of fragments in $\mathbf{F^{(j,k,r,d)}}$ are non-decreasing.

Example 4. $\mathbf{J_P}$ and $\mathbf{F^{(j,k)}}$ are defined in Examples 1 and 3. $\mathbf{F^{(j,k,r,d)}}$ is defined follows.

$$\begin{aligned}\mathbf{F^{(j,k,r,d)}} = [&(0,0,0,75),(3,0,0,120),(2,0,75,120),(1,0,90,120),\\&(2,1,90,150),(3,1,90,225),(4,0,150,225)]\end{aligned}$$

4.3 Step 3: Generating $\mathbf{F^{(j,k,r,d,c)}}$

This step augments $\mathbf{F^{(j,k,r,d)}}$ to $\mathbf{F^{(j,k,r,d,c)}}$ by assigning the execution time c for each fragment. At the beginning of this step, we augment $\mathbf{F^{(j,k,r,d)}}$ to $\mathbf{F^{(j,k,r,d,x)}}$, representing the execution time of each fragment as a variable; then we solve the variables with a Linear Programming (LP) solver under three sets of constraints: non-negative constraints, sufficiency constraints and slack-reserving constraints, which are defined as follows.

Non-negative constraints require that the execution times to be non-negative; i.e., for every fragment F, $F.x \geq 0$.

Notice that a fragment may have zero execution time. A fragment with zero execution time is called a zero fragment; otherwise it is a non-zero fragment. Zero fragments are trivial in the sense that we can either delete or add them from or to a pre-schedule, and the schedule produced according to the pre-schedule will not be modified at all.

Sufficiency constraints require that for every periodic job J in $\mathbf{J_P}$, the aggregate execution time of its fragments in the pre-schedule shall be equal to the execution time of J; i.e., $\sum F.x = J.c$, where F is any fragment of J.

A slack-reserving constraint requires that the aggregate execution time of fragments and all sporadic jobs that must be completely scheduled within a time interval shall be less than or equal to the length of the time interval. Since an execution may start at time 0 but may last forever, therefore the number of time intervals is infinite. In order to make the pre-scheduling problem solvable, we must establish a finite number of *critical* slack-reserving constraints, such that if all critical slack-reserving constraints are satisfied by a pre-schedule, all slack-reserving constraints are satisfied. For this purpose, we define critical time intervals.

A time interval (b, e) is *critical* if and only if all of the following conditions are true. First, there exists a periodic job J in $\mathbf{J_P}$ and $J.r = b$. Second, the length of the time interval is shorter than or equal to the hyper-period; i.e., $e - b \leq P$. Third, at least one of the following cases is true: there exists a periodic job J' in $\mathbf{J_P}$ and either $J'.d = e$ or $J'.d + P = e$; or there exists a sporadic task T in $\mathbf{T_S}$, such that $e - b = T.p \cdot n + T.d$, where n is a natural number.

In order to define slack-reserving constraints on critical intervals, we introduce two functions, $E(b, e)$ and $Slack(l)$. Function $E(b, e)$ represents the aggregate execution time of all fragments that must be completely scheduled between critical time interval (b, e), and function $Slack(l)$ represents the maximal aggregate execution time of sporadic jobs that must be completely scheduled within a time interval of length l. The slack-reserving constraint on a critical time interval (b, e) is $E(b, e) \leq e - b - Slack(e - b)$.

Function $E(b, e)$ is computed with the following two cases. For the first case, time interval (b, e) is within one hyper interval $(0, P)$, i.e., $e \leq P$. Let F_b be the first fragment with a ready time greater than or equal to b, F_e be the last fragment with a deadline less than or equal to e; then $E(b, e) = \sum F.x$, where F is between and including F_b and F_e. For the second case, time interval (b, e) straddles hyper interval $(0, P)$ and hyper interval $(P, 2P)$, i.e., $e > P$. Then let F_b be the first fragment with a ready time equal or after to b, and let F_e be the last fragment with a deadline earlier than or equal to $e - P$, function $E(b, e) = \sum F.x$, where F is any fragment including and after F_b or before and including F_e.

Function $Slack(l)$ is computed as follows. Let function $n(T, l)$ be the maximal number of jobs of T that must be completely scheduled within l. If $l - \lfloor \frac{l}{T.p} \rfloor \cdot T.p < T.d$, $n(T, l) = \lfloor \frac{l}{T.p} \rfloor$; otherwise, $n(T, l) = \lfloor \frac{l}{T.p} \rfloor + 1$. Function $Slack(l)$ is equal to $\sum_{T \in T_S} T.c \cdot n(T, l)$.

