

Commenced Publication in 1973

Founding and Former Series Editors:

Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

David Hutchison

Lancaster University, UK

Takeo Kanade

Carnegie Mellon University, Pittsburgh, PA, USA

Josef Kittler

University of Surrey, Guildford, UK

Jon M. Kleinberg

Cornell University, Ithaca, NY, USA

Friedemann Mattern

ETH Zurich, Switzerland

John C. Mitchell

Stanford University, CA, USA

Moni Naor

Weizmann Institute of Science, Rehovot, Israel

Oscar Nierstrasz

University of Bern, Switzerland

C. Pandu Rangan

Indian Institute of Technology, Madras, India

Bernhard Steffen

University of Dortmund, Germany

Madhu Sudan

Massachusetts Institute of Technology, MA, USA

Demetri Terzopoulos

University of California, Los Angeles, CA, USA

Doug Tygar

University of California, Berkeley, CA, USA

Moshe Y. Vardi

Rice University, Houston, TX, USA

Gerhard Weikum

Max-Planck Institute of Computer Science, Saarbruecken, Germany

Rogério de Lemos
Cristina Gacek
Alexander Romanovsky (Eds.)

Architecting Dependable Systems IV

Volume Editors

Rogério de Lemos
University of Kent, Computing Laboratory
Canterbury, Kent CT2 7NF, UK
E-mail: r.delemos@kent.ac.uk

Cristina Gacek
Alexander Romanovsky
Newcastle University, School of Computing Science
Newcastle upon Tyne, NE1 7RU, UK
E-mail: {cristina.gacek, alexander.romanovsky}@ncl.ac.uk

Library of Congress Control Number: 2007931900

CR Subject Classification (1998): D.2, D.4

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN 0302-9743
ISBN-10 3-540-74033-3 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-74033-9 Springer Berlin Heidelberg New York

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

Springer is a part of Springer Science+Business Media

springer.com

© Springer-Verlag Berlin Heidelberg 2007
Printed in Germany

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12102341 06/3180 5 4 3 2 1 0

Foreword

On a recent visit to Sweden I had the pleasure of traveling by train between Stockholm and Malmö over several segments that spanned a few days. The trains always ran on time and were very comfortable. Particularly convenient was the fact that a passenger could get on the Internet during the trip simply by using her ticket number as the access code. One of the features on the on-line provider's home page was a map of that area of Sweden, with the train's current location updated in real-time. Impressed by this, I made a point of mentioning it to my Swedish host, and the conversation quickly turned to how much today's systems, such as my train, rely on and are controlled by software.

My host subsequently relayed a somewhat less pleasant experience with the same type of train on which I had just arrived. During one of his recent trips, the software controlling the angle at which one of the train's cars entered and exited curves was not functioning properly. As a result, the G-force experienced by the passengers during turns had almost doubled. The problem was fixed at the next station, where the train sat idle for some time while it literally rebooted. I found myself having two reactions to this story. As a traveler, my first thought was that it is a good thing we do not have to reboot airplanes in mid-flight. As a software engineer, I wondered exactly how the software was constructed and what caused this particular problem.

As this story illustrates, as "regular" people we constantly depend on software in our daily lives, yet frequently do not realize it and rarely, if ever, stop to analyze the implications of that dependence and the extent of the software's actual *dependability*. On the other hand, as software engineering professionals, we are not only becoming increasingly aware of the importance of software dependability, but have amassed an arsenal of techniques and tools to help us ensure it. Many of these techniques and tools have traditionally been used to ensure dependability in existing systems "after the fact," that is, after the system has been designed, and possibly implemented and even deployed. However, a new class of emerging techniques gives dependability first-class status in the development of software-intensive systems by integrating dependability into software engineering processes from their inception. These techniques rely on a software system's architecture as the principal driver of dependability.

This book is the fourth in a series of collected papers on software architecture-based dependability solutions. The book addresses a number of on-going challenges (such as system modeling and analysis for dependability and ensuring dependability in distributed systems) as well as some timely issues (such as the role of the Architecture Analysis and Design Language—AADL—standard in modeling dependable systems, architecture-driven dependability in the automotive domain, and the benefits of following the model-driven architecture paradigm in ensuring software dependability). This book joins its three companion volumes in forming an indispensable source for the fast-growing community of software researchers and practitioners who are confronting the challenges posed by this important topic and architecting the software systems on which we rely every day.

Nenad Medvidovic
University of Southern California

Preface

This is the fourth book in a series on Architecting Dependable Systems we started five years ago that brings together issues related to software architectures and the dependability of systems. This book includes expanded and peer-reviewed papers based on the selected contributions to the Workshop on Architecting Dependable Systems (WADS), organized at the 2006 International Conference on Dependable Systems and Networks (DSN 2006), and a number of invited papers written by recognized experts in the area.

Identification of the system structure (i.e., architecture) early in its development process makes it easier for the developers to make crucial decisions about system properties and to justify them before moving to the design or implementation stages. Moreover, the architectural level views support abstracting away from details of the system, thus facilitating the understanding of broader system concerns. One of the benefits of a well-structured system is the reduction of its overall complexity, which in turn leads to a more dependable system that typically has fewer remaining faults and is capable of dealing with errors and faults of different types in a well-defined, cost-effective and disciplined way.

System dependability is defined as the reliance that can be justifiably placed on the service delivered by the system. It has become an essential aspect of computer systems as everyday life increasingly depends on software. It is therefore a matter for concern that dependability issues are usually left until too late in the process of system development.

Making decisions and reasoning about structure happen at different levels of abstraction throughout the software development cycle. Reasoning about dependability at the architectural level has recently been in the focus of researchers and practitioners because of the complexity of emerging applications. From the perspective of software engineering, traditionally striving to build software systems that are fault-free, architectural consideration of dependability requires the acceptance of the fact that system models need to reflect that it is impossible to avoid or foresee all faults. This requires novel notations, methods and techniques providing the necessary support for reasoning about faults (including fault avoidance, fault tolerance, fault removal and fault forecasting) at the architectural level.

This book comes as a result of bringing together research communities of software architectures and dependability, and addresses issues that are currently relevant to improving the state of the art in architecting dependable systems. The book consists of four parts: Architectural Description Languages, Architectural Components and Patterns, Architecting Distributed Systems, and Architectural Assurances for Dependability.

The first part entitled “Architectural Description Languages” (ADLs) includes four papers focusing on various aspects of defining and using ADLs with an aim to ensure system dependability. The first paper of this part, “Architecting Dependable Systems with the SAE Architecture Analysis and Description Language (AADL),” is prepared by J. Tokar. The Avionics Systems Division of the Society of Automotive Engineers (SAE) has recently adopted this language to support incorporation of formal methods

and engineering models into analysis of software and system architectures. The SAE AADL is a standard that has been specifically developed for embedded real-time safety critical systems. It supports the use of various formal approaches to analyzing the impact of system composition from hardware and software components and allows the generation of system glue code with the performance qualities predicted. The paper highlights features of AADL that facilitate the development of system architectures and demonstrates how the features can be used to conduct a wide variety of dependability analysis of the AADL architectural models. To help in the understanding of AADL, the paper begins with a discussion of software and systems architecture and then shows how the AADL supports these concepts.

The second paper, written by A.-E. Rugina, K. Kanoun and M. Kaâniche and entitled “A System Dependability Modeling Framework using AADL and GSPNs,” describes a modeling framework that generates dependability-oriented analytical models from Architecture Analysis and Design Language (AADL) specifications, which are then used for evaluating dependability measures, such as reliability or availability. The proposed stepwise approach transforms an AADL dependability model into a Generalized Stochastic Petri Net (GSPN) by applying model transformation rules that can be automated and then processed by existing tools.

P. Cuenot, D. Chen, S. Gérard, H. Lönn, M.-O. Reiser, D. Servat, R. T. Kolagari, M. Törnngren and M. Weber contribute to the book with the paper “Towards Improving Dependability of Automotive Systems by Using the EAST-ADL Architecture Description Language.” Management of engineering information is critical for developing modern embedded automotive systems. Development time, cost efficiency, quality and dependability all benefit from appropriate information management. System modeling based on an architecture description language is a way to keep this information in one information structure. EAST-ADL is an architecture description language for automotive embedded systems. It is currently refined in the ATESSST project. The focus of this paper is on describing how dependability is addressed in the EAST-ADL. The engineering process defined in the EASIS project is used as an example illustrating support for engineering processes in EAST-ADL.

The final paper of the first part is “The View Glue” written by A. Radjenovic and R. Paige. It focuses on domain-specific architecture description languages (ADLs), particularly for safety critical systems. In this paper, the authors outline the requirements for safety critical ADLs, the challenges faced in their construction, and present an example – AIM – developed in collaboration with the safety industry. Explaining the key principles of AIM, the authors show how to address multiple and cross-cutting concerns through active system views and how to ensure consistency across such views. The AIM philosophy is supported by a brief exploration of a real-life jet engine case study.

The second part of this book is entitled “Architectural Components and Patterns” and contains five papers. In the first paper, entitled “A Component-Based Approach to Verification and Validation of Formal Software Models,” D. Desovski and B. Cukic present a methodology for the automated decomposition and abstraction of Software Cost Reduction (SCR) specifications. The approach enables one to identify components in an SCR specification, perform the verification component by component, and apply compositional verification methods. It is shown that the algorithms can be used in large specifications.

In the paper “A Pattern-Based Approach for Modeling and Analyzing Error Recovery,” A. Ebnenasir and B. H. C. Cheng present an object analysis pattern, called the corrector pattern, that provides a generic reusable strategy for modeling error recovery requirements in the presence of faults. In addition to templates for constructing structural and behavioral models of recovery requirements, the corrector pattern also contains templates for specifying properties that can be formally verified to ensure the consistency between recovery and functional requirements. Additional property templates can be instantiated and verified to ensure the fault-tolerance of the system to which the corrector pattern has been applied. This analysis method is validated in terms of UML diagrams and demonstrated in the context of an industrial automotive application.

The third paper of this part, “Architectural Fault Tolerance Using Exception Handling,” is written by R. de Lemos. This paper presents an architectural abstraction based on exception handling for structuring fault-tolerant software systems. The proposed architectural abstraction transforms untrusted software components into idealized fault-tolerant architectural elements (iFTE), which clearly separate the normal and exceptional behaviors, in terms of their internal structure and interfaces. The feasibility of the proposed approach is evaluated in terms of a simple case study.

R. Buskens and O. Gonzalez contribute to the book with the paper “Model-Centric Development of Highly Available Software Systems.” They present the Aurora Management Workbench (AMW) as a solution to the problem of integration a high availability (HA) middleware with the system that uses it. AMW is an HA middleware and a set of tools for building highly available distributed software systems. It is unique in its approach to developing highly available systems: developers focus only on describing key architectural abstractions of their system as well as system HA needs in the form of a model. Tools then use the model to generate much of the code needed to integrate the system with the AMW HA middleware, which also uses the model to coordinate and control HA services at run-time. The paper discusses initial successes using the approach proposed in developing commercial telecom systems.

The final paper of this part, written by L. Grunske, P. Lindsay, E. Bondarev, Y. Papadopoulos and D. Parker and entitled “An Outline of an Architecture-Based Method for Optimizing Dependability Attributes of Software-Intensive Systems,” provides an overview of 14 different approaches for optimizing the architectural design of systems with regard to dependability attributes and cost. As a result of this study, the authors present a meta-method that specifies the process of designing and optimizing architectures with contradicting requirements on multiple quality attributes.

Part three of the book is on “Architecting Distributed Systems” and includes six papers focusing on approaches to architectural level reasoning about dependability concerns of distributed systems. This part starts with a paper by P. Inverardi and L. Mostarda that is entitled “A Distributed Monitoring System for Enhancing Security and Dependability at an Architectural Level.” The paper presents the DESERT tool that allows the automatic generation of distributed monitoring systems for enhancing security and dependability of a component-based application at the architectural level. The DESERT language permits one to specify both the component interfaces and interaction properties in terms of correct component communications. DESERT uses these specifications to generate one filter for each component. Each filter locally detects when its component communications violate the property and can undertake a set of reaction policies.

In their paper, entitled “Architecting Dynamic Reconfiguration in Dependable Systems,” A. T. A. Gomes, T. V. Batista, A. Joolia and G. Coulson introduce a generic approach to supporting dynamic reconfiguration in dependable systems. The proposed approach is built on the authors’ view that dynamic reconfiguration in such systems needs to be causally connected at runtime to a corresponding high-level software architecture specification. More specifically, two causally connected models are defined, an architecture-level model and a runtime-level model. Dynamic reconfiguration is applied either through an architecture specification at the architectural level, or through reconfiguration primitives at the runtime level. This approach supports both foreseen and unforeseen reconfigurations—these are handled at both levels with a well-defined mapping between them.

T. Dumitraş, D. Roşu, A. Dan and P. Narasimhan, in their paper “Ecotopia: An Ecological Framework for Change Management in Distributed Systems,” present Ecotopia, a framework for change management in complex service-oriented architectures (SOA) that is ecological in its intent: it schedules change operations with the goal of minimizing the service-delivery disruptions by accounting for their impact on the SOA environment. Ecotopia handles both external change requests, such as software upgrades, and internal changes requests, such as fault-recovery actions. The authors evaluate the Ecotopia framework using two realistic change-management scenarios in distributed enterprise systems.

In the fourth paper, entitled “Generic-Events Architecture: Integrating Real-World Aspects in Event-Based Systems,” A. Casimiro, J. Kaiser, and P. Veríssimo describe an architectural solution consisting of an object model environment, which can be easily composed, representing software/hardware entities capable of interacting with the environment, and an event model that allows one to integrate real-world events and events generated in the system. The architectural solution and the event-model permit one to compose large applications from basic components, following a hierarchical composition approach.

The fifth paper is by C. Heller, J. Schalk, S. Schneelee, M. Sorea, and S. Voss and is entitled “Flexible Communication Architecture for Dependable Time-Triggered Systems.” The authors propose an approach expressed in terms of a dependable and flexible communication architecture that supports flexibility in the use of time-triggered technologies and delivers a highly effective, reliable and dependable system design. This work is undertaken in the context of safety-critical aerospace applications.

The final paper of this part is by L. Baresi, S. Guinea, and M. Plebani and is entitled “Business Process Monitoring for Dependability.” This paper proposes a dynamic technique for ensuring that dependability requirements of service-based business processes are maintained during runtime. The approach is based upon the concept of supervision rules, which are the union of user-defined constraints. These rules are used to monitor how a BPEL process evolves, and specify corrective actions that must be executed when a set of constraints is violated. For facilitating the specification of these rules, the authors provide suitable languages and tools that enable one to abstract from the underlying technologies, and to hide how the system guarantees the dependability requirements.

The fourth part of this book is on “Architectural Assurances for Dependability” and contains three papers. The first paper, “Achieving Dependable Systems by Synergistic Development of Architectures and Assurance Cases” by P. J. Graydon, J. C.

Knight and E. A. Strunk, explains the basic principles of assurance-based development, and shows how the proposed approach can be used to provide assurance case goals for architectural choices. In this approach, first the architecture is developed to provide evidence required in the assurance case, and then the assurance case is refined as architectural choices are made. In this context, choices are better informed than an architecture chosen in an ad hoc manner.

The next paper, entitled “Towards Evidence-Based Architectural Design for Safety-Critical Software Applications,” is prepared by W. Wu and T. Kelly. This paper proposes a Triple Peaks process framework, within which a system model, deviation model, and mitigation model are proposed and linked together. The application of this framework is supported by the use of Bayesian Belief Networks and collation of relevant evidence. The link between the three models is elaborated by means of a case study. The core contribution of this paper is addressing safety using evidence available at the architectural level.

The paper “Extending Failure Modes and Effects Analysis Approach for Reliability Analysis at the Software Architecture Design Level,” by H. Sozer, B. Tekinerdogan and M. Aksit, shows how the Failure Mode and Effect Analysis (FMEA) and Fault Tree Analysis (FTA) can be extended and used in combination for conducting reliability evaluation of software systems at the architecture design level. The extensions of FMEA and FTA are related to using a failure domain model for systematic derivation of failures, prioritization of failure scenarios based on a user’s perception, and an FTA impact analysis model that does not explicitly require a running system. The software architecture reliability analysis approach (SARAH) proposed in the paper is illustrated using an industrial case for analyzing the reliability of the software architecture of a digital TV.

Architecting dependable systems is now a well-recognized area, attracting interest and contributions from many researchers. We are certain that this book will prove valuable for both developers designing complex applications and researchers building techniques supporting them. We are grateful to many people who made this book possible. Our thanks go to the authors of the contributions for their excellent work, the DSN 2006 WADS participants for their active participation in the discussions, and Alfred Hofmann from Springer for believing in the idea of a series of books on this important topic and for helping us to get it published. Last but not least, we very much appreciate the efforts of our reviewers who helped us in ensuring the high quality of the contributions. They are L. Baresi, L. Bass, T. V. Batista, J. Bryans, R. Buskens, F. Castor Filho, B. H.C. Cheng, A. C. Costa, B. Cukic, D. Desovski, T. Dumitras, J. Durães, A. Ebneenasir, L. Grunske, C. Heller, N. Henderson, M. Kaâniche, K. Kanoun, T. Kelly, S. Kharchenko, M. Klein, H. Lönn, T. Maxino, L. Mostarda, P. Narasimhan, R. F. Paige, P. Pelliccione, A. Radjenovic, S. Riddle, G. Rodrigues, D. Rosu, A.-E. Rugina, S. Schneelee, E. Strunk, B. Tekinerdogan, M. Tichy, J. L. Tokar, S. Voss and several anonymous reviewers.

Rogério de Lemos
Cristina Gacek
Alexander Romanovsky

Table of Contents

Part 1. Architectural Description Languages

Architecting Dependable Systems with the SAE Architecture Analysis and Description Language (AADL)	1
<i>Joyce L. Tokar</i>	
A System Dependability Modeling Framework Using AADL and GSPNs	14
<i>Ana-Elena Rugina, Karama Kanoun, and Mohamed Kaâniche</i>	
Towards Improving Dependability of Automotive Systems by Using the EAST-ADL Architecture Description Language	39
<i>Philippe Cuenot, DeJiu Chen, Sébastien Gérard, Henrik Lönn, Mark-Oliver Reiser, David Servat, Ramin Tavakoli Kolagari, Martin Törngren, and Matthias Weber</i>	
The View Glue	66
<i>Alek Radjenovic and Richard Paige</i>	

Part 2. Architectural Components and Patterns

A Component-Based Approach to Verification and Validation of Formal Software Models	89
<i>Dejan Desovski and Bojan Cukic</i>	
A Pattern-Based Approach for Modeling and Analyzing Error Recovery	115
<i>Ali Ebneenasir and Betty H.C. Cheng</i>	
Architectural Fault Tolerance Using Exception Handling	142
<i>Rogério de Lemos</i>	
Model-Centric Development of Highly Available Software Systems	163
<i>Rick Buskens and Oscar Gonzalez</i>	
An Outline of an Architecture-Based Method for Optimizing Dependability Attributes of Software-Intensive Systems	188
<i>Lars Grunske, Peter Lindsay, Egor Bondarev, Yiannis Papadopoulos, and David Parker</i>	

Part 3. Architecting Distributed Systems

A Distributed Monitoring System for Enhancing Security and Dependability at Architectural Level	210
<i>Paola Inverardi and Leonardo Mostarda</i>	
Architecting Dynamic Reconfiguration in Dependable Systems	237
<i>Antônio Tadeu A. Gomes, Thais V. Batista, Ackbar Joolia, and Geo Coulson</i>	
Ecotopia: An Ecological Framework for Change Management in Distributed Systems	262
<i>Tudor Dumitraş, Daniela Roşu, Asit Dan, and Priya Narasimhan</i>	
Generic-Events Architecture: Integrating Real-World Aspects in Event-Based Systems	287
<i>António Casimiro, Jörg Kaiser, and Paulo Verissimo</i>	
Flexible Communication Architecture for Dependable Time-Triggered Systems	316
<i>Christoph Heller, Josef Schalk, Stefan Schneelee, Maria Sorea, and Sebastian Voss</i>	
Business Process Monitoring for Dependability	337
<i>Luciano Baresi, Sam Guinea, and Marco Plebani</i>	

Part 4. Architectural Assurances for Dependability

Achieving Dependable Systems by Synergistic Development of Architectures and Assurance Cases	362
<i>Patrick J. Graydon, John C. Knight, and Elisabeth A. Strunk</i>	
Towards Evidence-Based Architectural Design for Safety-Critical Software Applications	383
<i>Weihsang Wu and Tim Kelly</i>	
Extending Failure Modes and Effects Analysis Approach for Reliability Analysis at the Software Architecture Design Level	409
<i>Hasan Sozer, Bedir Tekinerdogan, and Mehmet Aksit</i>	
Author Index	435

Architecting Dependable Systems with the SAE Architecture Analysis and Description Language (AADL)

Joyce L. Tokar

Pyrrhus Software,
P.O. Box 1352, Phoenix, AZ 85001, USA
tokar@pyrrhusoft.com

Abstract. Architecture Description Languages provide significant opportunity for the incorporation of formal methods and engineering models into the analysis of software and system architectures. The SAE AADL [1] is a standard that has been developed for embedded real-time safety critical systems which will support the use of various formal approaches to analyze the impact of the composition of systems from hardware and software and which will allow the generation of system glue code with the performance qualities predicted. This paper will highlight the components and features of AADL that facilitate the development of system architectures comprised of both hardware and software components. It will demonstrate how the features of AADL may be used to conduct a wide variety of dependability analysis on AADL architectural models. To help in the understanding of AADL the paper will begin with a discussion of software and systems architecture. It will then show how the AADL supports these concepts.

Keywords: Architecture description language, Architecture analysis, Dependability, Modeling.

1 Introduction

An architecture involves multiple views (perspectives) of the system [3] and relies, in whole or part, on patterns or styles of representation. These views enable the exchange of information about a system or system of systems (SOS) across a wide variety of domains of discourse. For example, a logical view of an architecture describes the logical relationships between various components of a system that may be used to assess the logical flow of information through a system. Whereas a physical view of an architecture describes how the architecture is realized in the physical environment.

Architecture is embodied in its components, both hardware and software; their relationships to each other and the environment; and the principles governing its design and evolution. The architecture of a program or computing system is the structure or structures of the system, which comprise software and hardware elements, the externally visible properties of those elements, and the relationships among them.

Thus, architecture helps to organize a system into components and interfaces between these components. There are both functional and nonfunctional

characteristics that can be modeled as properties of a component or system. These properties along with the model itself can then be used in analysis of the system.

1.1 Architecture: The Foundation of Good Software and Systems Engineering

Research in Architecture Description Languages (ADLs) has been focused on finding methods to reduce the cost of developing applications and for increasing the potential for commonality between different members of a closely related product family. Software development based on common architectural idioms has shifted from the lines-of-code view to coarser-grained architectural elements and their overall interconnection structure [4]. To support architecture-based development, formal modeling notations, analysis and development tools that operate on architectural specifications are needed. Architecture description languages and their accompanying toolsets have been proposed as the answer. An ADL for software applications focuses on the high-level structure of the overall application rather than the implementation details of any specific source module.

The AADL is an architecture descriptions language that includes support for the development of both the execution platform components and the software components in the system architectural specification. Thus, the characteristics of both the software and the execution platform are available for analysis.

AADL is based upon the ground-breaking work in architecture description languages funded by United States (US) Defense Advanced Research Projects Agency (DARPA) and the US Army Aviation and Missile Command (AMCM). Experiences from the use of the MetaH language and toolset developed by Honeywell Technology Laboratories [4] provided the foundation for the definition and development of the AADL.

1.2 Software and Systems Development with Modeling Languages

With modeling languages the approach to software and systems development is more integrated with the variety of participants from domains across the entire operational embedded system. The architecture model may be refined from the requirements phase through development into integration. This enables the detection of errors early in the process rather than at integration level. Functional interfaces and systems interface are integrated into the overall model development which provides a predictable system at the completion of development.

Analysis of the architecture may take place throughout the development cycle. Preliminary abstract models may be analyzed for feasibility prior to actual system construction. These models may also be used to evaluate interfaces and design constraints. Model analysis facilitates the early detection of errors and flaws in a system.

2 The SAE Architecture Analysis and Description Language (AADL)

A key to an AADL-based engineering process is an architectural specification that is an abstraction of the system. The architectural specification must be semantically strong enough to reflect relevant aspects of the application domain.

Since the AADL was designed for real-time embedded system, the architectural specification focuses on the task structure and interaction topology of a system and captures both the software architecture and hardware architecture. This real-time architecture model is the basis for various analyses, ranging from schedulability analysis to reliability and safety analysis.

The architectural specification is the basis for automated system generation and component integration. The actual components of a system may be hand-coded software, or components modeled in a domain-specific notation and auto-generated.

Although there is a considerable diversity in the capabilities of different ADLs, all share a similar conceptual basis, or ontology, that determines a common foundation of concepts and concerns for architectural description [5]. The main elements of this ontology include: components, connectors, systems, properties, constraints and styles.

2.1 The Elements of AADL

This section shows the correspondence between this ontology and AADL elements. These AADL entities are used to construct analyzable models of real-time, embedded, systems.

2.1.1 Components

Components represent the primary (computational) elements and data stores of a system. Intuitively, they correspond to the boxes in box-and-line descriptions of architectures. Typical examples of components include such things as clients, servers, filters, objects, blackboards, and databases. In most ADLs components may have multiple interfaces, each interface defining a point of interaction between a component and its environment.

In AADL, the definition of *components* is extended to include *execution platform components* such as buses and memories. A system is then the composition of software components, execution platform components, and possibly other system components. AADL supports multiple interfaces between components through the definition of ports. AADL also supports the concept of a family of components through the definition of multiple implementations that correspond to a component type definition.

2.1.2 Connectors

Connectors facilitate the communication channels between components and coordinate activities among components. Examples include simple forms of interaction, such as pipes, procedure call, and event broadcast. Connectors may also represent more complex interactions, such as client-server protocol or an SQL link between a database and an application. Connectors have interfaces that define the roles played by the various participants in the interaction represented by the connector.

In AADL, *connections* are represented as the actual linkage between components. *Ports* may be used to represent the flow of data and events between threads and execution platform components. *Data ports* are used for unqueued state data. *Event data ports* are used for queued message data. *Event ports* are used for events. A *port group* represents a grouping of ports or port groups. Outside a component a port

group is treated as a single unit. Inside a component the ports of a port group can be accessed individually. Port groups allow collections of ports to be connected with a single connection.

Interactions between components may also be represented by *subprogram call* sequences. *Flows* represent the logical information flow through components. Connections in AADL may be used to specify a mode change that may result in the change of component configuration and interaction.

2.1.3 Systems

Systems represent related collections of components and connectors. In modern ADLs, a key property of systems descriptions is that the overall topology of a system is defined independently from the components and connectors that make up the system. Systems may also be hierarchical: components and connectors may represent subsystems that have internal architectures.

AADL has a *package* configuration element that enables the collection of components and their connections into modules that may be reused in the definition of a system. The AADL *system* component is used to represent the composition of software components, execution platform components, and possibly other system components along with their corresponding connections. A system may be composed of a set of systems.

2.1.4 Properties

Properties represent semantic information about a system and its components that goes beyond structure. Different ADLs focus on different properties, but virtually all provide some way to define one or more extra functional properties together with tools for analyzing those properties. For example, some ADLs allow one to calculate overall system throughput and latency based on performance estimates of each component and connector.

AADL provides a predefined set of *properties* that are applicable to the components and connectors of the AADL. These properties specify various characteristics of a system such as the system start-up time, or the binding of elements to actual software or hardware modules. AADL includes support for *user defined property sets* that provide additional information about a specific system or family of systems that are unique to the user's domain.

2.1.5 Constraints

Constraints represent claims about an architectural design that should remain true even as it evolves over time. Typical constraints include restrictions on allowable values of properties, topology, and design vocabulary. For example, an architecture might constrain its design so that the number of clients of a particular server is less than some maximum value.

In AADL, *constraints* appear in several different forms. In the language definition itself, constraints are represented as assertions on the hybrid automata that describe the behavior of an AADL specification. Component and connector constraints are represented as limitations on the values of the corresponding properties. User may define a design vocabulary to further constrain an AADL specification using annex libraries and subclauses.

2.1.6 Styles

Styles represent families of related systems. An architectural style typically defines a vocabulary of design element types and rules for composing them. Examples include dataflow architectures based on shared data space and a set of knowledge sources, and layered systems. Some architectural styles additionally prescribe a framework as a set of structural forms that specific applications can specialize.

AADL offers a variety of capabilities to support architectural styles. At the lowest level, as mentioned earlier, component types may have *multiple implementations* thus enabling the definition of a single interface that has a family of underlying implementations. Components may be represented hierarchically. Thus, multiple levels of a system may correspond to different levels in a component or system hierarchy. Similarly, AADL supports the refinement of specifications that provides for the layering of specification definition.

For example, a communication system within an aircraft may be viewed at the highest level as a system consisting of a sending process, a receiving process, and a bus. In this configuration, the system would indicate that data flows from the out data port the sender to the in data port of the receiver via the bus.

Further analysis of this system may reveal that the bus is actually another system of components attached at a lower level to another bus. The AADL specification may be refined to reflect this enhancement to the definition of the system architecture. The system may be refined further to reveal the actual execution platform components. At each stage of the refinement process, the architecture may be analyzed to determine more about the overall behavior of the system.

In addition, packages in AADL may be used to collect common architectural elements together for reuse. Property sets may be used to define property values that are unique to a given style. And annexes may be used to introduce additional representations needed to enhance a given specification or family of specifications.

2.2 Combining Elements

Based on the common foundation for ADLs, AADL components represent the elements of a system including software elements and execution platform elements. Shared data, ports, and parameters represent the interfaces between components. Components are linked together using subprogram call sequences and connections. Logical data connections are represented using flows. Systems may be represented as collections of components, hierarchies of components, or systems of systems. Properties represent both functional and non-functional characteristics of an architecture. Property values are intrinsically important to model analysis tools as well as code generation and system generation. Properties may be used to represent some of the design constraints of an architecture. Constant property values may be used to represent invariants on a design. Users may build property sets to define properties to be used to constrain a particular system or family of systems even further. A component may have multiple implementations to supply support for families of components.

Figure 1 shows the graphical symbols of the component categories in AADL.

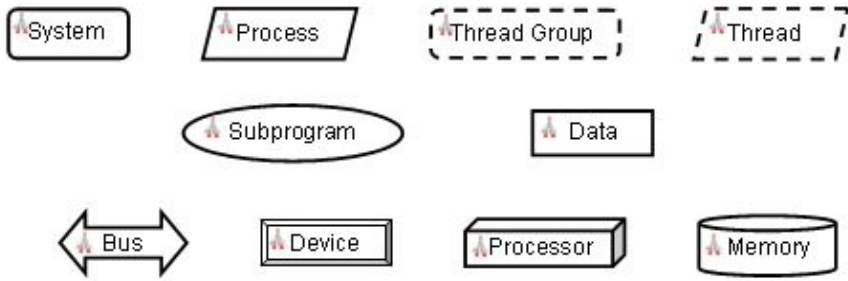


Fig. 1. AADL components graphical symbols

2.2.1 Application Components

Components represent the primary computational elements and data of a system. A component may have multiple implementations, each implementation defining a particular representation of a component in its specified environment.

The AADL *system* component represents the composition of software components and execution platform components. Systems support the hierarchical organization of threads and processes. A system may be comprised of a system of systems.

Software and execution platform components provide the basic modeling concepts that are needed to describe the runtime architecture of an application system in terms of concurrent tasks and their interaction as well as their mapping to the underlying execution platform.

Processes represent virtual address spaces whose boundaries may be enforced at execution time. *Threads* represent schedulable units of concurrent execution. *Thread groups* are used to organize collections of threads within a process.

Data components represent potentially shareable data. And *subprograms* represent callable sequences of code.

2.2.2 Execution Platform Components

AADL is designed to capture the physical components and the bindings of the application components onto the execution platform. Hence, the language includes the definition of execution platform components.

The *processor* component is defined to provide the scheduling and execution services. *Memory* components represent storage for data and source code.

Devices provide interfaces to external and environmental components.

Bus components provide the physical connectivity between processors, memory and devices.

2.2.3 AADL Component Type Extension Hierarchy

Component types can be declared in terms of other component types, i.e., a component type can *extend* another component type – inheriting its declarations and property associations. If a component type extends another component type, then features, flows, and property associations can be added to those already inherited. A component type extending another component type can also refine the declaration of

inherited feature and flow declarations by more completely specifying partially declared component classifiers and by associating new values with properties.

Component type extensions form an *extension hierarchy*, i.e., a component type that extends another component type can also be extended. Figure 2 illustrates the extension hierarchy. In this example, the component type GPS extends component type Position System inheriting ports declared in Position System. It may add a port, refine the data type classifier of a port incompletely declared in Position System, and overwrite the value of one or more properties. Component types being extended are referred to as *ancestors*, while component types extending a component type are referred to as *descendents*.

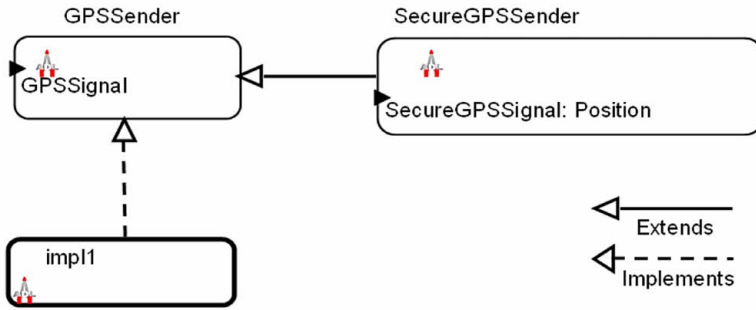


Fig. 2. Graphical representation of an AADL type, type extension and implementation

AADL offers a variety of features that may be utilized in a component implementation to refine and complete the component type definition. The *refines type* clause refines the properties and features in the component type. *Subcomponents* declare component parts. The *calls* clause specifies a sequence of subprogram calls. *Flows* declare the logical flows through components. *Modes* declare the modes of operation that are applicable to the component implementation. *Properties* define the property associations for the component implementation.

2.2.4 AADL Subcomponents

A *subcomponent* represents a component contained within another component, i.e., declared within a component implementation. Subcomponents contained in a component implementation may be instantiations of component implementations that contain subcomponents themselves. This results in a component containment hierarchy that ultimately describes the whole physical system as a system instance.

A *subcomponent declaration* may resolve required *subcomponent access* declared in the component type of the subcomponent. A subcomponent may be declared to apply to specific modes defined within the component implementation. Subcomponents can be refined as part of component implementation extensions.

Figure 3 provides an illustration of a containment hierarchy using the graphical AADL notation. In this example, Sys1 represents a system. The implementation of the system contains subcomponents named C3 and C4. Component C3, a subcomponent in Sys1's implementation, contains subcomponents named C1 and C2.

Component C4, another subcomponent in Sys1's implementation, contains a second set of subcomponents named C1 and C2. The two subcomponents named C1 and those named C2 do not violate the unique name requirement. They are unique with respect to the local namespace of their containing component's local namespace.

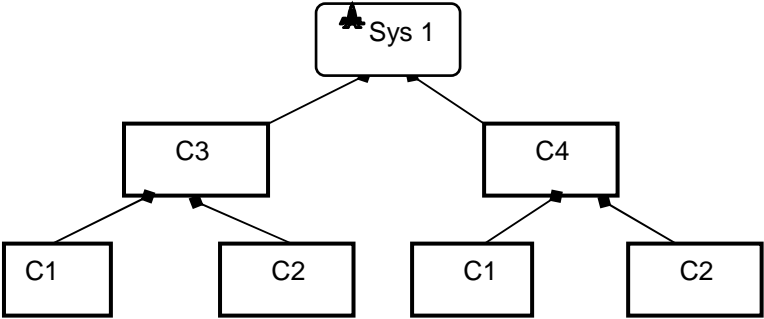


Fig. 3. Graphical representation of an AADL component containment hierarchy

2.2.5 AADL Interfaces and Connections

Ports represent the flow of data and events between threads and execution platform components. *Data ports* are used for the transmission of unqueued state data. *Event data ports* are used for queued message data associated with an event, such as data transferred as part of a system interrupt. *Event ports* are used for events.

Connections represent the actual linkage between components. *Immediate connections* represent data and events that are transmitted in the middle of a frame of execution. *Delayed connections* represent data and events that are transmitted at the deadline of the originating thread. Connections are one of three forms of interface interaction supported by AADL. The other two are *synchronous subprogram call* and *shared data access*.

The graphical representation of ports and connections is given in Figure 4.

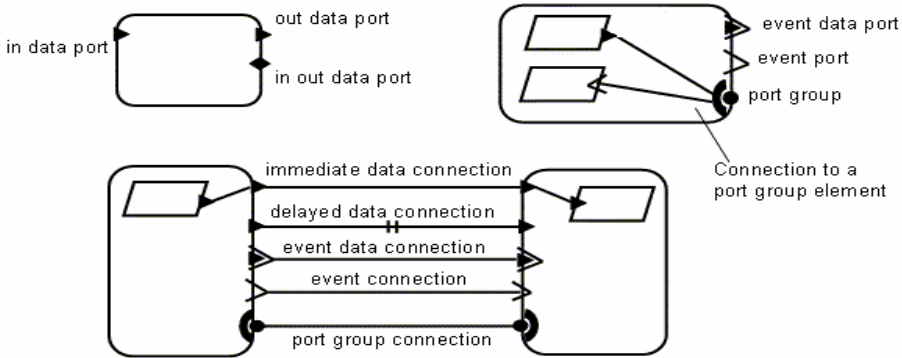


Fig. 4. Graphical representation of AADL ports and connections

The data that is transferred between components may be typed and characterized by additional properties such as unit of measurement, range constraints on the data values and constraints on the values of successive stream elements.

2.3 AADL Scheduling

AADL provides a precise specification of execution characteristics of threads and processes. At the same time it does not prescribe a particular scheduling protocol. AADL properties may be used to define the details of the scheduling policy and thread dispatching. In addition, AADL threads may have properties that describe their availability and priority for scheduling. Figure 5 shows the graphical representation of the four types of AADL threads each with annotations that capture their unique behaviors.

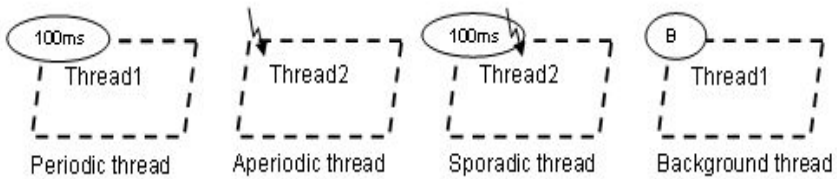


Fig. 5. Graphical representation of AADL threads

The AADL standard includes the specification of a hybrid automata that defines the various states of execution that are applicable to the thread scheduling and dispatching. In addition, since AADL supports the definition of modes of operation, the hybrid automaton includes states for mode transition and mode hibernation.

2.4 AADL Dependability, Faults and Modes

There are often requirements in embedded real-time systems for high dependability, fault-tolerance, and error recovery [6]. *Dependability* is the ability of a system to continue to produce the desired service to the user when the system is exposed to undesirable conditions. [7]. A *fault* is an anomalous undesired change in thread execution behavior, possibly resulting from an anomalous undesired change in data being accessed by that thread or from violation of a compute time or deadline constraint [8].

There is a fault framework described within the AADL standard that enables the user to describe what happens when a fault occurs. In addition, the Error Annex [9] provides support for error models and analysis. The AADL standard also supports the specification of mode transition actions.

2.4.1 AADL Specification of a Modal System

One method utilized to improve the dependability of a system is through the replication of hardware components, software components, or both. The example given Figures 6 and 7 demonstrates how AADL may be used to specify a system that is comprised of a primary and backup mode of operation.

```

system sys
features
    insignal: data port;
    outsignal: data port;
end sys;

system PrimaryBackupPattern
features
    insignal: data port;
    outsignal: data port;
    fault:    in out event port ;
    restart:  in out event port ;
    reinit:   in out event port ;
end PrimaryBackupPattern;

```

Fig. 6. Textural representation of the AADL specification of the system type Primary BackupPattern for a system whose external interface is comprised of two data ports: insignal and outsignal; and three event ports: fault, restart, and reinit

The implementation of the PrimaryBackupPattern system is provided in Figure 7. This implementation refines the type definition with the specification of two subsystems: Primary and Backup. The implementation also specifies each of the data port connections. It also defines three modes of operation: Primarymode, Backupmode, and Reinitmode. The implementation indicates that this implementation of the system will start up in Primarymode. It will transition from Backupmode to Reinitmode as the result of a restart event. And will transition from Reinitmode to Primarymode as the result of a reinit event.

```

system implementation PrimaryBackupPattern.impl
subcomponents
    Primary: system sys;
    Backup:  system sys;
connections
    inPrimary: data port insignal -> Primary.insignal;
    inBackup:  data port insignal -> Backup.insignal;
    outPrimary: data port Primary.outsignal -> outsignal
                in modes (Primarymode);
    outBackup: data port Backup.outsignal -> outsignal
                in modes (Backupmode);
modes
    Primarymode: initial mode;
    Backupmode:  mode;
    Reinitmode:  mode;
    Backupmode -[restart]-> Reinitmode;
    Reinitmode -[Reinit.Complete]-> Primarymode;
end PrimaryBackupPattern.impl;

```

Fig. 7. Textural representation of an AADL implementation of the system PrimaryBackup Pattern

Note that the specification of the connections to the `outsignal` data port include the specification of the mode to which the connection applies, thereby clearly identifying what event has trigger the event that is supplying the output signal data.

The graphical representation of the system `PrimaryBackupPattern` is given in Figure 8. Notice that the modes are encapsulated in the system components and the backup modes and corresponding system are grey.

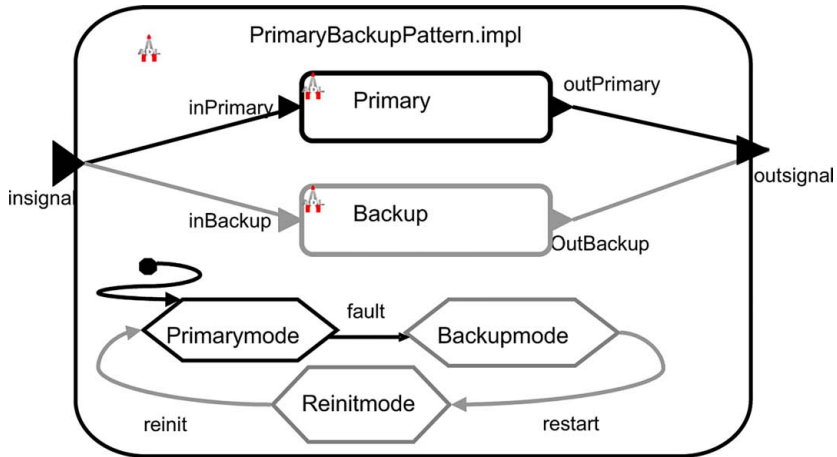


Fig. 8. Graphical representation of an AADL implementation of the system `PrimaryBackupPattern`

This example demonstrates many of the features of AADL as well as the textual representation of the model. Architectures specified in AADL may use the graphical notation or the textual notation.

2.5 AADL Annexes and Extensibility

Extensions to accommodate new analyses and unique hardware attributes take the form of new properties and analysis specific notations that can be associated with components may be defined as annex components. Users or tool vendors may define extension sets to facilitate additional capabilities. Extension sets may be proposed for inclusion in this standard. Such extensions will be defined as part of a new Annex appended to the standard.

Presently, there are four standard annexes: the Graphical AADL Notation Annex defines a set of graphical symbols for the graphical AADL notation. These graphical symbols can be used to express relationships between components, features, and connections in an AADL model. The AADL Meta Model and Interchange Formats, defines the AADL meta model and XML-based interchange formats for AADL models. The Language Compliance and Application Program Interface Annex defines language-specific rules for source text to be compliant with an architecture specification written in AADL. And the Error Model Annex defines features to enable the specification of redundancy management and risk mitigation methods in an

architecture, and enable qualitative and quantitative assessments of system properties such as safety, reliability, integrity, availability, and maintainability.

Use of the features of the Error Model Annex in the evaluation of dependency and reliability of components is defined further in the next chapter.

The standardization of additional annexes is under development including an annex that defines a UML profile for AADL and a Behavior annex that supports detailed component behavior modeling.

3 The SAE AADL Development Environment

The success of any new technology is dependent upon the availability of tools that support the use of the technology. As such, the Software Engineering Institute (SEI) in conjunction with the US Army and several other universities has developed an open source tool kit for AADL called the Open Source AADL Tool Environment (OSATE).

OSATE has been built as a set of plug-ins to the Eclipse environment and is itself extensible. OSATE includes an AADL parser that translates textual AADL specifications into in-core declarative AADL models. Those declarative AADL models get persistently stored in XML according to the AADL Meta model specification. OSATE also includes a semantic checker, various architecture analysis plug-ins, an AADL XML to text translator, an AADL object model editor, and an AADL graphical editing front-end.

4 Summary and Conclusions

The core AADL supports modeling of application systems and execution platforms as interacting components with specific semantics and bindings. Such systems are configurable in that components have multiple implementations. Semantics defined as part of the component categories and their predefined properties address timing and resource consumption as well as interaction consistency in terms of matching port types and data communicated through the ports. Behavior descriptions allow for model checking of behaviors as well as mode-specific analyses with less conservative results. The core language does not provide properties and semantics for all possible architecture analyses. Instead the AADL has been made extensible both in terms of language notation and in terms of standard annexes to accommodate further analyses.

Model-based, architecture driven software system engineering is critical to predictably developing and maintaining large-scale systems. Architecture analysis early and throughout the life cycle improves predictability of non-functional properties of mission-critical systems.

The SAE AADL, as an industry standard, provides a stable common framework for contractors to cooperatively evolve large-scale systems and for tool vendors to provide tools for a common architecture representation.

References

1. Society of Automotive Engineers (SAE) Avionics Systems Division (ASD) AS-2C Subcommittee. Avionics Architecture Description Language Standard, AS5506, vol. 1.0 (November 2004)
2. Clements, Paul, et al.: Documenting Software Architectures: Views and Beyond. SEI Series in Software Engineering. Addison-Wesley, Reading (2002)
3. IEEE Std 1471-2000 IEEE Recommended Practice for Architectural Description of Software-Intensive Systems –Description (2000)
4. Binns, Pamela, Englehart, M., Jackson, M., Vestal, S.: Domain Specific Software Architectures for Guidance, Navigation and Control, Honeywell Technology Center, Minneapolis, MN. International Journal of Software Engineering and Knowledge Engineering 6(2), 201–227 (1996)
5. Garlan, David, Monroe, R.T., Wile, D.: Acme: Architectural Description of Component-Based Systems. In: Foundations of Component Based Systems, Cambridge University Press, Cambridge (2000)
6. Feiler, Peter, H., Gluch, D.P., Hudak, J.H., Lewis, B.A.: Embedded System Architecture Using SAE AADL. Technical Note CMU/SEI-2004-TN-005 (June 2004)
7. LaPrie, J.-C.: Dependable Computing and Fault Tolerance: Concepts and Terminology. In: Proceedings of the 15th International Symposium on Fault-Tolerant Computing (FTCS-15), Ann Arbor, MI, pp. 2–11 (1985)
8. IFIP WG10.4 on Dependable Computing and Fault Tolerance. In: Laprie, J.-C. (ed.) Dependability: Basic Concepts and Terminology, Dependable Computing and Fault Tolerance, vol. 5, Springer, Wien, New York (1992)
9. Society of Automotive Engineers (SAE) Avionics Systems Division (ASD) AS-2C Subcommittee. SAE Architecture Analysis and Design Language (AADL) Annex vol. 1: Annex A: Graphical AADL Notation, Annex C: AADL Meta-Model and Interchange Formats, Annex D: Language Compliance and Application Program Interface Annex E: Error Model Annex, AS5506/1, vol. 1.0 (June 2006)

A System Dependability Modeling Framework Using AADL and GSPNs

Ana-Elena Rugina, Karama Kanoun, and Mohamed Kaâniche

LAAS-CNRS, University of Toulouse

7 avenue Colonel Roche

31077 Toulouse Cedex 4, France

Phone: +33(0)5 61 33 62 00, Fax: +33(0)5 61 33 64 11

{rugin,a,kanoun,kaaniche}@laas.fr

Abstract. For efficiency and cost control reasons, system designers' will is to use an integrated set of methods and tools to describe specifications and design, and also to perform dependability analyses. The SAE (Society of Automotive Engineers) AADL (Architecture Analysis and Design Language) has proved to be efficient for architectural modeling. We present a modeling framework allowing the generation of dependability-oriented analytical models from AADL models, to facilitate the evaluation of dependability measures, such as reliability or availability. We propose a stepwise approach for system dependability modeling using AADL. The AADL dependability model is transformed into a GSPN (Generalized Stochastic Petri Net) by applying model transformation rules that can be automated. The resulting GSPN can be processed by existing tools. The modeling approach is illustrated on a subsystem of the French Air Traffic Control System.

Keywords: dependability modeling, evaluation, AADL, GSPN, model transformation.

1 Introduction

The increasing complexity of new-generation systems raises major concerns in various critical application domains, in particular with respect to the validation and analysis of performance, timing and dependability-related requirements. Model-driven engineering approaches based on architecture description languages aimed at mastering this complexity at the design level have emerged and are being increasingly used in industry. In particular, AADL (Architecture Analysis and Design Language) [1] has received a growing interest during the last years. It has been recently developed and standardized under the auspices of the International Society of Automotive Engineers (SAE), to support the design and analysis of complex real-time safety-critical systems in avionics, automotive, space and other application domains. AADL provides a standardized textual and graphical notation for describing software and hardware system architectures and their functional interfaces. AADL may be used to perform various types of analysis to determine the behavior and the performance of

the system being modeled. The language has been designed to be extensible to accommodate analyses that the core language does not support.

Besides describing the systems' behavior in the presence of faults, the developers are interested in obtaining quantitative measures of relevant dependability properties such as reliability, availability and safety. For pragmatic reasons, the system designers using an AADL-based engineering approach are interested in having an integrated set of methods and tools to describe specifications and design, and to perform dependability evaluations. The *AADL Error Model Annex* [2] has been recently standardized to complement the description capabilities of the core language by providing features with precise semantics to be used for describing dependability-related characteristics in AADL models (faults, failure modes, repair policies, error propagations, etc.). However, at the current stage, no methodology and guidelines are available to help the developers in the use of the proposed notations to describe complex dependability models reflecting real-life systems with multiple interactions and dependencies between components. One of our objectives is to propose a structured method for AADL dependability model construction.

The AADL Error Model Annex mentions that stochastic automata such as fault trees and Markov chains can be generated from AADL specifications enriched with dependability-related information. Indeed, Markov chains are recognized to be powerful means for modeling system dependability taking into account dependencies between system components. Usually, they are automatically generated from higher level formalisms such as Generalized Stochastic Petri Nets (GSPNs). The latter allow structural model verification, before the Markov chain generation. Such verification support facilities are very useful when dealing with large models.

During the last decade, various approaches have been defined to support the systematic construction and validation of dependability models based on GSPNs and their extensions (see e.g. [3-5]). We propose to take advantage of such approaches in the context of an AADL-based engineering process, to i) build the dependability-oriented AADL model and to ii) generate dependability-oriented GSPN models from AADL models by model transformation. In this way, the complexity of GSPN model generation is hidden to users familiar with AADL but who have a limited knowledge of GSPNs. The AADL and GSPN models are built iteratively, taking into account progressively the dependencies between the components, and validated at each iteration. The dependability-related information is not embedded in the AADL architectural model. Instead, it is described separately and then plugged in the system's components. The user can easily unplug or replace the dependability-related information. This feature enhances the reusability and the readability of the AADL architectural model that can be used as is for other analyses (e.g., formal verification [6], scheduling and memory requirements [7], resource allocation with the Open Source AADL Tool Environment (OSATE)¹, research of deadlocks and un-initialized variables with the Ocarina toolset²).

To summarize, our objectives are threefold: i) present a structured and stepwise approach for building AADL dependability model, ii) show examples of model transformation rules to generate GSPNs from AADL dependability models and iii)

¹ <http://www.aadl.info/OpenSourceAADLToolEnvironment.html>

² <http://ocarina.enst.fr>

exemplify the proposed approach on a subsystem of the French Air Traffic Control System. The set of model transformation rules is meant to be the basis for the implementation of a model transformation tool completely transparent to the user. Such a tool can be interfaced with one of the existing GSPN processing tools (e.g., Surf-2 [8], Möbius [9], Sharpe [10], GreatSPN [11], SPNP [12]) to evaluate dependability/performability measures.

Compared to our work presented published in [13] and [14], we offer here a global view of our method's steps, by presenting a case study reflecting a real system. In particular, the AADL to GSPN transformation rules are developed and illustrated.

The remainder of the paper is organized as follows. Section 2 discusses related work. Section 3 presents the AADL concepts that are necessary for understanding our modeling approach. Section 4 gives an overview of our framework for system dependability modeling and evaluation using AADL and GSPNs. Section 5 presents examples of rules for transforming AADL into GSPN models. Section 6 applies our approach to a subsystem of the French Air Traffic Control System and Section 7 concludes the paper.

2 Background and Related Work

To the best of our knowledge there are no contributions similar to our work in the current state of the art. Most of the published work on analyses using AADL has focused on the extension of the language capabilities to support formal verifications. For example, the COTRE project [6] provides a design approach bridging the gap between formal verification techniques and requirements expressed in Architecture Description Languages. AADL system specifications can be imported in the newly defined COTRE language. A system specification in COTRE language can be transformed into timed automata, Time Petri nets or other analytical models. Also, a transformation from AADL models to Colored Petri Nets, aiming at formally verifying certain properties through model checking, is presented in [15]. However, as far as we are aware of, published work does not address generation of dependability-oriented quantitative evaluation models from AADL specifications.

Considering the problem of generating dependability evaluation models from model-driven engineering approaches in a more general context, a significant amount of research has been carried out based on UML (Unified Modeling Language) [16]. For example, the European project HIDE ([17], [18]) proposed a method to automatically analyze and evaluate dependability attributes from UML models. It defined several model transformations from subsets of UML diagrams to i) GSPNs, Deterministic and Stochastic Petri Nets and Stochastic Reward Nets to evaluate dependability measures, ii) Kripke structures for formal verification and iii) to Stochastic Reward Nets for performance analysis. Also, [19] proposes an algorithm to synthesize dynamic fault trees (DFT) from UML system models. Other interesting approaches have been developed, aiming at obtaining performance measures by transforming UML diagrams (activity diagrams in [20], sequence and statechart diagrams in [21]) into GSPNs.

Similarly to UML users, the AADL users are interested in using modeling approaches allowing them to derive dependability evaluation models from AADL specifications. The approach proposed here aims at fulfilling this objective.

3 AADL Concepts

In the AADL, systems are particular composite components modeled as hierarchical collections of interacting application components (processes, threads, subprograms, data) and a set of execution platform components (processors, memory, buses, devices). The application components are bound to the execution platform. The AADL allows analyzing the impact of different architecture choices (such as scheduling policy or redundancy scheme) on a system's properties [22].

Each AADL system component has two levels of description: the *component type* and the *component implementation*. The type describes how the environment sees that component, i.e., its properties and features. Examples of features are *in* and *out* ports that represent access points to the component. One or more component implementations may be associated with the same component type, corresponding to different implementation structures of the component in terms of subcomponents, connections (between subcomponents' ports) and operational modes.

Dynamic aspects of system architectures are captured with the AADL operational mode concept. Different operational modes of a system or a system component represent different system configurations and connection topologies, as well as different sets of property values to represent changes in non-functional characteristics such as performance and reliability. Mode transitions model dynamic operational behavior and are triggered by events arriving through ports. Operational modes may represent fault-tolerance modes or different phases in a phased-mission system. This dynamics may influence dependability measures (i.e., availability), thus operational modes are taken into account in the dependability model.

An *AADL architectural model* can be annotated with dependability-related information (such as faults, failure modes, repair policies, error propagation, etc.) through the standardized Error Model Annex. *AADL error models* are defined in libraries and can be associated with application components, execution platform components, and device components, as well as the connections between them. When an error model is associated with a component, it is possible to customize it by setting component-specific values for the arrival rate or the probability of occurrence for error events and error propagations declared in the error model.

In the same way as for AADL components, error models have two levels of description: the *error model type* and the *error model implementation*. The error model type declares a set of error states, error events (internal to the component) and error propagations³. Occurrence properties specify the arrival rate or the occurrence probability of events and propagations. The error model

³ We will refer to error states, error events, error propagations and error transitions without the qualifying term *error* in contexts where the meaning is unambiguous (note that error states can model error-free states, error events can model repair events and error propagations can model all kinds of notifications).

implementation declares error transitions between states, triggered by events and propagations declared in the error model type.

Figure 1 shows a simple error model, without propagations, considering two types of faults: temporary and permanent. A temporary fault leads the component in an erroneous state while a permanent fault leads it in a failed state. A temporary fault can be processed and the component recovers regaining its error free state. A permanent fault requires restarting the component.

Error Model Type [independent]
<pre> error model independent features Error_Free: initial error state; Erroneous: error state; Failed: error state; Temp_Fault: error event {Occurrence => poisson λ_1}; Perm_Fault: error event {Occurrence => poisson λ_2}; Restart: error event {Occurrence => poisson μ_1}; Recover: error event {Occurrence => poisson μ_2}; end independent; </pre>
Error Model Implementation [independent.general]
<pre> error model implementation independent.general transitions Error_Free-[Perm_Fault]->Failed; Error_Free-[Temp_Fault]->Erroneous; Failed-[Restart]->Error_Free; Erroneous-[Recover]->Error_Free; end independent.general; </pre>

Fig. 1. Error model example without propagations

Interactions between the error models of different components are determined by interactions between components of the architectural model through connections and bindings. Out propagations are sent out of a component through all features connecting it to other components. Thus, out propagations have an impact on any receiving component that declares an in propagation with the same name. In some cases, it is desirable to model how error propagations from multiple sources are handled. This is modeled by further customizing an error model to a system component by specifying filters and masking conditions for propagations by using Guard properties associated with its features.

AADL allows modeling logical error states independently from the operational modes of a component. It also allows establishing a connection between the logical error states and the operational modes. For example, operational mode transitions may be constrained, through the use of Guard_Transition properties applied to ports, to occur depending on the error state configuration of several components.

Several examples are given throughout the paper to illustrate how propagations and Guard_Transition properties are handled in AADL.

4 The Modeling Framework

For complex systems, the main difficulty for dependability model construction arises from dependencies between the system components. Dependencies are of several types, identified in [4]: structural, functional, those related to the fault-tolerance and those associated with recovery and maintenance policies. Exchange of data or transfer of intermediate results from one component to another is an example of functional dependency. The fact that a thread runs on a processor induces a structural dependency between them. Changing the operational mode of a component according to a fault tolerance policy (e.g., leader/follower) represents a fault tolerance dependency. Sharing a maintenance facility between several execution platform components leads to a maintenance dependency. Having to follow a strict recovery order for application components is an example of recovery dependency. Functional, structural and fault tolerance dependencies are grouped into an architecture-based dependency class, as they are triggered by physical or logical connections between the dependent components at architectural level. On the other hand, recovery and maintenance dependencies are not always visible at architectural level.

A structured approach is necessary to model dependencies in a systematic way, to avoid errors in the resulting model of the system and to facilitate its validation. In our approach, the AADL dependability-oriented model is built in an iterative way. More concretely, in the first iteration, we build the model of the system's components, representing their behavior in the presence of their own faults and repair events only. They are thus modeled as if they were isolated from their environment. In the following iterations, we introduce dependencies in an incremental manner.

The rest of this section is structured as follows. A general overview of our modeling framework is presented in subsection 4.1. In subsection 4.2, we illustrate how dependencies are modeled in AADL in the context of our approach. Subsection 4.3 presents briefly how a GSPN model is generated from the AADL model.

4.1 Overview

An overview of our iterative modeling framework, which is decomposed in four main steps, is presented in Figure 2.

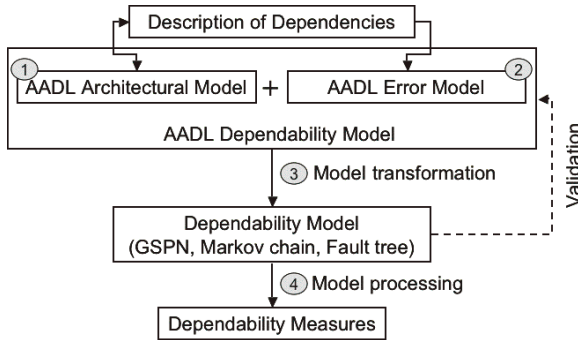


Fig. 2. Modeling framework

The *first step* is devoted to the modeling of the system architecture in AADL (in terms of components and operational modes of these components). This AADL architectural model may be available if it has been already built for other purposes.

The *second step* concerns the building of the AADL error models associated with components of the architectural model. The error model of the system is a composition of the set of components' error models, taking into account the dependencies between these components.

The description of architecture-based dependencies between components of the system is based on the analysis of the connections and bindings present in the architectural model. The corresponding error model is built based on the description of dependencies. Making maintenance and recovery assumptions may lead to the addition of components in the architectural model.

The architectural model and the error model of the system form a dependability-oriented AADL model, referred to as the *AADL dependability model* further on.

The *third step* aims at building a dependability evaluation model, from the AADL dependability model, based on model transformation rules. Here, we focus on generating a GSPN from the AADL model.

The *fourth step* is devoted to the processing of the dependability evaluation model (in our case under the form of a GSPN) to evaluate quantitative measures characterizing dependability attributes. This step is entirely based on existing GSPN processing algorithms and tools. Therefore, it is not considered here.

To obtain the AADL dependability model, the user must perform the first and second steps described above. The third step is intended to be automatic in order to hide the complexity of the GSPN to the user.

The iterative approach can be applied to the first two steps only or to the first three steps together. In both cases, the AADL dependability model is updated at each iteration. Modeling a dependency may either require to only add information in the model or to modify the existing model and to add new information (i.e., states and propagations). In the latter case, the AADL dependability model can be validated against its specification, based on the analysis and validation of the GSPN model, after each iteration.

To evaluate dependability measures, the user must specify state classes for the overall system. For example, if the user wishes to evaluate reliability or availability, it is necessary to specify the system states that are to be considered as failed states. If in addition, the user wishes to evaluate safety, it is necessary to specify the failed system states that are considered as catastrophic. In AADL, state classes are declared by means of a *derived error model* for the overall system describing the states of a system as Boolean expressions referring to its subcomponents' states.

4.2 Modeling with Dependencies in AADL

Architecture-based dependencies can be derived from the AADL architectural model. To these dependencies one has to add recovery and maintenance dependencies. The full set of dependencies can be summarized in a *dependency block diagram* to provide a global view of the system components and interactions. In the dependency block diagram, each component and each dependency are represented as distinct blocks. Blocks are connected through arcs. Their directions identify the directions of

dependencies. This diagram and the AADL architectural model are used to build the AADL error model progressively. Once the AADL error models of the components are built, the dependencies are added gradually. The order for introducing dependencies does not impact the final AADL dependability model. However, it may impact the reusability of parts of the model. Thus, the order may be chosen according to the context of the targeted analysis. Generally, fault tolerance and maintenance dependencies are modeled at the end, as their description strongly depends on the architecture.

It is noteworthy that not all the details of the architectural model are necessary for the AADL dependability model. Only components that have associated error models and all connections and bindings between them are necessary.

The rest of this subsection presents guidelines for modeling i) an architecture-based dependency and ii) a maintenance or recovery dependency.

4.2.1 Architecture-Based Dependency Modeling

The architecture-based dependency is supported by the architectural model and must be modeled in the error models associated with dependent components, by specifying respectively outgoing and incoming propagations and their impact on the corresponding error model. An example is shown in Figure 3. Figure 3-a presents the AADL architectural model (*Component 1* sends data to *Component 2*). Figure 3-b shows the corresponding dependency block diagram (the behavior of *Component 2* depends on that of *Component 1*). Figure 3-c presents the AADL dependability model where an error model is associated with each component to describe the dependency.

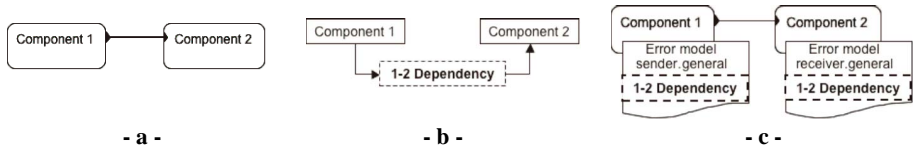


Fig. 3. Architecture-based dependency

The error model of Figure 4 is associated with *Component 1*. It takes into account the sender-side dependency from *Component 1* to *Component 2*. This error model is an extension of the one of Figure 1 that represents the behavior of a component as if it were isolated. The error model of Figure 4 declares an out propagation *Error* (see line d1) in the type and an AADL transition triggered by the out propagation in the implementation (see line d2).

The error model *receiver.general* associated with *Component 2* is not shown here but is similar. The only difference is the direction of the propagation *Error*. This in propagation triggers a state transition from *Error_Free* to *Failed*.

When *Component 1* is in the erroneous state, it sends a propagation through the unidirectional connection. As a consequence, the incoming propagation *Error* causes the failure of the receiving component *Component 2*. The in – out propagations *Error* defined respectively in the error model instance associated with *Component 2* and with *Component 1* have identical names. In the rest of the paper, such propagations are referred to as *name matching propagations*.

Error Model Type [sender]	
(d1)	<pre> error model sender features Error_Free: initial error state; Erroneous: error state; Failed: error state; Temp_Fault: error event {Occurrence => poisson λ_1}; Perm_Fault: error event {Occurrence => poisson λ_2}; Restart: error event {Occurrence => poisson μ_1}; Recover: error event {Occurrence => poisson μ_2}; Error: out error propagation {Occurrence => fixed p}; end sender; </pre>
Error Model Implementation [sender.general]	
(d2)	<pre> error model implementation sender.general transitions Error_Free-[Perm_Fault]->Failed; Error_Free-[Temp_Fault]->Erroneous; Failed-[Restart]->Error_Free; Erroneous-[Recover]->Error_Free; Erroneous-[out Error]->Erroneous; end sender.general; </pre>

Fig. 4. Error model example with dependency

In real applications, architecture-based dependencies usually require describing how error propagations from multiple sources are handled by the receiver component. This is achieved by using Guard properties in which Boolean expressions are used to specify the consequences of a set of propagations occurring in a set of sender components on a receiver component.

4.2.2 Maintenance and Recovery Dependency Modeling

Maintenance dependencies need to be described when repair facilities are shared between components or when the maintenance or repair activity of some components has to be carried out according to a given order or a specified strategy.

Components that are not dependent at architectural level may become dependent due to the fact that they share maintenance facilities or to the synchronization of the maintenance activities. Thus, the architectural model might need some adjustments to support the description of dependencies related to the maintenance policy. As error models interact only via propagations through architectural features (i.e., connections, bindings), the maintenance dependency between components' error models must also be supported by the architectural model. This means that besides the system architecture components, we may need to add a component representing the shared repair facilities to model the maintenance dependencies. Figure 5-a shows an architectural model example where *Component 3* and *Component 4* do not interact (there is no architecture-based dependency between them). However, if we assume that they share one repairman, it is necessary to represent the repairman at the level of the architectural model, as shown in Figure 5-b in order to model the maintenance dependency between these components.

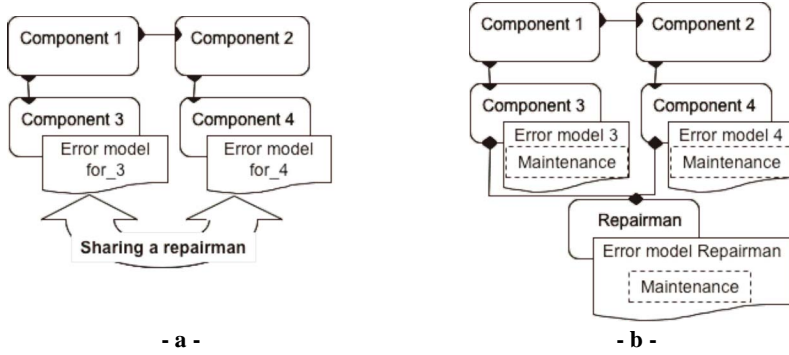


Fig. 5. Maintenance dependency

Also, the error models of dependent components with regards to their recovery might need some adjustments. For example, to represent the fact that *Component 3* can only restart if *Component 4* is running, one needs to distinguish between a failed state of *Component 3* and a failed state where *Component 3* is allowed to restart.

4.3 AADL to GSPN Model Transformation

The GSPN model of the system is built from the transformation of the AADL dependability model following a modular approach and taking into account the dependency block diagram.

The GSPN of the global system is structured as a set of interacting subnets, where a subnet is associated with a component or a dependency block identified in the dependency block diagram. Two types of GSPN subnets are distinguished: 1) a *component subnet* is associated with each component and describes the component's behavior in the presence of its own faults and repair events; and 2) a *dependency subnet* models the behavior associated with the corresponding dependency. In the AADL dependability model, each dependency is modeled as part of each of the error models involved in the dependency. GSPN dependency subnets are obtained from information concerning a particular dependency existing in (at least) two dependent error models. The global GSPN contains one subnet for the behavior of each component in the presence of its own faults and repair events, and one subnet for each dependency between components. It has the same structure as the dependency block diagram. The modular structure of the GSPN allows the user to validate the model progressively; as the GSPN is enriched with a subnet each time a new dependency is added in the error model of the system. So, if validation problems arise at GSPN level during iteration i , only the part of the current error model corresponding to iteration i is questioned.

5 Transformation Rules

In the next three subsections we present successively AADL to GSPN transformation rules for 1) isolated components, 2) name matching in - out propagations in







dependent components and 3) systems with operational modes necessary to describe fault tolerance dependencies. All transformation rules are defined to ensure that the obtained GSPN is correct by construction: bounded, live and reversible. They are aimed to be systematic in order to prepare the transformation automation. Also, the resulting GSPN is tool-independent, i.e., we do not use tool-specific features or predicates. It is worth noting that this section only presents a small set of transformation rules. A more complete set is presented in [23].

5.1 Isolated Components

In the case of an isolated component or in the case of a set of independent components, the AADL to GSPN transformation is rather straightforward, as an error model represents a stochastic automaton. The number of tokens in a component subnet is always one, as a component can only be in one state.

Table 1 shows the basic transformation rules.

Table 1. Basic AADL error model to GSPN transformation rules

AADL error model element	GSPN element		
State	Place		
Initial state	Token in the corresponding place		
Event	GSPN transition (timed or immediate)		
Occurrence property of an event	Distribution or probability characterizing the occurrence of associated GSPN transition		Timed
			Immediate
AADL transition (Source_State-[Event] -> Destination_State)	Arcs connecting places (corresponding to AADL Source_State and Destination_State) via GSPN transition (corresponding to AADL Event)		
			

By applying the transformation rules presented in Table 1 to the error model shown in Figure 1, we obtain the GSPN of Figure 6.

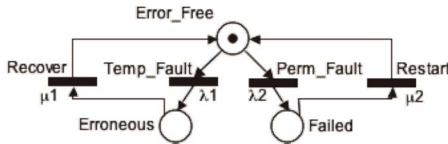


Fig. 6. GSPN corresponding to the error model of Figure 1

5.2 Transforming in - out Name Matching Propagations

In the most general case, an out propagation declared in a propagation sender error model could trigger n AADL transitions in this same error model (e.g., a *Failed* propagation could be propagated out both from a *FailStopped* and a *FailRandom* states). Name matching in propagations could be declared in $r \geq 2$ propagation receiver error models and trigger m_j AADL transitions in each j ($j = 1 \dots r$) receiver error model. We identified and analyzed several transformation rules for the same AADL specification of in - out name matching propagations. Some of the rules are convenient when an out propagation has only one receiver. On the other hand, these rules are hard to automate in case there are several receivers (i.e., the in propagation is declared in several components' error models) for the same out propagation. Also, the choice of a transformation rule for in - out name matching propagations impacts the transformation rules for systems with operational modes. The transformation rule for in - out name matching propagations we present here is very well adapted for the case where an out propagation has several receivers. It also simplifies the definition of the transformation rule for systems with operational modes. We first present an example of a pair of in - out name matching propagations declared in two connected components in Figure 7. Then we illustrate the chosen transformation rule on this example.

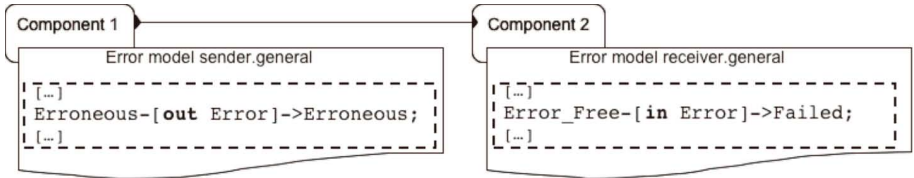


Fig. 7. Sender and Receiver – name matching propagations

In Figure 7, *Component 1* plays the role of the propagation sender and it sends propagations named *Error* through the connection that arrives at *Component 2*. *Component 2* plays the role of a receiver. If it receives a propagation named *Error*, it moves from *Error_Free* to *Failed* state.

The transformation rule consists in decoupling the in and out propagations in the GSPN through an intermediary place that represents the fact that the out propagation *Error* occurred, as shown in Figure 8 (*InOut_Error* place). A token arrives in the *InOut_Error* place when a GSPN transition (*Out_Error*) corresponding to the out propagation (and characterized by its Occurrence property) occurs. The existence of a token in the *InOut_Error* place leads to the firing of an immediate GSPN transition *In_Error* (if the place *Error_Free* in *Component 2* is marked) that corresponds to the in propagation. The intermediary place is emptied when the place corresponding to the source of the out propagation is empty and the GSPN transition corresponding to the in propagation is not enabled. We do not empty this place at the occurrence of the GSPN transition corresponding to the in propagation, as we need to memorize

the occurrence of the *out* propagation until all effects (immediate GSPN transitions) of the propagation occur. This memory is used in other transformation rules.

The GSPN place *NoPropag* and the associated immediate transitions with probability $(1-p)$ and 1 respectively model the situation where the propagation does not occur when *Component 1* is in an *Erroneous* state. If the probability of occurrence of the *out* propagation is equal to 1, then this subnet is not necessary.

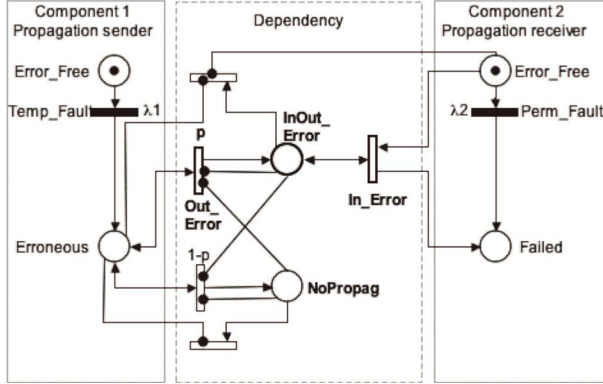


Fig. 8. Propagation from sender to receiver - transformation rule

In the general case of n AADL transitions triggered by an *out* propagation, with name matching *in* propagations in several receiver error models, one GSPN transition is created for each AADL transition triggered by the *out* propagation in the sender error model. Also, one intermediary place is created for each *out* propagation. One GSPN transition is created for each AADL transition triggered by the *in* propagation in the receiver error models. Consequently, the number of GSPN transitions (N_{tr}) describing the AADL propagation is given by:

$$N_{tr} = 4 * n + n * \sum_{j=1}^r m_j, \forall r \geq 1 \quad (1)$$

where n = the number of AADL transitions triggered by the *out* propagation in the sender error model;
 r = the number of receiver error models;
 m_j = the number of AADL transitions triggered by the *in* propagation in the receiver error model j .

The first term of equation (1) represents the number of GSPN transitions that model the *out* propagation, i.e., 4 GSPN transitions for each one of the n *out* propagations. The second term represents the number of GSPN transitions that model the *in* propagation, i.e., one GSPN transition for each pair of *in* - *out* propagations.

Figure 9-a shows an example of AADL dependability model with one sender and two receivers. It is transformed into the GSPN of Figure 9-b.

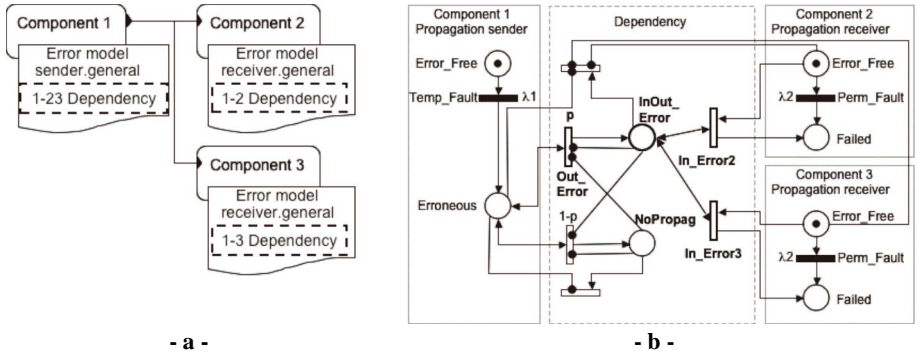


Fig. 9. Propagation from sender to two receivers

Naturally, when transforming large AADL models formed of many components, the size of the corresponding GSPN increases. The state space size depends on the number of components and on the dependencies between them. We have analyzed the state space for GSPNs obtained using our transformation rules from AADL models with several dependent components. Indeed, the more independent or loosely coupled components are, the larger the state space gets. To address this problem, GSPN reduction methods, such as those mentioned in [24], may be efficiently used before processing it to obtain the underlying Markov chain.

5.3 Systems with Operational Modes

In AADL, there are several mechanisms for connecting logical error states and operational mode transitions. For space limitation reasons, in this section, we focus on the AADL to GSPN transformation rules for *Guard_Transition* properties, which allow constraining a mode transition to occur depending on the error state configuration of several components of a system. The rest of this subsection presents successively the AADL modeling of an example of a system with operational modes, using *Guard_Transition* properties, and illustrates the proposed transformation rule on this example.

5.3.1 AADL Dependability Modeling of *Guard_Transition* Properties

We first present an example of a modal AADL system in Figure 10 and we show in Figure 11 the association of a *Guard_Transition* property with the ports involved in mode transitions. The error state configuration necessary to allow a mode transition is expressed as a Boolean expression referring to error states and propagations.

In Figure 10, the system is represented using the AADL graphical notation. It contains two identical active components and two operational modes (*Comp1Primary* and *Comp1Backup*). The system is initially in mode *Comp1Primary*. The transition from mode *Comp1Primary* to mode *Comp1Backup* occurs when propagations arrive through the ports *Send2* of *Comp1* and *Send1* of *Comp2*. In this case, the second

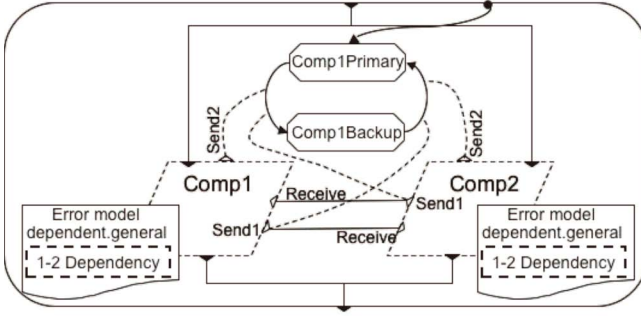


Fig. 10. Example architectural model for a system with operational modes

```

thread Comp
features
  Send1, Send2: out event port;
  Receive: in event port;
end Comp;

thread implementation Comp.generic
annex Error_Model {**
  Model => dependent.general;
**};
end Comp.generic;

system SystemLevelModes
end SystemLevelModes;

system implementation SystemLevelModes.generic
modes
  Comp1Primary: initial mode;
  Comp1Backup: mode;
  Comp1Primary-[Comp1.Send2, Comp2.Send1]->Comp1Backup;
  Comp1Backup-[Comp1.Send1, Comp2.Send2]->Comp1Primary;
subcomponents
  Comp1: system Comp.generic;
  Comp2: system Comp.generic;
connections
  event port Comp1.Send1->Comp2.Receive;
  event port Comp2.Send1->Comp1.Receive;
annex Error_Model {**
(g1)    Guard_Transition =>
(g2)    (Comp1.Send2[FailedVisible] and Comp2.Send1[Error_Free])
(g3)    applies to Comp1.Send2, Comp2.Send1;
(g4)    Guard_Transition =>
(g5)    Comp2.Send2[FailedVisible] and Comp1.Send1[Error_Free])
(g6)    applies to Comp1.Send1, Comp2.Send2;
**};
end SystemLevelModes.generic;

```

Fig. 11. Guard_Transition property associations

component must take over and provide the service. The transition from mode *Comp1Backup* to mode *Comp1Primary* occurs when propagations arrive through the ports *Send2* of *Comp2* and *Send1* of *Comp1*. The same error model is associated with both *Comp1* and *Comp2*. It is based on the error model for isolated components (see Figure 1). It declares in addition an out propagation *FailedVisible*, which notifies the failure of the component and which is used in the *Guard_Transition* properties.

Guard_Transition properties are associated with the ports involved in mode transitions. A mode transition occurs only if the *Guard_Transition* property associated with the port named in it evaluates to TRUE. In our example, the mode transition from mode *Comp1Primary* to *Comp1Backup* occurs when *Comp1* sends the *FailedVisible* out propagation while *Comp2* is *Error_Free* (see lines g1-g3 of Figure 11). The complementary condition must hold for the occurrence of the transition from *Comp1Backup* to *Comp1Primary* (see lines g4-g6 of Figure 11).

5.3.2 Transforming Guard_Transition Properties

We illustrate the transformation rule on the example of a system with operational modes described in Figure 10 and Figure 11. Then, we discuss the use of this rule.

Figure 12 shows the GSPN corresponding to the first *Guard_Transition* property (lines g1-g3) of Figure 11. The GSPN models of *Comp1* and *Comp2* are incomplete in this figure.

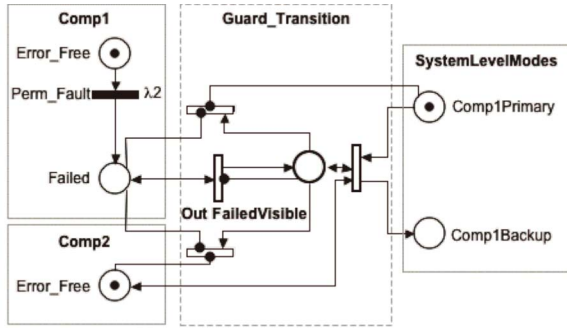


Fig. 12. GSPN modeling of the *Guard_Transition* property

Operational modes are directly mapped to Petri net places.

The transformation rule assumes that the Boolean expression of the *Guard_Transition* property is in disjunctive normal form. If it is not the case, the Boolean expression must first be transformed into disjunctive normal form. Each conjunction (referring to states and/or propagations) is transformed into an immediate GSPN transition connected with:

- places corresponding to the states and out propagations referred to in the AND expression via bi-directional arcs or inhibitor arcs (depending whether there are negations in the Boolean expression or not).
- places corresponding to operational modes referred to in the mode transition triggered by the port having the *Guard_Transition* property.

If, in our example above, the Boolean expression in disjunctive normal form were formed of several conjunctions, then several GSPN transitions would be connected to the places *Comp1Primary* and *Comp1Backup*.

An intermediary place corresponding to an out propagation is emptied when the place corresponding to the source of the out propagation is empty and the GSPN transitions corresponding to *Guard_Transition* conjunctions are not enabled.

If an out propagation name-matches an in propagation in a receiver component and is referred to in a *Guard_Transition* property declared in another receiver component, the same intermediary place is used both for the name matching GSPN subnet and for the *Guard_Transition* subnet. The intermediary place is emptied when both emptying conditions related to the name-matching propagations rule and to the *Guard_Transition* rule are true. An example is shown in Figure 13. We consider the system presented in Figure 9 contains a third component, *Comp3*, and that *Comp1* is connected to it. The error model associated with the newly introduced *Comp3* declares an in propagation *FailedVisible*. The *Guard_Transition* is transformed as above and the name matching propagation of *Comp1* and *Comp3* reuses the intermediary place representing the occurrence of the *FailedVisible* propagation.

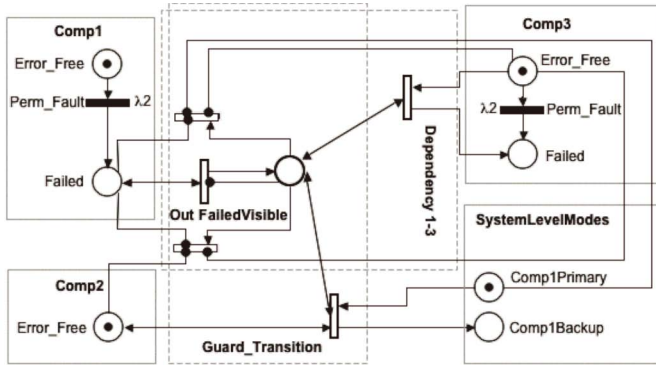


Fig. 13. GSPN modeling of the *Guard_Transition* property taking into account a name matching propagation

6 Case Study

In this section we show how our modeling framework can be used to compare the availability of two candidate architectures for a subsystem of the French Air Traffic Control System. This subsystem is designed to achieve high levels of service availability and is further detailed in [25]⁴.

In the rest of this section, we first present the AADL architectural models of the two candidate architectures in subsection 6.1. The dependency analysis based on these models is presented in subsection 6.2. Subsection 6.3 details the error models

⁴ The AADL modeling of the subsystem and the AADL to GSPN model transformation are not presented in the paper cited here.

describing some of these dependencies. Subsection 6.4 deals with the AADL to GSPN transformation while subsection 6.5 presents an example of dependability evaluation for the two candidate architectures.