Function $E(b, e)$ is a linear function of a set of variables, and $Slack(e - b)$ is a constant; therefore, the slack-reserving constraint is a linear constraint.

Non-negative constraints, sufficiency constraints and slack-reserving constraints on critical intervals are all linear and their total number is finite; therefore, the execution time assignment problem is a LP problem, which can be solved optimally in the sense that if there exists an assignment to the execution times under these constraints, the assignment will be found in finite time. Optimal LP solvers are widely available. If an optimal LP solver returns an assignment to execution times, SRP produces a fully defined pre-schedule with the execution times; otherwise, SRP returns failure.

Example 5. Assume that $\mathbf{J_P}$ and $\mathbf{T_S}$ are defined in Example 1, and $\mathbf{F^{(j,k,r,d)}}$ is defined in Example 4.

All non-negative constraints are listed as follows.

$$F_0.x \geq 0; \quad F_1.x \geq 0; \quad F_2.x \geq 0; \quad F_3.x \geq 0; \quad F_4.x \geq 0; \quad F_5.x \geq 0$$

All sufficiency constraints are listed as follows.

$$F_0.x = 15; \quad F_3.x = 5; \quad F_2.x + F_4.x = 15; \quad F_1.x + F_5.x = 150; \quad F_6.x = 15$$

The slack-reserving constraints on many critical time intervals will be trivially satisfied. For instance, time interval $(0, 75)$ is a critical time interval, however, $E(0, 25) = F_0.x = 15$, $Slack(75-0) = 25$, therefore the slack-reserving constraint on this time interval is always satisfied. We list all non-trivial slack-reserving constraints below.

$$\begin{array}{ll} F_1.x + F_2.x \leq 75 & \text{critical interval } (0, 120) \\ F_4.x + F_5.x \leq 90 & \text{critical interval } (90, 225) \end{array}$$

The pre-schedule **F** in Example 2 satisfies all the above constraints, so it is a valid pre-schedule. ■

4.4 Optimality and Computational Complexity

We prove the optimality and complexity of SRP in Theorem 1 and 2. Due to the page limitation, proofs for Lemma 1 and Lemma 2 can not be provided in this paper. They are available on-line at [13].

Lemma 1. *Slack-reserving constraints on all time intervals are satisfied by a pre-schedule produced by SRP.*

Lemma 2. *If SRP produces a pre-schedule* **F**, *then* **F** *is valid.*

Lemma 3. *If a valid pre-schedule exists, SRP produces one pre-schedule.*

Proof. The strategy of our proof is as follows. Suppose $\mathbf{F'}$ is a valid pre-schedule. First we transform $\mathbf{F'}$ into another valid pre-schedule $\mathbf{F^1}$ which can be obtained by augmenting attributes r, d, and c to each fragment of $\mathbf{F^{(j,k)}}$, the partially defined pre-schedule generated by Step 1 of SRP. Then we further transform $\mathbf{F^1}$ to $\mathbf{F^2}$, which can be obtained by augmenting attribute c to each fragment of $\mathbf{F^{(j,k,r,d)}}$ generated by Step 2 of SRP. Because $\mathbf{F^2}$ is a valid pre-schedule, there exists a pre-schedule satisfying all constraints in Step 3 of SRP, and so SRP must produce a pre-schedule. We define the transformations and show the correctness of the claims below.

Transformation 1: from $\mathbf{F'}$ to $\mathbf{F^1}$

Apply the following rules one at a time to the pre-schedule, until no rule can be further applied.

Rule 1: If F_b and F_e are two consecutive fragments of the same job J, merge them into one fragment F of job J, with $F.c = F_b.c + F_e.c$, $F.r = F_b.r$ and $F.d = F_e.d$.