6.1 AADL Architectural Models

The subsystem we consider here is formed of two fault-tolerant distributed software units that are in charge of processing flight plans (FPunit) and radar data (RDunit). Two processors can host these units. We consider two candidate architectures for this subsystem, referred to as *Configuration1* and *Configuration2*. Figure 14 presents both candidate architectures using the AADL graphical notation.

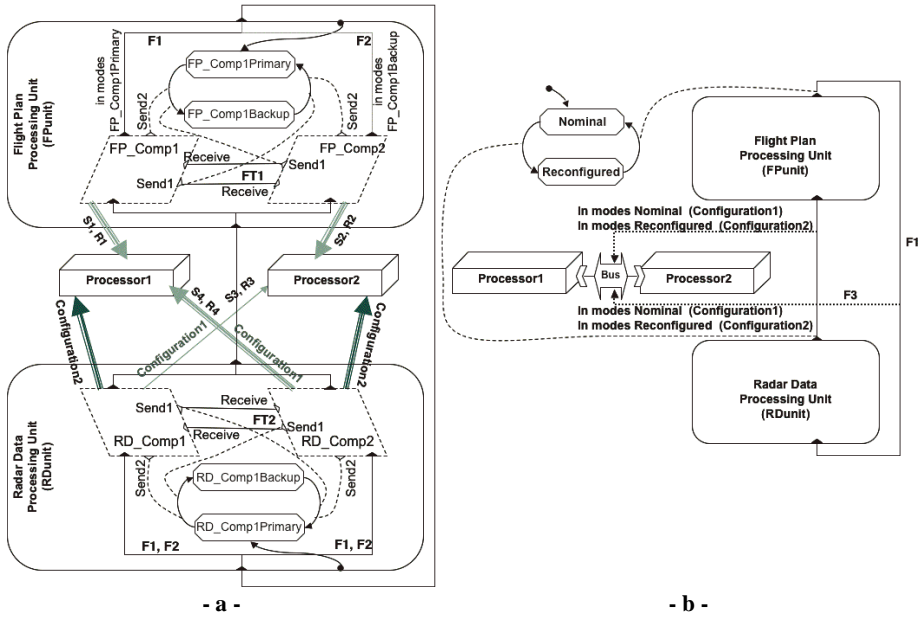


Fig. 14. AADL architectural model of Air Traffic Control System candidate architectures

The FPunit and the RDunit have the same structure (presented in Figure 10), i.e., they are formed of two replicas (threads): one having the primary role (provides the service) while the other one has a backup role (monitors the primary). Both candidate architectures use two processors. The two replicas of each software unit are bound to separate processors. In *Configuration1*, the initially primary replicas of the FPunit and RDunit (*FP_Comp1* and *RD_Comp1*) are bound to separate processors (*FP_Comp1* bound to *Processor1* and *RD_Comp1* bound to *Processor2*). In *Configuration2*, the initially primary replicas of the FPunit and RDunit are bound to the same processor, *Processor1*. The whole subsystem has two operational modes: *Nominal* and *Reconfigured*. Connections between replicas bound to separate processors are bound to a bus. Thus, the connection bindings to the bus depend on the operational mode of the subsystem. A bus failure causes the failure of the RDunit replica. The primary

replica of the FPunit exchanges data with both replicas of the RDunit. For the sake of clarity, we show the thread binding configurations in Figure 14-a and the bus and the connection bindings to the bus separately in Figure 14-b.

6.2 Dependency Analysis

The various interactions between this subsystem's components induce dependencies between them. Most of them are architecture-based, thus they are visible on the architectural model. We took into account the following dependencies:

- structural dependency between each processor and the threads that run on top of it. We assume that hardware faults can propagate and influence the software running on top of it. These dependencies (S1, S2, S3 and S4 in Figure 14) are supported by the architectural bindings of threads to processors.
- recovery dependency between each processor and the threads that run on top of it. If a thread fails, it cannot be restarted if the processor on top of which it runs is in a failed state. These dependencies (R1, R2, R3 and R4 in Figure 14) are supported by the architectural bindings of threads to processors.
- maintenance dependency between the two processors that share a repairman that is not simultaneously available for the two components. This maintenance dependency is not visible on the architectural model of Figure 14.
- fault tolerance dependency between the two RDunit threads and the two FPunit threads. If the replica that delivers the service fails but the other one is error free, the two software replicas switch roles. Then, the failed replica is restarted. These dependencies (FT1 and FT2 in Figure 14) are supported by the connections between the replicas of each software unit.
- structural dependency between the bus and the threads of the RDunit. If the bus fails, the broken connections bound to it make the RDunit fail in mode *Nominal* of *Configuration1* and in mode *Reconfigured* of *Configuration2*. This dependency (F3 in Figure 14) is supported by the binding of the connection from the FPunit to the RDunit to the bus.
- functional dependencies between the FPunit and the RDunit. The active FPunit thread may propagate errors to both RDunit threads. These dependencies (F1 and F2 in Figure 14) are supported by the connections of the FPunit replicas to the RDunit replicas. Note that we consider that RDunit errors do not propagate to the FPunit even though there is a connection from the RDunit to the FPunit.

Figure 15 shows the dependency block diagram describing the dependencies between components of the *Configuration1* of the Air Traffic Control System. We built the AADL dependability model iteratively, by integrating first the structural and functional dependencies and then the maintenance, recovery and fault tolerance dependencies. Due to space limitations, we further focus on the two grey-color blocks, which represent the functional dependency between a FPunit replica and both RDunit replicas and the fault tolerance dependency between the FPunit replicas.

The dependency block diagram for *Configuration2* is similar. In *Configuration2*, *Processor1* is linked to *RD_Comp1* via structural and maintenance dependency blocks and *Processor2* is linked to *RD_Comp2* via structural and maintenance

and the FPunit goes to the corresponding mode. To model this behavior, we associate error models with *FP_Comp1* and *FP_Comp2* and we use *Guard_Transition* properties on the out ports *Send* of both components. These *Guard_Transition* properties are extensions of those presented in Figure 11. The behavior in the case of a double failure requires including the notification of the end of the restart procedure before moving to *Error_Free* state.

Figure 16 presents the error model associated with the FPunit threads. Lines f1-f2 correspond to the functional dependency presented above while lines t1-t4 correspond to the fault tolerance dependency. The rest of the error model is similar to the one presented in Figure 1 for isolated components. The component may propagate errors (out propagation *Error*) but it cannot be influenced by *Error* propagations, as it does not declare an in propagation *Error*. The end of the restart procedure is notified (*IAmRestarted* out propagation) before moving to *Error_Free* state.

The only difference between the error model associated with the FPunit threads and the RDunit threads is the direction of the propagation *Error* and the AADL transition triggered by it. In the error model associated with the RDunit threads, *Error* is an in propagation triggering an AADL transition from *Error_Free* to *Failed*.

Error Model Type [forFP_Comp]	
	<pre> error model forFP_Comp features Error_Free: initial error state; Erroneous: error state; Restarted: error state; Failed: error state; Temp_Fault: error event {Occurrence => poisson λ_1}; Perm_Fault: error event {Occurrence => poisson λ_2}; Restart: error event {Occurrence => poisson μ_1}; Recover: error event {Occurrence => poisson μ_2}; (f1) Error: out error propagation {Occurrence => fixed p}; (t1) FailedVisible: out error propagation {Occurrence=>fixed 1}; (t2) IAmRestarted: out error propagation {Occurrence=> fixed 1}; end forFP_Comp; </pre>
Error Model Implementation [forFP_Comp.general]	
	<pre> error model implementation forFP_Comp.general transitions Error_Free-[Perm_Fault]->Failed; Error_Free-[Temp_Fault]->Erroneous; Failed-[Restart]->Restarted; (f2) Erroneous-[out Error]->Erroneous; (t3) Restarted-[out IAmRestarted]->Error_Free; (t4) Failed-[out FailedVisible]->Failed; Erroneous-[Recover]->Error_Free; end forFP_Comp.general; </pre>

Fig. 16. Error model forFP_Comp

Figure 17 presents the *Guard_Transition* properties that specify the conditions under which mode transition occur, according to the fault tolerance behavior described above. Mode transitions occur if one of the components sends the *FailedVisible* out propagation while the other one is *Error_Free* or if one of the components sends the *IAmRestarted* out propagation while the other component is not *Error_Free* (meaning that a double failure occurred and the first component has been restarted before the second one).

```

Guard_Transition =>
  (Comp1.Send2[FailedVisible] and Comp2.Send1[Error_Free])
  or (Comp2.Send1[IAmRestarted] and not Comp1.Send2[Error_Free])
  applies to Comp1.Send;
Guard_Transition =>
  (Comp2.Send2[FailedVisible] and Comp1.Send1[Error_Free])
  or (Comp1.Send1[IAmRestarted] and not Comp2.Send2[Error_Free])
  applies to Comp2.Send;

```

Fig. 17. *Guard_Transition* properties associated with Send ports of FPunit threads

6.4 AADL to GSPN Model Transformation

For these two dependencies, we use only the AADL to GSPN transformation rules presented in section 5. We first took into account the functional dependency from *FT_Comp1* to *RD_Comp1* and *RD_Comp2* and then the fault tolerance dependency between *FT_Comp1* and *FT_Comp2*. Before adding the fault tolerance dependency, the GSPN subnet FT1 did not exist and the *FP_Comp1* and *FP_Comp2* subnets were identical to the *RD_Comp1* and *RD_Comp2* subnets.

Figure 18 presents the part of the GSPN corresponding to the functional dependency between the *FP_Comp1* replica of the FPunit and the two replicas of the RDunit and to the fault tolerance dependency between the threads of the FPunit. For clarity reasons, immediate GSPN transition that empty intermediary places corresponding to out propagation are not shown.

6.5 Evaluation of Quantitative Measures

Figure 19 gives the unavailability of the two candidate architectures. Such quantitative measures are obtained from the processing of the GSPN derived from the AADL model. In Figure 19, the varying parameter is the occurrence rate of a bus failure; λ_c . $\lambda_c \leq 10^{-6}/h$ corresponds to a redundant bus. For *Configuration1*, the impact of this parameter is important when $\lambda_c \geq 10^{-5}/h$. *Configuration2* is much less influenced by λ_c , as in *Nominal* mode, the communication between the two units does not go through the bus. From a practical point of view, if $\lambda_c \geq 10^{-5}/h$, *Configuration2* is recommended. Otherwise the two candidate architectures are equivalent.

Other analyses can be carried out on the same model. The results of several analyses allow taking a decision about what candidate architecture best suites the application. For example, performance analyses can be performed to determine the impact of the choices made to achieve dependability goals on a system's performance.

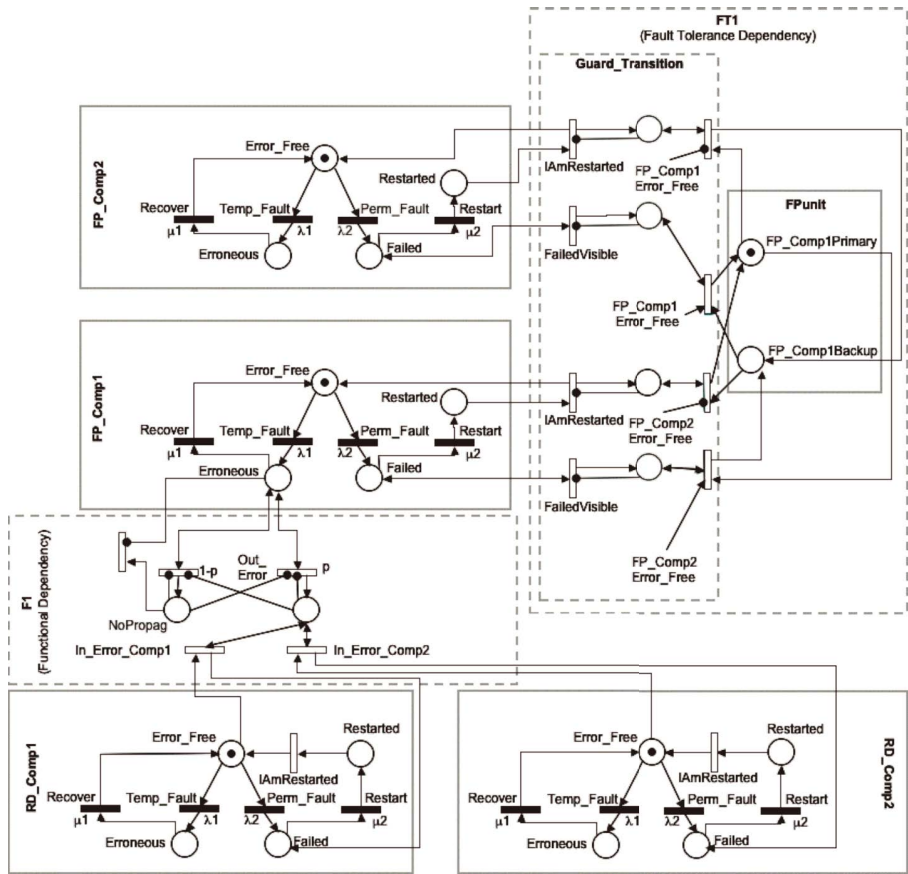


Fig. 18. GSPN model of the Air Traffic Control System – two dependencies

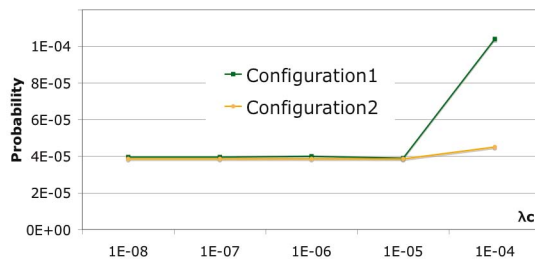


Fig. 19. Unavailability

7 Conclusion

We presented a stepwise approach for system dependability modeling using AADL and GSPNs. The aim of this approach is to hide the complexity of traditional analytical models to end-users acquainted with AADL. In this way, we ease the task of evaluating dependability measures. Our approach assists the user in the structured construction of the AADL dependability model that is transformed into a GSPN to be processed by existing tools. To support and trace model evolution, this approach proposes that the user builds the AADL dependability model iteratively. Components' behaviors in the presence of faults are modeled in the first iteration as if they were isolated. Then, each iteration introduces a new dependency between system's components in the AADL dependability model. The AADL to GSPN model transformation is meant to be transparent to the user. Thus, it is based on rigorous and systematic rules aimed at supporting tool-based transformation automation. The model transformation can be performed iteratively, each time the AADL dependability model is enriched. In this way, the GSPN model can be validated progressively (hence the corresponding AADL architecture and error models can be validated progressively and corrected accordingly, if required). Finally, we illustrated the proposed approach on a subsystem of the French Air Traffic Control System. We have shown the principles of the transformation and some of the rules. The work in progress concerns the implementation of a model transformation tool to be easily integrated into AADL and GSPN based tools.

Acknowledgements. This work is partially supported by 1) the European Commission (ASSERT European IP No. IST 004033 and ReSIST NoE No. IST 026764), 2) the European Social Fund and 3) Zonta International Foundation.

References

1. SAE-AS5506: SAE Architecture Analysis and Design Language (AADL), International Society of Automotive Engineers, Warrendale, PA, USA (November 2004)
2. SAE-AS5506/1: SAE Architecture Analysis and Design Language (AADL) Annex vol. 1, Annex E: Error Model Annex, International Society of Automotive Engineers, Warrendale, PA, USA (June 2006)
3. Bondavalli, A., Chiaradonna, S., Di Giandomenico, F., Mura, I.: Dependability Modeling and Evaluation of multiple-phased systems, using DEEM. *IEEE Transactions on Reliability* 53, 509–522 (2004)
4. Kanoun, K., Borrel, M.: Fault-tolerant systems dependability. Explicit modeling of hardware and software component-interactions. *IEEE Transactions on Reliability* 49, 363–376 (2000)
5. Bernardi, S., Bobbio, A., Donatelli, S.: Petri Nets and Dependability. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) *Lectures on Concurrency and Petri Nets*. LNCS, vol. 3098, pp. 125–179. Springer, Heidelberg (2004)
6. Farines, J.-M., et al.: The Cotre project: rigorous software development for real time systems in avionics. In: 27th IFAC/IFIP/IEEE Workshop on Real Time Programming, Zielona Góra, Poland (2003)
7. Singhoff, F., Legrand, J., Nana, L., Marcé, L.: Scheduling and Memory Requirements Analysis with AADL. In: *SIGAda Int. Conf. on Ada*, Atlanta, GE, USA (2005)

8. Béounes, C., et al.: Surf-2: a program for dependability evaluation of complex hardware and software systems. In: 23rd IEEE Int. Symposium on Fault Tolerant Computing, Toulouse, France, IEEE Computer Society Press, Los Alamitos (1993)
9. Deavours, D.D., et al.: The Mobius Framework and its Implementation. *IEEE Transactions on Software Engineering* 28, 956–969 (2002)
10. Hirel, C., Sahner, R., Zang, X., Trivedi, K.: Reliability and performability modeling using SHARPE 2000. In: 11th Int. Conf. on Computer Performance Evaluation: Modelling Techniques and Tools, Schaumburg, IL, USA (2000)
11. Bernardi, S., Bertinello, C., Donatelli, S., Franceschinis, G., Gaeta, R., Gribaudo, M., Horvath, A.: GreatSPN in the new millenium. In: Tool Session of 9th Int. Workshop on Petri Nets and Performance Models, Aachen, Germany (2001)
12. Ciardo, G., Trivedi, K.S.: SPNP: The Stochastic Petri Net Package (Version 3.1). In: 1st Int. Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, San Diego, CA, USA (1993)
13. Rugina, A.E., Kanoun, K., Kaâniche, M.: An Architecture-based Dependability Modeling Framework using AADL. In: 10th IASTED Int. Conf. on Software Engineering and Applications, Dallas, USA (2006)
14. Rugina, A.E., Kanoun, K., Kaâniche, M.: Modélisation de la sûreté de fonctionnement à partir du langage AADL. In: 15ème Congrès de Maîtrise des Risques et de Sûreté de Fonctionnement, Lille, France (2006)
15. Hugues, J., Kordon, F., Pautet, L., Vergnaud, T.: A Factory To Design and Build Tailorable and Verifiable Middleware. In: Kordon, F., Sztipanovits, J. (eds.) Monterey Workshop 2005. LNCS, vol. 4322, pp. 121–142. Springer, Heidelberg (2007)
16. OMG: Unified Modelling Language Specification (October 2004), <http://www.omg.org>
17. Majzik, I., Bondavalli, A.: Automatic Dependability Modeling of Systems Described in UML. In: Int. Symposium on Software Reliability Engineering (1998)
18. Bondavalli, A., et al.: Dependability Analysis in the Early Phases of UML Based System Design. *Int. Journal of Computer Systems-Science&Engineering* 16, 265–275 (2001)
19. Pai, G.J., Bechta Dugan, J.: Automatic Synthesis of Dynamic Fault Trees from UML System Models. In: 13th Int. Symposium on Software Reliability Engineering, Annapolis, USA (2002)
20. López-Grao, J.P., Merseguer, J., Campos, J.: From UML Activity Diagrams To Stochastic Petri Nets: Application to Software Performance Engineering. In: 4th Int. Workshop on Software and Performance, Redwood City, CA, USA (2004)
21. Bernardi, S., Donatelli, S., Merseguer, J.: From UML Sequence Diagrams and Statecharts to Analysable Petri Net Models. In: 3rd Int. Workshop on Software and Performance, Rome, Italy (2002)
22. Feiler, P.H., Gluch, D.P., Hudak, J.J., Lewis, B.A.: Pattern-Based Analysis of an Embedded Real-time System Architecture. In: 18th IFIP World Computer Congress, ADL Workshop, Toulouse, France (2004)
23. Rugina, A.E., Kanoun, K., Kaâniche, M.: AADL-based Dependability Modelling, LAAS-CNRS Research Report n°06209 (April 2006)
24. Ajmone Marsan, M., et al.: Modelling With Generalized Stochastic Petri Nets. John Wiley & Sons, Chichester (1995)
25. Kanoun, K., Borrel, M., Morteveille, T., Peytavin, A.: Availability of CAUTRA, a Subset of the French Air Traffic Control System. *IEEE Transactions on Computers* 48, 528–535 (1999)

Towards Improving Dependability of Automotive Systems by Using the EAST-ADL Architecture Description Language

Philippe Cuenot¹, DeJiu Chen², Sébastien Gérard³,
Henrik Lönn⁴, Mark-Oliver Reiser⁵, David Servat³,
Ramin Tavakoli Kolagari⁵, Martin Törngren², and Matthias Weber⁶

¹ Siemens VDO, 1 Avenue Paul Ourliac, BP 1149 31036 Toulouse Cedex 1, France
philippe.cuenot@siemens.com

² Royal Institute of Technology, SE-100 44 Stockholm, Sweden
{chen,martin}@md.kth.se

³ CEA List, Commissariat à l'Énergie Atomique Saclay, F-91191 Gif sur Yvette Cedex, France
{sebastien.gerard,david.servat}@cea.fr

⁴ Volvo Technology Corporation, Electronics and Software, SE-405 08 Gothenburg, Sweden
henrik.lonn@volvo.com

⁵ Technical University of Berlin, Software Engineering Group, D-10587 Berlin, Germany
{moreiser,tavakoli}@cs.tu-berlin.de

⁶ Carmeq GmbH, D-10587 Berlin, Germany
matthias.weber@carmeq.com

Abstract. The complexity of embedded automotive systems calls for a more rigorous approach to system development compared to current state of practice. A critical issue is the management of the engineering information that defines the embedded system. Development time, cost efficiency, quality and most importantly, dependability, all benefit from appropriate information management. System modeling based on an architecture description language is a way to keep the engineering information in one information structure. The EAST-ADL was developed in the EAST-EEA project (www.east-eea.org) and is an *architecture description language* for automotive embedded systems. It is currently refined in the ATESSST project (www.atesst.org). This chapter describes how dependability is addressed in the EAST-ADL. The engineering process defined in the EASIS project (www.easis-online.org) is used as an example to illustrate the support for engineering processes in EAST-ADL.

Keywords: architecture description language, automotive systems, systems engineering.

1 Introduction

Current development trends in automotive software feature increasing standardization of the embedded software structure. The need to integrate software from different suppliers, supporting dependable real-time execution, and managing changes, all call

for an integrated approach for software-based vehicle systems. Unfortunately, the last decade has shown that even connecting rather simple stand-alone systems to integrated systems has led to a number of unexpected vehicle failures. The complexity of embedded automotive systems calls for a more rigorous approach to system development than is current state of practice. A critical issue is the management of the engineering information that defines the embedded system. Development time, cost efficiency, quality and most importantly, dependability all benefit from appropriate information management.

Dependability is a broad concept covering many qualities that are otherwise considered separately, including reliability, availability, safety and security (see [22], [33], [35]). It refers to the overall property of a system that justifies placing one's reliance or trust on it [2]. Design for dependable computer systems involves techniques from, at least, three major engineering approaches [33]: safety-engineering, dependability-engineering, and real-time system engineering. The safety engineering approach addresses the environmental consequences of faults and often relies on dependability services (e.g. fault-tolerance) to control hazards that cannot be eliminated by design. The dependability engineering approach emphasizes the quality of services when faults occur, with a focus on reliability. The engineering of real-time systems often takes both fault tolerance and safety solutions into consideration but pays special attention to hazardous timing-dependent behaviors. In such approaches, models play a central role in making and justifying design decisions and in providing early quality feedbacks. From a system development point of view, one challenge is to enable an integrated analysis and design and hence to make the information align with each other during design or changes.

System modeling based on an architecture description language (ADL) is a way to keep the engineering information in a well-defined information structure. The EAST-ADL was developed in the EAST-EEA project (www.east-eea.org), and is an architecture description language for automotive embedded systems. It is currently refined in the ATESSST project (www.atesst.org). The guidelines for this refinement process are the identification and integration of the most adequate approaches and techniques for each need (specific to the automotive domain). The EAST-ADL2 language contains thus UML2 basic constructs, the requirement concepts from SysML, practical variability approaches for highly complex product lines developed from the automotive domain, function modeling from SysML, behavior from SOTA tools and UML2, error behavior from AADL, implementation modeling from AUTOSAR, and finally non functional properties from MARTE are reused.

The differences between SysML and EAST-ADL are the following: the SysML language is reused as far as possible, EAST-ADL providing for the framework/ontology to guide the use of SysML concepts in an automotive context. The SysML-based part of EAST-ADL2 is linked to the automotive implementation concepts from AUTOSAR and augmented with concepts from AADL and MARTE. Variability constructs and verification and validation constructs are further contributions beyond plain SysML.

This chapter describes how the refinement made on several sub parts of the EAST-ADL language address the issue of system dependability. Section 2 presents an overview of the language constructs, the subsequent sections deal in turn with several aspects of dependability: requirements, variability modeling, analysis methods and

engineering process support. The latter uses EASIS engineering process as an example. Finally, future steps which form the ongoing work in the ATESSST project are underlined and some conclusions drawn.

2 Overview of the EAST-ADL

EAST-ADL is an architecture description language, dedicated to automotive embedded electronic systems, developed in the context of the ITEA cooperative project EAST-EEA (<http://www.east-eea.net/>) finished in 2004.

This language is intended to support the development of automotive embedded software, by capturing all the related engineering information. The scope is the embedded system (hardware and software) and its environment. On top of the formal description of the elements, the language defines different abstraction levels that reflect different detail level of the architecture and implicitly different stages of an engineering process. The detailed process definition is company specific.

The EAST-ADL language constructs support:

- vehicle feature modeling including variability concepts to support product families
- vehicle environment modeling to define context and perform validation
- Structural and behavioral modeling of software and hardware entities supporting refinement to code and binaries in the context of distributed system
- requirements modeling and tracing with all modeling entities
- other information part of the system description, such as a definition of component timing and failure modes, necessary for system verification purposes.

The language is structured in five abstraction layers, each with corresponding system representation (in brackets):

0. operational level supporting final binary software deployment (Operational Architecture)
- I. implementation level with reusable code (platform independent) and AUTOSAR compliant software and system configuration for hardware deployment (Implementation Architecture)
- II. design level for detailed functional definition of software including elementary decomposition (Design Architecture)
- III. analysis level for abstract functional definition of features in system context (Functional Analysis Architecture)
- IV. vehicle level for elaboration of electronic features (Vehicle Feature Model)

Note that environment model spans all abstraction levels, and that requirements and variability constructs apply to modeling elements regardless of abstraction level. Depending from the different abstraction view, structural decomposition of the automotive electronic system is decomposed based on functional definition of the system. At vehicle level, feature are stamped from their interaction with vehicle environment, and then refined during Analysis and Design level with ADL function

description. Toward ADL function description, external behavioral description is documented to support integration in the architecture with EAST-ADL semantics, while internal behavioral definition is either defined with native ADL representation (state chart for example), or mainly referencing external tool with dedicated feature for algorithm numerical representation. At implementation level, relation between ADL function and software component is set, to allow software deployment on hardware architecture at operational level. Starting from Design level, hardware elements are part of the core language, and allow to model sensor, actuator and hardware elements to represent core, peripheral, pins, communication bus, and middleware service applicable to automotive context and standardized with AUTOSAR initiative.

The ATESSST project is currently aiming to refine the EAST-ADL language in the context of dependability concerns, supporting OMG standard alignment and the new automotive domain standardization AUTOSAR (<http://www.autosar.org/>).

To support a sound EAST-ADL language in relation to the new automotive standard, the lower levels of the language have been reworked to support software and hardware model entities standardized in the AUTOSAR templates. Detailed adjustments are in progress to assess matching of AUTOSAR ontology with abstract representation at design level.

To cover dependable systems, on going activities enrich requirement constructs to satisfy the needs of different integrity levels, refine the modeling entities to support necessary analysis methods, to finally being able to support an engineering process for safety. Transversal to these concepts, with same consideration for dependability, the variability constructs of EAST-ADL are improved to support vehicle product lines, the major productivity driver in automotive industry.

3 Dependability Requirements

In order to better support the development of dependable systems, the EAST-ADL does not only include means to create analysis and design models of the system to be developed (at varying abstraction levels), but also language means

- to specify required properties of the system (at varying degrees of abstraction),
- to trace requirements between system refinement and system decomposition levels
- to require satisfaction of requirements for system components,
- to refine the specification of requirements by behavioral models
- to verify requirements by verification and validation activities.

The EAST-ADL does not start from scratch but closely aligns its requirements concepts to SysML 1.1 [27], as currently standardized by the OMG organization. However, extensions and adjustments are made to these proposals based on the needs of the automotive application domain.

3.1 Basic Requirements Relations

Four requirement relations from SysML are used in the EAST ADL, Figure 1.

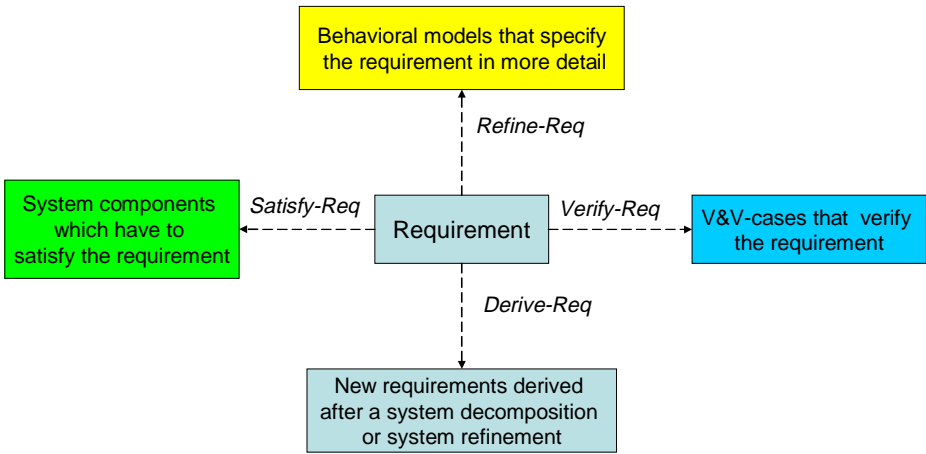


Fig. 1. The basic requirements relations *refine*, *verify*, *derive*, and *satisfy*. (adopted from SysML).

First and foremost, requirements may be used to textually specify required properties of the system to be developed. The textual specification of a requirement can be refined (using the “refine” relationship from SysML) by attaching behavioral models – such as use-cases, activity diagrams or state machines – to requirements.

Requirements are refined into more detailed requirements after a system refinement or a system decomposition step. This concept is supported by the “derived requirement” relationship to allow a hierarchical view of requirements to be defined. It allows analysis of requirements to determine multiple derived requirements that support a source of requirement, and the document requirement over the system decomposition, but also over the different abstraction view point of feature development from different disciplines.

Specific UML constructs such as use case or activity diagram assist requirement analysis for better description or prepare further refinement. These constructs are associated to requirement entities via a “refine” construct.

Requirements apply to the various system components which are introduced to satisfy them. This concept is modeled using the “satisfy” relation from SysML.

Requirements traceability is completed by the verification and validation relationships. These defines how the verification and validation activities, such as the testing activities fit criteria explicit for the verification and validation goals and how its associated verification and validation cases, such as test cases, verify the requirement. This is modeled using the SysML “verifies” relationship along with constructs that capture and relate V&V to the requirement and system components.

The “satisfy” and “derive” relations are illustrated in Figure 2.

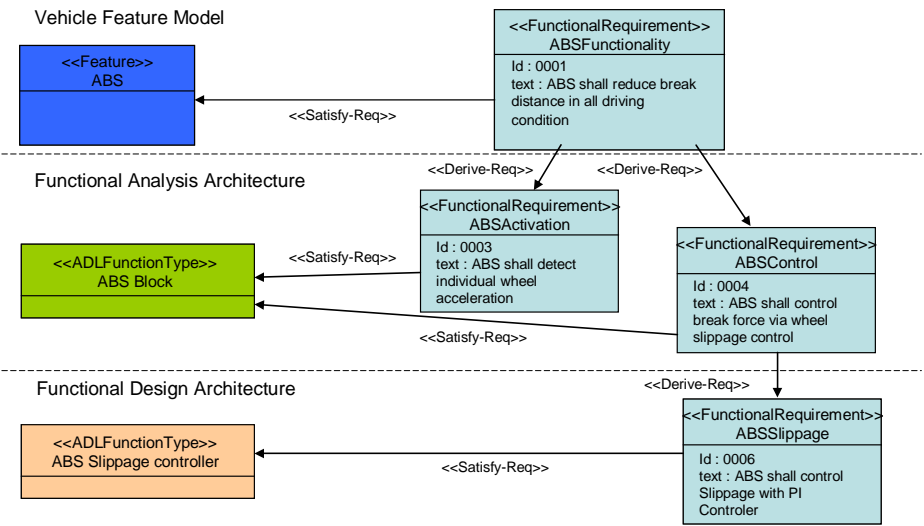


Fig. 2. The figure illustrates requirements tracing and linking to system components

The refinement of a requirement by means of a use case is illustrated in Figure 3.

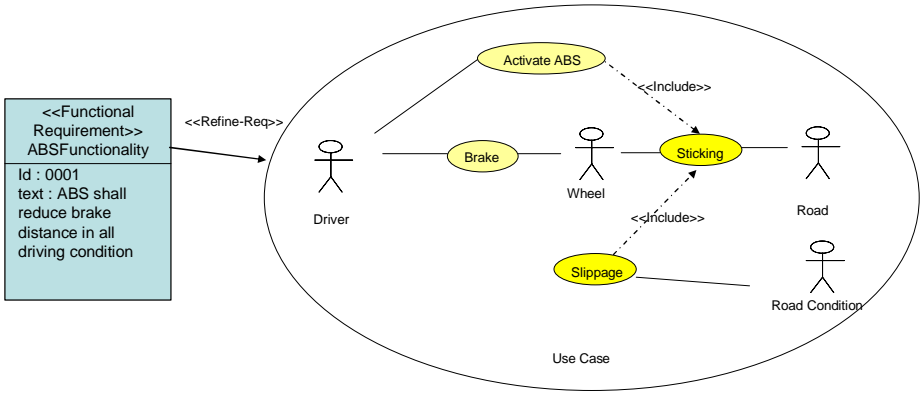


Fig. 3. The figure illustrates the refinement relation

3.2 Requirements Types

Methodically, EAST ADL differentiates between *functional requirements*, which typically focus on some part of the “normal” functionality that the system has to provide (e.g. “ABS shall control brake force via wheel slip control”), and *quality requirements*, which typically focus on some external property of the system seen as a whole (e.g. “ABS shall have an MTTF of 10.000 hours”).

Quality requirements are further classified from standardized enumeration list: Performance, Dependability, HMI, Configurability, Ergonomy, Safety, Security,

Others. Below, the two examples “safety requirements” and “timing requirements” are discussed in more detail.

3.3 Safety-Related Requirements

In order to perform safety assessments of the vehicle systems, safety requirements in the refined EAST-ADL have attributes and related entities to define the requirement and the hazard it mitigates. Hazards or hazardous events are part of the environment model and are characterized by attributes for severity, exposure and controllability [16]. The hazardous event may be further detailed by e.g. use cases, sequence or activity diagrams.

Safety requirement attributes includes safety integrity level (SIL), operation state, fault time span, emergency operation times, safety state, and functional redundancy to record dependability characteristics [16]. A requirement can be traced from the abstract vehicle model all the way to its derived requirements allocated to the final hardware and software components. Depending on abstraction level, some or all of these attributes are applicable.

3.4 Timing Requirements

Embedded systems have several timing requirements. On the top level, there are performance requirements based on e.g. ergonomics or safety. To meet such top level timing requirements, or to sustain the selected design regarding resource scheduling or interaction between components, timing requirements can be seen on all abstraction levels of an automotive system. Timing errors are the source of many failures, and it is thus important to correctly express and subsequently analyze timing properties. Examples of timing requirements that are supported by the EAST-ADL include end-to-end deadlines, period timing and worst case execution time. The goal is to be able to support the analysis techniques necessary for high integrity automotive systems.

3.5 Explicit Modeling of Verification and Validation (V&V) Artifacts

In order to support the development of dependable systems, the EAST-ADL offers detailed means to explicitly model central artifacts of verification and validation activities and to relate these artifacts to requirements. This allows for explicitly and continuously planning, tracking, updating and managing important V&V-activities and their impact on the system in parallel to the development of the system.

The combination of a V&V-case, its environment (element V&V Stimuli, V&VIntendedOutcome, V&VActualOutcome) and its target object (element V&VTarget) is described as a V&V context. A V&V-case will take very different forms, depending on the kind of V&V activity performed, e.g. safety analysis, specification, design or implementation review, functional analysis by simulation, SIL-testing, HIL-testing, or vehicle testing. In general it consists of a number of V&V-procedures to be applied to the target object, which are recorded close the modeling artifact. Each procedure may contribute a dedicated aspect in the verification of some requirement and the EAST-ADL allows documenting this relationship by means of the “verifies” relation. These informations and relations

between model, requirement and verification validation information are a way to ensure all requirements are satisfied in the configuration that correspond to produced vehicle. Also test coverage control is simplified by centralized information analysis, and guarantee adequate usage and representation of environment for validation, critical for correctness and safety of realized functions. The context of function reuse will also benefit from these formal associations, by a consistent functional package.

The basic association of a V&V-case to a requirement and to its V&V context is illustrated in Figure 4.

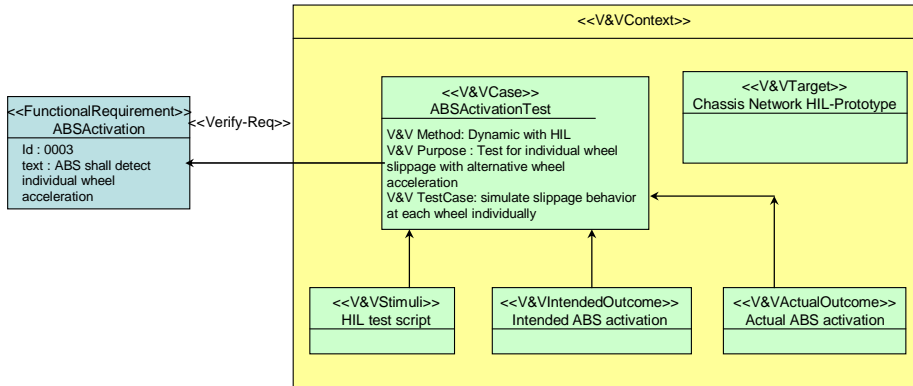


Fig. 4. The figure illustrates the verification of a requirement

4 Variability Modeling for Safety-Related Systems

Most of the systems developed today are variable systems. This holds in particular for automotive systems where the systems or parts of them are reused rather than developed from scratch. In order to develop dependable systems, reuse must be supported systematically to be safe, beneficial and effective. The main technique to systematize reuse is to manage the system's variability appropriately. There are two reasons:

- Reuse means handling of changes (i.e. variability); systematic reuse essentially means handling of expected and wished changes. So one has to ask how changes occur, which changes are right and wished, and which changes are occasional, uncoordinated or historically conditioned and hence possibly dispensable. Thus, if a product is changed in the course of time, these questions should be asked at all stages of development. Because in any stage individual solutions or approaches may need to be realized, or short-cuts or simplifications may be necessary as a result of short development schedules. These temporary changes should not become methodical, e.g. by putting them as a specific and reusable variant in a reuse platform; this means that techniques must be provided to prevent developers from making short-term solutions part of a reuse framework. Variability management provides for the right amount of the right changes to be communicated

throughout the system development process. As a consequence, future manufacturer development teams as well as suppliers will profit from variability management, since it can provide improved systems produced in shorter and more cost-effective development cycles.

- By managing the variability, differences are made explicit. Explicit documentation is beneficial for all stakeholders that need an abstract view on the system, like business management or customers. The system's variability is the basis on which they can decide for their choices: management becomes aware of the diversity and resulting complexity of the products with the consequence that variants are brought as assets to market consciously; and the customer is given access to the product portfolio with all its variants.

Variability in the automotive domain is highly complex. The complexity of automobile electronics arises because of the distribution of a variety of interacting functions over a number of different components and a large number of interfaces, as well as the variation resulting from inevitable product differentiation.

This situation is problematic, because so far there is no methodical support to reuse features or artifact elements between different model ranges and thus achieve economies of scale: we use the term model range in order to describe a set of vehicles that have recognizable commonalities; these vehicles can also differ from one another with respect to the choices e.g. made by a customer. This kind of differentiation within the model range we call model range specific variability. Forms of model range specific variability include variability through optional equipment, variability through special purpose vehicles (police cars, taxis ...), variability through country-specific equipment, and variability through specific design (cabriolet, estate ...).

The classical definition of a software product line is based on kinds of product families that are similar to model ranges and furthermore asks for explicit activities to manage the commonalities and differences of the products: "A software product line is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way" [7].

Model range specific variability must be managed in order to realize reliable and reusable artifact specifications and to make the model range specific variability visible.

In the automotive domain beneficial reuse is based on using artifacts between different model ranges. Different model ranges can also differ in their underlying variability approach and thus a technique is needed to express variability and commonality of model ranges, i.e. a technique to manage model range spanning variability. Such a technique is needed to develop systems because a practical and successful way to realize dependability is to profit from previous system development: previous artifact elements and variability approaches are improved by usage in practice.

In the remainder of this section we describe how model range spanning variability can be handled. We distinguish between two cases of variability management: the global perspective for the entire vehicle represented in EAST-ADL by the vehicle feature layer (cf. Section 4.1) and variability of individual development artifacts, e.g. the component diagrams and their related behavioral descriptions on the functional design layer (Section 4.2).

4.1 Variability Modeling on the Vehicle Level

Variability management is required on two levels: First, within each artifact – such as a requirements specification, a design model or test case – variability with respect to the artifact's contents has to be defined. For example, it has to be specified that certain requirements are only valid for selected models or in some markets. Second, all such variants and combinations defined on the artifact level have to be coordinated and strategically managed on a global perspective for the entire product range. In this section we focus on the latter, the big picture, while the next section is then devoted to a discussion of artifact level variability modeling.

Variability modeling for large industrial product ranges is faced with several challenges. First of all, the degree of complexity is enormously high. Today's global automotive manufacturers, for example, usually comprise several brands, most of them partitioned in divisions for passenger vehicles and commercial vehicles each including many model ranges. Those manufacturers also offer their products in many diverse markets and market segments with diverse legislation and customer demands. And clients expect to be able to further customize the model of their choice. All these different aspects lead to product variability. In recent years, parts of the industry tried to face this challenge by avoiding variability wherever possible. While this is certainly an important approach, it cannot be the only way to deal with the problem. From a marketing perspective, it is of utmost importance to tailor products as closely as possible to the customers' expectations. Consequently, variability of substantial complexity will remain an important issue for automotive development.

Another challenge for modeling variability in this context is related to the way automotive industry is organized. Development and production in this domain are traditionally characterized by a high degree of collaboration between an original equipment manufacturer (OEM) and its suppliers. From the perspective of a development methodology this means, that no actor, neither the OEM nor any of the suppliers, actually has a global view on the system and its development artifacts. In particular, even for the OEM, many subsystems appear as black-boxes in the overall design of the vehicle and vehicle range. Especially for variability modeling on the vehicle level – as a means to globally manage development from a central perspective – this is an important aspect to be taken into account.