To facilitate other rules, we first define a primitive $Swap(J_f, J_r, F_b, F_e)$, which swaps all non-zero fragments (fragments with non-zero execution times) of job J_f before all non-zero fragments of job J_r between and including fragments F_b and F_e. It may be implemented as follows. Let C_f be the aggregate execution time of all fragments of job J_f between and including F_b and F_e. Let $C_{[x,y)}$ be the aggregate execution time of all fragments of job J_f or J_r including and after F_x but before F_y. Let F_m be the latest fragment of either job J_f or job J_r, such that $C_{[b,m)}$ is less than or equal to C_f. For every fragment F of job J_r including and after F_b but before F_m, change it to a fragment of job J_f without changing its ready time, deadline, or execution time. Similarly, for every fragment F of job J_f after F_m but before and including F_e, change it to a fragment of job J_r. If $C_{[b,m)} = C_f$, make F_m a fragment of J_r. Otherwise, split F_m into two consecutive fragments F_{mf} of job J_f and F_{mr} of job J_r, such that the ready times and deadlines of F_{mf} and F_{mr} are the same as those of F_m, but $F_{mf}.c = C_f - C_{[b,m)}$ and $F_{mr}.c = F_m - F_{mf}.c$.

Rule 2: Assume that job J_f is before job J_r. Let F_b be the first non-zero fragment of J_r, and F_e be the last non-zero fragment of J_f. If F_b is before F_e in the pre-schedule, apply $Swap(J_f, J_r, F_b, F_e)$.

Rule 3: Assume that job J_f contains job J_r. Let F_b and F_e be the first and the last non-zero fragments of J_r. If there exists a non-zero fragment of J_r between F_b and F_e, apply $Swap(J_f, J_r, F_b, F_e)$.

Rule 4: Assume that non-zero fragment F_b of non-top job J_f is before non-zero fragment F_e of non-top job J_r, and F_b and F_e are within the same gap (defined in Step 1 of SRP), represented as (x, y), where x and y are either $\perp$ or the boundary fragments. Apply $Swap(J_f, J_r, F_b, F_e)$ if and only if one of the following cases is true. Case 1: J_f is contained by J_r, y is not $\perp$ and J_f contains the top job corresponding to y. Case 2: J_f contains J_r, and either y is $\perp$ or J_r does not contain the top job corresponding to y.

Claim 0: Transformation 1 terminates.

Proof of this claim can be found in [13].

Claim 1: Pre-schedule $\mathbf{F^1}$ is a valid pre-schedule.

None of the transformation rules will change the aggregate execution time of any periodic job or any critical time interval. Also, the ready time and deadline of a fragment in transformation is maintained within the valid scope of its corresponding job.

Claim 2: Pre-schedule $\mathbf{F^1}$ can be obtained by augmenting $\mathbf{F^{(j,k)}}$.

By the transformation rules, if $\mathbf{F^1}$ does not satisfy the constraints listed in Step 1 of SRP, Transformation 1 won't terminate. Notice that if there is no fragment F of J between any consecutive pair of fragments of top jobs in $\mathbf{F^1}$, we can always plug in a fragment F of job J with 0 execution time at proper position. Therefore, there is a one-to-one in-order mapping between $\mathbf{F^1}$ and $\mathbf{F^{(j,k)}}$, which implies the claim.

Transformation 2: from $\mathbf{F^1}$ to $\mathbf{F^2}$

Re-assign attribute r and d of every fragment according to the algorithm in Step 2 of SRP.

Claim 3: $\mathbf{F^2}$ is a valid pre-schedule.

For every fragment, its ready time and deadline are still within the valid range of its corresponding periodic job, and the execution time of each fragment remains unchanged. If $\mathbf{F^2}$ is not valid, there must exist an arrival function A, such that CEDF fails with it at a time e, which is either the deadline of a sporadic job or a periodic job. Let b be the latest idle time before e or time 0 if such an ideal time does not exist. Let F_b^2 be the first fragment in $\mathbf{F^2}$ with a ready time at or after time b and F_e^2 be the last fragment in $\mathbf{F_2}$ with a deadline before or at time e. Fragment F_b^2 must be the first fragment of its corresponding job J_b and $F_b^2.r = J_b.r$. Let F_b^1 be the corresponding fragment in $\mathbf{F^1}$, then $F_b^2.r \leq F_b^1.r$. Similarly, $F_e^1.d \leq F_e^2.d$. Then all fragments between and including F_b^1 and F_e^1 must be scheduled between time b and e when CEDF schedules according to $\mathbf{F^1}$. With the same arrival function A, CEDF must also fail with $\mathbf{F^1}$, which contradicts with Claim 1.

Claim 4: $\mathbf{F^2}$ can be obtained by augmenting $\mathbf{F^{(j,k,r,d)}}$.

Follows Claim 2 and the fact that the attributes r and d of fragments in $\mathbf{F^2}$ are defined by the algorithm in Step 2 of SRP.

Claim 5: SRP must produce a pre-schedule.

It is easy to show that all non-negative, sufficient execution time and slack-reserving constraints are necessary for a valid pre-schedule. Because of Claim 3, $\mathbf{F^2}$ satisfies all these constraints. Together with Claim 4, $\mathbf{F^2}$ implies the existence of an execution time augment to $\mathbf{F^{(j,k,r,d)}}$ that satisfies all those constraints. Therefore Step 3 of SRP must return a pre-schedule. ■

Immediately follows Lemma 2 and Lemma 3, we have the following theorem.

Theorem 1. *Given* $\mathbf{J_P}$ *and* $\mathbf{T_S}$*, SRP produces a valid pre-schedule if and only if it exists.*

Theorem 2. *The computational complexity of SRP is* $O(C(n_p^2, n_p \cdot (n_p + n_s))$*, where* n_p *and* n_s *are the number of periodic jobs and the maximal number of sporadic jobs in one hyper-period respectively, and* $C(n, m)$ *represents the complexity of the LP solver with* n *variables and* m *constraints.*

Proof. The complexity of the LP solver is the major factor. The maximal number of fragments is bounded by n_p^2. The number of non-negative constraints and the number of execution time constraints are bounded by n_p, and the number of slack-reserving constraints on critical intervals is bounded by $n_p \cdot (n_p + n_s)$. ■

5 The Non-existence of Universally Valid Pre-Schedule

The slack embedded in a pre-schedule by SRP specifically targets one given set of sporadic tasks. Is it possible to produce a one-size-fits-all pre-schedule? To formalize the discussion, we define the concept of *universally valid* pre-schedule. For a given periodic task set $\mathbf{T_P}$, a pre-schedule $\mathbf{F_U}$ is universally valid if and only if the following condition is true for any sporadic task set $\mathbf{T_S}$: either $\mathbf{F_U}$ is a valid pre-schedule for $\mathbf{T_P}$ and $\mathbf{T_S}$ or there exists no valid pre-schedule for them. If a universally valid pre-schedule exists for every periodic task set, then a more dynamic optimal integrating scheduling scheme could be constructed: A universally valid pre-schedule might be produced off-line without requiring the knowledge of the sporadic task set, and the sporadic task set might be allowed to change on-line, under the admission control of a schedulability test as defined in [6]. The optimality criterion for scheduling here means that if a specific sporadic task set is rejected on-line, then no valid pre-schedule of the periodic tasks exists for this sporadic task set. However, by Example 6, we can show that the one-size-fits-all pre-schedules are impossible: universally valid pre-schedule does not exist in general, so the more dynamic scheme we surmise above cannot be done.

Example 6. Suppose one periodic task set and two alternative sporadic task sets are defined as follows:

$$\mathbf{T_P} = \{(56, 100, 19, 9), (0, 100, 100, 71)\}; \quad \mathbf{T_s} = \{(50, 10, 10)\}; \ \mathbf{T'_s} = \{(20, 4, 4)\}$$

With both sets of sporadic tasks, hyper-period $P = 100$, and the periodic job list is defined as follows.

$$\mathbf{J_p} = [(56, 75, 9), (0, 100, 71)]$$

There exists a valid pre-schedule $\mathbf{F}$ for $\mathbf{T_P}$ and $\mathbf{T_S}$, and a valid pre-schedule $\mathbf{F'}$ for $\mathbf{T_P}$ and $\mathbf{T'_S}$, defined as follows.

$$\mathbf{F} = [(1, 0, 0, 75, 46), (0, 0, 56, 75, 9), (1, 1, 56, 100, 25)]$$
$$\mathbf{F'} = [(1, 0, 0, 75, 48), (0, 0, 56, 75, 9), (1, 1, 56, 100, 23)]$$

Suppose there is a universally valid pre-schedule $\mathbf{F_U}$. Let X_b be the total execution time of all fragments of J_1 before the last fragment of J_0 in $\mathbf{F_U}$; let X_a be the aggregate execution time of all fragments of J_1 after the first fragment of J_0 in $\mathbf{F_U}$. A universally valid pre-schedule $\mathbf{F_U}$ must satisfy the following set of contradicting constraints, therefore does not exist.