Finally, the development of the artifacts themselves poses an important challenge for variability modeling, or, more precisely speaking, the methods, languages and formalisms in which these artifacts are formulated and the tools in which they are maintained. These methods and tools are of very different form and nature, thus introducing a vast heterogeneity in automotive development. Due to the involvement of a multitude of different actors (esp. different companies), the size of automotive corporations, the long adoption cycles for new methods and tools and the need to maintain legacy artifacts relying on legacy methods and tools, this heterogeneity of development means cannot be eliminated, however desirable this may seem.

Therefore, we can summarize the most important challenges to be considered when providing a feasible concept for variability modeling on the vehicle level as follows:

- variability of very high complexity
- high degree of collaboration within and between automotive companies
- heterogeneity of development processes, methods and tools.

One of the most common concepts of variability modeling are *feature models*, which were introduced by Kang et al. [17], [18]. In particular, they are used to document what characteristics are common to all products and which characteristics differ from one product to another. Also dependencies between characteristics are defined in feature models. A *feature* in this context is a certain characteristic or trait that each product instance may or may not have. Usually, feature models are hierarchically structured as *feature trees* or directed acyclic graphs, i.e. tree-like structures in which a node – here a feature – may have more than one parent. A small sample feature tree is shown in Figure 5. Apart from documenting the commonalities and differences between products of a product range, feature models also serve as coarse-grained requirements. This last aspect often is not well covered by feature modeling methodologies in that they lack a detailed definition how features and requirements are related. It is one aim of the effort of refining the EAST-ADL in the ATESSST project to provide a detailed proposal for this which serves the needs of the automotive domain.

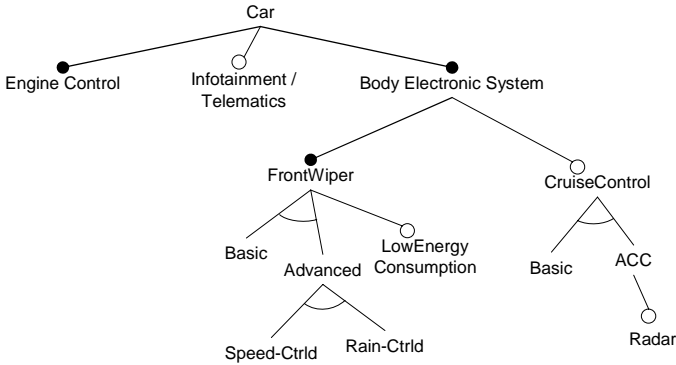


Fig. 5. The figure depicts a sample feature model. Edges connected with an arc denote alternative features (e.g. cruise controls Basic and ACC). A filled circle means that the corresponding feature is mandatory; an optional feature is supplied with an empty circle.

Feature models are, by now, a well-established instrument for variability modeling in traditional software engineering domains. However, when applying this technique for the development of complex software intensive systems, in particular automotive control systems, several open issues arise due to the challenges described above. This is true, even though feature models are to some extent an answer to the above challenges: By providing an abstract view on a system's variant and invariant characteristics, feature models are able to serve as a link between management, marketing and development within a single company. They also serve as a link between companies to assist communication, from contract negotiations to inspection of the supplied deliverables. Finally, they can provide a central view of variability in a wide range of development artifacts, thus becoming the core of all variability and evolution management. Some open issues remain however and these are subject to recent research activities of the software product line community. In the course of the

ATESST project, results emerging from these activities are integrated and, where necessary, adapted or complemented to meet the demands of the automotive domain.

For the purpose of this overview, we only introduce one of these issues briefly to provide an example. When applying feature modeling in traditional software domains, the process of configuration – i.e. selecting or deselecting optional features and thus choosing one of the model's possible products – is usually an interactive activity of a customer or an engineer acting on behalf of a customer and takes place for each delivered product separately. In contrast, feature models used as a central view on an automotive manufacturer's product range are far too complex and include far too many purely technical features to be directly configured by the customer. Many choices depend on the country for which the vehicle is built or what supplier is currently able to offer a certain subcomponent at the lowest price. Therefore, many configuration decisions have to be pre-defined and documented and the remaining variability has to be packaged and set up for customer configuration.



Fig. 6. The figure depicts an example of several orthogonal product decisions with different rationales and from different stakeholders, all influencing the same feature

The pre-defined configuration decisions can become impressively complex: selection decisions for several hundreds or even thousands of features are often each influenced by several orthogonal considerations (illustrated in Figure 6). Therefore it is not feasible to supply each feature with a logical expression stating when the corresponding feature will be selected, because these would each be influenced by many different orthogonal considerations and will therefore be extremely difficult to adapt to changes of individual considerations. To solve this problem, so called *product decisions* and *product sets* are used to clearly document the orthogonal configurations considerations, their rationale and the person responsible for them. For a detailed discussion of this problem and the mentioned solution concept please refer to [31] and [32].

4.2 Variability Modeling of Artifacts

On the artifact level, the situation is much different. While variability definitions can become fairly complex here too, the mere size of these definitions is not one of the

primary problems. Also the number of engineers or teams which are directly manipulating an individual artifact is comparatively manageable in most cases. In contrast, the main difficulty for variability modeling within artifacts consists in that the concepts for defining variability are closely coupled with the structure and nature of the artifact's contents. The challenge is to find a concept suitable to express variability in artifacts ranging from requirements specifications with their rather textual content over EAST-ADL's component diagrams – namely the functional analysis architecture and the functional design architecture – to test case descriptions.

In addition, the many semantic relations between artifacts pose another important challenge to variability modeling on this level, because they bring about manifold dependencies of the variability definitions across artifacts. For example, when a certain signal is defined as being variable within a component's interface in the functional design architecture, this must be reflected by the variability definition within the state machine specifying the component's behavior: all transitions triggered by the respective signal have to be labeled variable, if sufficient detail is considered.

Another challenge to be solved on this level is to check whether all variability defined as desirable on the global vehicle level can be realized by way of the variability provided within the artifacts. In other words, for all desired product variants, there must exist a corresponding valid configuration of the artifacts. Similarly, it must be possible to discover redundancies in the variability definitions, i.e. possible configurations which are superfluous according to the global variability definition of the vehicle level.

To solve these obstacles, an approach called *variation point propagation* [37] is used. The basic idea of this approach is that whenever a variation point is added to an artifact, a number of additional variation points are deduced based on a set of predefined rules. The fact that the EAST-ADL is designed as a comprehensive framework consisting of a fixed set of mutually aligned artifact types is a great opportunity for this approach.

Regardless of whether the internal - component view - or external - vehicle view - variability is considered, the complexity of variability calls for a rigorous approach to avoid the introduction of potentially dangerous failures. With the same means, the potential of improved reliability through re-use can be realized.

5 Dependability Analysis Methods

The purpose of an architecture description language is to capture engineering information for documentation as well as analysis. One focus of EAST-ADL is to enable an integrated dependability analysis and architecture design. This section will discuss some analyses that are of particular concern for EAST-ADL as a means to improve dependability and provide an overview of current language support for error modeling and the integration of an external tool for safety and reliability analysis.

As a key element of safety analysis, hazard analysis aims to identify hazards and the causes and consequences of system failures. Normally, hazard analysis also involves the assessment of hazard levels in terms of probability and criticality and the generation of safety requirements. For safety critical systems, it is preferable to

perform such an analysis at different design stages and levels of abstraction [35]. Two well-known techniques for identifying unknown hazards are HAZOP (Hazard and Operability Studies) [20] and FFA (Functional Failure Analysis) [34].

Revealing the factors causing hazards is an essential step toward specifying safety requirements and designing hazard control solutions. This activity is also referred to as hazard causal analysis [22]. Common techniques for this type of analysis include FMEA (Failure Modes and Effects Analysis) [28] and FTA (Fault Tree Analysis) [37]. FMEA is a bottom-up technique that derives the possible causes of system/component failures by reasoning about the effects of abnormal behaviors and other details within a system/component. The results from FMEA are often captured in tabular form. FTA is a top-down technique that searches for the possible causes of a given system/component hazard by going back from a top-level failure event to the lower-level events contributing to it. The results from FTA are presented as fault trees depicting the causal or logical relationships of events. In recent years, several adaptations of such classical techniques, originally developed for physical systems, have been proposed, targeting software programs at different levels of detail, such as the extensions of HAZOP in [5], [21], SW FMEA in [24] and SW FTA in [6] and [23].

The focus of classical safety analysis techniques lies, in general, on supporting the reasoning of possible failures (i.e. in terms of failure modes) and on recording the causal relationships in failure events (e.g. in tabular and tree structures). The analysis usually requires a description of the logical structure of systems, e.g. as a data flow diagram. However, it is up to the engineers, based on their understanding of the systems, to determine the actual failures of concern and the propagations. Another common limitation of classical techniques is the combinational effects of multiple component failures. While a component can have failure modes, it might make sense to consider the effect of multiple component failures in a certain time sequence. Historically, the discontinuity between system and software safety requirements has made the translation from system safety requirements to software requirements difficult. Fault trees, constructed for hazard causal analysis, have also been used or interpreted as safety requirements. However, since traditional fault trees cover only the causal aspect of failures in Boolean logic, they are considered insufficient for specifying software systems because of the lack of information concerning ordering, timing and synchronization.

Over the years, various formal safety analysis techniques have been proposed to extend and complement classical safety analysis techniques and also to automate the activities. These formal techniques differ from the classical ones in that they provide executable models, e.g. by the use of Petri-nets in [14] and temporal formulas in [15], and by using some formal/mathematical methods to check safety properties (e.g. model-checkers or theorem-provers). The use of formal safety analysis techniques, although not new, is still immature. Some major obstacles relate to integrating system design and formal safety assessment in respect of the engineering activities, the models, and the tools [11]. Recent research efforts addressing formal safety analysis of complex systems include the European project ESACS (FP5) [4] and its follow-up project ISAAC (FP6) [1] for aeronautical systems. The ESACS methodology distinguishes models of nominal and failure behaviors by providing an extended system model that refines the model for nominal system behaviors by adding failure

modes. The modeling is supported by formal languages like AltaRica [19] and the analysis by various tools. For example, the toolset and environment FSAP/NuSMV-SA [3] has been developed for safety analysis. It provides support for model construction with a library of predefined failure modes, automatic fault injection and definition of safety requirements in temporal logic formulas, automatic fault tree construction and simulation.

The aim of reliability and availability analysis is to estimate the failure rates and repair time of a system or its components. One way of performing the analysis is to first define a stochastic process describing the error propagations, e.g. by using Markov modeling, and then to derive the overall reliability and availability from the reliability of the parts. The analysis can be based on fault trees. For example, the Galileo fault tree analysis tool [36] supports reliability analysis by transforming fault trees into equivalent Markov models. It implements the DIFTree (Dynamic Innovative Fault Tree) analysis methodology, which combines static and dynamic fault tree analysis techniques. The dynamic fault trees extend traditional (static) fault trees by ordering failure behaviors and their functional dependencies.

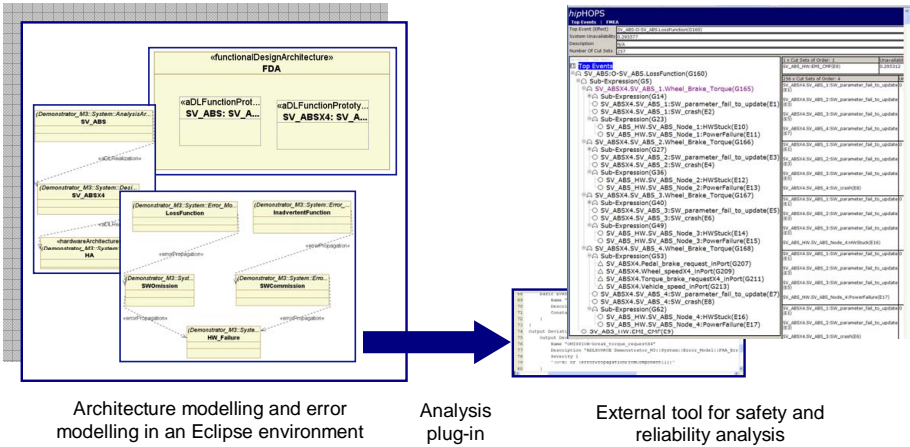


Fig. 7. A conceptual illustration of an EAST-ADL error models and its integration with the external analysis tool HiP-HOPS

The EAST-ADL will support several types of analysis, several of which directly or indirectly target safety. Examples of these include the mentioned safety and reliability analysis techniques. To this end, the EAST-ADL language provides explicit support for modeling error behaviors and propagations, deriving and managing the safety requirements, and integrating external analysis tools. As a first step towards this goal, a proof-of-concept integration of the HiP-HOPS method (Hierarchically Performed Hazard Origin and Propagation Studies) [29] has been development. Figure 7 illustrates the architecture modeling and error modeling in an Eclipse environment and an example output produced by HiP-HOPS. The integration is performed through the Eclipse plug-in technology. The HiP-HOPS method integrates a set of classical techniques for safety and reliability analysis, including FFA, FTA, and FMEA, and

provides the possibility of taking the combinations of errors and the temporal ordering into consideration. The method and tool has been incorporated earlier into the European projects TTA and SETTA. The analysis leverages include fault trees from functional failures to their causes in terms of software and hardware errors, calculations of minimal cut-sets, FMEA tables for component errors and their effects on the behaviors and reliability of entire system.

The language support for safety and reliability analysis is provided through the EAST-ADL error modeling package, targeting an architectural solution at different levels of abstraction captured in EAST-ADL. It also enables the traceability to system requirements and the associated V&V cases, environmental conditions, and architectural relationships such as communication and allocation. The concept is shown in the Figure 8, where the components *A* and *B* represent components of different types in an automotive EE system like function blocks, software and hardware components. The *A_Rel_B* represents an architectural relationship between two components, which can for example be an allocation or a communication.

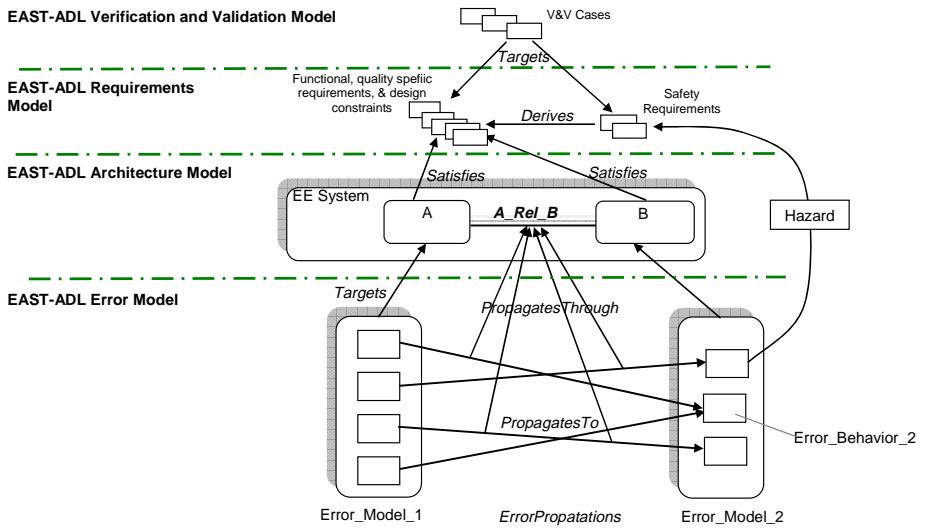


Fig. 8. The figure depicts the major relationships of EAST-ADL error modeling and other parts of the language

The EAST-ADL modeling package extends the architecture modeling support by allowing the failure semantics to be specified for every architectural entity and provides explicit information about the error propagations. The architectural entities of concern include abstract functional blocks in the Functional Analysis Architecture (FAA), software design components in the Functional Design Architecture (FDA), and hardware components in the Implementation Architecture (IA). Each architectural entity is associated with an error model, consisting of a set of local error behaviors. Each local error behavior specifies a particular failure semantics that relates a set of component internal failure events and a set of local effects of external failures (e.g., a

failure of underlying hardware component or a value failure of input) to a particular local failure mode that can propagate to the environment (e.g., value failures of an output). It is also possible to associate several alternative error models to an architectural entity, of which one particular instance will be chosen for a particular analysis through the use of EAST-ADL variability mechanism (see Section 4). The specifications of failure semantics can be based on logical or temporal expressions, depending on the analysis techniques and tools of interest and available. Safety requirements are derived through error behaviors of abstract functional blocks in the FAA in combinations with some environmental conditions of concern as well as the underlying software and hardware component failures. Such environmental conditions together with the system functional failures define system hazards, while the component failures reveal the causing factors of hazards.

The propagations of local error behaviors of the architectural entities are captured and controlled through an explicit error propagation construct, which provides an abstraction for describing and specifying the relationships of errors across abstraction levels and compositional hierarchies. Through this propagation construct, EAST-ADL supports the specifications of advanced properties of error propagations, such as the logical and temporal relationships of source and target errors, the enabling conditions of propagations, and the synchronizations of multiple propagation paths. Traceability is ensured through explicit associations from error propagations to the predefined architectural relationships like communication, synchronization, and allocation. In Figure 8, this is illustrated by the *propagationThrough* relationship from error propagations to the relationships between Component *A* and *B*. Currently, error propagations from software and hardware components to abstract functional blocks, and between hardware and software components are allowed in the language.

6 An Engineering Process for Safety

A key component for improving safety is an adequate development process. This section describes how the EAST-ADL entities can be used in an engineering process that has safety as a prerequisite. This process was developed in the EASIS project (<http://www.easis.org>) [8]. Note that the EAST-ADL itself is not tailored for a specific process, so this is only an example of its use.

The scope of the EASIS Engineering Process is the initial stages of development, from requirements to design. The implementation, integration and testing stages of a typical V-model development process are not considered.

The EASIS Engineering Process contains 5 main parts (see Figure 9):

- Part 1: Specify requirements Collection and integration of system and safety requirements
- Part 2: Development of functional architecture (FAA model) Definition of an abstract, hardware independent functional architecture based on identified requirements
- Part 3: Development of hardware architecture Initial definition of hardware architecture based on needs of the FAA model
- Part 4: Development of design architecture (basic FDA model) Definition of a basic design architecture, which realizes the abstract functional architecture

previously defined. The architecture is adapted to hardware architecture, OS, middleware services.

- Part 5: Refinement of the design architecture (FDA model) Handling of faulty hardware and signals is added to the basic design architecture.

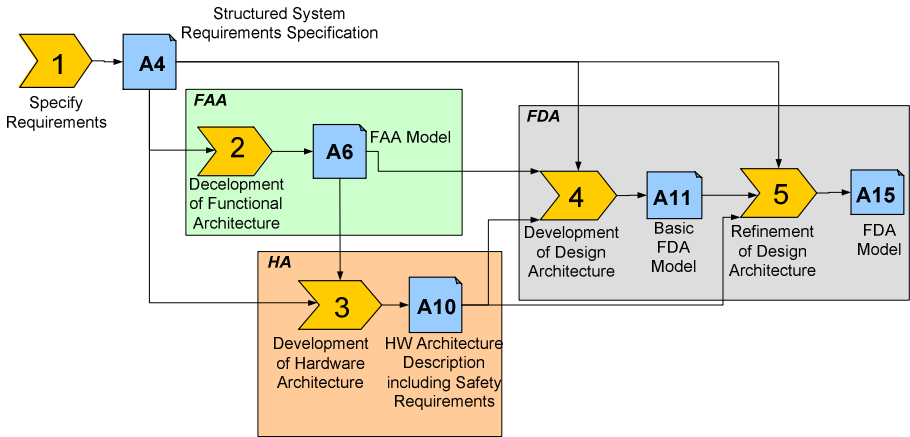


Fig. 9. The figure depicts the major process steps and artifacts of the EASIS engineering process

Although these 5 parts are presented sequentially, they are in practice iterated several times, as the design evolves and affects previous development steps.

Figure 10 shows an example system model that captures some of the items relevant for the EASIS engineering process. The rendering of each artifact may be adjusted depending on needs, e.g. tabular notations may be appropriate for Hazard lists and requirements and parts of the model may be viewed separately (e.g. per function, per ECU, per domain).

Below, the 5 parts of the EASIS Engineering Process will be discussed in relation to this example.

Part 1: Specify requirements

The EAST ADL System model has a Vehicle Feature Model to capture the intended functions and features of the vehicle. Requirements on these features, or on the entire vehicle, are recorded in the first step. The outcome of the “Specify requirements” part is a structured set of requirements, which includes both general requirements and those specifically concerned with safety.

Requirements are typically expressed in natural language or restricted natural language. They may also be refined with executable or structured models. In either case, requirements are associated to the target entity with an ADLSatisfy association. Requirements that are derived from other requirements are traced to the base requirement with the ADLDeriveReq association.

To identify Safety requirements, a Hazard Analysis may be performed. Hazards are recorded and linked to the related function or feature. Safety requirements identified to mitigate the Hazard are linked to the Hazard and to the related system entity.

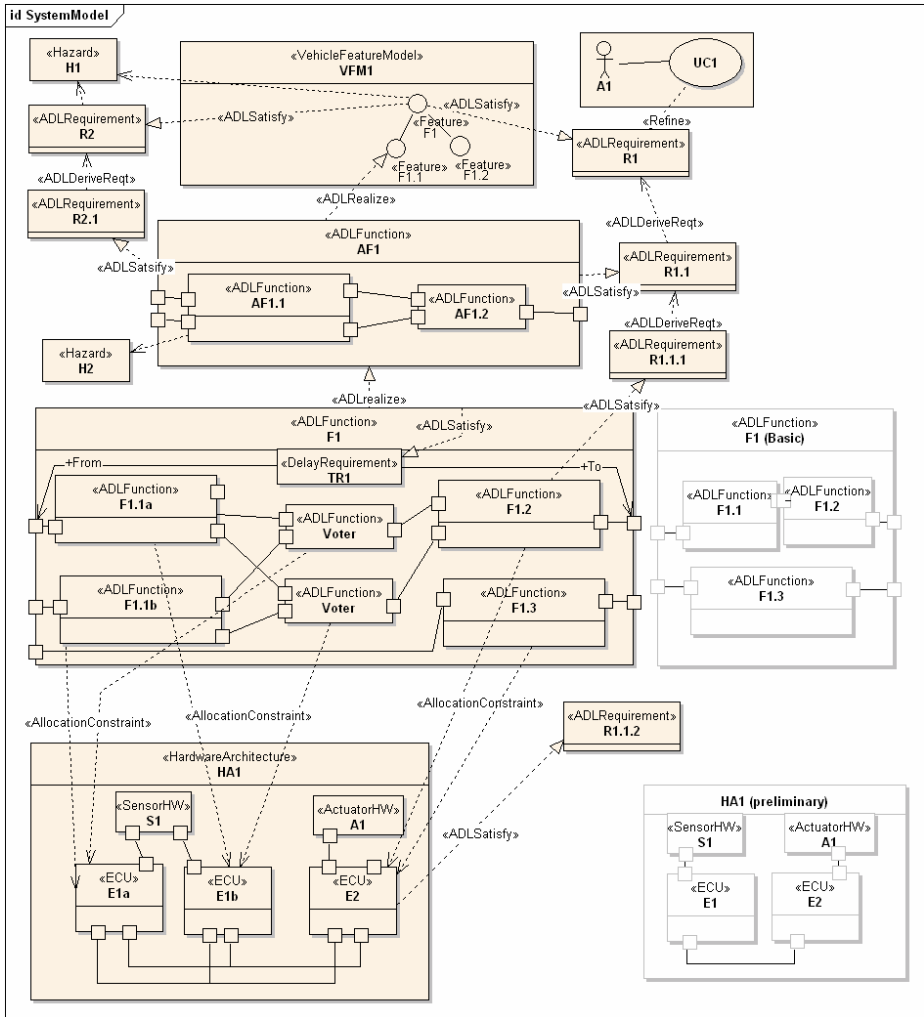


Fig. 10. Example model as it may evolve in the EASIS Engineering Process

Part 2: Development of functional architecture (FAA Model)

Based on the requirements identified in the previous part, a first step towards implementation can be taken. The functional architecture is an abstract description of the system, leaving out details and focusing on what the system should do. The FAA is specified in two steps, first specifying the functional architecture and then the functional behavior. This low-detail description is an important means to identify basic interfacing and interaction problems within and between systems at an early

stage. The ADLFunction AF1 in Figure 10 is an example of basic analysis model with associated requirements and an ADLRealizes association to the Feature it realizes. Sensor and actuator concerns are omitted from the example, except in the hardware architecture.

An FAA hazard analysis is performed in this stage to investigate the need for changes in the solution or further safety requirements. The FAA model is also verified to decide whether the requirements from Part 1 are satisfied. Analysis w.r.t. consistency, timing and formal properties is performed, and depending on the application domain and character of the models, simulation, prototyping, formal verification, inspection, etc. can be used. Examples of tools include Matlab/Simulink for control systems or statecharts for discrete systems. The list of the identified FAA blocks can also be used as basic input for dependability related activities (e.g. FMEA).

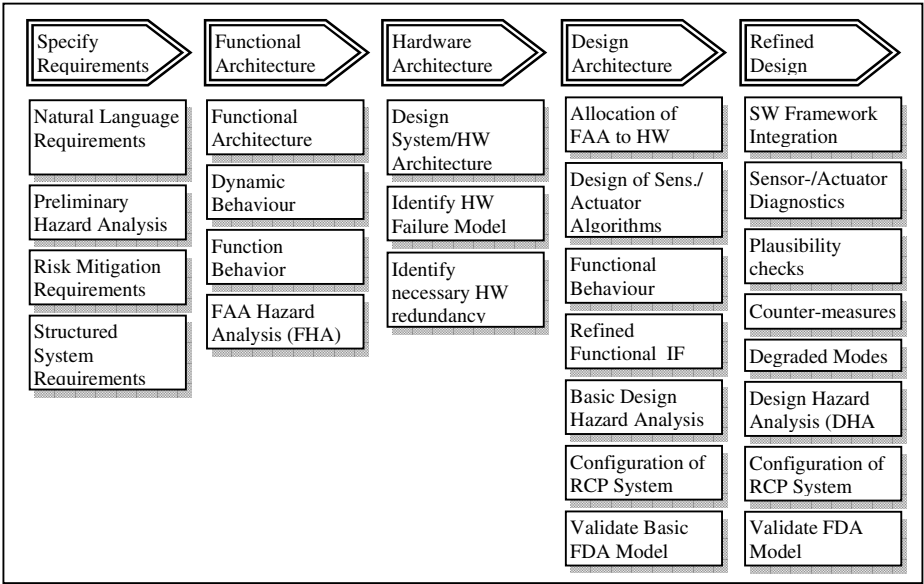


Fig. 11. The figure presents a summary of major and minor process steps

Part 3: Development of Hardware Architecture

Based on the system requirements and the FAA, a preliminary hardware architecture is defined, see preliminary HA1 in Figure 10. It includes e.g. electronic control units for computations, connections for data interchange and sensors and actuators for the abstract I/O defined in the FAA.

This hardware architecture is analyzed w.r.t. dependability requirements, leading to a description of necessary redundancy. Analysis will also identify the possible failure modes, and requirements on mitigation can be found, such as software and hardware redundancy.

Part 4: Development of design architecture (basic FDA model)

In this stage, a more concrete design of the functional aspects of the system is defined. The final allocation of functional components to the Hardware Architecture (as defined in the HA) is specified. Operating system concepts and platform services and its impacts on the functional architecture are added to the design. Domain specific services for diagnostics, network management, etc. are included. The behavior of each function is modeled under normal operation circumstances (no faults). The result is the basic FDA model, see F1 (Basic) in Figure 10.

Part 5: Refinement of the design architecture (FDA model)

With the basic FDA model as a basis, the handling of faulty hardware and signals is added in this step, giving a final FDA model which includes the safety concept of the function/system under development. The system is re-validated to cover the recent adjustment of the FDA. Figure 10, F1 is the final, redundant design architecture with allocation constraints defining the intended hardware allocation.

Figure 11 presents a summary of the process steps.

7 EAST-ADL Compliance with Standards

Safety through modeling techniques can only be achieved if the approach can be effectively used, which in turn relies on tools and mature concepts. Tool availability and validation of concepts is enabled by standardization or alignment with existing standards. Standardization of the EAST ADL is ensured in two contexts: AUTOSAR and the OMG.

The AUTOSAR platform is a future de-facto standard for Automotive embedded systems. It defines a set of middleware components that provides a standardized platform for application software. The modeling approach for application software components and hardware architecture contains the details necessary for correct integration. In ATESS, the entities corresponding to software components and hardware components are taken from the AUTOSAR standard, but put in a context where the EAST ADL system modeling concepts can be used. AUTOSAR compliant software architecture can thus be modeled with support for e.g. variability, requirements, traceability and verification and validation.

The OMG standards in the scope of EAST-ADL include UML2, SysML [27] and Marte (see below). The refined EAST-ADL will be aligned with these approaches. Alignment with UML2 is done by construction because EAST-ADL is designed as a UML2 profile. SysML concepts are re-used wherever applicable, for example regarding requirements and plant modeling constructs. Many of the EAST ADL concepts are thus reused SysML concepts, or specializations of these. Marte harmonization, finally, is done by integrating Marte concepts in the EAST ADL where real-time and embedded system properties are modeled.

The EAST-ADL2 language contains thus UML2 basic constructs, the requirement concepts from SysML, practical variability approaches for highly complex product lines developed from the automotive domain, function modeling from SysML, behavior from SOTA tools and UML2, error behavior from AADL, implementation modeling from AUTOSAR, and finally non functional properties from MARTE are reused.

The differences between SysML and EAST-ADL are the following: the SysML language is reused as far as possible, EAST-ADL providing for the framework/ontology to guide the use of SysML concepts in an automotive context. The SysML-based part of EAST-ADL2 is linked to the automotive implementation concepts from AUTOSAR and augmented with concepts from AADL and MARTE. Variability constructs and verification and validation constructs are further contributions beyond plain SysML.

Since Marte is work in progress, some background is provided here. Marte is an ongoing effort to define a standard on UML profile for Modeling and Analysis of Real-Time and Embedded system [9][12]. The background of MARTE, is that a consensus has emerged that, while a useful tool, UML is lacking in some key areas that are of particular concern to real-time and embedded system designers and developers. In particular, [13] noticed that firstly the lack of quantifiable notions of time and resources was an impediment to its broader use in the real-time and embedded domain. Secondly, the need for rigorous semantics definition is also a mandatory requirement for a widespread usage of the UML for RT/E systems development. And thirdly, specific constructs were required to build models using artifacts related the real-time operating system level such as task and semaphore.

Fortunately, and contrary to an often expressed opinion, it was discovered that UML had all the requisite mechanisms for addressing these issues, in particular through its extensibility facilities. This made the job much easier, since it was unnecessary to add new fundamental modeling concepts to UML – so called “heavyweight” extensions. Consequently, the job consisted in defining a standard way of using these capabilities to represent concepts and practices from the real-time and embedded domain. Yet special care had to be paid on the precise semantics definition of the profile itself. This was achieved by: 1) the explicitation of a domain language which gives the precise rationale for the constructs introduced, in a UML-independent fashion; 2) the resolution of all semantical variation points and ambiguities of the UML constructs used for the projection of this domain language onto a UML profile. The result is a UML profile with rigorous semantics, which, because it spans across a wide range of UML modeling elements, allows when used, to rely on a completely reassessed set of elements and thus gives a solid ground as far as semantics is concerned. Indeed such an approach should be the golden rule, but may not be adopted as often and as deeply as possible because of lack of time when defining a profile. This was made possible here by the definition of a fully devoted OMG consortium called the ProMarte consortium¹ who was responsible to answer the OMG request for proposal issued to add to UML modeling capabilities firstly for modeling Real Time and Embedded Systems (RTES), and secondly for analyzing schedulability and performance properties of UML specifications [25]. The profile called Marte is intended to replace the existing UML Profile for schedulability, performance and time [26]. Modeling capabilities have to ensure both hardware and software aspects of RTES in order to improve communication/exchange between developers. It has also to foster the construction of models that may be used to make quantitative analysis regarding hardware and software characteristics. Finally, it should enable interoperability between development tools used all along the development process.

¹ www.promarte.org

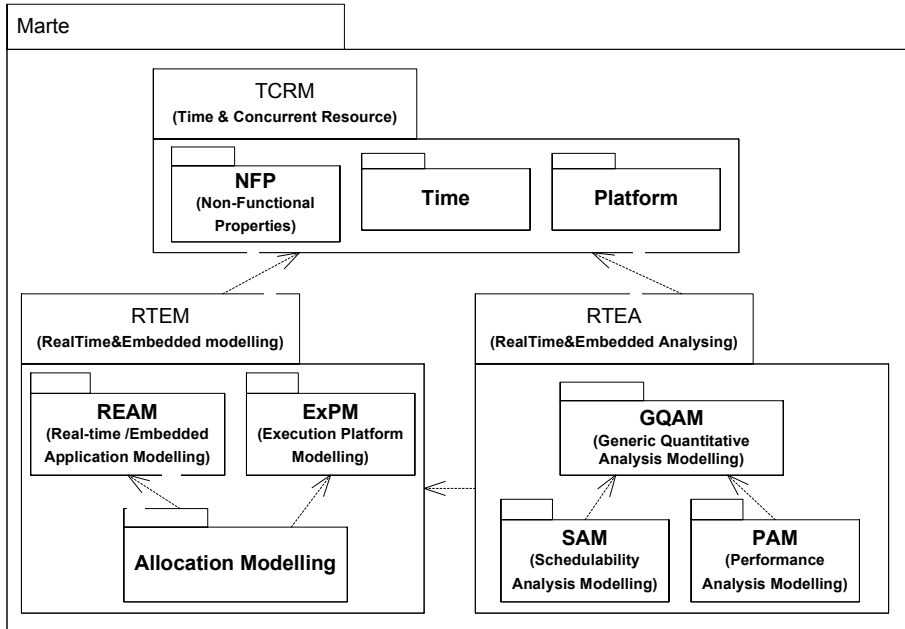


Fig. 12. Current architecture of the Marte profile

The alignment of EAST-ADL2 with MARTE will be centered on the expression of non functional properties – borrowing MARTE framework to rewrite elements currently defined by plain UML constructs – and the component model. The ProMarte consortium agreed to the introduction of an appendix section to the standard for the definition of EAST-ADL2. This section will deal with how to make EAST-ADL2 models using Marte, it will be initiated shortly after the vote of the initial release of the standard by the OMG and further refined in the context of the Marte Finalization Task Force (FTF).

8 Conclusions

Traffic safety can be improved in two fundamentally different ways. First, new technical systems may be introduced that directly address traffic safety by influencing traffic in some way, regardless of whether they are to be built into the vehicles themselves as active or passive safety systems or if they are to become part of the transport infrastructure.

The introduction of information technology in the automotive industries over the last decade, and the fact that electronic control units have become more and more inter-connected, has led to safety visions that were unimaginable fifteen years ago. But with such highly advanced, extremely complex functions – often spanning several classical sub-domains such as engine control and telematics – appropriate design methods and tools have become crucial. This is why a second approach is also needed

to increase traffic safety: Safety systems and technology should be complemented by improved development methodology.

The benefit of such additional perspective is threefold:

- By making the vehicles and the transport infrastructure more reliable in general – even with respect to vehicle features not directly related to safety – traffic will become safer. For example, if the failure rates of a motor control subsystem can be reduced by a certain amount, fewer breakdowns will impede the transportation infrastructure.
- Once development methods and tools are improved, completely new safety functions will become possible. For example, experts agree that dependencies between automotive sub-domains (such as motor control, chassis, telematics) are a significant source of errors in the electronics system, because they cannot be handled well during the development process. Therefore, many interesting functions for next-generation cars are currently not taken into consideration simply because they would introduce additional dependencies of this kind.
- The need for mechanical fall-back systems and the redundant design of safety-critical systems can be limited, provided design errors can be avoided, and component failures can be predicted or handled in an appropriate manner. The cost penalty of advanced systems is thus reduced, making them accessible in large volumes. In addition, vehicle weight and fuel consumption are reduced, making transportation more environmentally friendly.

This context gives the fundament for further research in model-based development to meet automotive needs. This is the overall context of the ATESSST project, which further investigates the definition of an architecture description language for automotive embedded systems. The result is a refinement of an existing approach, the EAST-ADL that was developed in the EAST-EEA project. The previous version of the EAST-ADL was mainly focused on the architecture-induced complexity of embedded systems. ATESSST extends this to adequately address the application- and environment-induced complexity. A further challenge in ATESSST is to consolidate EAST-ADL solutions and harmonize it with existing approaches in industry. The resulting language will be an ontology for automotive electronics and software, making system models unambiguous, consistent and exchangeable.

The purpose of this chapter was to show how the currently refined EAST-ADL – defined in the ATESSST project – provides extra support for modeling dependable embedded automotive systems, with emphasis on requirement specification and traceability, and support for dedicated engineering process for safety and standard analysis techniques.

In this area, the main contributions are the following:

- Modeling in general means that the engineering information is formalized and available for automatic analysis and synthesis. Modeling also enables visualization and information organization for enhanced understanding during manipulation and review. This is a way to secure correctness and thus improve dependability.

- The dependability is improved by adding an ADL which is tailored for automotive needs. It will thus be possible to apply in a way that fits existing processes.
- EAST-ADL2 integrates concepts from different approaches in a way that makes it possible to keep all information in one structure. The risk of inconsistencies and integration-related errors is thus reduced.
- Traceability between requirements and between requirements and design/implementation is a way to make sure all requirements are handled. Managing verification and validation information in the system model is a way to ensure that all requirements are satisfied in the configurations that correspond to produced vehicles. Both aspects are possible due to the integrated system model that EAST-ADL2 provides.
- Automotive systems interact with the dynamics of the vehicle and environment. Verification and validation with an adequate representation of the plant/environment is thus critical for the correctness and safety of realized functions. This is handled in a uniform way from early concepts to final implementation in the EAST-ADL2.

References

1. Åkerlund, O., Bieber, P., Boede, E., Bozzano, M., Bretschneider, M., Castel, C., Cavallo, A., Cifaldi, M., Gauthier, J., Griffault, A., Lisagor, O., Lüdtke, A., Metge, S., Papadopoulos, C., Peikenkamp, T., Sagaspe, L., Seguin, C., Trivedi, H., Valacca, L.: ISAAC, a framework for integrated safety analysis of functional, geometrical and human aspects. *Embedded Real Time Software*, Toulouse (2006)
2. Avizienis, A., Laprie, J.-C., Randell, B., Landwehr, C.: Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing* 1(1), 11–33 (2004)
3. Bozzano, M., Villaflorita, A.: Improving System Reliability via Model Checking: The FSAP/NUSMV–SA Safety Analysis Platform. In: Anderson, S., Felici, M., Littlewood, B. (eds.) *SAFECOMP 2003*. LNCS, vol. 2788, pp. 302–9743. Springer, Heidelberg (2003)
4. Bozzano, M., Villaflorita, A., Åkerlund, O., Bieber, P., Bougnol, C., Böde, E., Bretschneider, M., Cavallo, A., Castel, C., Cifaldi, M., Cimatti, A., Griffault, A., Kehren, C., Lawrence, B., Lüdtke, A., Metge, S., Papadopoulos, C., Passarello, R., Peikenkamp, T., Persson, P., Seguin, C., Trotta, L., Valacca, L., Zacco, G.: ESACS: an integrated methodology for design and safety analysis of complex systems. *ESREL*, Maastricht (2003)
5. Chudleigh, M.F., Catmur, J.R., Redmill, F.: A Guideline for HAZOP Studies on Systems which include a Programmable Electronic System. In: *SAFECOMP'95*, Belgirate, Italy, pp. 42–58 (1995)
6. Clarke, S.J., McDerimid, J.: Software Fault Trees and Weakest Preconditions: A Comparison and Analysis. *Journal of Software Engineering* (1993)
7. Clements, P., Northrop, L.: *Software Product Lines: Practices and Patterns*. The SEI Series in Software Engineering. Addison-Wesley, Boston (2002)
8. EASIS (Electronic Architecture and System Engineering for Integrated Safety Systems) URL: <http://www.easis.org>

9. Espinoza, H., Medina, J., Dubois, H., Gérard, S., Terrier, F.: Towards a UML-Based Modelling Standard for Schedulability Analysis of Real-Time Systems. MARTES Workshop at MODELS Conference (2006), available at <http://wo.uio.no/as/WebObjects/theses.woa/wa/these?WORKID=45427>
10. Espinoza, H., Dubois, H., Gérard, S., Medina, J., Petriu, D.C., Woodside, C.M.: Annotating UML Models with Non-functional Properties for Quantitative Analysis. In: Bruel, J.-M. (ed.) *MoDELS 2005*. LNCS, vol. 3844, pp. 79–90. Springer, Heidelberg (2006)
11. Fenelon, P., McDermid, J.A., Nicolson, M., Pumfrey, D.J.: Towards integrated safety analysis and design. *ACM SIGAPP Applied Computing Review* 2(1) (1994)
12. Gerard, S., Espinoza, H.: Rationale of the UML profile for Marte. Chapter of the book: *From MDD Concepts to Experiments and Illustrations*, pp. 43–52 (2006)
13. Gérard, S., et al.: Efficient System Modeling of Complex Real-time Industrial Networks Using The ACCORD/UML Methodology. In: *Architecture and Design of Distributed Embedded Systems (DIPES 2000)*, Paderborn University, Germany, Kluwer Academic Publishers, Dordrecht (2000)
14. Gorski, J., Wardzinski, A.: Deriving real-time requirements for software from safety analysis. In: *Proceedings of the Eighth Euromicro Workshop on Real-Time Systems*, pp. 9–14 (1996)
15. Hansen, K.M., Ravn, A., Stavridou, P.V.: From safety analysis to software requirements. *IEEE Transactions on Software Engineering* 24(7), 573–584 (1998)
16. ISO TC22 SC3 WG16 preliminary results for introduction of future Automotive standard ISO 26262 "Road vehicle - Functional Safety" (planned for 2008)
17. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature Oriented Domain Analysis (FODA) – Feasibility Study. Technical Report, CMU/SEI-90-TR-21 (1990)
18. Kang, K.C., Kim, S., Lee, J., Shin, E., Huh, M.: FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures. *Annals of Software Engineering* 5, 143–168 (1998)
19. Kehren, C., Seguin, C., Bieber, P., Castel, C., Bougnol, C., Heckmann, J.-P., Metge, S.: Advanced Multi-System Simulation Capabilities with AltaRica. In: *22nd Int. System Safety Conf. System Safety Society* (2004)
20. Kletz, T.: *HAZOP: and HAZAN: Identifying and assessing process industry standards*. 3rd edn. Washington, DC: Hemisphere (1992)
21. Lano, K., Clark, D., Androutsopoulos, K.: Safety and Security Analysis of Object-Oriented Models. In: Anderson, S., Bologna, S., Felici, M. (eds.) *SAFECOMP 2002*. LNCS, vol. 2434, Springer, Heidelberg (2002)
22. Leveson, N.G.: *Safeware: System safety and computers*. Addison-Wesley Publishing Company, Reading (1995)
23. Leveson, N.G., Cha, S.S., Shimeall, T.J.: Safety Verification of Ada Programs Using Software Fault Trees. *IEEE Software*, 48–59 (1991)
24. Lutz, R.R., Shaw, H.-Y.: Applying Adaptive Safety Analysis Techniques. In: *Proceedings of the 10th International Symposium on Software Reliability Engineering*, Boca Raton, FL (1999)
25. Object Management Group, UML Profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE) RFP, *realtime/05-02-06* (2005)
26. Object Management Group, UML Profile for Schedulability, Performance, and Time, Version 1.1. *formal/05-01-02* (2005)

27. Object Management Group, Systems Modeling Language (SysML) Specification, ptc/06-05-04 (2006)
28. Palady, P.: Failure Modes and Effects Analysis. PT Publications, West Palm Beach, FL (1995)
29. Papadopoulos, Y., McDermid, J.A.: Hierarchically Performed Hazard Origin and Propagation Studies. In: Felici, M., Kanoun, K., Pasquini, A. (eds.) SAFECOMP 1999. LNCS, vol. 1698, pp. 139–152. Springer, Heidelberg (1999)
30. ProMarte consortium, Joint UML Profile for MARTE Initial Submission, realtime/05-11-01 (November 2005) available at, <http://www.omg.org/cgi-bin/doc?realtime/05-11-01>
31. Reiser, M.-O., Weber, M.: Using Product Sets to Define Complex Product Decisions. In: Obbink, H., Pohl, K. (eds.) SPLC 2005. LNCS, vol. 3714, pp. 21–32. Springer, Heidelberg (2005)
32. Reiser, M.-O., Weber, M.: Managing highly complex product families with multi-level feature trees. In: Proceedings of the 14th IEEE International Requirements Engineering Conference, RE, pp. 146–155. IEEE Computer Society Press, Los Alamitos (2006)
33. Rushby, J.: Critical system properties: Survey and taxonomy. Reliability Engineering and System Safety 43(2), 189–214 (1994)
34. ARP-4761, S.A.E.: Aerospace recommended practice: guidelines and methods for conducting the safety assessment process on civil airborne systems and equipment. 12th edn. SAE, 400 Commonwealth Drive Warrendale PA United States (1996)
35. Storey, N.: Safety-Critical Computer Systems. Addison-Wesley, Reading (1996)
36. Sullivan, K.J., Dugan, J.B., Coppit, D.: The Galileo fault tree analysis tool. In: Proc. of the 29th Annual IEEE International Symposium on Fault-Tolerant Computing, pp. 232–235. IEEE Computer Society Press, Los Alamitos (1999)
37. Tessier, P., Gérard, S., Terrier, F., Geib, J.-M.: Using variation propagation for Model-Driven Management of a System Family. In: Obbink, H., Pohl, K. (eds.) SPLC 2005. LNCS, vol. 3714, pp. 222–233. Springer, Heidelberg (2005)
38. Vesely, W.E.: Fault Tree Handbook, US Nuclear Regulatory Committee Report NUREG-0492, US NRC, Washington, DC (1981)

The View Glue

Alek Radjenovic and Richard Paige

The University of York, Department of Computer Science, Heslington, York, YO10 5DD
{alek,paige}@cs.york.ac.uk

Abstract. In this paper we focus on domain-specific Architecture Description Languages (ADLs), particularly for safety critical systems. We argue that existing standards for architectural modelling are insufficient for achieving the necessary levels of control of the development process for such systems. We outline the requirements for safety critical ADLs, the challenges faced in their construction, and present an example - AIM - developed in collaboration with the safety industry. Explaining the key principles of AIM, we show how to address multiple and cross-cutting concerns through active system views and, how to ensure consistency across such views. The AIM philosophy is supported by a brief exploration of a real-life jet engine case study.

Keywords: architectural views, view consistency, software architectures, modelling, safety critical systems.

1 Introduction

There is wide-spread recognition that software systems are becoming increasingly large and complex. Such systems are inherently difficult to manage, understand, design, implement, test and maintain. This is partly a direct consequence of the number of features software is required to offer.

Moreover, software features are exposed to frequent requests for modification, largely due to changes in customer requirements. For real-time embedded systems, hardware obsolescence presents a great challenge.

As a large-scale system is developed, a number of deliverables are produced, e.g., requirements specifications and design documents. Changes impact on each deliverable, which should ideally remain consistent. However, changes that occur later in the lifecycle will have wider impact, thus increasing the cost of change. In large and complicated or complex software systems the cost of change is disproportionately larger than the size of change.

Recently, two approaches have begun to show promise in tackling the challenges associated with managing the size and complexity of modern complex software systems: software architectures and modelling.

Architecture is broadly defined as an organisation of a system describing its components, their relationships, the system's environment and the rationale behind such an organisation [1]. Research into architectures attempts to enable a better understanding of large and complex software systems, facilitate predictable reuse of its parts (components), assist in design and construction, support evolution, enable (formal)

analysis and, help overall management by clearer understanding of requirements, implementation strategies and potential risks

Modelling is a process that employs abstraction and focuses on higher-level system constructs [2]. Thus, a model is a representation of a system's essential structure and properties, obtained by omitting detail deemed unnecessary for a particular modelling method. Research into modelling complements that on architecture. Indeed, modelling is regarded as a vehicle necessary for describing an architecture.

Consequently, we have witnessed the emergence of numerous precisely defined languages to aid in systems modelling and creating architectural specifications. These notations are more commonly known as architecture description languages (or, ADLs). Although almost all ADLs have their own merits, they tend to focus on a particular, fairly narrow, aspect of system design and have not had much impact in being accepted in everyday engineering practice [3].

A key technique used when building models is layering of abstractions, e.g., constructing a model of a network, and atop it constructing a model of a protocol, which uses the network model. Most modelling approaches limit high-level design to one or two abstraction levels. Next generation ADLs should support multiple abstraction layers. This will let developers refine higher-level constructs into lower-level, more detail-rich artefacts that safety-critical systems require. Although we have ways of understanding function, state, and structure refinements, we need a mechanism that lets us refine other non-functional and diverse concepts while preserving traceability.

Architectural (or model) views must become a primary design mechanism. Views aren't novel-prominent approaches include the 4+1 View Model and the IEEE 1471 standard. However, the rigidity of the 4+1 View Model limits its support for specific engineering practices. More importantly, to be sufficiently rigorous in the HIRTS (High Integrity Real Time Systems) arena, view consistency is essential and neither approach provides it. Changes to the artefacts in one view need to be automatically applied to the relevant artefacts in another view [4].

All known modelling platforms today fail to ensure consistency across its set of views. As a result, we cannot have sufficient guarantees about the consistency and coherency of the system as a whole, even in cases when the analyses performed on partial models returns satisfactory results. In this paper, we present an approach - a modelling platform called AIM (Architectural Information Modelling). We argue about the importance and role of dependency links in modelling dependable systems and, demonstrate their role as a reliable '*view glue*' in ensuring architectural view consistency. Our argument is illustrated by examples from a real-life jet engine case study provided by Rolls Royce plc.

In section 2, we introduce the key principles of the AIM platform, followed by a brief description of the case study in section 3. Next, in section 4, we illustrate how to apply the modelling concepts of AIM to the presented case study. In the last section, we summarise the potential benefits of our approach in the design of safety critical systems.

2 AIM in a Nutshell

AIM is a generic, extensible platform for modelling software and systems in multi-team environments with particular focus on the HIRTS domain.

2.1 Top Level Organisation

At the top level, AIM is greatly similar to (and thus has been aligned with) the traditional 3-tier RDBMS (relational database management systems). Organising AIM in this way has the benefits of using proven and mature technologies (such as RDBMS, SQL, XML, PHP, etc) for the tool implementation making it straightforward, modern and cross-platform.

The traditional 3-tier database system comprises of the Data Layer, Business Logic and, Presentation Layer. The Data Layer holds the actual data, organised into various tables which are in turn linked together through declarations of dependency relationships. The Business Logic provides the rules according to which diverse information can be added, viewed, updated or removed from the tables and, by which groups of users. It can also provide a mechanism to synchronise access to the database by multiple users. The Presentation Layer makes sure that the interface for data manipulation is easy to use and includes filtering capabilities, i.e. allows the end user to deal only with a subset of data at any one time.

The three layers in AIM - Data, Rules and, Views - follow the same logic as described above. The AIM Data layer contains the model information such as the structure, the functionality or the data flow within the target modelled system. The Rules layer describes the dependency links, constraints and/or various design rules. The Views layer (which corresponds to the Presentation layer in the RDBMS) provides each of the users (or, each of the groups of users with the same set of concerns) with a view-driven design environment. Figure 1 below illustrates the top level organisation of the AIM platform.

The 3-layered approach described above provides clear separation of information that describes the system and restrictions imposed on its development and operation. It facilitates formalism with a very small set of syntactic and semantic rules. In addition, its view-driven development approach provides a platform for a straightforward application of various kinds of analyses, such as model checking, resource allocations and budgets, circularity, required connections, schedulability, priority inversions, and flow latency.

2.2 Data Layer

The AIM Data layer is the key repository of the information contained in the model. It is further divided into four abstraction sub-layers: *Core Notation*, *Language Factory*, *Modelling Language*, and *Model*. This was necessary because AIM was created to support multiple design and implementation languages and notations, including the programming languages.

By definition, a model is a representation of the essence of an artefact. During the process of modelling we omit detail. Consequently, models are necessarily incomplete and there is always a trade-off between the explanatory power and the complexity. Needless to say, all modelling technologies will omit differing levels – differing kind and differing amount – of detail.

Figure 2 illustrates this division into sub-layers. It is important to note that AIM defines only the two most abstract layers, the *Core Notation* and the *Language Factory*, in a precise manner. Various modelling languages (or their AIM representations)

are custom-defined in the Language Factory notation and will depend on the system being modelled, user preferences, or a specific company policy. The models, such as the Architectural, AADL and Ada in the picture below, are parts of a single integral AIM model which describes the target system through multiple levels of abstraction. This further allows multiple stakeholder groups (such as control engineers, system architects or, developers) to work under the same (AIM) umbrella simultaneously.

AIM Model

Traditional 3-tier RDBMS

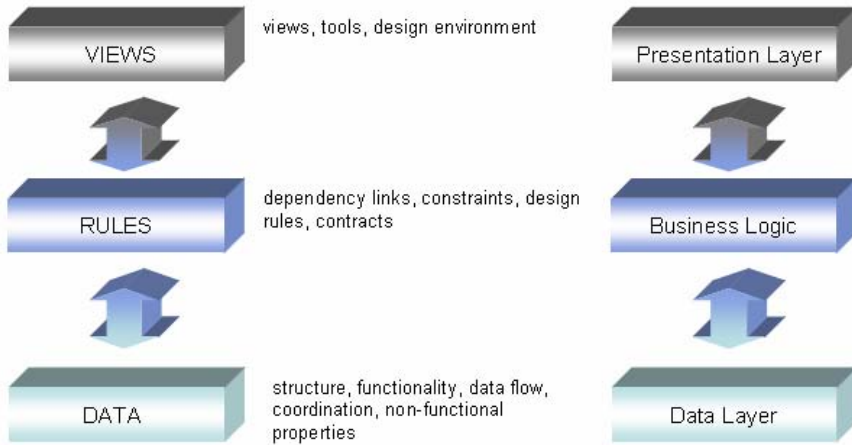


Fig. 1. Top level organisation of the AIM platform

AIM Core Notation, the most abstract of the four sub-layers, is based on and aligned with two key meta-models: the MOF (Meta Object Facility) defined by OMG, and Ecore which is the (meta) model used to represent models in EMF (Eclipse Modelling Framework, www.eclipse.org). MOF 2 is closely aligned and borrows from the UML Infrastructure 2.0 [5]. The main intention in defining the AIM Core Notation as a subset of MOF and Ecore was to enable data exchange and model transformations between AIM models, on one hand, and UML models and tool implementations in Eclipse, on the other.

AIM Core Notation is a *reflective* notation just like MOF or Ecore, i.e. it can be used to define itself, and so there is no need to define another language to specify its semantics. This is achieved by propagating the metadata through to the instances (objects) of the modelling constructs itself allowing the discovery and manipulation of metadata, as well as the use of objects, without prior knowledge of the objects' specific features.

The next level in the abstraction layers stack, the **Language Factory**, is an instance of its meta-model – the Core Notation. In other words, by using the modelling artefacts defined in the Core Notation, we are able to create the Language Factory layer.

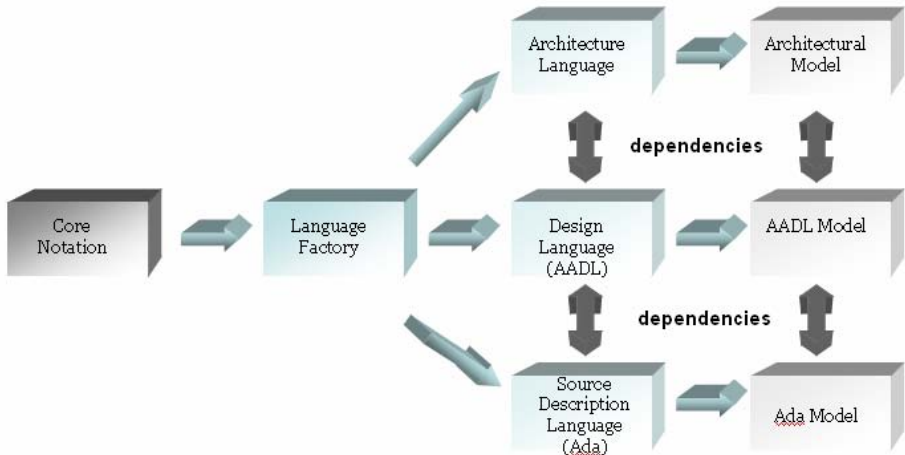


Fig. 2. The four sub-layers of the AIM Data layer

The design philosophy driving the organisation of this layer was to provide a common ground to most, if not all, modelling languages and notations. The hypothesis is that every model would need to describe some, or all, of the following aspects of a software system:

- structure
- functionality
- data and control flow
- coordination
- non-functional characteristics

Accordingly, we have defined a modelling construct to describe each of the five aspects above, as follows: *Component* for the structure, *Service* for the functionality, *Connection* for the data and control flow, *Coordination* for the coordination and, *Property* for the non-functional characteristics. Each of these is explained next in some detail.

At the top level of the Language Factory meta-model is the *model element* (Figure 3). It is composed of zero or more collections of properties, data types and components, organised (or, packaged) in their corresponding sets (each having a name and an optional namespace classifier).

Each *collection* is tagged with a name and a namespace, creating a unique, global footprint for every model element that is defined. This arrangement supports modularity, low coupling and high cohesion i.e. the component based approach to design. Later, we demonstrate how trusted components and composable architectures are supported by the 3-tier design described in an earlier section.

The AIM *data type system* predefines several basic types such as – integer, float, string, Boolean, reference and URI (uniform resource identifier). With the provision of the built-in template mechanism, additional, more complex, derived data types can be defined inside custom-made type sets. The templates provided include: enumerations, ranges, sets, arrays and records.

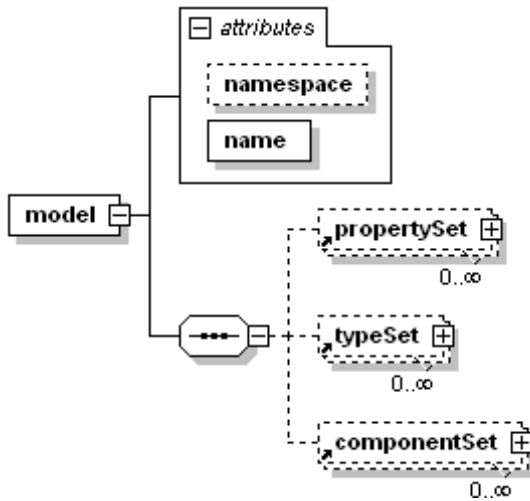


Fig. 3. AIM Language Factory - top level

Each property set is a collection of *property definitions*. A property definition includes a name and a type reference at a minimum. By default, a property can be attached to *all* model constructs (other than types or properties, obviously) unless its *scope* is restricted by the definition. Besides scope, other optional features can be specified. These include: default values, units and, whether assigning a value is required or not. It is worth mentioning here that AIM defines two special literal values: UNDEF and NULL. The former indicates that no value is assigned, and the latter represents a *zero-value* (which has different meanings for different types: 0 for integers, empty string for strings, false for Boolean, etc).

Next, we go into some detail describing the modelling elements that represent, in our opinion, the five ‘pillars’ of software design: structure, functionality, data and control flow, coordination and, non-functional properties.

The **Component** is the top level modelling construct which describes the *structure* of the target system. Components, like types and properties, are organised in their corresponding collections – component sets. They are further broken down into five specialised collections: *Services*, *Subcomponents*, *Connections*, *Coordination* and *Properties*.

Component **Services** describe the desired *functionality* of the component (i.e. its relationship to its environment) without going into detail of how such functionality is achieved. All services are grouped in the component’s *interface*. Most commonly the services will describe the functionality that the component provides. In addition, services required *by* the component can also be specified.

The **Connections** describe the data and control flow inside the system. In its simplest form, a connection is represented with two endpoints (attached to a component or a service) and a connector between them. In order to allow composition, a single connector can be attached to multiple endpoints representing more complex data flows.

The **Coordination** section of the component deal with state machines as well as modal operation of the component. In this (optional) section one or more states (or, modes) are named, where the first one in the list represents the initial state. Each state can also specify a list of components that are active in that state. If two or more states are named, then one or more transitions between the states can also be specified. A component can switch from one or more (source) states to a different (target) state when one or more events occur. The user can also specify one or more actions to take place before the switch between the states occurs.

The **Properties** section of a component lists zero or more *property assignments*. A property assignment associates a property (defined inside one of the property sets, mentioned earlier) with a value. In addition, one or more of the component's constituent parts (elements) can be associated with the property in order to support batch assignments. If no elements are specified, then the property is associated with the component itself. The types of elements must match one of the types declared in the *scope* section of the property declaration, and the literal value must be of the same type as the property itself.

2.3 Rules Layer

The Rules Layer in AIM provides a platform for describing the dependency links. This layer is vital in expressing the relationships that are cross-cutting and orthogonal to the ones defined by the meta-models. The presence of this layer enables clear separation between the information artefacts and their mutual dependencies. The Rules Layer represents the key traceability mechanism, enforces model integrity, defines inter-language mapping (transformation) and, provides support for change control.

We recognise two distinct types of dependency links: *implicit* and *explicit*.

The *implicit* dependencies are those implied by the make-up of the meta-models. These include:

- **Reflective:** types of elements, which correspond to types defined either in the meta-model or in one of the (data) type sets; also, in case of refinement or extension, types of ancestors and descendants (e.g. Components defined in the Language Factory meta-model are derived from Containers in the Core Notation meta-model).
- **Content:** relationship between containers and their contents (e.g. Components and their Subcomponents, Connections or, Services)
- **View:** automatically derived from view definitions (see View-driven Design section below)

The *explicit* dependencies are those that need intervention from designers and have to be defined explicitly. We categorise them in three groups:

- Constraint Logic
- Design Rules
- Transformation Rules

AIM modelling is essentially a component-based approach. The **constraint logic** addresses the compositional aspects of the design. For example, a networking server component is designed to operate in the presence of multiple client components which

can be attached to it. However, the sum of the maximal throughputs of the clients must not exceed the throughput of the server. Defining a constraint on the server component would prevent inadvertent over-allocation of the networking resources. Thus, the constraint logic helps to control predictable but undesired emergent behaviour.

The **design rules** govern the way model elements are designed. They enforce the architectural design decisions and prevent the design erosion. For example, a design rule may enforce the use of data types only from a particular *type set*, or from a particular *namespace*. Another example could be to ensure that all network enabled model elements communicate through a particular *server* component and not directly with each other.

The **transformation rules** define mapping between elements of two languages defined in the AIM Language Factory meta-model. Two languages focus on differing levels of abstraction and one-to-one mapping is possible only for a very limited number of artefacts. An example is given in Figure 4. In MASCOT language [6], sensors and clocks are modelled as *components*. In AADL [7], sensors are normally modelled as *components*, and clocks are (temporal) *properties*.

The capability needed in the Rules Layer in order to describe the explicit dependency links is based around a *common platform*.

Most importantly, we have to be able to refer to *any* model element or a list of elements. By this we mean both the meta-model artefacts as well as their instances in the model. Because all meta-models and model instantiations are expressed in XML, we use XPath [8] for this purpose. XPath is a language for finding information in an XML document. It is used to navigate through elements and attributes in an XML document. XPath includes a myriad of built-in functions for string values, numeric values, date and time comparison, node manipulation, sequence manipulation, Boolean values, and much more.

The AIM platform was designed to be extensible (this was demonstrated for the Data Layer earlier). AIM adopts the same kind of philosophy for the Rules Layer, too. Taking advantage of XPath, AIM provides an open and extensible platform for specifying constraints, design rules and language transformations.

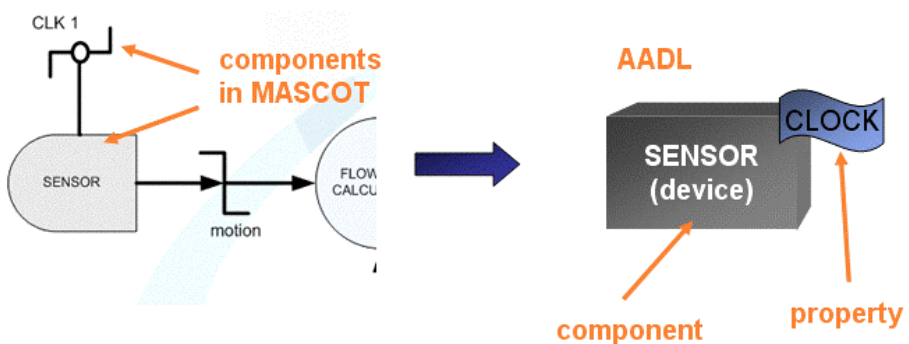


Fig. 4. Inter-language dependencies

The Rules Layer defines two basic kinds of rules:

- Boolean rules
- Generic dependency rules

Boolean rules are composed of AIM Boolean functions, XPath expressions, other Boolean rules and the standard (AND, OR and NOT) logic operators. XPath expressions are used as arguments (parameters) to the AIM Boolean functions which all return a Boolean TRUE or FALSE. Combining Boolean rules with AND, OR and NOT operators more complex rules can be formed. Rules can be attached to an element in order to define scope. Such rules will be checked for True value by an AIM compliant tool. Unattached rules will not be checked; for example, complex rules are often broken down into smaller and simpler sub-rules which can be reused and do not need to be attached to any particular element. Their scope is resolved from the top (i.e. from the parent rule(s)).

In the example below, a design rule named *Disconnected* is composed of rules *HasChildren* and *HasConnections*, both of which use an internal Boolean function *Empty()*, as well as the standard logic operator *NOT*. While *HasChildren* is an unattached rule, the other two rule (*HasConnections* and *Disconnected*) rule will only be performed on the elements of type *component*. Of course, all rules defined here are reusable and can be used to form other more complex rules in the model.

```
<rule type="bool" name="HasChildren" param="object">
  NOT Empty(object/)
</rule>
<rule type="bool" name="HasConnections" scope="component" param="object">
  NOT Empty(object/Connections/)
</rule>
<rule type="bool" name="Disconnected" scope="component" param="object">
  NOT (HasChildren(object) AND IsConnected(object))
</rule>
```

Both the *constraint logic* and the *design rules* use Boolean rules as their underlying mechanism. In addition, Boolean rules are optionally tagged by the *kind* label (of string type) to further categorise the rule. Predefined kinds include: PRECONDITION, POSTCONDITION, INVARIANT and, DESIGN, but more can be added in a straightforward fashion.

A **generic dependency rule** provides a non-specific association between model and meta-model artefacts. For each such rule we define:

- source artefacts (XPath expression)
- target artefacts (XPath expression)
- link type
- body

The *link types* are descriptive tag which categorise different kinds of cross-cutting dependencies. External tools use these tags to extract relevant information for further analysis. Body is of type TEXT and is an optional part of the declaration. It can be

used to store further information about the dependency and is normally processed by an external tool. For this reason, the format of the body text is not specified by AIM.

An example of a generic dependency rule is given in Section 4.3.

Transformation rules are instantiations of generic dependency rules. Their link type is a reserved keyword XFORM. The body of the XFORM generic dependency rule describes in detail how source artefacts are transformed into target artefacts.

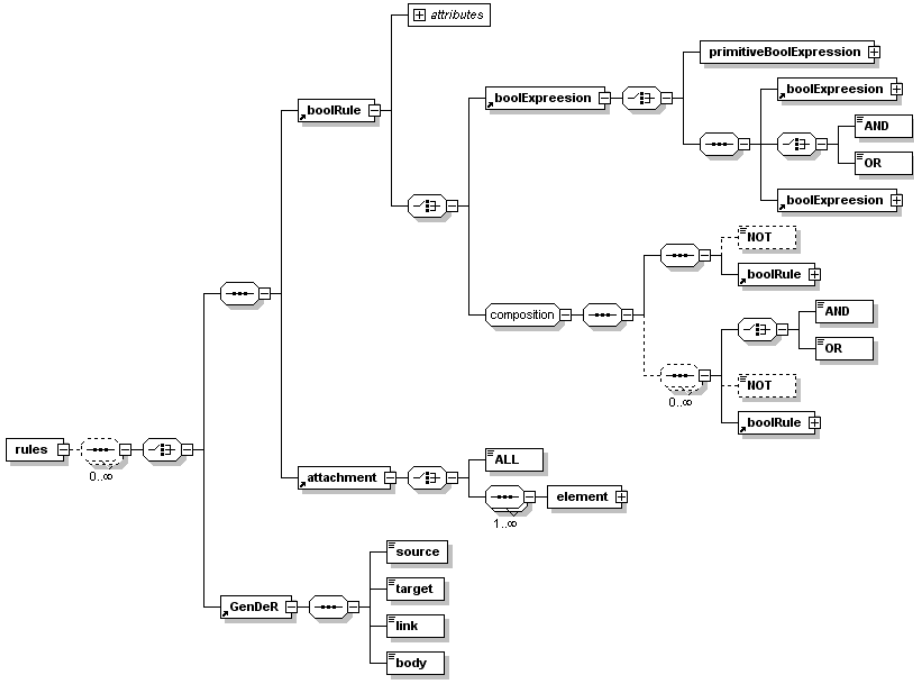


Fig. 5. Rules in AIM

2.4 Views Layer

The Views Layer contains a collection of user-defined *views* of the modelled system and/or its parts. Five key issues had to be addressed in order to describe this layer. They are:

1. **View definition:** what is a view?
2. **View template:** what comprises a view in general terms and how do we define a *mechanism* by which to develop a view?
3. **View instantiation:** how do we *extract* the necessary information from the underlying model database in order to populate the view?
4. **View presentation:** how do we *present* the extracted information?
5. **View-driven design:** how do we allow users to actively use views as their primary means of system design?

Next, we delve into each of these issues and give detailed answers to the questions raised.

The ‘conventional’ perception of an (architectural) view and the common practice of defining it can probably best be described one of the most encompassing definitions which states that the view is “*a representation of a whole system from the perspective of a related set of concerns*” [1]. Indeed, the two of the most widely accepted view-based design methods, “The 4+1 View of the System” [9] and the “Siemens Views” [10], fit well with this definition. However, such definition states *explicitly* that the view takes into account the *whole system*. This is the natural consequence of the research at the time. Views in these two methods are created separately, much like the diagrams in UML. Considering the whole system in each of the views was thought to be a sufficient condition for the consistency of the views across the system. This proved not to be the case, as illustrated by the significant body of work on consistency checking in, e.g., UML.

In AIM, we take a somewhat different approach. We believe that the Data and Rules layers ensure sufficient cohesion of the data within the model that the projection of the system as a whole is not necessary within any of the views. On the contrary, apart from very few people involved in the development of the system (such as the system architects), in real world many of the other stakeholder groups will *not want* to view the whole of the system and yet, their contribution to the system development can hardly be called negligible. Most users will more likely be interested in just a part of the system (in which they have the expertise) and its interface to the rest of the system.

Although our approach to views is in many ways substantially different from the conventional approach, our **view definition** differs only slightly from that in IEEE 1471:

- *A **view** is a representation of a whole system, or its part, from the perspective of a related set of concerns.*

There are two kinds of **view templates** we can define in AIM.

The first kind, called *language view templates*, is used for defining views within a model specified in a particular AIM language. The language view template declaration consists of three distinct declarations:

1. *language*: one of the languages specified by the Language Factory meta-model
2. *scope*: part of the system that is the view’s focus of interest
3. *viewpoint*: a set of meta-model artefacts from which the view is developed

Views are not meant to be a cross-language design mechanism. This would be similar to a text in English interspersed with French and German phrases, and written in Arabic alphabet in places. A view template needs to specify precisely which language it is to use.

By declaring the *scope*, the users specify which part of the model they are interested in exploring and/or modifying. Here, they can either select one or more from a list of already declared components or use the keyword MODEL to view the modelled system from the top. Normally, system architects would use the latter option.

A *viewpoint* represents a set of artefacts from the specified language which should appear in the specified view. Naturally, *all* information present in the model should be available for the view creation. This is only to be expected since the view is the primary design mechanism as explained later in the text.

If the view's scope is declared with the keyword **MODEL**, then the viewpoint declaration has to use one of the three keywords: **PSET**, **TSET** or, **CSET**. They indicate that the content of the view will either be the property sets, the type sets or, the component sets.

Most of the time, however, a view will be specified with its scope pointing to a component or a collection of components. The viewpoint declaration in this case is only slightly more complex. Again, we use the XPath expressions as a filter to extract a subset of the available information from the Data Layer that is of interest to a particular group of stakeholders.

By specifying multiple, incongruent components in the *scope* declaration, and by the powerful filtering capabilities of the *viewpoint* declaration, we are able to address all classes of cross-cutting concerns, ranging from e.g. the safety engineers' perspective of the system to the e.g. source version control system's.

Users are not restricted to any particular set of views. All views in AIM are custom made, much like the queries into a standard database. Figure 6 suggests some of the kinds of views possible in AIM.

Finally, because viewpoint declarations are language dependent and model independent, once declared they can be saved and reused across multiple projects. Similarly, through a change of the *scope* declaration (which is model dependent, except when using the **MODEL** keyword), the view templates can also be reused.

An example of a view template definition is given below:

```
<viewtemplate type="language" language="AL" scope="package">
  <viewpoint>
    <item name="component"/>
    <item name="interface"/>
  </viewpoint>
</viewtemplate>
```

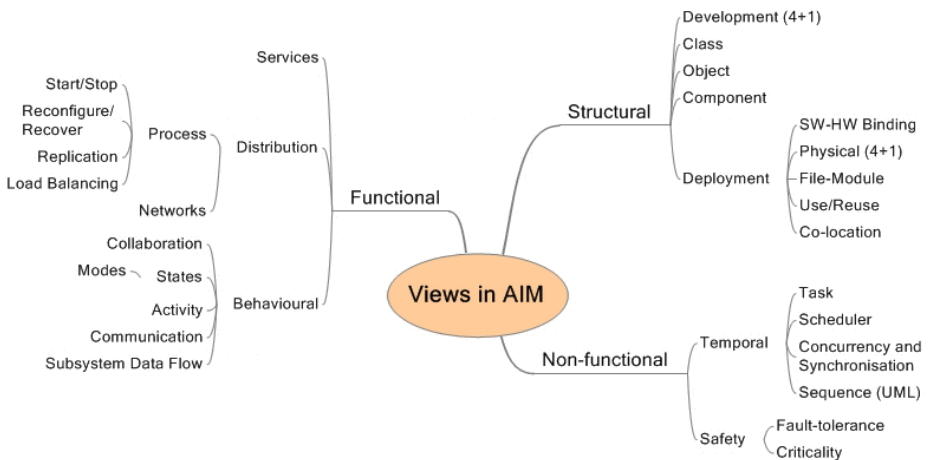


Fig. 6. An example of views possible in AIM

Here, we are interested only in the *package* elements and their subparts, and would like to display the components and their interfaces defined in the *architectural model* (described in the Architectural Language, or AL – see Section 4.1 below).

The other type of a view template is called a *rule* view template. Views defined using this mechanism allow us to specify concrete values for placeholders in the source and target XPath expressions of a generic dependency rule. We do this by ‘attaching’ a rule between model artefacts, and as a result, instantiate a generic rule. This is a cross-language type of view meaning that, in this case, model artefacts *can* come from different AIM languages.

A view in AIM is an *instantiation* of an AIM view template. With the help of an AIM compliant tool, the display of a selected view (based on the chosen view template) is a three-stage process:

1. **Sub-model selection:** the tool selects the Data Layer section that represents a model described in the language named in the *language declaration* of the view template
2. **Scope selection:** the tool narrows down its search space by targeting only those sections of the sub-model that match the *scope declaration* pattern
3. **View population:** XPath expressions specified in the viewpoint declaration are applied to the selected search space in order to extract the matching model information.

Initially, of course, all views are empty. As the development of the system evolves, more artefacts are added to each of the views. These artefacts represent real elements in the final system model (the right-most blocks in Figure 2).

Potentially, there may be considerable diversity of possible views in AIM. Hence, we do not recommend any particular approach to **view presentation**. Possible options are tabular, or tree-like approach, or a mixture of the two - which have been implemented in our prototype tool. Other alternatives include translation to UML diagrams. During our research, we have shown that this is possible; however, work in this direction was not pursued for two reasons. Firstly, there are significant differences in the XML representation of diagrams in various commercial UML tools and, we lacked the resources to investigate them. Secondly, and more importantly, views in AIM were devised to be first class citizens and a primary tool for model specification. This is explained in some detail next.

Views in AIM are its primary design mechanism. Although model modifications *are* possible through direct editing of the XML text, an AIM tool is the more advisable option. Clearly, one-way flow of information – from the model database to a view displayed by a tool – is not sufficient. In a **view-driven design** environment, any model manipulation through views must be propagated back to the model database (the Data Layer). This means that addition, deletion or modification of model artefacts is as important as the display of such information.

In AIM, the synchronisation of views with the model database, i.e. the synchronisation between the Data Layer and the Views Layer, is achieved in the Rules Layer. View dependencies are automatically defined from view templates. For example, model artefacts belonging to two or more views will be tagged with an *overlap* flag. This mechanism ensures that any modifications to the artefacts made in one view, are propagated to the other view(s) and that the owners of those other views are notified

of the change. It is also possible to flag these artefacts as *read-only* (either temporarily or permanently): thus, although they are part of more than one view, they can be modified in only one of them.

In conclusion, view consistency is ensured by two things:

1. Single model repository
2. View dependencies

3 Case Study

The case study presented here is an electronic engine controller (EEC) software for the TAY jet engine manufactured by Rolls Royce plc. The case study material was provided in two parts: the top level software architecture documentation and the source code in SPARK Ada 83 programming language.

3.1 TAY Software Architecture

The documentation provided is written in English narrative style. It describes the rationale behind the design decomposing the entire software system into smaller subsystems and Ada packages. Arguments are supported throughout with a series of ‘box and line’ diagrams adhering to no particular graphical notation.

At the top level, the authors address the following three issues:

1. The *scope* diagram shows the decomposition of software into three major areas and indicates the primary source of software requirements for each (Figure 7).
2. The *context* diagram explains the environment with which the software will interact/communicate (Figure 7).
3. The *top level design* diagram depicts the design philosophy breaking down the software system into a series of subsystems (Figure 8).

This introduction is followed by a scrutiny of various aspects of the target system, such as the functionality, data flows and package dependencies.

For example, the functionality is described with a diagram shown in Figure 9. Thirteen major functional groups are recognised (each of which is to be further split into multiple packages in the final implementation) as well as the key types of communication between them.

This is followed by the hardware support ACG (application communications graph) (Figure 10) which shows EEC software in the context of its hardware, their interfaces to each other and, dependencies in terms of control and data exchange between various software and hardware entities.

In addition to Functional and Hardware Support ACG’s, the authors also suggest a host of utilities to be used by all software packages. These include: Control, Data, IO, Timer and Validation utilities. Although these are described in a section called ‘Utilities ACG’, no graph is actually provided. An assumption could therefore be made that utility packages are not interdependent. It will be shown later, during model extraction from the source code that this is not the case

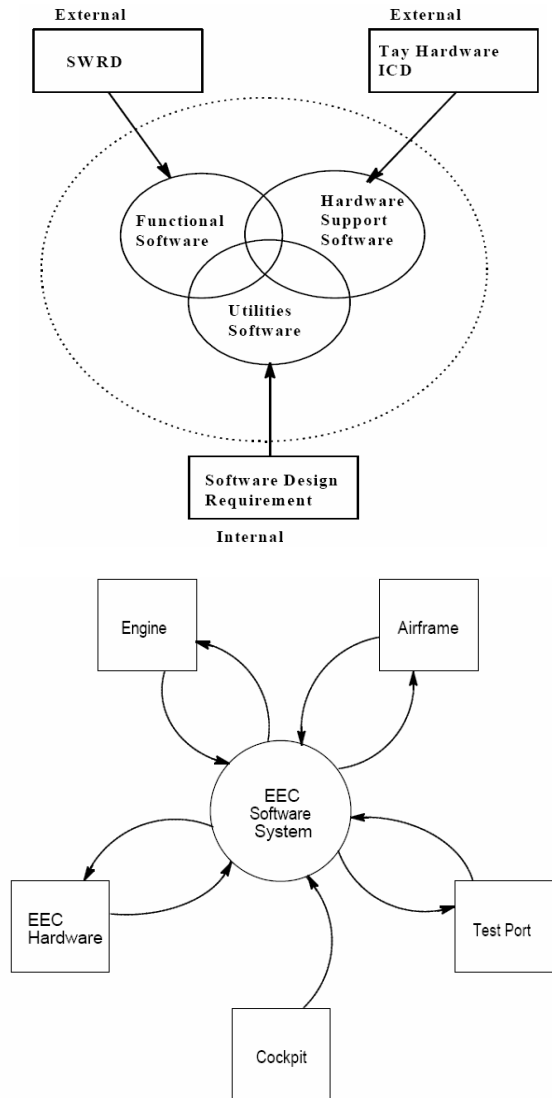


Fig. 7. TAY software: *scope* (left) and *context* (right) diagrams

What follows is a decomposition of each application into a set of packages showing major dependencies between packages within an application and with packages in other applications. However, this approach is somewhat inconsistent throughout the document. To explain this more clearly, an example is given in Figure 11.

Here, the decomposition and dependencies of the *Lane Control* application and its packages are presented. The diagram indicates application dependencies such as: *Starting and Shutdown* application depends on the *Lane Control* application which in

turns depends on the *Box Management* application. It also indicates package dependency on application (*LaneChange* depends on *Box Management*), application dependency on package (*Starting and Shutdown*, and *LaneChange*), and package-package dependency (*LaneChange* and *HealthIO*). This mixed approach is confusing to some extent. One might ask questions: “*what application does HealthIO belong to?*” or “*which package in Box Management is LaneChange dependent on?*”

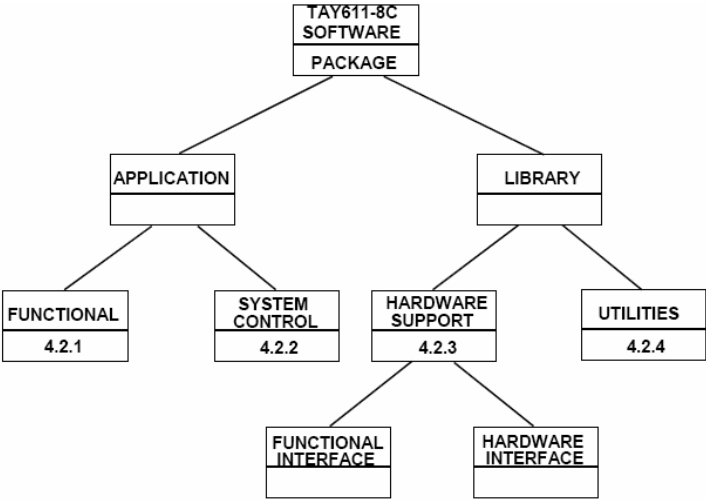


Fig. 8. Top level design diagram

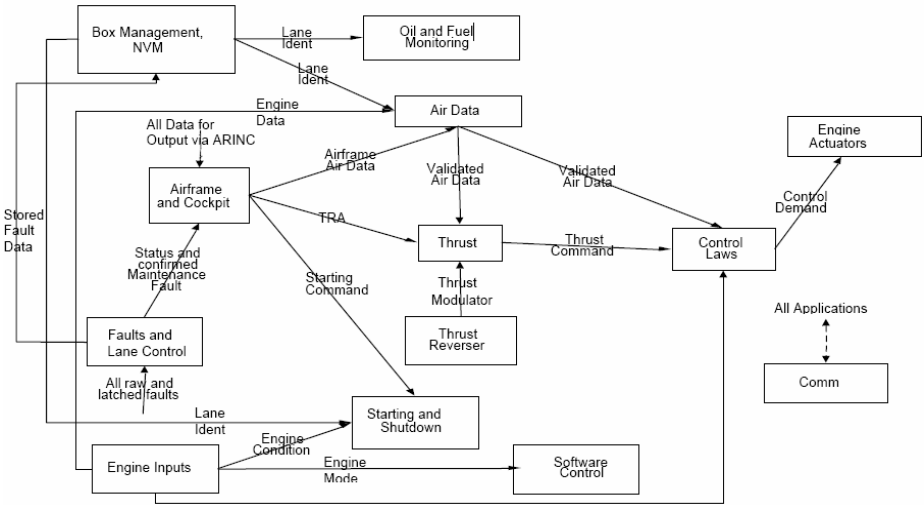


Fig. 9. Functional Applications Communications Graph

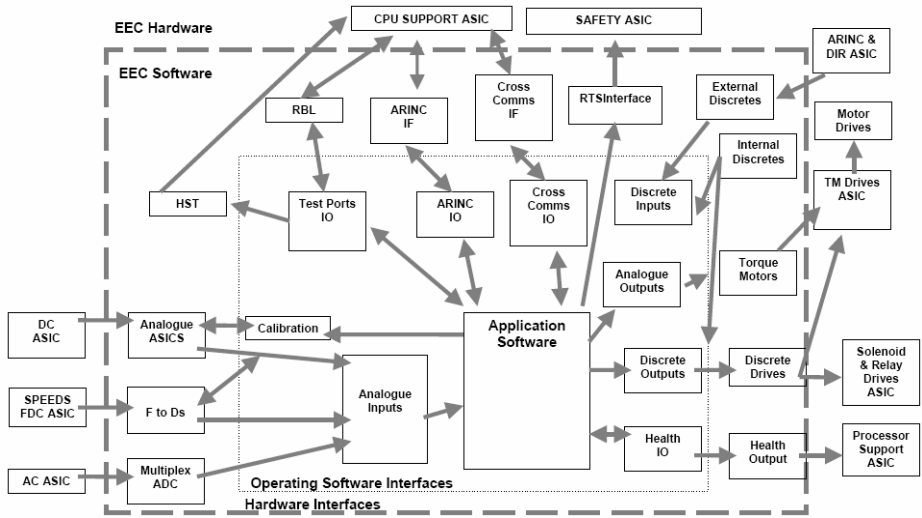


Fig. 10. Hardware Support Application Communications Graph

However, it was shown during the work on the case study that even this kind of logic is easily modelled in AIM and that the model merging of the top level design with the low level implementation reveals the missing detail and restores the ‘hidden’ dependencies.