$X_b + X_a \geq 71$	sufficiency constraint for J_1
$X_b \leq 46$	slack-reseving constraint on $(0, 75)$ for $\mathbf{T_S}$
$X_a \leq 23$	slack-reseving constraint on $(56, 100)$ for $\mathbf{T'_S}$

6 Experiments

We investigate the performance of SRP in terms of the success rate of pre-schedule generation. As we mentioned, [5] and [6] address the issue of validity testing on pre-schedules but do not provide any specific pre-scheduling algorithm. So for comparison purposes, we need to consider "reasonable" pre-scheduling algorithms which do not take into account the sporadic workload. Accordingly, we choose a well-understood pre-scheduling algorithm called Offline-EDF(OEDF). Given a set of periodic jobs, OEDF first schedule the periodic jobs according to EDF in one hyper period, which is a sequence of fragments. Without changing the sequence of fragments, OEDF then minimizes the ready-times and maximize the deadlines of fragments under the following constraints: The sequence of all ready-times and the sequence of all deadlines are both non-decreasing, and the ready-time and deadline of each fragment is within the range of its corresponding periodic job. OEDF is aggressive in preparing pre-schedules to accommodate the impact of sporadic tasks; we deem it a reasonable pre-scheduler for a fair comparison with SRP.

We measure the success rate of both SRP and OEDF on the same eight groups of test cases. There are 100 cases for each group. Tasks in each test case are randomly generated, but they must satisfy the following requirements. The total utilization rate of sporadic tasks is always between 10% and 20%. The relative deadline of each sporadic task is between its execution time and its period. The total utilization rate of periodic tasks are set to different ranges in different test groups, as shown in Table 1. The relative deadline of each periodic task is always equal to its period.

Table 1. SRP vs. OEDF

Sporadic Utl. Rate (%)	10-20							
Periodic Utl. Rate (%)	0.01-10	10-20	20-30	30-40	40-50	50-60	60-70	70-80
SRP Success Rate (%)	100	99	97	98	98	97	97	89
OEDF Success Rate (%)	100	96	77	57	35	33	29	28

The measurement shows that when total utilization rate is not low, the difference between SRP and OEDF is significant. Take the last group as an example: When the total utilization rate is between 80% and 100% (70% to 80% periodic task utilization plus 10% to 20% sporadic task utilization), SRP can successfully produce valid pre-schedule for 89 cases out of 100 cases, while OEDF can produce valid pre-schedule for only 28 cases.

7 Related Work

In the introduction section, we have reviewed the Slot Shifting scheme [5] [6], CE scheduler [1] and EDF scheduler [8]. In this section, we review other related work.

Gerber *et al* propose a parametric scheduling scheme [7]. They assume that the execution order of real-time tasks is given, the execution times of tasks may range between upper and lower bounds, and there are relative timing constraints between tasks. The scheduler consists of an off-line component and an on-line component. The off-line component formulates a “calendar” which stores two functions to compute the lower and upper bounds of the start time for each task. The bound functions of a task take the parameters of tasks earlier in the sequence as variables and can therefore be solved on-line. Based on the bounds on the start time, the on-line dispatcher decides when to start scheduling the real-time task. The parametric scheduling scheme and our integrated approach share the following similarities: (1) Both schemes make use of off-line computation to reduce on-line scheduling overhead and yet guarantee schedulability. (2) Both schemes make use of slacks to handle a non-deterministic workload represented either as variable parameters or as sporadic tasks. Yet, the parametric scheduling scheme assumes that the sequencing order of the tasks is given, but our integrated approach does not. Instead, the order is established by the pre-scheduler in our case. The techniques applied in these two schemes are also quite different.

The pre-scheduling techniques in our integrated approach are more related to a line of research by Erschler *et al* [4] and Yuan *et al* [14], even though their work focuses on non-preemptive scheduling of periodic tasks. Erschler *et al* introduce the concept of “dominant sequence” which defines the set of possible sequences for non-preemptive schedules. They also introduce the concept of “top job”. Building upon the work of Erschler *et al*, Yuan *et al* propose a “decomposition approach”. Yuan *et al* define several relations between jobs, such as “leading” and “containing”, and apply them in a rule-based definition of “super sequence” which is equivalent to dominant sequence. The partially defined pre-schedule $\mathbf{F}^{(\mathbf{j},\mathbf{k})}$ in our paper is similar to the dominant sequence or the super sequence, and we adopt some of their concepts and terminology as mentioned. However, in view of the NP-hardness of the non-preemptive scheduling problem, approximate search algorithms are applied to either the dominant sequence or super sequence to find a schedule in [4] and [14]. In this paper, the execution time assignment problem on the partially defined pre-schedule can be solved optimally by LP solver.