3.2 TAY Source Code

TAY source code, written in SPARK Ada 83 programming language, is a medium size software system with around 200,000 lines of code in 991 files. The source text

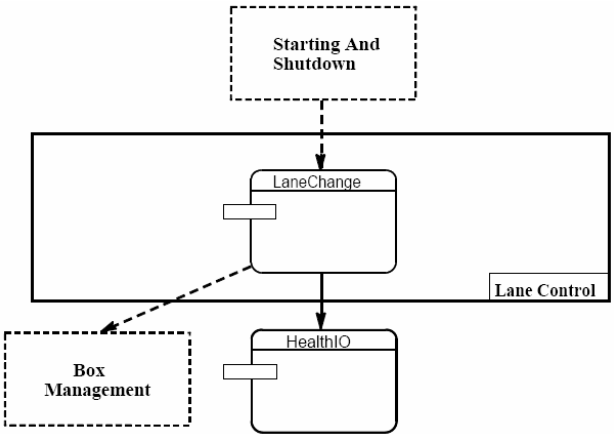


Fig. 11. An example of application and package dependencies

was delivered with a series of linked HTML files describing Ada packages, their purpose, their internal make-up (such as access subprograms, inputs, outputs, and test-points) and any requirements imposed on the package (such as the scheduling requirements and, required subprograms and types).

4 TAY Model in AIM

AIM principles and methods described above are demonstrated in this section on the TAY case study. Due to space constraints, only relatively simplified models are shown. This is, however, sufficient to illustrate AIM modelling applied in practice.

4.1 The Architecture Meta-Model

In order to create an AIM model from the design documentation, it was necessary to recognise and categorise the kind of information that is being described. In simple words, we needed to create a language which would be used to generate the architectural model. We shall call this language the Architecture Language (AL) and we define it by using the Language Factory meta-model. Again, AL is a meta-model (a language) which is used to create the model of the software architecture documentation. The table below summarises the AL elements.

It is worth noting that states and transitions do not get any mention in the design documentation, hence no coordination elements are defined in AL.

All ‘boxes’ in Figure 8, apart from the top one, are modelled as *subsystems*. The top box is the software system and is described by the model element. Observing the relationships in the same diagram, it is clear that subsystems can be composed of other subsystems.

Figure 9 depicts applications and their communication. Although not obvious, these functional applications grouped together form subsystems from the previous graph. Closer scrutiny of the documentation reveals that not all applications are shown in Figure 9. A few others are present in Figure 10, whereas the rest are named throughout the text. It is already easy to see that systematic usage of a modelling platform such as AIM brings immediate benefits by detecting discrepancies like these.

Table 1. Architectural language elements

Component	Service	Subcomponent	Connection End	Connector	Property
SUBSYSTEM		SUBSYSTEM APPLICATION	SUBSYSTEM APPLICATION		ReqDoc: URI
APPLICATION		PACKAGE		-> -->	ArchDoc: URI
PACKAGE	INPUT OUTPUT		PACKAGE APPLICATION		FilePfx: string

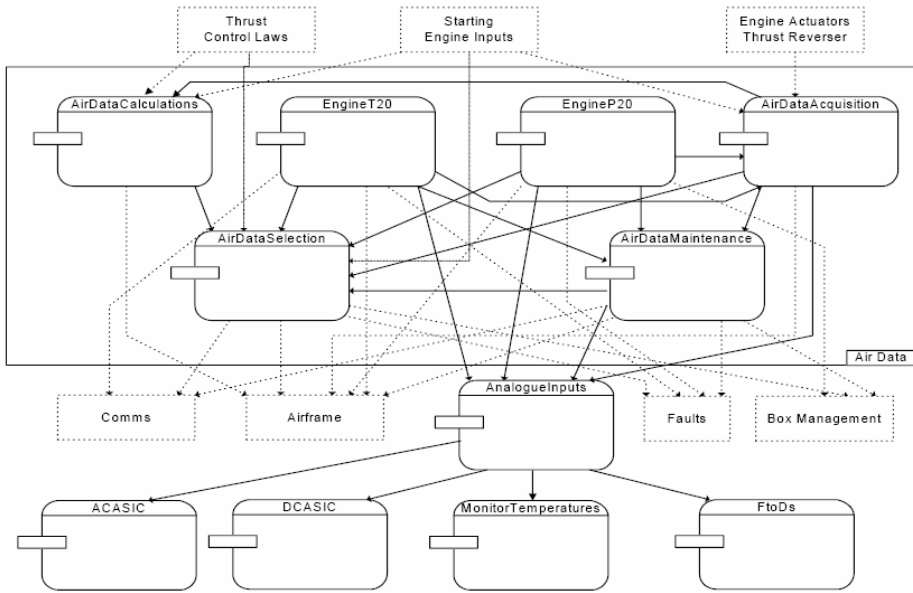


Fig. 12. Air data software architecture design

Further along in the software architecture documentation, applications are broken down into Ada packages and each is accompanied by a diagram such as the one shown in Figure 12. A diagram of this type provides information on:

1. Decomposition of the application into its packages
2. Internal communication between application's packages with the direction of data flow
3. Dependency of the application's packages on external (other applications') packages
4. Dependency of application's packages on other applications

(Note that the solid line rounded rectangles represent packages, the dashed line rectangles are external applications, the large solid line rectangle with the label in its bottom right corner is the application being described, the solid arrowed lines represent package-to-package communication, and the dashed arrowed lines are communication paths between applications and packages).

The first two in the list above provided enough information for the creation of the AL model *structure* and the *communication flow* between various entities. The bottom two in the same list, enabled us to specify a collection of *generic dependency rules* in the Rules Layer.

4.2 The Ada Meta-Model

The Ada language meta-model (ADAL) is another language meta-model that had to be defined via the AIM Language Factory. Although it is really a meta-model, we call it so instead of 'Ada Language' in order to avoid confusion with the Ada

programming language. ADAL is clearly less abstract than AL, in the sense that this ‘abstraction layer’ resides much closer to the real system (source code). One way to look at it is that ADAL *describes* the Ada source text. Table 2 shows a somewhat simplified list of ADAL elements.

Table 2. Ada language elements

Component	Service	Subcomponent	Connection End	Connector	Property
PACKAGE	INPUT OUTPUT TESTPOINT	FILE	PACKAGE	WITH	ReqDoc: URI
FILE	INPUT OUTPUT	OBJECT SUBPROG RAM	OBJECT SUBPROG RAM		Path: URI FileName: URI
OBJECT	DATA			-> <-	Constant: Boolean Public: Boolean Declared: FileLoc Initialised: Boolean
SUBPROG RAM	RETURN	OBJECT	OBJECT	PARAM	Public: Boolean Declared: FileLoc Body: FileLoc

For model extraction, a custom-made Ada language parser was used in order to ‘map’ the source code into the ADAL model. During this process of reverse engineering, all data types were stored in the newly (automatically) created AIM *type sets*. Each file became a FILE component, and they were grouped under the PACKAGE component. The contents of the files were parsed and Ada objects (constants, variables) were converted into OBJECT components and inserted as subcomponents of the corresponding FILE components. Similar processing took place with Ada subprograms. The bodies of subprograms were ignored. However, file locations (line numbers) were recorded and stored in a property attached to the SUBPROGRAM component. It is easy to see how, for example, an external analysis tool could parse the model, locate each subprogram body, analyse its contents and attach additional temporal properties such as Execution Period, WCET (worst case execution time), Memory Size, and so on.

Equally, developers can use the generated ADAL model to navigate effortlessly through hundreds of files and thousands of lines of source text.

4.3 Model Merging

So far, we have created two models. In a top-down approach, we ‘described’ the software architecture from the documentation received. We used the Architecture Language we defined for this purpose and created an architectural model (ARCHMOD, from now on). Because the TAY software was already implemented, we were also able to model in a bottom-up fashion and reverse engineer the source code. We extracted the second model (ADAMOD, from now on) using the Ada Meta-model which we created from the same platform (AIM Language Factory) as the Architecture Language.

One of the key tasks in carrying out this case study was to examine if the implemented system was correctly aligned with its architectural blueprint, and if not, to evaluate the degree of ‘model erosion’. These findings are outside scope of this article. However, the first step in this line of investigation after the two models were created was to somehow attempt to link them together. This was achieved by defining generic dependency rules and using rule view templates to create active AIM views for linking artefacts from one model to the other.

For example, we first defined a generic dependency rule with link type called PKGDEP (standing from ‘package dependency’). We define source artefacts as packages from the architectural model. For example, an XPath expression used could be:

```
/ARCHMOD/SUBSYSTEM/APPLICATION/PACKAGE{ [ @name=" %PackageName% " ] }
```

During the rule instantiation, the following steps take place:

1. AIM scans the expression above and removes all text inside the curly brackets [11]
2. The remaining text is evaluated as a standard XPath expression. In this case, it will return all PACKAGE nodes from the ARHCMOD
3. In the view created by a rule view template, the user selects one node from the list of PACKAGE nodes
4. The selected node’s *name* (for example *AirData*) replaces the %PackageName% placeholder and the concrete XPath expressions now becomes:
/ARCHMOD/SUBSYSTEM/APPLICATION/PACKAGE[@name="AirData"]

We define the target artefacts in a similar fashion.

Finally, a rule instance would look something like this:

Source:

```
/ARCHMOD/SUBSYSTEM/APPLICATION/PACKAGE[ @name="AirData" ]
```

Target: /ADAMOD/PACKAGE[@name="AirData"]

Link: PKDEP

The mechanism described above allowed us to create permanent relationships between elements in the architectural model and those in the low-level Ada model. The creation of dependency links across the two languages ensures that the changes created, for example, by the system architects are propagated back to the developers, and vice versa.

In conclusion, using the AIM platform, we have been able to create two models for the TAY engine EEC. The first one describes the *architecture* of the system software which until now was specified only in plain English. To do this, we used a specially created language which we called AL. The second model describes the existing SPARK Ada 83 source text, in another specially created language which we called ADAL. English and Ada are very different languages in every aspect. On the other hand, AL and ADAL have been defined from the same platform and are compatible with each other. It is precisely because of this that we have been able to define the dependencies between the two languages in a straightforward fashion and merge the two models. The model data resides in a single repository and the relationships are defined

in an orthogonal fashion within the Rules Layer. In addition, both AL and ADAL (once defined) have become reusable assets within the organisation. We have also defined a number of *rule* view templates which enabled us to instantiate cross-language types of views. This was the key tool in identifying the discrepancies between the system design (i.e. the architectural specification) and the system implementation (i.e. the Ada source code). This powerful and easy-to-use feature of the AIM platform helped us, very quickly, to isolate parts of the system where the implementation diverged from the intended design.

Under the assumption that TAY software is correct (because it was already implemented, tested, certified and deployed at the time of our work on this case study), we rectified the inconsistencies in the design documentation. As a result, the architectural specification is now up-to-date and in sync with the implementation.

Furthermore, because the system is now described in a precise fashion (as an AIM compliant model), a range of formal analyses is possible to further validate the fault-free behaviour of the system.

5 Summary

We have created a modelling platform, AIM, to help us overcome some of the key limitations of ADLs that prevent them from being deployed in the safety critical systems arena. We acknowledged the fact that the development of large scale software systems such as these is driven by a multitude of stakeholder groups. These groups often have divergent sets of concerns.

Views are commonly used in architectural design and modelling to address different perspectives of the system. The difficulty arises because it is extremely beneficial that such views are brought together into a single design process. The consistency across all views is difficult, if not impossible to achieve. Models created are therefore either kept separate or discarded instead of being integrated on the same platform.

In order to address this issue, we designed AIM around three layers and kept model data, cross-cutting dependencies and views separate. This allowed us to display different perspectives of the system as well as create orthogonal relationships without jeopardising the integrity of the model. Consequently, the single model repository and the separate layer for dependencies represent the true *view glue*, ensuring view consistency in the model.

References

1. IEEE, IEEE Recommended Practice for Architectural Description of Software-Intensive Systems, IEEE. p. 29 (2000)
2. Finkelstein, A., Kramer, J., Nuseibeh, B.: Software process modelling and technology. Advanced software development series. Wiley, Chichester (1994)
3. Medvidovic, N.T.R.: A Classification and Comparison Framework for Software Architecture Description Languages. *Software Engineering* 26(1), 70–93 (2000)
4. Radjenovic, A., Paige, R.: Architecture Description Languages for High-Integrity Real-Time Systems. *IEEE Software* 23(2), 71–79 (2006)
5. OMG, Unified Modeling Language: Infrastructure. OMG (2006)

6. Simpson, H.R.: The MASCOT Method. *Software Engineering Journal* 1(3), 103–120 (1986)
7. SAE, Architecture Analysis & Design Language (AADL). Society of Automotive Engineers (AS-2C) (2004)
8. Clark, J., DeRose, S.: XML Path Language (XPath) Version 1.0, in W3C Recommendation (1999)
9. Kruchten, P.: Architectural Blueprints - The '4+1' View Model of Software Architecture, Rational Software Corp. p. 15
10. Soni, D., Nord, R., Hofmeister, C.: Software Architecture in Industrial Applications. In: *International Conference on Software Engineering* (1995)
11. Kande, M.C.V., Strohmeier, A., Sendall, S.: Bridging the Gap between IEEE 1471, Architecture Description Languages and UML. Swiss Federal Institute of Technology: Lausanne. p. 16 (2002)

A Component-Based Approach to Verification and Validation of Formal Software Models

Dejan Desovski and Bojan Cukic

Lane Department of CSEE
West Virginia University, Morgantown, WV, 26506, USA
{desovski, cukic}@csee.wvu.edu

Abstract. Formal methods for verification of software systems often face the problem of state explosion and complexity. We present a divide and conquer methodology that leads to component based analysis and verification of formal requirements specifications expressed using Software Cost Reduction (SCR) models. The proposed methodology has the following steps: model partitioning, partition verification and composition of verification results. We define a novel decomposition methodology for SCR specifications based on minimum cut graph algorithms. Experimental validation of our methodology brought to light the importance of several concepts that have been advocated in the software development community for a long time: modularity, encapsulation, information hiding and the avoidance of global variables. The advantages of the compositional verification strategy are demonstrated in the case study, which analyses the Personnel Access Control System. Our approach offers significant savings in terms of time and memory requirements needed to perform formal system verification.

1 Introduction

By definition, a high assurance system is a system for which compelling evidence is required to demonstrate that it delivers its services in a manner that satisfies certain critical properties. Consequently, during the development of a high assurance system we must prove that it does not contain any faults or, if that is out of our reach, that failures are highly unlikely to occur. High assurance software systems continue to pose significant challenges for verification and validation. Software industry relies mostly on informal methods (e.g., code review or testing) for quality assurance purposes. However, even the most experienced quality assurance engineers can often overlook faults.

Formal methods hold a promise for the development of provably correct software applications. Despite enthusiasm in the research community, formal methods are rarely used in software industry. Continual complaints regarding the difficulty of applying them to practical software applications are due, in part, to scalability problems. The challenges of formal approaches are caused by the large state-spaces and the complexity of any practical software application. The automated formal verification methodologies, such as model checking, often hit the state-space explosion barrier, while the approaches involving theorem proving require advanced skills and

knowledge. Several techniques have been investigated to combat the complexity and state-space explosion. For example, partial order reduction [2], abstractions [6][11], assume guarantee reasoning [13] and other approaches have been proposed, but practical applications of formal methods to real-life software systems remain as uncommon as they were a decade ago.

In order to apply formal methods, the underlying software model must be recorded in some form of a formal notation. For our research purposes, we selected one of the most mature requirements specification notations: Software Cost Reduction (SCR) [12]. Typical SCR requirement specification consists of two parts. The *operational* part describes system operation, while the *property-based* part encodes the logical and temporal properties that the system must satisfy in operation. SCR tables in the operational part represent the system as a finite state machine, while the properties are first order logic formulas representing state or transition (two-state) invariants which must hold for the system. Having this separation makes it possible to perform verification and detect possible inconsistencies using formal methods like theorem proving or model checking. *When we talk about “verification” in the remainder of the paper, we are referring to the formal consistency verification between the stated properties and the operational description of a given system.*

The role of the properties is similar to the notion of *checklists* used for certification of systems in other engineering disciplines. If all items (properties) on the checklist are satisfied, the system is certified for use; if not, the engineer must present sufficient evidence that they are true so that the system can become certified. Consequently, this list must be *complete* in the sense that it must contain all needed properties that establish system correctness.

SCR specifications deal with reactive systems that monitor variables in the environment and react accordingly by changing the controlled variables in a single step. There are no explicit loops in the control flow, except for one main loop, which iterates through the reading of monitored variables and producing the control variables.

We present a methodology for automated decomposition and abstraction of SCR specifications. The main hypothesis is that components of complex system specifications can be identified at the points of minimal coupling (minimal control and/or information exchange). By applying graph-theoretic minimum cut algorithms, we can identify these points and decompose the specification.

Automated decomposition of specifications remains one of the most elusive research goals. An approach to specification decomposition has been presented in [8]. The authors propose an algorithm for slicing system specifications represented with Colored Petri Nets. Slicing the specification improves the understanding of the complex system models and helps with identifying high-risk components early in the life cycle. The underlying ideas of [8] are similar to our approach. We want to decompose complex system specifications into smaller parts with manageable complexity. Instead of ad-hoc decomposition criteria, we propose using *minimum coupling*. In other words, our approach to decomposition creates system components such that have minimal coupling (information exchange and/or control connectivity) with the rest of the system.

Several abstraction and slicing criteria for SCR have been proposed in [11]. We demonstrate how minimum cut graph algorithms can be used to decompose SCR specifications, as well as to automate the specific SCR abstraction method. Proposed

methodology provides automated abstraction of “irrelevant” monitored variables and provides guidance on how to perform verification and validation of system models. Consequently, domain and specification experts can focus their formal verification efforts on the identified components, combine the results and provide evidence regarding the correctness of the system as a whole. We present the existing theory and propose a strategy for verification of decomposable SCR requirements specifications for efficient verification of SCR models.

The paper is organized as follows. Section 2 provides background and introduces a motivational example that is used throughout the exposition. Section 3 reviews the theoretical basis for compositional verification of SCR models. Section 4 presents the proposed decomposition methodology and the strategy for component-based verification. The case study of the Personnel Access Control System (PACS) SCR specification is given in Section 5. Section 6 concludes the paper.

2 Background and a Motivating Example

In this section, we present notational background, i.e., the SCR requirements capture methodology. To illustrate the key concepts we present a small SCR specification of a nuclear reactor safety injection system, which was introduced in [7] and is available as an example with the SCR toolset.

2.1 Software Cost Reduction (SCR) Method

In SCR we represent the environmental quantities as monitored and controlled variables. The environment non-deterministically produces a sequence of input events, where an input event represents a change in some monitored quantity. The system, represented as a finite state machine, begins execution in an initial state. It responds to each input event by changing the state and, possibly, by producing one or more output events. Output events are changes in controlled variables. An assumption of the model is that at each state transition exactly one monitored variable changes its value, often referred to as “one input” assumption. To concisely capture the system behavior, SCR specifications may include two types of internal auxiliary variables: terms, and mode classes. Mode classes and terms often capture historical information.

In the SCR operational model, system Σ is defined as a finite state machine $\Sigma = (V, S, \Theta, \rho)$, where $V = \{ r_1, r_2, \dots, r_n \}$ is the set of state variables, S is the set of states, $\Theta : S \rightarrow \text{boolean}$ is the initial state predicate, and $\rho : S \times S \rightarrow \text{boolean}$ is the next state predicate representing the transition relation. A *state* s in S is a function that maps each variable $x \in V$ to a value $x(s)$ from its set of legal values. Usually, the evolution of the system described by the transition relation ρ is deterministic, i.e., it can be represented as a function that maps a given state and an input event to only one possible next state. To construct the next state predicate ρ , SCR uses the composition of smaller functions described in a tabular notation, thus improving the readability and understandability of the specification. There are three kinds of tables in SCR requirements specifications, event tables, condition tables, and mode tables. These

tables describe the values of each dependent variable, that is, each controlled variable, mode class, or term.

In SCR, a state is a function that maps each variable in the specification to a value, a condition is a predicate defined on a system state, and an event is a predicate defined on a pair of adjacent system states implying that the value of at least one state variable has changed. When a variable changes value, we say that an event “occurs”. The following notation is used to denote an event in which some condition becomes true:

$$@T(c) \stackrel{\text{def}}{=} \neg c \wedge c' \quad (1)$$

where c is a condition evaluated in the current state, and the primed condition is evaluated in the next state. Informally the notation $@T(c)$ can be read as “*at true c*,” meaning that we are interested in the event when the logical value of the predicate c changes from *false* in the current state to *true* in the next state. Similarly an opposing event is denoted with $@F(c)$ - “*at false c*”, meaning that we are interested in the event when the logic value of the predicate c changes from *true* to *false*.

$$@F(c) \stackrel{\text{def}}{=} @T(\neg c) = c \wedge \neg c' \quad (2)$$

We can also have conditioned events, denoting that we are interested in the event only when some predicate already holds. The expression “ $@T(c)$ WHEN d ” represents a conditioned event, which is defined by:

$$@T(c) \text{ WHEN } d \stackrel{\text{def}}{=} \neg c \wedge c' \wedge d \quad (3)$$

where the unprimed conditions c and d are evaluated in the current state and the primed condition c' is evaluated in the next state.

The keyword NEVER denotes a special event that is always false. We use the NEVER event in the SCR tables to denote transitions which should never occur during the operation of the system. In addition, there is a special condition keyword INMODE, which is shorthand for specifying that the value of the mode variable is equal to the mode specified in the first column of the row. Similarly, $@T(\text{INMODE})$ denotes the event when the mode variable becomes equal to the mode specified in the first column of the row. We give an example of an SCR requirements specification using all these constructs and keywords in the next section.

The SCR model requires the entries in each table to satisfy consistency and completeness properties. The completeness property, in the sense of SCR, is defined as complete definition of the tables, i.e. there cannot be cells which are empty or missing, and each enumerated value must be used as a possible assignment in the corresponding variable specification table. The consistency property ensures that the conditions or events used in specifying the variable are disjoint, i.e. there cannot be non-deterministic assignments of two or more values to a single variable at any point in time. These properties are automatically checked by the SCR toolset and guarantee that all of the tables describe total functions [10]. We must note that the defined completeness and consistency properties do not guarantee the correctness of the developed specification; they just provide assurance in the structural accuracy.

2.2 Motivating Example – Safety Injection System

To illustrate the key SCR constructs and the problems we are facing when performing verification we presents a simple specification of a nuclear safety system. The Safety Injection System (SIS) SCR specification describes the safety system of a water coolant system in a nuclear reactor [7]. Based on the pressure sensor readings the system decides whether a safety injection of water is needed in the reactor cooling system. The sensor system uses triple modular redundancy to deal with possible sensor failures and erroneous readings. Depending on the pressure monitored by three sensors and using majority voting, the system performs injection if the pressure is too low. Fig. 1 presents a visual interface used for simulation and testing of the SIS specification.

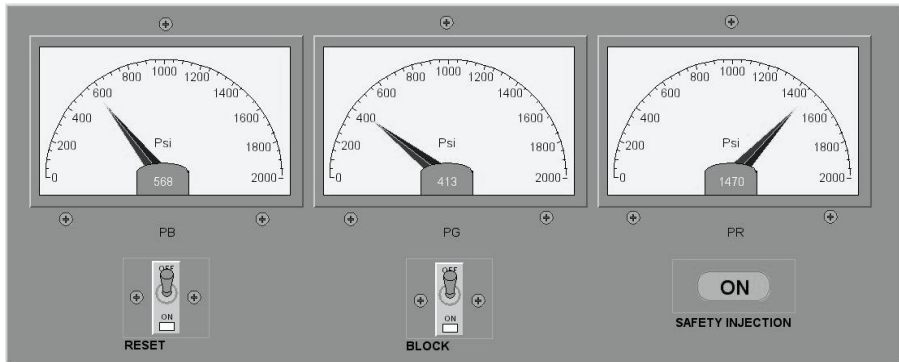


Fig. 1. Visual Interface of the SIS specification

2.2.1 SCR Tables of the SIS Specification

The SIS specification is a small specification with 5 monitored variables, 1 controlled variable, 3 term variables, and 2 mode variables. The SCR specification first defines the variables by giving their types and domains. The pressure sensors PG, PR and PB are monitored variables defined as integers having the range between [0, 2000] psi. The other two monitored variables are RESET and BLOCK representing switches on the control table with two possible values {ON, OFF}.

Table 1. MajorityPermit Condition Function

	Conditions	
	$((PG < permit) \text{ AND } (PR < permit))$ OR $((PR < permit) \text{ AND } (PB < permit))$ OR $((PG < permit) \text{ AND } (PB < permit))$	NOT $((PG < permit) \text{ AND } (PR < permit))$ OR $((PR < permit) \text{ AND } (PB < permit))$ OR $((PG < permit) \text{ AND } (PB < permit))$
MajorityPermit =	TRUE	FALSE

The MajorityPermit term variable is a boolean variable and its specification is given by the condition Table 1. From Table 1 we see that MajorityPermit is set to TRUE when any two pressure sensors are less than the value of the permit constant, which is defined as

1000 psi in the specification document. When we do not have two sensors which are less than permit the MajorityPermit variable is set to FALSE.

The MajorityLow term variable (Table 2) is specified in an analogous way as the MajorityPermit variable. The only difference is that the low constant is now used, which is defined as 900 psi in the specification document. Table 1 and Table 2 are examples of modeless condition tables, since their definition does not involve a mode variable.

Table 2. MajorityLow Condition Function

	Conditions	
	((PG < low) AND (PR < low)) OR ((PR < low) AND (PB < low)) OR ((PG < low) AND (PB < low))	NOT (((PG < low) AND (PR < low)) OR ((PR < low) AND (PB < low)) OR ((PG < low) AND (PB < low)))
MajorityLow =	TRUE	FALSE

Table 3. PermitStatus Mode Transition Function

Source Mode(s)	Events	Destination Mode
BelowPermissive	@F(MajorityPermit)	AbovePermissive
AbovePermissive	@T(MajorityPermit)	BelowPermissive

Table 3 presents the mode table for the PermitStatus mode variable. The first row of the table specifies that if the previous value of the PermitStatus is BelowPermissive and the boolean variable MajorityPermit becomes *false* then PermitStatus is assigned AbovePermissive as the new value. Similarly, the second row states that if the previous value was AbovePermissive and MajorityPermit becomes *true*, the next value of PermitStatus will be BelowPermissive. So, the assignment of a new value to a mode variable depends on its previous value, and an event causing the transition.

Table 4. M_Pressure Mode Transition Function

Source Mode(s)	Events	Destination Mode
Low	@F(MajorityLow)	Normal
Low	@F(MajorityPermit)	VoterFailure
Normal	@T(MajorityLow) WHEN MajorityPermit	Low
Normal	@T(MajorityLow) WHEN (NOT MajorityPermit)	VoterFailure
VoterFailure	@T(MajorityPermit)	Low
VoterFailure	@F(MajorityLow)	Normal

The mode variable M_Pressure is defined by the function described in Table 4. The table specifies the following transitions for the M_Pressure variable: 1) in case the current mode is Low and the MajorityLow variable becomes *false*, then the new mode for M_Pressure will be Normal; 2) in case the current mode is Low and the MajorityPermit variable becomes *false*, then the new mode for M_Pressure will be VoterFailure; 3) if the mode is Normal and MajorityLow becomes *true* when MajorityPermit is *true*, then the next value for the M_Pressure will be Low; 4) if the current mode is Normal and the MajorityLow

variable becomes *true* when MajorityPermit is *not true*, then the new mode will be VoterFailure; 5) if the current mode is VoterFailure and the MajorityPermit variable becomes *true* then the new mode for M_Pressure will be low; and 6) if the current mode is VoterFailure and the MajorityLow variable becomes *false*, then the new mode will be Normal.

Table 5. OVERRIDDEN Event Function

Modes for PermitStatus	Events	
AbovePermissive	Never	@T(Inmode)
BelowPermissive	@T(Block = ON) WHEN (Reset = OFF)	@T(Reset = ON)
BelowPermissive	Never	@T(Inmode)
OVERRIDDEN =	TRUE	FALSE

The OVERRIDDEN term variable is specified by the regular SCR event table given in Table 5. Depending on the current value of the PermitStatus mode variable, the new value of OVERRIDDEN is assigned to TRUE or FALSE. In essence, OVERRIDDEN becomes TRUE only if PermitStatus is BelowPermissive and the monitored variable Block becomes equal to ON when the Reset variable is equal to OFF. Whenever PermitStatus becomes equal to AbovePermissive or BelowPermissive, or in the event that the Reset variable becomes equal to ON when the PermitStatus is BelowPermissive, the OVERRIDDEN variable becomes equal to FALSE.

Table 6. Safety_Injection Condition Function

Modes for M_Pressure	Conditions	
Normal	FALSE	TRUE
Low	NOT OVERRIDDEN	OVERRIDDEN
VoterFailure	TRUE	FALSE
Safety_Injection =	ON	OFF

The controlled variable Safety_Injection (Table 6) determines whether a safety injection is performed or not depending on the current value of the M_Pressure mode variable. When M_Pressure is Normal, the value of the Safety_Injection variable is OFF; when the M_Pressure is Low and the OVERRIDDEN variable is FALSE, the Safety_Injection is ON, otherwise in the same mode when the OVERRIDDEN variable is TRUE the Safety_Injection is OFF; finally if the mode of M_Pressure is VoterFailure then Safety_Injection is ON.

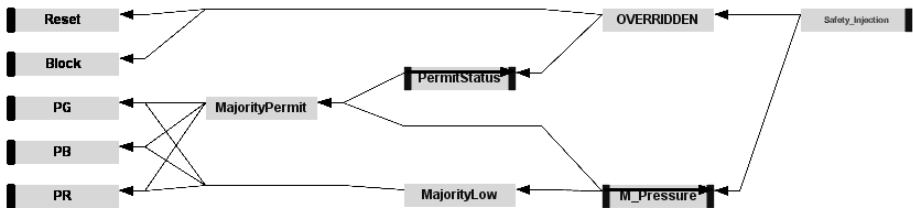


Fig. 2. Dependency Graph of the Safety Injection SCR specification

Given the SCR specification in tabular notation, we can observe that there are dependencies between the variables (Fig. 2). In other words, the value of a particular variable depends on values of some other variables. This dependency chain starts with the monitored variables, used by the system to observe changes in the environment, and ends with controlled variables, which are produced by the system in order to affect the environment. The SCR toolset [12] includes *dependency graph browser*, which is used for navigation through or manual slicing of the specification under development.

2.2.2 Problems in the Verification

Once the operational part of the SIS is specified, we proceed with verification of the stated properties that the specification must satisfy. We are focusing on safety properties, which can be either state or transition properties.

For example, we would like to prove that the following property holds:

$$\text{Override Works: Reset} = \text{ON} \Rightarrow \text{not OVERRIDDEN.} \quad (4)$$

This property states that the system cannot be stuck in overridden mode, i.e., if the monitored variable Reset is ON, then Overriden should be *false*.

```

pan: out of memory
(Spin Version 4.2.5 -- 2 April 2005)
Warning: Search not completed
Full statespace search for:
    never claim - (not selected)
    assertion violations +
    cycle checks - (disabled by -DSAFETY)
    invalid end states +
State-vector 40 byte, depth reached 499999, errors: 0
1.1879e+07 states, stored
2.2326e+06 states, matched
1.41116e+07 transitions (= stored+matched)
0 atomic steps
hash conflicts: 1.92053e+06 (resolved)
Stats on memory usage (in Megabytes):
522.675    equivalent memory usage for states
475.349    actual memory usage for states (compression: 90.95%)
           State-vector as stored = 36 byte + 4 byte overhead
33.554    memory used for hash table (-w23)
14.000    memory used for DFS stack (-m500000)
27.836    other (proc and chan stacks)
0.082     memory lost to fragmentation
536.822    total actual memory usage

```

Fig. 3. Execution of the SPIN on the complete model

We first tried to run the model checker SPIN on the whole model. As expected, this was unsuccessful because we run out of memory (see Fig. 3). Since PG, PB, PR are integer variables with a large range we expect state space explosion when using explicit model checkers like SPIN. The application of the symbolic model checking tool SMV was successful and proved the property in 16 seconds. We also tried to apply the inductive theorem proving tool SALSA, however, it was unsuccessful in proving the property because of the incompleteness of the tool.

2.2.3 Proposed Approach

We explore how to utilize the modular structure of the SIS specification in order to enable more efficient verification of the stated properties. For example, the SIS specification can be decomposed into two parts: the triple modular redundancy sensor module (lower left corner of Fig. 4), and the control module (upper part in Fig. 4). The properties that involve variables within a single component can be soundly verified by abstracting the rest of the system. The properties that involve variables within several components can be verified either by combining the affected components or by using compositional proof rules.

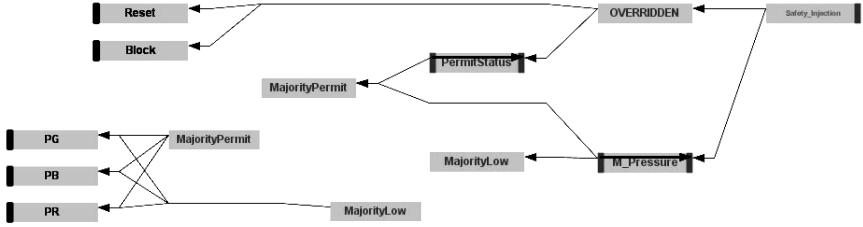


Fig. 4. Decomposition of the SIS specification

In order to verify the *Override Works* property, we observe that we can focus on the identified control module. Application of the SPIN model checker verifies the property in 0.062 seconds avoiding the state explosion problem. SMV also verifies the property in 0.016 seconds, which is 1000 times faster than performing verification on the complete system.

We just demonstrated a single property, which is not enough to establish the correctness of the whole system (i.e., our checklist representing required properties is far from *complete*). However, we see that being able to verify properties focusing just on components of the specification is beneficial from performance standpoint. We must note that in general this is not as easy as illustrated in the given example. The underlying assumptions for this methodology to work are that the original specification is modular in nature and that the properties have been stated on component boundaries, or they involve only few components. Emergent system properties should be presented as conjunctions of component properties to ease their compositional verification. Next, we present a decomposition methodology that identifies the components of a given SCR specification and present a strategy for efficient verification of decomposable specifications.

3 Theoretical Basis for the Verification of Decomposable Models

In this section, we review the theoretical foundation for the compositional verification of SCR requirements specifications based on [15]. As mentioned previously, the SCR model of a given system Σ can be viewed as a finite state machine, represented by the quadruple $\Sigma = (V, S, \Theta, \rho)$ where:

- V is a set of variables. It contains the monitored, controlled, and internal (term and mode class) variables of the system;
- S is the set of system states. Each state $s \in S$ maps each variable $x \in V$ to a value in its set of legal values. The value of a variable x in state s is denoted by $x(s)$;
- $\Theta : S \rightarrow \text{boolean}$ is a one-state predicate defining the set of initial states;
- $\rho : S \times S \rightarrow \text{boolean}$ is a two-state predicate defining the transitions of Σ . A state s may transition to a state s' if $\rho(s, s')$ is true.

The properties the system must satisfy, called system invariants, are represented by the first order logical formulae (predicates), defined on a single state, or a transition. The truth value of a single state predicate $\phi(s)$ is calculated by replacing the variables it involves with the values from s . Two-state predicate $\phi(s, s')$ is evaluated with values from s replacing unprimed variables and values from s' replacing primed variables. The following are standard definitions for reachability and invariants (e.g. see [4]).

Definition 1: Given a state machine $\Sigma = (V, S, \Theta, \rho)$, a state $s \in S$ is reachable, denoted $\text{Reachable}_\Sigma(s)$, if and only if it is one of the initial states or if there exists another reachable state s_1 from which we can make a transition to the state s .

$$\text{Reachable}_\Sigma(s) \Leftrightarrow \Theta(s) \vee \exists s_1 \in S : \text{Reachable}_\Sigma(s_1) \wedge \rho(s_1, s) \quad (5)$$

Often we need to use induction on the number of steps, so the following definition is beneficial. A state $s \in S$ reachable in n steps denoted $\text{Reachable}^n_\Sigma(s)$ is defined by:

$$\text{Reachable}^n_\Sigma(s) \Leftrightarrow \begin{cases} \Theta(s), n=0 \\ \exists s_1 \in S : \text{Reachable}^{n-1}_\Sigma(s_1) \wedge \rho(s_1, s), n > 0 \end{cases} \quad (6)$$

■

Definition 2: Given a state machine $\Sigma = (V, S, \Theta, \rho)$, a one-state predicate $\phi(s)$ is a state invariant of Σ if and only if it holds for all reachable states.

$$\phi \in \text{Inv}(\Sigma) \Leftrightarrow \forall s \in S : \text{Reachable}_\Sigma(s) \Rightarrow \phi(s) \quad (7)$$

■

Definition 3: A two-state predicate $\phi(s, s')$ is a transition invariant of Σ if and only if it holds for all pairs of adjacent reachable states.

$$\phi \in \text{Inv}(\Sigma) \Leftrightarrow \forall s, s' \in S : (\text{Reachable}_\Sigma(s) \wedge \rho(s, s')) \Rightarrow \phi(s, s') \quad (8)$$

■

The following theorems (Theorem 1 and 2) are also given in [4].

Theorem 1: Let $\Sigma = (V, S, \Theta, \rho)$, then $\phi(s)$ is a state invariant of Σ if the following holds:

$$(\forall s \in S : \Theta(s) \Rightarrow \phi(s)) \wedge (\forall s, s' \in S : \phi(s) \wedge \rho(s, s') \Rightarrow \phi(s')) \quad (9)$$

The proof follows from definitions 1 and 2, by using induction on the number of steps. Basically we ensure that the property $\phi(s)$ holds for the initial states, and then

show that if it holds in the current state, it must also hold in the next state defined by the transition relation, thus covering all reachable states as required by definition 2. However, it covers more than just the reachable states, thus causing incompleteness. Some unreachable states can cause the right side of the premise to fail and dispute the validity of some probable invariant. ■

This theorem gives us the initial deductive rule for proving single state invariants (Eq. 10). In this shorter notation ϕ is a single state property and ϕ' denotes the same property evaluated in the next state.

$$\frac{\Theta \Rightarrow \phi, (\phi \wedge \rho) \Rightarrow \phi'}{\phi \in \text{Inv}(\Sigma)} \quad (10)$$

We can strengthen the rule by using additional previously proven invariants. This allows us to reduce the incompleteness in the previous rule by using invariants that restrict the unreachable states causing it to fail. The deduction rules (Eq. 11) are obtained, where α represents a single state invariant and β represents a transition invariant for the system Σ .

$$\frac{\alpha \in \text{Inv}(\Sigma), (\alpha \wedge \Theta) \Rightarrow \phi, (\phi \wedge \alpha \wedge \alpha' \wedge \rho) \Rightarrow \phi'}{\phi \in \text{Inv}(\Sigma)}; \text{ and } \frac{\beta \in \text{Inv}(\Sigma), \Theta \Rightarrow \phi, (\phi \wedge \beta \wedge \rho) \Rightarrow \phi'}{\phi \in \text{Inv}(\Sigma)} \quad (11)$$

Theorem 2: Let $\Sigma = (V, S, \Theta, \rho)$, then $\varphi(s, s')$ is a transition invariant of Σ if the following holds: $\forall s, s' \in S : \rho(s, s') \Rightarrow \varphi(s, s')$.

The proof follows from definition 3 by removing the requirement that the state s should be reachable. This introduces incompleteness, meaning that there might be invariants that we might not be able to prove by this theorem. ■

Like in the previous case, this theorem gives us the initial deduction rule (Eq. 12) for proving transition invariants.

$$\frac{\rho \Rightarrow \varphi}{\varphi \in \text{Inv}(\Sigma)} \quad (12)$$

It also can be strengthened by using auxiliary state or transition invariants as presented in (Eq. 13).

$$\frac{\alpha \in \text{Inv}(\Sigma), (\alpha \wedge \alpha' \wedge \rho) \Rightarrow \varphi}{\varphi \in \text{Inv}(\Sigma)}; \text{ and } \frac{\beta \in \text{Inv}(\Sigma), (\beta \wedge \rho) \Rightarrow \varphi}{\varphi \in \text{Inv}(\Sigma)} \quad (13)$$

The problem in using these rules for verification of system properties is that they rely on the specification of entire system $\Sigma = (V, S, \Theta, \rho)$. We would like to apply our decomposition methodology to divide the system into smaller subsystems $\Sigma_1, \Sigma_2, \dots, \Sigma_n$, and perform the verification task on these components instead on the complete system.

3.1 Compositional Verification Rules

Abadi and Lamport [1] argue that the most natural way of representing composition of systems is by using conjunction of their specifications. This idea of representing

the system specification as a conjunction of subsystems is central in the underlying theory on compositional verification of SCR requirements specifications. Similar to [1], we also limit our interest to safety properties, and advocate specifying SCR properties that are associated with components.

Without the loss of generality, we consider how to decompose a given system $\Sigma = (V, S, \Theta, \rho)$ into two subsystems Σ_1 and Σ_2 allowing us to perform verification on the components. We use the same definition of parallel composition of state machines as given by Jeffords and Heitmeyer in [15].

Definition 4: Given two state machines $\Sigma_1 = (V, S, \Theta_1, \rho_1)$ and $\Sigma_2 = (V, S, \Theta_2, \rho_2)$, with the same set of variables and the same allowed values for each of the variables, their parallel composition $\Sigma_1 \parallel \Sigma_2$ is defined as conjunction, i.e., $\Sigma_1 \parallel \Sigma_2 = (V, S, \Theta_1 \wedge \Theta_2, \rho_1 \wedge \rho_2)$. ■

Theorem 3: All reachable states within n steps of $\Sigma_1 \parallel \Sigma_2$ are those that are reachable in both Σ_1 and Σ_2 within n steps.

$$Reachable^n_{\Sigma_1 \parallel \Sigma_2}(s) \Leftrightarrow Reachable^n_{\Sigma_1}(s) \wedge Reachable^n_{\Sigma_2}(s) \quad (14)$$

The proof follows from definitions 1 and 4 using induction on the number of steps needed to reach the state. Since the set of all reachable states is obtained when we let the number of steps to go to infinity

$$\{s : s \in S \wedge Reachable_{\Sigma}(s)\} = \{s : s \in S \wedge Reachable^{n \rightarrow \infty}_{\Sigma}(s)\} \quad (15)$$

we conclude that all reachable states of $\Sigma_1 \parallel \Sigma_2$ are those that are reachable in both Σ_1 and Σ_2 . ■

Corollary 1: Invariants of each of the subsystems are also invariants of the composition, i.e., $Inv(\Sigma_i) \subseteq Inv(\Sigma_1 \parallel \Sigma_2)$ for $i = 1, 2$. From Theorem 3 it follows that each component contains all reachable states of the composition and possibly some more. Although this seems contradictory for verification purposes, we will show in Section 4 that focusing the verification on properties that are stated on components boundaries and performing abstractions will reduce the state spaces of the derived components. ■

Since the components may have additional reachable states, possibly invalidating properties that might hold true, we can use already proven invariants from the other components to strengthen the deduction rules. The rules (Eq. 16) can be used for this purpose, where α represents a single state invariant and β represents a transition invariant for the subsystem Σ_1 .

$$\frac{\alpha \in Inv(\Sigma_1), (\alpha \wedge \Theta_2) \Rightarrow \phi, (\phi \wedge \alpha \wedge \alpha' \wedge \rho_2) \Rightarrow \phi'}{\phi \in Inv(\Sigma_1 \parallel \Sigma_2)}; \frac{\beta \in Inv(\Sigma_1), \Theta_2 \Rightarrow \phi, (\phi \wedge \beta \wedge \rho_2) \Rightarrow \phi'}{\phi \in Inv(\Sigma_1 \parallel \Sigma_2)} \quad (16)$$

A similar sound compositional rule for proving a two-state property φ is invariant is given in (Eq. 17):

$$\frac{\alpha \in Inv(\Sigma_1), (\alpha \wedge \alpha' \wedge \rho_2) \Rightarrow \varphi}{\varphi \in Inv(\Sigma_1 \parallel \Sigma_2)}; \text{ and } \frac{\beta \in Inv(\Sigma_1), (\beta \wedge \rho_2) \Rightarrow \varphi}{\varphi \in Inv(\Sigma_1 \parallel \Sigma_2)} \quad (17)$$

Thus, the strategy for verification would be to decompose a given system into components which if composed using parallel composition result in the system itself. Then we prove invariants that hold for the components by using the deduction rules. From the Corollary 1 it follows that the proven invariants also hold for the complete system. Jeffords and Heitmeyer in [15] present a strategy for verification that performs abstraction by removing a single variable from the specification and uses the automatically generated invariants [14] for the abstracted variable to strengthen the deduction rules. Our approach extends this strategy by considering components of the specification (i.e. sets of variables) which have low coupling and information exchange with the rest of the specification. Instead of automatically generating invariants for the abstracted components, our approach is to specify properties on component level and later try to prove that they are invariants and use them in the certification process of the system as a whole.

Since each component usually assumes some properties about the environment for its correct operation, the rules (Eq. 16) and (Eq. 17) can be considered as assume-guarantee rules. We use them to prove and guarantee some properties of the component and consequently the system based on assumptions about the environment. These assumptions are either properties stating the correctness of components that the specific component interacts with, or assumptions about the monitored variables.

4 Component-Based Verification

Complex systems typically contain subsystems, which include different and possibly disjoint sets of monitored variables. For example, in an avionic system we have a navigation subsystem that depends on the altitude, speed, and direction; another life-support subsystem would deal with the cabin pressure and temperature. Several specification slicing techniques [8][11] have been proposed in order to tame a system's complexity and avoid state space explosion in formal verification. However, most of them are done ad hoc and require user's experience in order to be applied successfully. Verification of such complex specifications can be extremely time-consuming, especially if we are not familiar with the specified system.

Since the early days of the SCR Method, Parnas et al. [16] have argued that complex systems must be built by utilizing a modular structure. They demonstrate how information hiding and abstraction principles are to be followed in the design of the Onboard Flight Program (OFP) for the A-7E aircraft by writing a hierarchical module guide document. The guide is intended to achieve the following goals:

1. A software engineer should be able to understand the responsibility of a module without understanding the module's internal design;
2. A reader with a well-defined concern should easily be able to identify the relevant modules without studying irrelevant modules;
3. The number of branches at each non-terminal module in the hierarchy graph should be *small enough* so that the designers can prepare convincing arguments demonstrating that the sub-modules have no overlapping responsibilities and that the module covers all of the intended responsibilities.

These principles, since their introduction, have lead to the development of the object oriented programming methodologies. We argue that the same principles should be followed when writing the operational part (SCR tables) of any large SCR specification. The responsibilities of the components should be stated by the property-based part of the specification, creating a checklist for verification and validation of the components and the system as a whole. The verification process is simplified if the properties are stated on component boundaries (involving only the variables within a single component), as advocated by Abadi and Lamport [1].

Given an SCR specification in tabular notation, dependencies between the variables can be observed easily. Dependency chains end with controlled variables, which are generated by the system and affect the environment. They start with the monitored variables, which are used by the system to observe the changes in the environment. How the monitored variables are used, and how the controlled variables are produced is formally defined by the SCR specification and its constructs: tables, term variables, mode class variables, etc.

Our hypothesis is that if a given SCR model has modular structure as advocated by Parnas et al. in [16], then the boundaries of different subsystems in complex SCR specifications can be automatically identified at the points of minimal coupling with the rest of the specification. The expectation is that the subsystems generally transform monitored variables into a smaller set of derived variables (term variables), which are then further utilized to calculate the values of controlled variables.

We use minimum cut graph algorithms [5] on the dependency graphs to identify these points of minimal coupling and decompose the system in smaller components. The overarching idea is to apply a *divide and conquer* approach in the verification of SCR specifications. The minimum cuts result in *partitioning* of the set of variables. Each partition represents a smaller component of the specification. Only the properties that involve the variables contained in the given partition need to be used for its verification.

Because of the decreased state space of the components, we expect to avoid the state explosion problem while performing model checking, or if it happens, the same decomposition procedure can be applied recursively to the components of interest. After each component has been verified, the resulting abstract model will contain fewer states that should be verified.

4.1 Decomposition of Variable Dependency Graphs

One approach to specification decomposition has been presented in [8]. The authors propose an algorithm for slicing system specifications represented with Colored Petri Nets. The slicing is performed by selecting a particular node of interests (CPN place or transition) and following the control flow of the CPN backwards in order to identify arcs, places, guards and transitions that lead to that node. Cukic et al. [8] argue that slicing the specification improves the understanding of the complex system models and helps with identifying high-risk components early in the life cycle.

The underlying ideas in [8] are very similar to our current approach: We want to decompose complex system specifications to smaller, more manageable and understandable parts. Instead of ad-hoc decomposition criteria, we propose using *minimum*

coupling, i.e., the boundaries of system components should have minimal coupling (information exchange and/or control connectivity) with the rest of the system.

Several abstraction and slicing methods for SCR have been proposed in [3][11]. We summarize them below.

1. *Removal of irrelevant variables.* This method is similar to the “program slicing” technique [17], which removes irrelevant variables for the purpose of program analysis and understanding. Based on the property that we want to verify, only the relevant variables for this particular property obtained by reflexive and transitive closure on the dependency relation are preserved. This method is fully automated and already implemented in the existing SCR toolset.
2. *Removal of detailed monitored variables.* If a relevant variable for a given property is the only one that depends on a set of monitored variables, then these monitored variables can be removed, and the relevant variable can be treated as a monitored variable.
3. *Replacement of a detailed variable with an abstract variable.* In this case, the domain of a detailed variable is partitioned in equivalence classes and its size (the number of distinct input values) is reduced to the number of classes taking one representative for each class.

Methods 2 and 3 have not been fully automated [9]. We show how the minimum cut graph algorithm can be used to decompose SCR specifications, as well as to automate the abstraction Method 2. Our methodology provides automatic abstraction of irrelevant monitored variables and provides guidance on how to perform verification and validation of system models. The specification and domain experts can focus their verification and validation efforts by combining the results and providing evidence for the correctness of the system as a whole.

In order to apply the minimum cut algorithms, we transform the dependency graph produced by the SCR tool to an undirected graph having specific properties of interest. In general, we want to identify internal system variables that represent the points of minimal coupling and, consequently, module boundaries. Therefore, partitioning cuts can be applied to the vertices of the dependency graph only, and not to the edges.

We will represent each variable v of the specification by two vertices v_{in} and v_{out} connected by the undirected edge (v_{in}, v_{out}) . The weight of the newly introduced edge will depend on the type of the variable. We do not want to split monitored (input) and controlled (output) variables, since they cannot imply meaningful system partitions. Therefore, these edges will have infinite weight, $w(v_{in}, v_{out}) = \infty$. Conversely, term and mode variables will have their weights set to 1, $w(v_{in}, v_{out}) = 1$. Each undirected edge (v, u) of the dependency graph (v_{out}, u_{in}) will assume weight $w(v_{out}, u_{in}) = \infty$, as we do not want to cut any existing dependency edges either.

This transformation is linear on the number of vertices and edges requiring $O(n + m)$ time. It produces an undirected graph with $2n$ vertices and $m + n$ edges. The choice of the cost of the edges in our transformed graph will force the minimum cut algorithm to produce cuts, as intended, on vertices only. Fig. 5 presents an example of the dependency graph transformation.

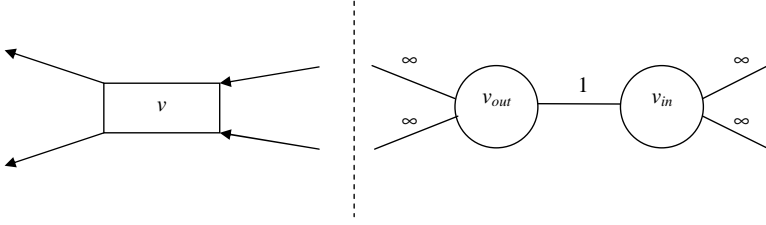


Fig. 5. Transformation of the dependency graph to undirected graph

The application of the minimum cut algorithm will find a partitioning of the vertices of the transformed graph in two sets, which minimizes the cost of the edges connecting the both sets. Thus, the cut will contain one or more edges with cost 1 and no edges with infinite cost, thus performing partition of the variables of the specification that crosses the minimal number of variables. The minimum cut algorithm can be recursively applied to the obtained partitions in order to further decompose the system. Since our transformation is linear and does not expand significantly the size of the problem, the time required to calculate the minimum cut is still polynomial with respect to the size of the specification.

Theorem 4: If an SCR variable r is the only variable which depends on several monitored variables r_1, r_2, \dots, r_i , then a minimum cut of the transformed graph will split this variable.

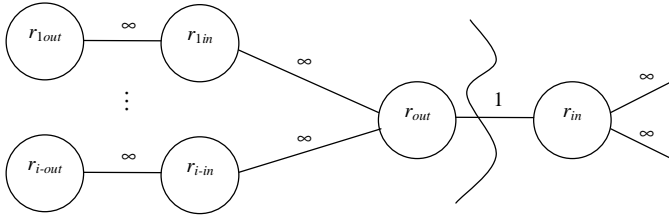


Fig. 6. Minimum cut of the transformed graph of a variable directly dependant on several monitored variables

The proof is intuitive, by construction (see Fig. 6). If we generate the transformed graph of this specific dependency, then the cut splitting the variable r has a cost of 1. This is also a regular cut of the transformed graph because there are no other variables that depend on the given set of monitored variables. Consequently, it is a valid minimum cut, which will be identified by the algorithm. ■

Corollary 2: The proposed methodology automates the abstraction Method 2 – Removal of Detailed Monitored Variables.

Corollary 2 follows directly from Theorem 4. Calculating the minimal cuts of the transformed graph will automatically identify the variables to which this abstraction method can be applied. ■

The proposed methodology not only identifies the variables that directly depend on a set of monitored variables, but it also identifies the variables that are deeper in the dependency graph and have minimal coupling with the other parts of the system. These variables, according to our hypothesis, are the “semi-controlled” variables defining the boundaries of the subsystems contained in the specification. However, we must note that this hypothesis might not hold, i.e., there are many ways one can write an SCR specification for a given system, some of them better than others. Different specifications might not follow generally recognized rules for writing good specifications. There is a list of properties that a specification should have; for example: being complete, unambiguous and minimal. However, from the V&V perspective, the most desired property and the most difficult to achieve is for the specification to be *easily verifiable*. We argue that one measure that provides insight if a specification is “open” to verification methodologies is whether it is decomposable, and one such test is our proposed decomposition methodology.

4.2 The Strategy for Verification of Decomposable Models

Based on the theory presented in Section 3, we propose the following strategy for verification of decomposable models. As stated previously, the *overarching* idea is to apply a *divide and conquer* approach in the verification of SCR specifications. The minimum cuts decomposition results in *partitioning* of the variables, where each partition represents a smaller component of the complete specification. The verification of these parts is performed against the properties involving only variables contained in the given partition.

Jeffords and Heitmeyer in [15] present a strategy for verification that performs abstraction by removing a single variable from the specification and uses the automatically generated invariants [14] for the abstracted variable to strengthen the deduction rules. They state that the problem on how to decompose a given system for performing compositional verification is still open, and should be automatable and accessible to non-experts. Our approach extends their work by considering components of the specification (i.e. sets of variables) which have low coupling and information exchange with the rest of the specification. Instead of automatically generating invariants for the abstracted components, our approach is to specify properties at the component level and later try to prove that they are invariants and use them in the certification process of the system as a whole.

Given a specification of a system $\Sigma = (V, S, \Theta, \rho)$ and a set P of desired properties to be proven as invariants, we apply the following steps.

- **Step 1.** Apply the decomposition algorithm to obtain partitioning of the set of variables V , to disjoint subsets V_1, V_2, \dots, V_n .
According to our heuristic hypothesis, each of these sets should represent one component of the system.
- **Step 2.** Construct each component system Σ_i from V_i , $i = 1$ to n , and Σ as follows:
 1. Delete from Θ the initial state definitions of variables in $V - V_i$, and assign the result to Θ_i .

2. Delete from ρ the functions defining the variables in $V - V_i$, and assign the result to ρ_i .

This step essentially makes all variables in $V - V_i$ to behave as monitored variables for the subsystem Σ_i . In other words, by removing the functions defining them, these variables can assume any allowed value non-deterministically. This introduces incompleteness in the verification process in the following sense: in order to prove some properties for the component that are necessary for the system correctness, we might need to rely on some explicit or implicit assumptions that this component makes about its environment or other interfacing components. The goal is to make all needed assumptions explicit during the specification process, i.e. the created verification checklist of properties should be complete in order to allow verification of the system.

- **Step 3.** For each component Σ_i select those properties from the set P that depend only on the variables found in V_i . Try to prove that these properties are invariants for the component and, consequently, from the corollary of Theorem 3, invariants for the system.

Our minimum cut decomposition approach guarantees that the component Σ_i has the least number of dependencies on other system variables. Consequently, this selection of properties ensures that most of the variables $V - V_i$ will be abstracted away by using the abstraction method 1 and reduces the incompleteness of the component verification process.

- **Step 4.** Use the compositional verification rules together with the invariants proven in the previous step to prove the rest of the properties, or those properties which have failed in step 3.

Each derived component represents an abstraction of the complete system. However, components usually allow auxiliary behaviors that might invalidate the properties we would like to prove. The minimum cut decomposition approach minimizes the dependency of a single component on external variables, thus reducing auxiliary behaviors and making it often possible to prove invariants for the system without using extra invariants.

- **Step 5.** The properties that depend on variables contained in several components can be demonstrated in parts if they can be represented as conjunction of component properties. Or, they can be proven by creating larger subsystems from unions of the required sets of variables $V_i \cup V_j$ and repeating the steps 2, 3 and 4.

Assuming that a given system specification Σ is decomposable; the proposed strategy should improve the time and memory requirements for verification of the system properties. In Section 5 we demonstrate how an SCR specification, which originally was not decomposable, can be refactored into a decomposable specification. We identify the main principles for designing decomposable specifications and use the proposed strategy to verify the required system properties.

5 Case Study

We demonstrate our methodology on an SCR specification of Personnel Access Control System (PACS), originally described in a prose requirements document from the National Security Agency [18]. The SCR specification had been originally developed as an example on how to write high quality formal requirements specification.

A high quality requirements specification must not only be easy to understand and change, precise, and unambiguous, it must also avoid implementation bias. In addition, it should be complete and consistent and organized as a reference document. Unfortunately, requirements specifications with all of these attributes are extremely rare. The original specification was developed focusing on two important aspects of a high quality requirements specification: the formulation of a set of system modes, which make the specification more concise and easier to understand, and the design of the specifications for ease of change.

5.1 PACS Description

PACS checks information on magnetic cards and uses PIN numbers to limit physical access to a restricted area to authorized users. To gain access, the user swipes an ID card containing the user's name and Social Security Number (SSN) through a card reader. After using its database of names and SSNs to validate that the user has the required access privileges, the system instructs the user to enter a four-digit personal identification number (PIN). If the entered PIN matches a stored PIN in the system database, PACS allows the user to enter the restricted area through a gate. To guide the user through this process, PACS displays messages on a single-line display screen. A security officer monitors and controls PACS using a console with the second single-line display screen, an alarm, a reset button, and a gate override button.

To initiate the validation process, PACS displays the message "Insert Card" on the user display. Upon detecting a card swipe it validates the user name and SSN. If the card is valid, PACS displays "Enter PIN." If the card is unreadable or the information on the card fails to match the information in the systems database, PACS displays "Retry" for a maximum of three tries. If after three tries the user's card is still invalid or there is no match, the system displays "See Officer" on both the user's display and the officer's display, and turns on an alarm (either a sound or a light) on the officer's console.

Before system operation can resume, the officer must reset PACS by pushing the reset button. The user, who also has three tries to enter a PIN, has a maximum of five seconds to enter each of the four digits before PACS displays the "Invalid PIN" message. If an invalid PIN is entered three times or the time limit is exceeded, the system displays "See Officer" on both the user and the officer display. After receiving a valid PIN, PACS unlocks the gate and instructs the user to "Please Proceed." After 10 seconds, the system automatically closes the gate and resets itself for the next user.

Fig. 7 presents a Visual Interface of the PACS system, which was used during the development of the specification.

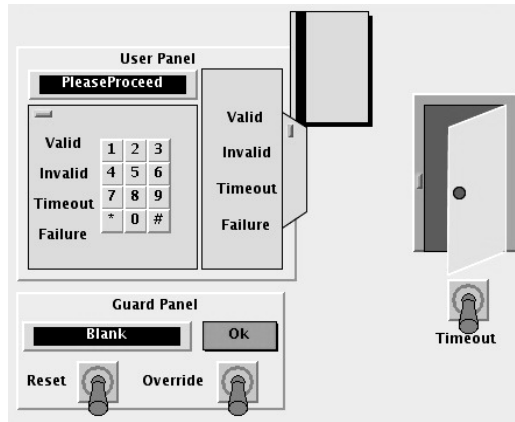


Fig. 7. Visual Interface of the PACS specification

5.2 Applying the Decomposition Procedure

The initial application of our decomposition algorithm on the PACS dependency graph was not successful in the sense that it did not identify the components we were expecting. Namely, from [18] we can see that PACS system has at least two components: the card reader and the PIN reader. Our decomposition heuristic could not detect these two components from the original specification leading to the conclusion that information hiding principles (advocated by Parnas et al. in [16]) have not been followed. After the careful review of the original specification and its dependency graph, we concluded that there are several factors limiting the decomposability of this specification:

- The behavior of the card reader is specified in the system status mode table, instead of being encapsulated as a separate component;
- The behavior of the PIN reader is separated from the system status mode table, however its information hiding can be improved (e.g. there are variables which are not completely encapsulated, adding unnecessary dependencies);
- The specification contains two variables which directly affect most of the other variables in the system (mReset and mOverride). Although the behavior introduced by these two variables is simple (e.g. reset of all the variables in the system by mReset), they introduce additional dependency links that our algorithm cannot break.

To remedy the encapsulation problems, we refactored the original specification by applying the following principles:

1. Define term variables to represent the result of the operation of each component, thus encapsulating the internal component behavior.
2. Use mode classes for component specification.
3. Use the defined term variables when referring to the results of other components (do not break the encapsulation and information hiding by using internal component variables).
4. Avoid the use of global variables.

To break the dependencies on the global variables, we had to remove them from the specification. There are several ways this could be done in the future – we can denote these variables as global and relax the cost of the dependencies links that are connected to them, allowing the decomposition algorithm to break these links.

For the PACS specification, we defined term variables `tCardValid` and `tPINValid` to encapsulate the behavior of the two components. Both of them have the same domain $\{Unknown, Yes, No, Error\}$ denoting that, for example, `tCardValid` is either *Unknown* – we do not currently know the result of the card entry and validation, *Yes* – the card is successfully swiped and validated, *No* – the card is not valid, or *Error* – the user exceeded the number of allowed non-valid swipes. The *Error* state allows us to encapsulate the `tNumCReads` variable, counting the number of non-valid swipes within the component. For each component we used a mode class (`mcCard` and `mcPIN`), which simplifies the specification of their behavior. The system mode status table `mcStatus` and the other external variables to the components are changed only to refer to the defined terms.

```

Input the name of the dependencies file:
PACS2NewDependencies.dg

Cut 1: cost = 1
mcCard_OUT, mcCard_IN, mCardInput_OUT, mCardInput_IN, mCardValid_OUT,
mCardValid_IN, tCardValid_OUT, tNumCReads_OUT, tNumCReads_IN,
Cut 2: cost = 1
mcPIN_OUT, mcPIN_IN, mDigit1_OUT, mDigit1_IN, mDigit2_OUT,
mDigit2_IN, mDigit3_OUT, mDigit3_IN, mDigit4_OUT, mDigit4_IN, mPIN-
Valid_OUT, mPINValid_IN, tNumPReads_OUT, tNumPReads_IN, tPINValid_OUT,

the rest of the system:
mcStatus_OUT, mcStatus_IN, cGate_OUT, cGate_IN, cGuardAlarm_OUT,
cGuardAlarm_IN, cGuardDisplay_OUT, cGuardDisplay_IN, cUserDisplay_OUT,
cUserDisplay_IN, mGate_OUT, mGate_IN, tCardValid_IN, tPINValid_IN,
No more cuts.

```

Fig. 8. Decomposition of the refactored PACS specification

After this refactoring, our decomposition algorithm correctly identified the two components of the system, leaving only the control component (Fig. 8).

These changes in the operational part of the specification imply only small changes in the specification of properties. Because we removed the `mReset` and `mOverride` variables, we had to remove one of the properties which referred to these variables. The rest of the properties were either unchanged or only had replacement of the `mCardValid` or `mPINValid` monitored variables, with the corresponding term variables `tCardValid` and `tPINValid`.

We used the SPIN model checker with default options and increased the depth limit to 180000 in order to achieve complete verification to verify all properties for the complete refactored PACS specification. This took 35.405 seconds and required 553.342 MB of memory on a machine with Pentium M 1.5GHz processor and 1GB of RAM.

5.3 Applying the Compositional Verification Strategy

- Step 1 of our verification strategy gives the following partitioning of the variables:
 $V_1 = \{mCardValid, mCardInput, tNumCReads, mcCard, tCardValid\}$,
 $V_2 = \{mDigit1, mDigit2, mDigit3, mDigit4, mPINValid, mcPIN, tNumPReads, tPINValid\}$,
 $V_3 = \{mGate, mcStatus, cGuardAlarm, cGuardDisplay, cGate, cUserDisplay\}$.
- In step 2, we derived the three components presented in Fig. 9, Fig. 10, and Fig. 11. In the figures, we omitted the variables on which the components do not have explicit dependency. Since the properties of interests for each component are only those that depend on the variables within the component, the variables from the set $V - V_i$ are going to be abstracted by applying the abstraction method 1 described in Section 4.1.

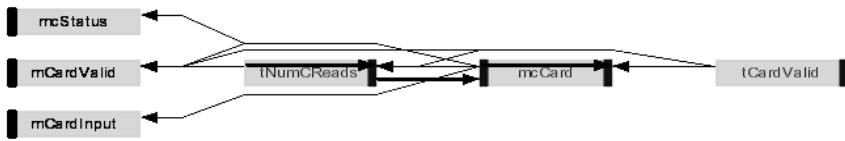


Fig. 9. The Card reading component

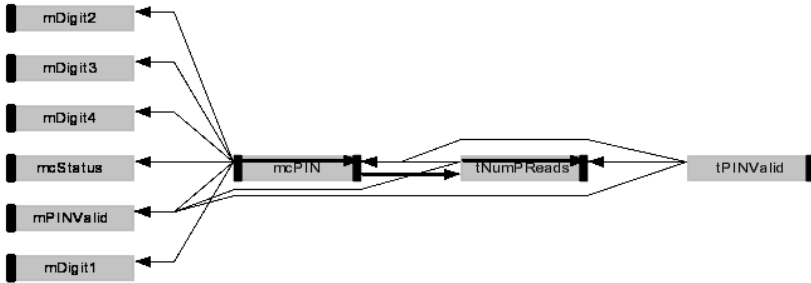


Fig. 10. The PIN Reading component

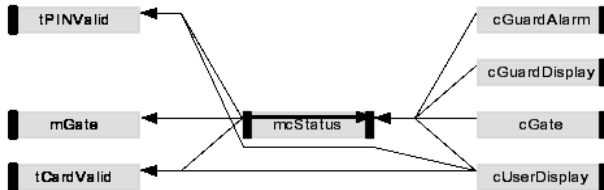


Fig. 11. The Control component

- Next, in step 3 we need to prove the properties that hold for the identified components.

The verification of the Control component (Fig. 11) with SPIN took 0.093 seconds and required 2.827MB of memory, establishing the following invariants for the system:

AlarmStatus: $(cGuardAlarm = On) \Leftrightarrow (cUserDisplay = SeeOfficer)$

CardSuccess: $((cUserDisplay = InsertCard \text{ OR } cUserDisplay = Retry) \text{ AND } mcStatus = CheckCard \text{ AND } @T(tCardValid = Yes)) \Rightarrow (cUserDisplay' = EnterPIN)$

GateStatus: $(cUserDisplay = PleaseProceed) \Leftrightarrow (cGate = Open)$

PINSuccess: $((cUserDisplay = EnterPIN \text{ OR } cUserDisplay = InvalidPIN) \text{ AND } mcStatus = CheckPIN \text{ AND } @T(tPINValid = Yes)) \Rightarrow (cUserDisplay' = PleaseProceed)$

Safety: $(cUserDisplay = SeeOfficer) \Leftrightarrow (cGuardDisplay = SeeOfficer)$.

The verification of the Card-Reading component takes 0.062 seconds and requires 2.724MB of memory, establishing the following invariants for the system:

CardErrors: $(mcStatus = CheckCard \text{ AND } @T(tCardValid = No)) \Rightarrow (tNumCReads' = tNumCReads + 1)$

NumCardErrors: $(tNumCReads \leq MaxCardError)$.

Similarly, the verification of the PIN-Reading component takes 2.327 seconds and requires 55.320MB of memory, establishing the following properties as invariants:

NumPINErrors: $(tNumPReads \leq MaxPINError)$

PINErrors: $(mcStatus = CheckPIN \text{ AND } @T(tPINValid = No)) \Rightarrow (tNumPReads' = tNumPReads + 1)$.

We should note that the verification of the PIN-Reading component requires most resources, because its state space is larger. Consequently, by applying step 3 of our verification strategy, we verified 9 out of the 14 invariants, in total of 2.482 seconds and requiring maximum 55.320MB of memory, which is for an order of magnitude more efficient than performing verification on the complete system.

The rest of the properties include variables that span across more than one component. We were not successful in applying the deduction rules in step 4 to prove the remaining 5 properties, partly because the automated translation from the SCR toolset to Promela currently does not handle assumptions. The results in using Salsa and SMV suggest that the problem in verification of these 5 properties is the over-approximation introduced by the decomposition. We would need to come up with additional invariants or rewrite these properties, but that was not our goal in this study.

CardDisplay1: $(tNumCReads > 0 \text{ AND } tNumCReads < MaxCardError) \Leftrightarrow (cUserDisplay = Retry)$

CardDisplay2: $(tNumCReads = MaxCardError) \Rightarrow (cUserDisplay = SeeOfficer)$

PINDisplay1: $(tNumPReads > 0 \text{ AND } tNumPReads < MaxPINError) \Leftrightarrow (cUserDisplay = InvalidPIN)$

PINDisplay2: $(tNumPReads = MaxPINError) \Rightarrow (cUserDisplay = SeeOfficer)$

PINEntry: $(@C(tPINValid) \Rightarrow (mcStatus = CheckPIN))$

By using step 5 of our strategy and creating unions of components, we are successful in verifying the rest of the properties. Combining the Card-Read and Control components allows verification of the *CardDisplay1* and *CardDisplay2* properties in 0.124 seconds with 3.544MB of memory. More demanding is the union of the

PIN-Read and Control components, which allows verification of the remaining 3 properties in 16.734 seconds and 273.060MB of memory. Table 7 summarizes the results of the case study.

Table 7. The resources needed and number of verified properties for each component

Module	Time (seconds)	Memory (MB)	Properties Proven
Complete PACS	35.405	553.342	14
Control	0.093	2.827	5
Card Read	0.062	2.724	2
PIN Read	2.327	55.320	2
Card Read + Control	0.124	3.544	2
PIN Read + Control	16.734	273.060	3

6 Conclusions

In order to perform verification and validation of complex system specifications, it is beneficial to identify the sub-components, apply abstraction and compositional verification methods. Our approach is based on the hypothesis that specification components can be automatically identified at the points that have minimal coupling with the rest of the system. In the case of SCR specifications, these points are presented by the variables in the specification, and coupling appears in the dependency graph. Applying the minimum cut algorithms, we can identify these points in the graph and perform specification decomposition.

We observed that our hypothesis holds true for specifications that follow the principle of information hiding. The proposed algorithms are polynomial and therefore applicable to large specifications. We demonstrated that our algorithm automates the abstraction procedure for removal of detailed monitored variables in the SCR specifications.

We presented the theoretical framework for compositional verification of the SCR specification properties. The identified deduction rules are sound, but incomplete in general, possibly requiring additional inductive invariants that should be used to prove some of the system invariants. Although we are focusing on the SCR requirements model, the same approach should be applicable to similar formal models (e.g. Reactive Modules [13]), and this is one of the subjects for future research.

Assuming that the specification under review is decomposable, we proposed verification strategy that provides significant reductions in the required time and memory. Each derived component represents an abstraction of the complete system; however, it usually allows additional behaviors that might invalidate the properties we would like to prove. The minimum cut decomposition approach minimizes the dependency of a single component on external variables. Our approach considers components of the specification (i.e. sets of variables) which have low coupling and limited information exchange with the rest of the specification. We specify properties at component level, prove that they are system invariants, and use them in the certification process. This does not always work, because of the additional behavior that the abstract components might have when considered by themselves. The invariants proven this way

can be used to prove additional invariants by using the strengthening compositional and assume-guarantee rules.

During our experiments with the PACS system, we identified the basic principles that need to be followed when writing decomposable SCR specifications. They echo the long time advocated principles of modularity, encapsulation, and information hiding in the software development process:

1. Define component border variables that will represent the result of the operation of each component, thus enabling encapsulation of the internal component behavior. In SCR models, these variables are represented as term variables.
2. Use variables that capture the internal state of the system modules in order to achieve specification modularity and ease of change. In SCR models, these variables are represented as mode class variables.
3. Use the defined border variables (from 1) when referring to the results of other components (do not break the encapsulation by using internal component variables).
4. Avoid the use of global variables, or break the dependencies caused by them.

The results of the PACS case study demonstrate the advantages of decomposable specifications. Following the proposed verification strategy we achieved significant reduction in time and memory required for automated verification. The strategy allows us to perform the verification component by component, and combine the obtained results.

References

1. Abadi, M., Lamport, L.: Conjoining Specifications. *ACM Transactions on Programming Languages and Systems* 17(3), 507–534 (1995)
2. Alur, R., Brayton, R.K., Henzinger, T.A., Qadeer, S., Rajamani, S.K.: Partial-Order Reduction in Symbolic State-Space Exploration. *Formal Methods in System Design* 18, 97–116 (2001)
3. Bharadwaj, R., Heitmeyer, C.L.: Model Checking Complete Requirements Specifications Using Abstraction. *Automated Software Engineering* 6, 37–68 (1999)
4. Bharadwaj, R., Sims, S.: Salsa: Combining Constraint Solvers with BDDs for Automatic Invariant Checking. In: *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems* (2000)
5. Chekuri, C., Goldberg, A., Karger, D., Levine, M., Stein, C.: Experimental study of minimum cut algorithms. In: *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'97)*, New Orleans, pp. 324–333 (1997)
6. Clarke, E.M., Grumberg, O., Long, D.E.: Model Checking and Abstraction. *ACM Transactions on Programming Languages and Systems* 16(5), 1512–1542 (1994)
7. Courtois, P.J., Parnas, D.L.: Documentation for Safety Critical Software. In: *proceedings of 15th International Conference on Software Engineering*, Baltimore, MD (May 17- 21, 1993)
8. Cukic, B., Ammar, H.H., Lateef, K.: Identifying High-Risk Scenarios of Complex Systems Using Input Domain Partitioning. In: *proceedings of the 9th International Symposium on Software Reliability Engineering (ISSRE'98)*, November 4-7, 1998 Paderborn, Germany, pp. 164–173 (1998)

9. Heitmeyer, C., Archer, M., Bharadwaj, R., Jeffords, R.: Tools for constructing requirements specifications: The SCR toolset at the age of ten. *International Journal of Computer Systems Science and Engineering* 5, 95–114 (2005)
10. Heitmeyer, C.L., Jeffords, R.D., Labaw, B.G.: Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology* 5(3), 231–261 (1996)
11. Heitmeyer, C., Kirby Jr., J., Labaw, B., Archer, M., Bharadwaj, R.: Using Abstraction and Model Checking to Detect Safety Violations in Requirements Specifications. *IEEE Transactions on Software Engineering* 24(11), 927–948 (1998)
12. Heitmeyer, C., Kirby, J., Labaw, B., Bharadwaj, R.: SCR*: A Toolset for Specifying and Analyzing Requirements. In: Vardi, M.Y. (ed.) *CAV 1998. LNCS*, vol. 1427, Springer, Heidelberg (1998)
13. Henzinger, T.A., Qadeer, S., Rajamani, S.K.: You Assume, We Guarantee: Methodology and Case Studies. In: Vardi, M.Y. (ed.) *CAV 1998. LNCS*, vol. 1427, pp. 440–451. Springer, Heidelberg (1998)
14. Jeffords, R., Heitmeyer, C.: Automatic Generation of State Invariants from Requirements Specifications. In: Vaudenay, S. (ed.) *FSE 1998. LNCS*, vol. 1372, Springer, Heidelberg (1998)
15. Jeffords, R.D., Heitmeyer, C.L.: A Strategy for Efficiently Verifying Requirements Specifications Using Composition and Invariants. In: *proceedings of 9th European Software Engineering Conference held jointly with 11th International Symposium on Foundations of Software Engineering (ESEC/FSE'03)*, Helsinki, Finland (September 1–5, 2003)
16. Parnas, D.L., Clements, P.C., Weiss, D.M.: Modular Structure of Complex Systems. In: *Proceedings of the 7th International Conference on Software Engineering*, pp. 408–417 (1984)
17. Weiser, M.: Program Slicing. *IEEE Transactions on Software Engineering* SE-10(4), 352–357 (1984)
18. Requirements Specification for Personnel Access Control System: National Security Agency (2003)

A Pattern-Based Approach for Modeling and Analyzing Error Recovery^{*}

Ali Ebzenasir^{1,**} and Betty H.C. Cheng²

¹ Department of Computer Science
Michigan Technological University
Houghton, Michigan 49931, USA
aebzenas@mtu.edu

<http://www.cs.mtu.edu/~aebzenas>

² Department of Computer Science and Engineering
Michigan State University
East Lansing, Michigan 48824, USA
chengb@cse.msu.edu
<http://www.cse.msu.edu/~chengb>

Abstract. Several approaches exist for modeling recovery of fault-tolerant systems during the requirements analysis phase. Most of these approaches are based on design techniques for recovery. Such design-biased analysis methods unnecessarily constrain an analyst when specifying recovery requirements. To remedy such restrictions, we present an object analysis pattern, called the *corrector* pattern, that provides a generic reusable strategy for modeling error recovery requirements for embedded systems. In addition to templates for constructing structural and behavioral models of recovery requirements, the corrector pattern also contains templates for specifying properties that can be formally verified to ensure the consistency between recovery and functional requirements. Additional property templates can be instantiated and verified to ensure the fault-tolerance of the system to which the corrector pattern has been applied. We validate our analysis method in terms of UML diagrams, where we (1) use the corrector pattern to model recovery in UML behavioral models, (2) generate and model check formal models of the resulting UML models, and (3) visualize the model checking results in terms of the UML diagrams to facilitate model refinement. We demonstrate our analysis method in the context of an industrial automotive application.

Keywords: Requirements Analysis, Fault-Tolerance, Formal Methods, Error Recovery, Corrector, UML.

^{*} This work was partially sponsored by NSF grants EIA-0000433, EIA-0130724, CDA-9700732, CCR-9901017, CNS-0551622, CCF-0541131, NSF CAREER CCR-0092724, ONR grant N00014-011-0744, DARPA Grant OSURS01-C-1901, Siemens Corporate Research, a grant from the Michigan State University's Quality Fund, and a grant from Michigan Technological University.

^{**} The work presented here was performed largely while this author was a postdoctoral researcher at Michigan State University with support from NSF and ONR.

1 Introduction

High costs and complexity of developing fault-tolerant distributed systems are largely due to the crosscutting nature of fault-tolerance concerns and the requirement for coordinated recovery by system components [1]. The complexity of developing *fault-tolerant embedded systems* is exacerbated as embedded systems should operate under the constraints of physical systems. Several existing approaches (such as ROPES [2] and COMET [3]) put more emphasis on the requirements analysis phase by developing and analyzing object-oriented conceptual models (i.e., abstract implementation-independent models that capture system requirements) of embedded systems before entering design and implementation phases. For example, ROPES [2] requires the development of both structural and behavioral models that specify correctness conditions for design solutions refined from a conceptual model. To facilitate the rigorous development of *fault-tolerant* embedded systems, we follow the above approaches in presenting a method for modeling and analyzing *nonmasking* fault-tolerance in embedded software systems at the requirements analysis phase, where, in the ideal case, a nonmasking fault-tolerant system guarantees error recovery [4].¹

Numerous approaches exist for the *design and implementation* of recovery from error conditions in sequential [6, 7] and concurrent (respectively, distributed) programs [1, 8, 9]. For example, Randell [6] presents the concept of recovery blocks for implementing recovery in sequential programs and uses atomic actions for the design of error recovery in asynchronous concurrent programs [1]. Cristian [7] focuses on the concept of exceptional conditions and systematic handling thereof. Schneider [8] presents a replication-based method for recovery from failures in client-server distributed systems. Saridakis [10] presents a set of design patterns based on existing recovery mechanisms [9]. The UML profile for fault-tolerance [11] and several aspect-oriented approaches [12, 13, 14] use redundancy of services to *mask* faults, which is sometimes impractical and costly [15]. Moreover, most existing *analysis* methods for fault-tolerance [16, 17, 18, 19] assume that a specific fault-tolerance design mechanism will be used (e.g., exception handling, redundancy) and specify analysis requirements within those design constraints. As such, error recovery requirements may be overly constrained and preclude useful solutions or even finding a solution. For example, it is difficult to specify and model self-stabilization [20] solely based on exception handling. While a specific error recovery mechanism should certainly be considered at design time based on the constraints of the problem at hand, we believe that, at the requirements analysis level, an abstract specification of error containment and state restoration in distributed systems helps developers to detect the inconsistencies between recovery and functional requirements independent of the design and implementation techniques.

¹ We emphasize that our proposed approach facilitates the creation and analysis of the *conceptual models* of nonmasking fault-tolerant systems. For such models to be realized in practice, one has to use fault-tolerance preserving refinements [5] to develop design and implementation artifacts.

In order to specify and analyze recovery for embedded systems, we introduce an object analysis pattern, called the *corrector* pattern, that provides a reusable strategy for eliciting and specifying error correction constraints in UML object models. Object analysis patterns apply a similar approach to that used by design patterns [21], but instead of focusing on design they address the construction of the conceptual model of a system [2]. Patterns for the analysis stage of software development are not new (see [22, 23]). For example, Fowler [22] presents a method for characterizing recurring ideas in business modeling as reusable analysis patterns. Konrad *et al.* [23] present domain-specific object analysis patterns for analyzing the conceptual models of embedded systems. We introduce the corrector pattern that serves to modularize the requirements of error recovery, thereby facilitating tracing and reasoning about recovery in different stages of system development.

Our method comprises fault modeling, recovery modeling, and automated analysis of the UML models of fault-tolerant embedded systems. Specifically, to construct the UML model of a nonmasking Fault-Tolerant System (FTS), we start with the UML model of its fault-intolerant version, where a Fault-Intolerant System (FIS) meets its functional requirements in the absence of faults (i.e., when no faults occur) and provides no guarantees in the presence of faults (i.e., when faults occur). Then we model faults in the UML model of the FIS to produce a *model with faults*. We use the notion of state perturbation to model different types of faults in UML behavior diagrams [20, 4, 15]. Next, we specify error states reached due to the occurrence of faults from where recovery should be provided. Subsequently, we add instances of the corrector pattern to the model with faults to model recovery and to generate a *candidate* UML model of a nonmasking FTS. To generate a valid UML model of the nonmasking FTS, we have to ensure that the candidate UML model is *interference-free*. That is, in the absence of faults, the candidate model meets all functional requirements of the FIS, and in the presence of faults, the candidate model meets recovery requirements; i.e., when faults stop occurring, the system will eventually recover from error conditions. To ensure interference-freedom, we extend McUmber and Cheng's UML formalization framework [24] to generate formal specifications of faults, fault-tolerance and functional concerns in the Promela modeling language [25]. Subsequently, we use the SPIN model checker [25] to detect inconsistencies between the corrector pattern and the functional UML model. The automated analysis with the SPIN model checker coupled with a new visualization tool, called Theseus [26], that animates counterexample traces in terms of the original UML diagrams and generates corresponding sequence diagrams enables a roundtrip engineering process for modeling and analyzing recovery requirements.

We demonstrate our approach by modeling and analyzing an adaptive cruise control (ACC) system in UML obtained from industry. We have also validated the corrector pattern for several other industrial examples [27] including a diffusing computation program for a hierarchical distributed system [28]. The remainder of this paper is organized as follows. Section 2 presents an overview of the proposed

approach. Section 3 introduces an approach to modeling faults and nonmasking fault-tolerance in terms of UML state and sequence diagrams. Section 4 presents a systematic method for eliciting and specifying error conditions. Section 5 presents the corrector pattern. Section 6 focuses on formal analysis of the UML model of FTSs using the model checker SPIN [25]. Section 7 discusses related work. Finally, Section 8 gives concluding remarks and discusses future work.

2 Overview

In this section, we present an overview of our pattern-based modeling approach. Figure 1 illustrates the steps of our approach (including modeling faults, specifying error conditions, instantiating the corrector pattern, and automated analysis) annotated with the relevant paper section number on the lower left corner of each step. For a given FIS \mathcal{S} and a fault-type f , we start from a *valid* UML model of \mathcal{S} that captures all global properties of \mathcal{S} that should hold in the absence of faults f . A UML model of \mathcal{S} uses class diagrams to capture structural constraints of \mathcal{S} and uses behavior diagrams to capture the behaviors of \mathcal{S} . Subsequently, we model the effect of f on each component of \mathcal{S} modeled as an object in UML. Then we specify the error conditions that denote the set of states from where recovery should be provided. Subsequently, to specify the requirements of detecting and correcting error conditions, we compose instances of the proposed corrector pattern with the UML model of \mathcal{S} , which results in creating a candidate UML model of a nonmasking version of \mathcal{S} . To ensure the correctness of such a composition, we first employ an extended version of the Hydra [24] formalization tool to generate the Promela specifications of the candidate UML model. Then we use the SPIN model checker to verify the correctness of the composition. If the model checking is successful, then the candidate model is indeed a UML model of a nonmasking fault-tolerant version of \mathcal{S} . Otherwise, using the Theseus visualization tool [26], we animate the analysis errors (illustrated as counterexamples) in the state and sequence diagrams. Using such a visualization, we help developers to revise the candidate model to eliminate the inconsistencies between functional and recovery requirements. The revised model can again be model checked until analysis is successful or an upper bound is reached in the number of model checking attempts. The latter case indicates that, given the available resources, we have not been able to verify that the current instance of the corrector pattern is consistent with the functional concerns.