General composition approaches have been proposed in recent years, such as the open system environment [3] by Deng and Liu, temporal partitioning [10] by Mok and Feng, and hierarchical schedulers [11] by Regehr and Stankovic. The general composition schemes usually focus on the segregation between components or schedulers, which means that the behavior of a component or scheduler will be independent to the details of other components or schedulers. In contrast, our integrated approach focuses on paying the scheduling overhead off-line and yet provides strict ordering and deadline guarantees by making intensive use of the information available about all tasks.

8 Conclusion

This paper proposes an integrated approach that combines the advantages of CE and EDF schedulers. We present an off-line Slack Reserving Pre-scheduling algorithm which is optimal in the sense that given a periodic task set and a sporadic task set, SRP produces a valid pre-schedule if and only if one exists. Experiment shows that SRP has a much higher success rate than that of an EDF-based pre-scheduling algorithm. We also demonstrate the non-existence of universally valid pre-schedules, which implies that every optimal pre-scheduling algorithm requires both the periodic task set and the sporadic task set as input.

For future work, SRP may be extended to accommodate workload models other than periodic and sporadic tasks with preperiod deadlines, such as complex inter-task dependencies. We may also apply the pre-scheduling techniques to other forms of integrated schedulers, such as a combination of CE and the fixed priority scheduler.

References

1. T. P. Baker, A. Shaw. The cyclic executive model and Ada, Proceedings of IEEE Real-Time Systems Symposium, pp.120–129, December 1988.
2. G. B. Dantzig. Linear Programming and Extensions, Princeton University Press, 1963.
3. Z. Deng and J. Liu. Scheduling Real-Time Applications in an Open Environment. Real-Time Systems Symposium, pp. 308–319, December 1997.
4. J. Erschler, F. Fontan, C. Merce, F. Roubellat. A New Dominance Concept in Scheduling n Jobs on a Single Machine with Ready Times and Due Dates, Operations Research, 31:114-127.
5. G. Fohler. Joint Scheduling of Distributed Complex Periodic and Hard Aperiodic Tasks in Statically Scheduled Systems, Real-Time Systems Symposium, pp. 152–161, December 1995.
6. D. Isovic, G. Fohler. Handling Sporadic Tasks in Off-line Scheduled Distributed Real-Time Systems, the 11th EUROMICRO Conference on Real-Time Systems, pp. 60–67, York, England, July 1999.
7. R. Gerber, W. Pugh, and M. Saksena. Parametric Dispatching of Hard Real-Time Tasks, IEEE Trans. on Computers, Vol.44, No.3, pp. 471–479, Mar 1995.
8. C.L. Liu and J.W. Layland. Scheduling Algorithms for Multi-programming in Hard Real-time Environment. Journal of ACM 20(1), 1973.
9. A. K. Mok. Fundamental Design Problems of Distributed Systems for the Hard-Real-Time Environment. Ph.D. thesis. MIT. 1983.
10. A. K. Mok, X. Feng. Towards Compositionality in Real-Time Resource Partitioning Based on Regularity Bounds. Real-Time Systems Symposium, pp. 129–138, 2001.
11. J. Regehr, J. A. Stankovic. HLS: A Framework for Composing Soft Real-Time Schedulers. Real-Time Systems Symposium, pp. 3–14, 2001.

12. Duu-Chung Tsou. Execution Environment for Real-Time Rule-Based Decision Systems. PhD thesis, Department of Computer Sciences, The University of Texas at Austin, 1997.
    esting for Real-Time Tasks, Real-Time Systems, Vol. 11, No. 1, pp. 19–39, 1996.
13. W. Wang, A. K. Mok, G. Fohler. Pre-Scheduling: Integrating Off-line and On-line Scheduling Techniques, http://www.cs.utexas.edu/users/mok/RTS/pubs.html.
14. X. Yuan, M.C. Saksena, A.K. Agrawala, A Decomposition Approach to Non-Preemptive Real-Time Scheduling, Real-Time Systems, Vol. 6, No. 1, pp. 7–35, 1994.

Author Index