In our approach, we separate the functional concerns from fault-tolerance concerns where we start with a valid UML model of the FIS and incrementally add fault-tolerance concerns in terms of the instances of the corrector pattern. The motivation behind such a separation of concerns is two-fold. First, fault-tolerance is added only for dealing with faults, and the added fault-tolerance concerns should not conflict with functional requirements in the absence of faults. Second, fault-tolerance requirements evolve as we encounter new types of faults. Thus, there exist two options: either (1) develop from scratch a system that meets its functional requirements in the absence of f and provide desired functionalities

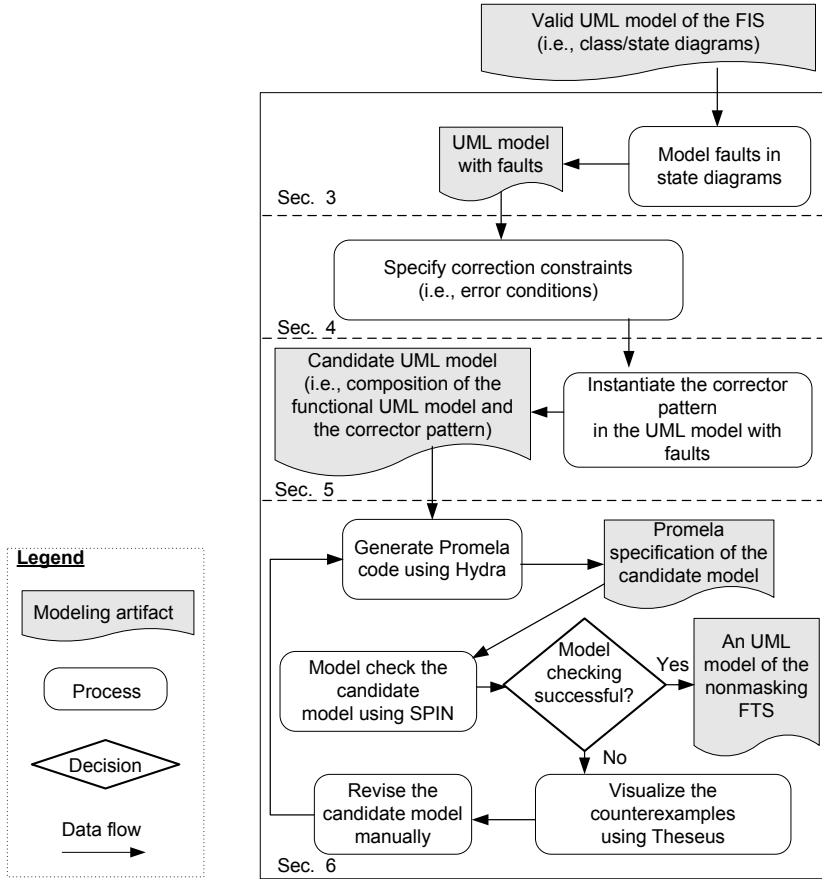


Fig. 1. An overview of the proposed approach

in the presence of f , or (2) incrementally add new fault-tolerance functionalities while preserving the existing functionalities in the absence of f . We adopt the latter as it seems to be less expensive than the former approach and better handles legacy systems.

3 Modeling

In this section, we present the basic concepts of modeling FISs, faults, and nonmasking fault-tolerance in UML. The motivation behind using UML is two-fold. First, UML is a modeling language well-accepted in both academia and industry. Second, UML state diagrams enable us to capture any form of recovery that can be expressed in a state machine-based formalism.

3.1 UML Models

We use conventional UML notations [29] to represent the UML-based conceptual models of FISs (respectively, FTSs). Since our focus is on modeling and analyzing fault-tolerant embedded systems, we follow Douglass [2] in using UML class diagrams to model structural constraints of (software and hardware components of) embedded systems during the object analysis phase. We use UML behavior diagrams to capture high-level *behavioral* information of UML object models. The combination of class and behavior diagrams yields a conceptual *object analysis model*.

Depending on the semantics of object interactions, the complexity of automatic analysis of an FTS varies from polynomial (in a shared memory model [30]) to undecidable (in an asynchronous message-passing model [31]). For example, Kulkarni and Arora [30] show that automated analysis of nonmasking fault-tolerance for models of distributed systems has an exponential complexity (in the size of the model). To facilitate an automated analysis method with a manageable complexity, we consider a *high atomicity* model where transitions of state diagrams are executed atomically and any instance of message passing between two objects takes place in an atomic step. An atomic transition is executed in a *test-and-set* fashion. A motivation behind this assumption is that modeling fault-tolerance in a high atomicity model provides an *impossibility test* in the early stages of development (which could potentially reduce development costs). That is, if a conceptual model of an FTS cannot be derived from the conceptual model of its fault-intolerant version in the high atomicity model, then it would be impossible to derive a model of the FTS in a lower atomicity level. A theoretical investigation of this claim can be found in [32].²

Underlying Computational Model. In a UML object model M with n objects O_1, \dots, O_n , we denote the state transition diagram of each object O_i by $SD_i = \langle S_i, \delta_i \rangle$, where S_i is the set of states in the state diagram SD_i and δ_i denotes the set of transitions of SD_i ($1 \leq i \leq n$). A state of an object O_i is a valuation of its state variables (i.e., attributes). A transition of O_i is of the form $(a, evt[grd]/act, b)$, where a and b are states, evt denotes a triggering event, grd represents a guard condition and act denotes an action that O_i executes during a transition from a to b . A *global state predicate* is defined over a set of states of multiple objects. A *local state predicate* is specified over the set of states of only one object O_i (i.e., S_i). A *scenario* is a sequence of states $\langle s_0, s_1, \dots \rangle$, where each s_i is a state of some object O_j ($1 \leq j \leq n$). The messages in the sequence diagrams correspond to the transitions in the state diagrams. We use UML sequence diagrams to represent scenarios in that a sequence diagram may capture multiple scenarios. A *behavior* of an object O_j ($1 \leq j \leq n$) is a scenario $\langle s_0, s_1, \dots \rangle$ such that $\forall s_i : i \geq 0 : s_i \in S_j$.

² In cases such atomicity assumptions do not hold (e.g., distributed systems), one can use existing tolerance-preserving refinement techniques (e.g., [5]) to generate a refined model from a high atomicity model developed using our proposed approach.

Modeling Functional Requirements. In order to model functional requirements, we extend Gouda and Arora's [4] notion of *closure*, where a fault-tolerant system remains in a set of legitimate states as long as no faults have occurred. A set of legitimate states (also called an *invariant*) can be computed by identifying the set of states that are reachable by system actions from a given set of initial states. (Techniques of how to extract an invariant from user requirements are beyond the scope of this work. Examples can be found in [33,34].) In the invariant, no system action violates its *safety* and *liveness* requirements. Intuitively, safety requirements stipulate that nothing bad ever happens and the liveness requirements specify that something good will eventually occur. For example, in a cruise control system, the actual speed of the car must not exceed 1% of the desired speed set by the driver (i.e., safety), and when the driver applies the brakes, the cruise control system will eventually be deactivated (i.e., liveness). We represent safety requirements by a set of transitions, say \mathcal{B} , that must not occur in the behaviors of any object. Since we start with a functional model of an FIS system that meets its safety and liveness requirements in the absence of faults (i.e., when no faults occur) and incrementally model fault-tolerance, we do not explicitly specify liveness requirements. Nonetheless, we require that while modeling fault-tolerance concerns, no deadlock states (states with no outgoing transitions) should exist in the invariant of the nonmasking FTS. The deadlock freedom requirement captures the fact that, in the absence of faults, fault-tolerant embedded systems have non-terminating computations and always react to their environment. We say a scenario $\langle s_0, s_1, \dots \rangle$ *meets safety requirements* iff (if and only if) $\forall i : i \geq 0 : (s_i, s_{i+1}) \notin \mathcal{B}$. A scenario $\langle s_0, s_1, \dots \rangle$ *meets liveness requirements* iff every state s_i , for $i \geq 0$, has a next state and some desired conditions eventually become true in that scenario. Thus, an invariant has two properties: (1) starting from any state in the invariant, the subsequent states are also in the invariant (i.e., *closure*), and (2) from every state in the invariant, all scenarios meet safety and liveness requirements. A UML model M *meets its functional requirements* iff there exists a non-empty invariant \mathcal{I} for M . A *functional scenario* is a scenario whose states all belong to the invariant.

Running Example: Adaptive Cruise Control (ACC). The ACC system comprises a standard cruise control system and a radar system to automatically adjust the distance between the car and the front vehicle (i.e., *target vehicle*) for collision avoidance; the ACC requirements were obtained from industrial collaborators. The ACC system has different modes of operation (see Figure 2), namely *closing*, *coasting*, *matching*, *disengaged* or *resume* mode. When the radar detects a target vehicle, the ACC system enters the *closing* mode. In the closing mode, the goal is to control the way that the car approaches the target vehicle, and to keep the car in a fixed *trail distance* from the target vehicle with a zero relative speed. The *trail distance* is the distance that the target vehicle travels in a fixed amount of time (e.g., 2 seconds). The distance to the target vehicle must not be less than a *safety zone*, which is 90% of the *trail distance*. The ACC system calculates a *coasting distance* that is the distance at which the car should

start decelerating in order to achieve the trail distance; i.e., the car enters the *coasting* mode. In the *matching* mode, the relative speed of the car is zero; i.e., the speed of the car matches the speed of the target vehicle. In cases where the speed of the car is so fast (greater than a maximum speed v_{max}) that a collision is unavoidable, the ACC system must raise an alarm for the driver, and must deactivate the cruise control system, i.e., the *disengaged* mode. When the radar loses the target vehicle and the cruise control system is active, the system is in the *resume* mode.

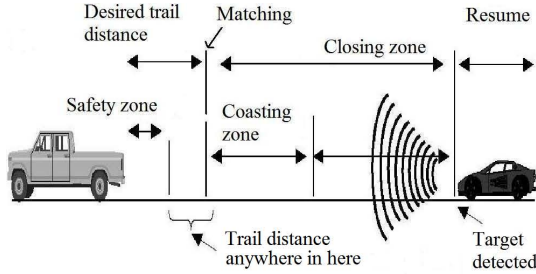


Fig. 2. The adaptive cruise control system

The class diagram of the ACC system includes three main classes, namely Control, Car, and Radar (see Figure 3). (We use Sans Serif font to denote state variables, methods and classes.) (i) The Control class has a set of Boolean state variables that represent different modes of the ACC system. The Brakes state variable is set when the control receives a signal from the brakes subsystem indicating that the brakes have been applied. The ACC system must be disengaged when the Control receives a Brakes signal. The method `setpUpdate()` updates the *setpoint*, which is the desired speed determined by the driver. The Radar sets `Control.target` to true by invoking the `targetDet()` method. Depending on the computed trail distance, the Control object also calculates the *safetyZone* such that the distance to the target vehicle never becomes the *safetyZone* value. (ii) The Car class models the engine management functionalities such as acceleration and deceleration. A Car object matches the real speed of the car, denoted *realv*, with the setpoint (using the method `matchSpeed()`). The car calculates its real speed using the data received from the speed sensors located in the car. The `getRealV()` method may be invoked by the Control to receive the real speed of the car. (iii) The Radar measures the distance of the car to the target vehicle, kept in the state variable *currDist*, which is also used by the Control by invoking `Radar.getDistance()`. The Radar also measures the speed of the target vehicle (kept in `Radar.targetSpeed`) that can be accessed using the `Radar.getTargetSpeed()` method.

An invariant of the ACC system (denoted \mathcal{I}_{ACC}) is a global state predicate that specifies a set of states, in which (i) if a target vehicle has been detected then the ACC system is in one of the following modes: *closing*, *coasting*, *matching*, or *disengaged*; (ii) the distance with the target vehicle is greater than the safety

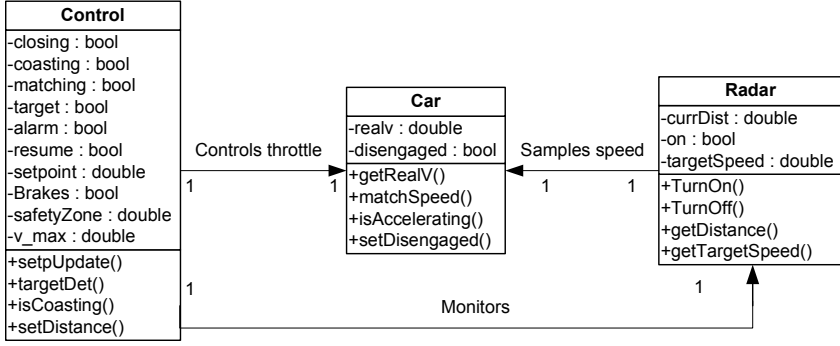


Fig. 3. Excerpted class diagram of the ACC system

zone distance; (iii) if the ACC system is in the *cruise* mode and the target is lost then ACC will go to the *resume* mode; (iv) if the driver applies the brakes then the ACC system must be in the *disengage* mode, and (v) if the closing speed of the car is greater than the maximum speed v_{max} , then the ACC system must alarm the driver of a potential collision and must disengage. Therefore, the invariant \mathcal{I}_{ACC} is equal to the following set of states:

$$\{s : (target(s) \Rightarrow (closing(s) \vee coasting(s) \vee matching(s) \vee disengaged(s))) \wedge (safetyZone(s) < currDist(s)) \wedge ((\neg target(s) \wedge cruise(s)) \Rightarrow resume(s)) \wedge (Brakes(s) \Rightarrow disengaged(s)) \wedge (realv(s) > v_{max}(s) \Rightarrow (alarm(s) \wedge disengaged(s)))\}$$

Notation. $var(s)$ denotes the value of a system variable var in a state s .

Note that the above predicate is specified in terms of the variables of all three objects. The variables `target`, `closing`, `coasting`, `matching`, `resume`, `alarm`, `Brakes`, v_{max} and `safetyZone` belong to the **Control** object. `disengaged` and `realv` are state variables of the **Car** object, and `currDist` is in the **Radar** object.

3.2 Modeling Faults in UML

In this section, we illustrate how to model faults in UML behavior diagrams in the context of the ACC system. Since our focus is on the behavioral object models, we omit the fault modeling at the class diagram level (see [27] for details).

Modeling Faults in State Diagrams. We systematically model a fault-type as a *set of transitions* in UML state diagrams (see Figure 4). Representing faults as a set of transitions has already appeared in previous work [20, 15], and it is known that state perturbation is sufficiently expressive to represent different types of faults (e.g., crash, input-corruption, Byzantine) from different behavioral categories (e.g., transient, intermittent, permanent) [20, 15]. Moreover, we assume that faults stop occurring in a finite amount of time so that eventually

recovery can occur [20, 35]. Depending on the occurrence of faults, we classify faults into two categories of *conditional* and *arbitrary* faults. A *conditional* fault-type may occur only in particular states of the state transition diagram of an object. An *arbitrary* fault-type has no precondition and may occur at any state (e.g., environmental noise). Given a conditional fault-type f , we model the effect of f on the state diagram SD_i of each object O_i by introducing a new set of transitions in SD_i denoted f_i , for $1 \leq i \leq n$ (see Figure 4). We denote the set of *transitions of SD_i in the presence of faults f_i* by $\delta_i \cup f_i$.

We model an arbitrary fault-type as a separate fault state transition diagram (e.g., FD_1 in Figure 5) that executes concurrently with an object state diagram (e.g., SD_1 in Figure 5). In Figure 5, the transitions of the arbitrary faults in FD_1 may trigger at any state of the state diagram SD_1 . Modeling fault transitions is a modeling activity similar to drawing regular transitions of an object O_i in its state diagram SD_i . The key difference is in the semantics of fault transitions in that an object O_i does not have control over the execution of faults f_i (see dashed arrows in Figure 4), whereas the execution of regular transitions (see solid arrows in Figure 4) is controlled by the thread of execution in O_i .

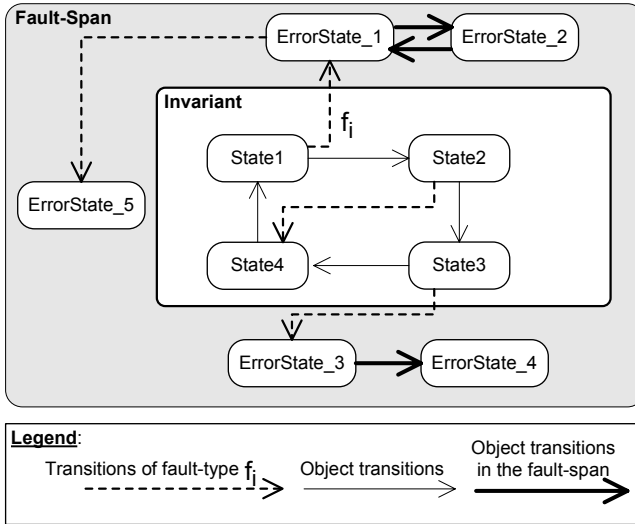


Fig. 4. Modeling conditional faults in UML state diagrams

When modeling a fault type f_i in a state diagram SD_i of the UML model of the FIS, modelers should identify the effect of f_i on the behavior of each object O_i . In the absence of faults, an object O_i of the FIS is in its invariant.³ Modelers should identify the scope of the states reachable by a combination of fault and regular transitions from the invariant, which is called the *fault-span* of O_i for

³ Faults are the cause of errors, and from error states failures may occur [36].

fault f_i (denoted f_i -span of O_i) [30]. For example, in Figure 4, all error states are only reachable when faults occur. Thus, introducing faults in a state diagram may require new states and transitions to be added to that state diagram. In fact, given a UML model M , its invariant \mathcal{I} and a fault-type f , starting from \mathcal{I} , the set of states reachable by a combination of fault and system transitions comprises the *global fault-span* of M , denoted f -span of M . More precisely, the f -span of M has two properties: (1) the f -span of M contains \mathcal{I} , and (2) starting from every state in the f -span of M , any fault or system action will result in another state in the f -span; i.e., *closure* of the f -span of M in the set of system and fault actions. In Figure 5, the fault-span is identified by calculating the asynchronous automata-theoretic product of the two state machines SD_1 and FD_1 , which results in a new state diagram that simultaneously includes fault and regular transitions. A behavior of an object that originates in its fault-span outside its invariant may lead to failures (i.e., violate safety requirements, fall into non-progress cycles, or reach a deadlock state). For example, in Figure 4, if the object is in State1 then the faults f_i may non-deterministically transition to ErrorState_1 from where the object may either be trapped in a non-progress cycle (comprising ErrorState_1 and ErrorState_2) or be deadlocked in ErrorState_5.

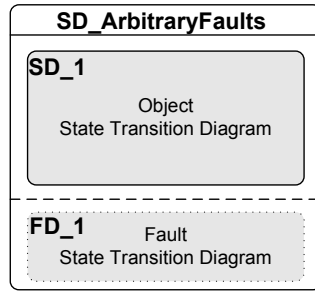


Fig. 5. Modeling arbitrary faults in a state diagram

ACC Example: The ACC system is subject to an arbitrary fault-type f_{ACC} that may non-deterministically set the disengaged signal in the Car object to false. Hence, we model the effect of f_{ACC} on the Car object as a state machine concurrent with the car state machine (see Figure 6). Note that in the f_{ACC} state machine in Figure 6, once the fault transition from State1 to State2 sets Car.disengaged to false, Car.disengaged remains false until a system recovery action resets it back to true.

Modeling Faults in Sequence Diagrams. In UML sequence diagrams, we model the effect of a fault-type f_i on an object O_i as a self message to O_i that may occur non-deterministically (see Figure 7). Such a representation of faults in sequence diagrams is based on how faults are modeled in the state diagram of O_i . Thus, modeling faults in SD_i affects all sequence diagrams in which O_i is involved. Such sequence diagrams represent *scenarios with faults*. Formally, a scenario with fault f is a sequence of states $\sigma = \langle s_0, s_1, \dots \rangle$, where for every

transition $t = (s_i, s_{i+1})$ in σ either t is a valid transition of a functional object or t is a fault transition. To identify scenarios with faults, modelers should update every scenario in which O_i is involved, and should discover failure scenarios that could take place due to the occurrence of faults. The identification of such failure scenarios helps developers to recognize the functional objects that could participate in recovery.

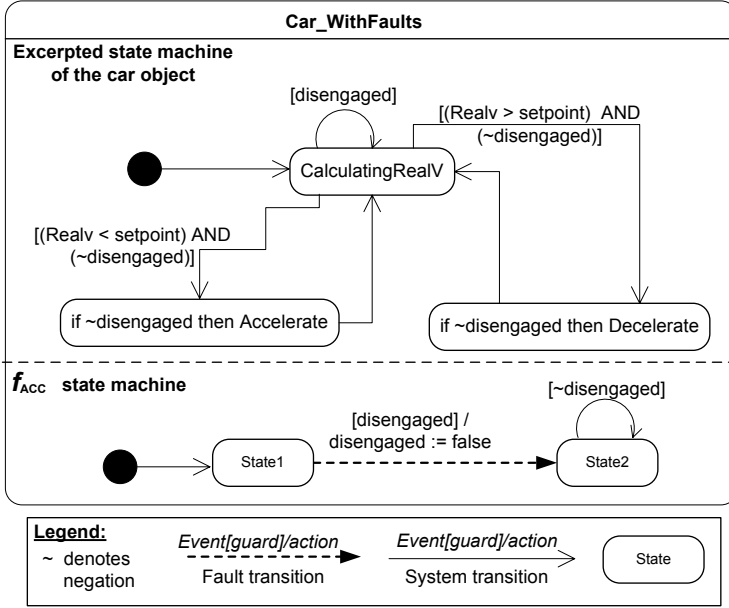


Fig. 6. Modeling faults in the state transition diagram of the car

ACC Example: In Figure 7, once the Control detects a Brakes signal, it invokes `Car.setDisengaged()` illustrating that the engine controller should be deactivated (i.e., *disengaged*). However, if faults f_{ACC} occur, then the *disengaged* flag will be reset to false, which in turn results in the reactivation of the engine controller, possibly resulting in acceleration while brakes are applied. This is a scenario with faults that must be corrected.

UML models with faults. Modeling a fault-type f in the state diagrams of a UML model M creates a UML model M_f that has been augmented with fault f . We call M_f a UML model with faults f .

Comment on the complexity of modeling faults, fault-span and scenarios with faults. The proposed modeling approach in this section includes three main tasks, namely modeling (conditional or arbitrary) faults in state diagrams, modeling fault-spans in state diagrams and modeling scenarios with faults. The fault modeling task should be done manually and the other two tasks can be automated. While modeling (conditional and arbitrary) fault transitions in state

diagrams may seem to be a tedious task for large systems, we argue that (1) the scale of such a modeling activity does not go beyond the complexity of modeling regular transitions in the state diagrams of all functional objects, and (2) techniques that facilitate the modeling of regular transitions can directly be reused to facilitate fault modeling. In cases where more than one type of fault should be modeled, UML extension techniques (e.g., stereotyping) would help developers to distinguish the transitions of different fault-types and their corresponding fault-spans. Since it is difficult to manually identify all scenarios with faults (respectively, model the fault-span of an object) and the number of such scenarios may increase exponentially, we are currently investigating the integration of a software tool we have previously developed [37, 38], called Fault-Tolerance Synthesizer (FTSyn), in UML as FTSyn automatically generates fault-span and scenarios with faults in terms of finite-state automata.

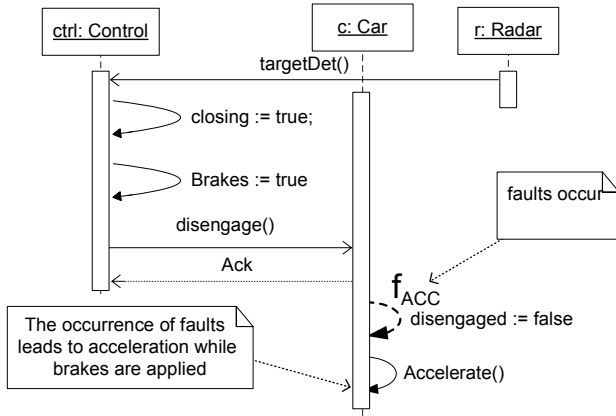


Fig. 7. Coasting scenario in the presence of f_{ACC}

3.3 Modeling Nonmasking Fault-Tolerance

In this section, we extend the definition of nonmasking fault-tolerance from Arora [15] in the context of UML models. Intuitively, *nonmasking* fault-tolerance requires recovery to the invariant after faults stop occurring [15]. More precisely, let \mathcal{S} be an FIS, \mathcal{I} be an invariant of \mathcal{S} , and f be a given fault-type perturbing the state of \mathcal{S} . (Note that the FIS \mathcal{S} provides no guarantees about its behavior when f occurs.) A system \mathcal{S}' is a nonmasking f -tolerant (i.e., nonmasking fault-tolerant against f) version of \mathcal{S} if and only if the following conditions are satisfied: (1) in the absence of f , the FTS \mathcal{S}' meets the functional requirements specified for \mathcal{S} , and (2) in the presence of f , the FTS \mathcal{S}' guarantees recovery to \mathcal{I} .

Before defining what we mean by a nonmasking fault-tolerant UML model, we define *recovery scenarios*. Let M be a UML model of \mathcal{S} and \mathcal{I} be an invariant of \mathcal{S} defined in M . We say a scenario $\sigma = \langle s_0, s_1, \dots \rangle$ in M *recovers to the invariant* \mathcal{I} iff $\exists i : i \geq 0 : s_i \in \mathcal{I}$. Note that once a state in the invariant is reached, the

closure property guarantees that the system remains in the invariant as long as there are no faults. We say a scenario σ in the UML model M *violates recovery requirements* iff σ does not recover to the invariant of M . Violation scenarios could take place if a deadlock state or a non-progress cycle is reached due to the occurrence of f . We say a UML model M *recovers to the invariant* \mathcal{I} iff all scenarios of M recover to \mathcal{I} . Accordingly, a UML model M *violates recovery requirements* iff there exists a scenario that violates recovery requirements. We say a UML model M' (derived from M) is nonmasking f -tolerant if M' satisfies the following conditions: (1) the set of functional scenarios of M' is a non-empty subset of the set of functional scenarios of M starting in a subset of \mathcal{I} , and (2) all scenarios with fault f recover to \mathcal{I} .

4 Specifying Error Conditions

In order to model recovery, we need to specify the set of error states from where recovery should be provided. The notion of invariant simplifies the task of specifying error states as it characterizes the set of states from where functional requirements are guaranteed to be met in the absence of faults. The occurrence of faults may falsify the invariant, thereby reaching states from where failures may occur. Thus, for a given UML model M and its invariant \mathcal{I} , the weakest set of error states is $\neg\mathcal{I}$. However, some states in $\neg\mathcal{I}$ may be unreachable by either system or fault actions. In fact, for a specific fault-type f , the set of reachable error states is equal to the intersection of $\neg\mathcal{I}$ and $\mathcal{FS} - \mathcal{I}$, where \mathcal{FS} denotes the f -span of M . A fault-intolerant system may stay in $\mathcal{FS} - \mathcal{I}$ forever for two reasons: reaching a deadlock state or falling in a cycle whose states all belong to $\mathcal{FS} - \mathcal{I}$ (called a *non-progress cycle*).

In order to ensure recovery, we have to resolve deadlock states and non-progress cycles. For programs whose processes can read and write all program variables in an atomic step, resolving deadlock states amounts to the addition of actions that establish the truth value of \mathcal{I} once it is falsified. Such actions are called *convergence* actions [4] as they guarantee the convergence of system behaviors to its invariant. Likewise, non-progress cycles can be resolved by breaking cycles and adding convergence actions. However, in concurrent and distributed programs, resolving deadlock states and non-progress cycles is a non-trivial task [30, 39]. To illustrate the complexity of providing recovery, consider a distributed program with two processes and an invariant $((x - y) = c) \wedge (y \geq z)$, where x, y , and z are program variables and c is a constant. In an error state where the invariant does not hold, a process that cannot read z may decrease the value of y to establish the equality $(x - y = c)$ for the sake of recovery. This recovery action may potentially violate the second conjunct of the invariant due to decreasing the value of y . In such cases, convergence should be provided in a coordinated fashion. In the above example, if y is left unchanged and each process is allowed to modify only one of the variables x and z , then coordinated recovery is achievable. In the next section, we present the corrector pattern, which

facilitates modeling and analysis of the recovery of concurrent and distributed programs and provides a means to verify the correctness of such recovery.

5 Corrector Pattern

In this section, we introduce a template for the corrector pattern that we use for modeling and analyzing nonmasking fault-tolerance. We have also developed a corresponding detector pattern [27] to specify error detection, but due to space constraints, we do not include it here. While design patterns are traditionally classified in terms of structural, behavioral, and creational patterns [21], the corrector pattern provides a reusable strategy (for decomposing error conditions) that can be refined to different design mechanisms. In order to facilitate its use, we define a template for the corrector pattern based on the fields used in the design patterns presented by Gamma *et al.* [21], with modifications to reflect analysis-level information. For example, we do not use the **Implementation** and **Sample Code** fields. The **Structure** field captures structural constraints of the corrector pattern represented by UML class diagrams. The corrector pattern also includes several new fields that are added for the purpose of specifying and analyzing fault-tolerance concerns. For example, the corrector pattern includes the *Correction Requirements* field that specifies a set of requirements that must be met by the corrector pattern to ensure that the corrector pattern is itself nonmasking fault-tolerant. We use the ACC system to demonstrate how to use the corrector pattern to add nonmasking f_{ACC} -tolerance to the ACC system. Next, we describe the fields of the corrector pattern. Example application of each field to the ACC system is denoted in *italics*.

Intent. The corrector pattern captures the recurring problem of restoring the state of a computing system from one state predicate to another (e.g., from outside an invariant to the invariant).

Correction Predicate. A *correction predicate*, say X , is a condition whose truth value should be established (e.g., invariant). In a UML model M , a correction predicate is a state predicate that could be either local or global. In a distributed system, it is difficult for an object to correct a global correction predicate X in an atomic step [40]. Thus, it is desirable to decompose X into a set of local predicates X_1, \dots, X_n , and to specify the correction of X based on the correction of X_1, \dots, X_n , where each X_i ($1 \leq i \leq n$) represents the local state of a system component. Since global deadlock and non-progress conditions are often specified in terms of a conjunction of the local state of all objects, we limit the scope of the application of the corrector pattern to the correction of conjunctive error predicates.⁴

ACC Example: The occurrence of f_{ACC} may perturb the ACC system to a state s , where the cruise control system is engaged in engine management even though

⁴ Conjunctive predicates comprise an important class of predicates in distributed systems [41].

brakes have been applied. This introduces a deadlock state as long as the brakes are applied. We represent this error condition by the conjunctive predicate $(X_{control} \wedge \neg X_{car})$ that should be corrected, where $X_{control} \equiv \text{Control.Brakes}$ and $X_{car} \equiv \text{Car.disengaged}$. The correction predicate X_{ACC} is the negation of the above error condition, i.e., $X_{ACC} \equiv \neg(X_{control} \wedge \neg X_{car}) \equiv (X_{control} \Rightarrow X_{car})$. Note that if X_{ACC} is false (i.e., error has occurred), then the invariant \mathcal{I}_{ACC} (specified in Section 3) is violated. To provide nonmasking f_{ACC} -tolerance, we must ensure that the condition X_{ACC} will eventually hold after f_{ACC} stops occurring. Moreover, in the context of the ACC example, there is only one way to correct X_{ACC} ; it is by setting the state variable Car.disengaged to true (because the state variable Control.Brakes represents an input signal and cannot be changed).

Corrector Elements (Participants). We use corrector elements c_i , $1 \leq i \leq n$, such that each c_i is responsible for correcting X_i . Each corrector element c_i is indeed a participant of the corrector pattern and has its own correction predicate X_i .⁵

Distinguished Element. An element c_{index} ($1 \leq index \leq n$) that finalizes the correction of X based on the correction of X_1, \dots, X_n is called the *distinguished element*.

Structure. We present two basic structures for the corrector pattern: *sequential* and *parallel*. The correction of X can be done either (i) sequentially, where participants c_i , for $1 \leq i \leq n$, correct their correction predicates X_i one after another, or (ii) in parallel, where all elements c_i correct their correction predicates concurrently. For example, if the inter-object associations in an UML object model form a linear (respectively, hierarchical) structure then a sequential (respectively, parallel) corrector is more appropriate. We illustrate the structure of the sequential corrector pattern in Figure 8. The shadowed objects represent the elements of the corrector pattern encapsulated in a dashed box that denotes an instance of the corrector pattern. The distinguished element of the corrector pattern is depicted by the dark shading. A combination of sequential and parallel correctors may also be used for the correction of a predicate. Due to space constraints, we omit the presentation of such combinations and the parallel corrector (see [27] for details).

In Figure 8, each corrector participant c_i is associated with a class $Class_i$ in which the predicate X_i ($1 \leq i \leq n$) should be corrected. (Note that $Class_i$ may be associated with multiple corrector elements, each belonging to a different instance of the corrector pattern.) The distinguished element is associated with the participant c_n , which establishes the correction of X_n and X . Figure 9 illustrates the application of an instance of the sequential corrector to the class diagram of the ACC system.

⁵ In the design and implementation phases, the corrector elements may be realized as independent software/hardware components that execute concurrently with other components of an embedded system. We conjecture that any additional execution overhead on system performance incurred by adding corrector elements would not be worse than the use of conventional redundancy mechanisms.

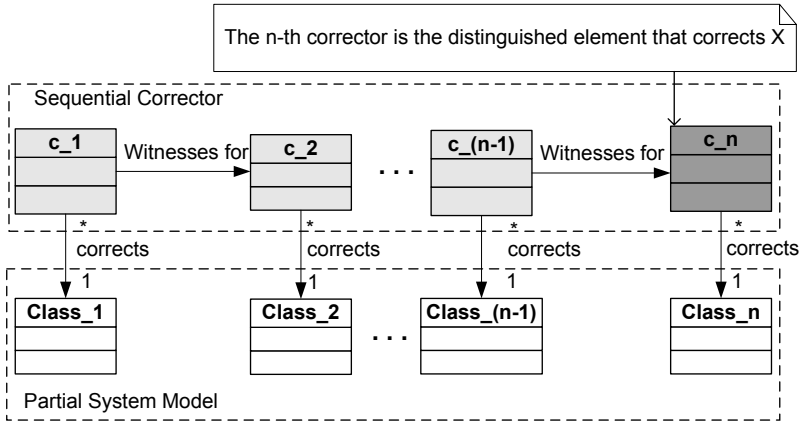


Fig. 8. The structure of the sequential corrector

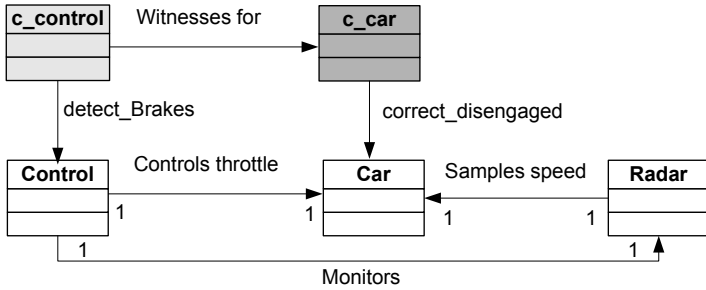


Fig. 9. Composition of a sequential corrector pattern with the ACC system

Witness Predicate. Since we decompose the global correction predicate X into a set of local correction predicates X_1, \dots, X_n , we should specify what implies the truth value of X . Towards this end, we introduce the notion of a *witness* predicate Z that is a local condition belonging to the distinguished element of the corrector pattern. The truth value of the witness predicate is an indication that X has been corrected. We also consider a witness predicate Z_i for each element c_i to represent that c_i corrects X_i . We say c_i *witnesses* iff Z_i is *true*. In the case of the sequential corrector, the distinguished element c_{index} (i.e., c_n) sets the value of Z_{index} (i.e., Z_n) to *true* if c_1, \dots, c_{n-1} witness their correction predicates and X_n holds.

Invariant. The invariant of the correction pattern is a state predicate \mathcal{I}_C such that $\mathcal{I}_C = \{s : (\forall i : 1 < i \leq n : (Z_i(s) \Rightarrow (\forall j : 1 \leq j < i : Z_j(s))))\}$. Intuitively, it means that, in an invariant state s , if a corrector element c_i witnesses, then all its predecessors should also witness.

Behavior. The state diagram of each corrector element c_i in Figure 8 is concurrently composed with the state diagram of its associated class $Class_i$ in order

to create a composite concurrent state transition diagram. Figure 10 depicts a possible scenario for correcting a predicate $X \equiv (X_1 \wedge X_2 \wedge \dots \wedge X_n)$ in a sequential fashion. The distinguished element can witness if all its predecessors c_1, \dots, c_{n-1} have already witnessed their correction predicates. In other words, if Z holds then $Z_1 \wedge \dots \wedge Z_n$ must hold as well. Notice that, for nonmasking fault-tolerance, we do not explicitly impose any order on the recovery of corrector elements as long as recovery to the invariant is guaranteed. Nonetheless, depending on the problem at hand, satisfying the above requirement may require us to impose a specific recovery order. For example, in a token ring protocol, the direction of token circulation should be consistent with the order of recovering elements.

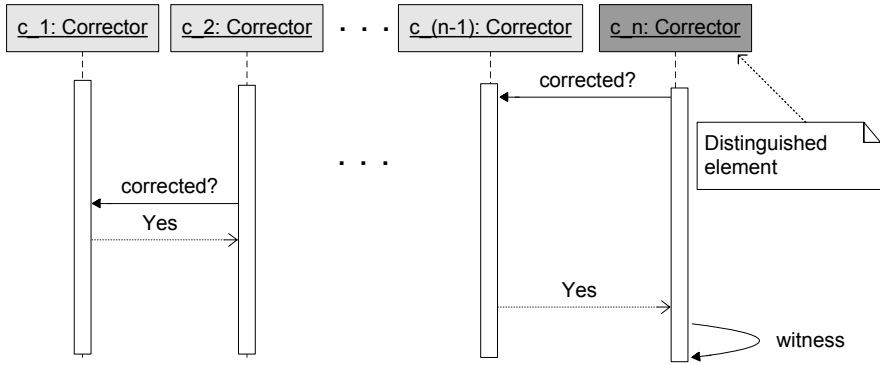


Fig. 10. The behavior of a sequential corrector

ACC Example: The instance of the corrector pattern applied to the ACC system comprises two elements $c_{control}$ and c_{car} modeled as two new objects in the UML model of the ACC system (see Figures 9 and 11). The element $c_{control}$ behaves as a detector that only monitors the state of the Control.Brakes signal (i.e., detects $X_{control}$) and the element c_{car} should correct X_{car} when it is false; i.e., when $Car.disengaged$ is false, it should be set to true. The element $c_{control}$ sets its witness predicate $Z_{control}$ to true when $X_{control}$ holds; i.e., when brakes are applied. The element c_{car} sets its witness predicate Z_{car} to true when $X_{control}$ and X_{car} hold. The invariant of the corrector pattern is equal to $Z_{car} \Rightarrow Z_{control}$. More specifically, c_{car} continuously checks with $c_{control}$ to see whether brakes are applied or not. If $c_{control}$ witnesses, then c_{car} corrects its correction predicate (i.e., $X_{car} \equiv Car.disengaged$) if necessary. Such a correction is established by a local corrective action that sets the state variable $Car.disengaged$ and the witness predicate Z_{car} to true.

Correction Requirements. In order to ensure the recovery of the composition of an instance of the corrector pattern with the UML model of an FIS, the behavior of the corrector pattern and its participants should meet the following requirements (from [42]): (1) *Safeness*. It is never the case that the witness

predicate Z is *true* when the correction predicate X is *false*; i.e., the corrector pattern never lies. (2) *Progress*. It is always the case that if X becomes true then Z will eventually hold. (3) *Stability*. It is always the case that once Z becomes *true*, it will remain *true* as long as the predicate X is *true* (i.e., Z remains *stable*). (4) *Convergence*. The correction predicate X will eventually hold and will continuously remain true. Each participant c_i should also meet the above requirements for Z_i and X_i . The first three requirements (safeness, stability, and progress) specify the requirements for the detection of the predicate X while the convergence states that X will eventually hold. A pattern that only meets the safeness, progress, and stability requirements guarantees to detect the correction predicate X if it ever holds, but does not guarantee to establish X if it is falsified. In special instances of the corrector pattern, we may have some participants c_i that perform only as a detector.

The correction requirements can be specified in Linear Temporal Logic (LTL) [43] using (i) the universal operator \Box , where $\Box Y$ means that the state predicate Y always holds; (ii) the next state operator \bigcirc , where $\bigcirc Y$ means that in the next state Y holds, and (iii) the eventuality operator \Diamond , where $\Diamond Y$ means that the state predicate Y eventually holds. We respectively specify *safeness* and *stability* as $\Box(Z \Rightarrow X)$ and $\Box(Z \Rightarrow (\bigcirc(Z \vee \neg X)))$. We specify *progress* as the following LTL expression: $\Box(X \Rightarrow \Diamond Z)$, and the LTL formula $\Diamond(\Box X)$ specifies the convergence requirement.⁶

Nonmasking fault-tolerance of the corrector pattern. Since the instances of the corrector pattern are also subject to faults, we must ensure that the corrector pattern is itself nonmasking fault-tolerant to *the effect of faults*. To guarantee the nonmasking fault-tolerance of the corrector pattern, it has to eventually recover to its invariant \mathcal{I}_C . For example, if faults occur after some corrector element c_i ($i > 1$) witnesses, then the witness predicate of some c_j , for $1 \leq j < i$, may be falsified due to the effect of faults. As a result, the invariant $Z_i \Rightarrow (\forall j : 1 \leq j < i : Z_j)$ will no longer hold. However, since X_j holds (notice that Z_j was set to true because X_j had become true at some point), after faults stop occurring, the progress property of the element c_j guarantees that Z_j will again become true, thereby resulting in the recovery of the entire corrector pattern to its invariant \mathcal{I}_C . In another scenario, the effect of faults may cause Z_i to become true while none of its predecessors has witnessed, thereby violating the invariant predicate \mathcal{I}_C . (In this case, faults directly violate the safety of the corrector pattern, which is not a concern since recovery to \mathcal{I}_C is the only requirement.) Since the convergence requirement guarantees that all predecessors of c_i will eventually witness, the invariant \mathcal{I}_C will eventually be established. Therefore, a design of the corrector pattern that recovers to its invariant after faults stop occurring is itself nonmasking fault-tolerant to the effect of faults.

⁶ The correction requirements can also be specified using Dwyer *et al.* [44] specification patterns. For example, the safeness and stability can be represented in terms of the *Universality* specification pattern defined by Dwyer *et al.* [44].

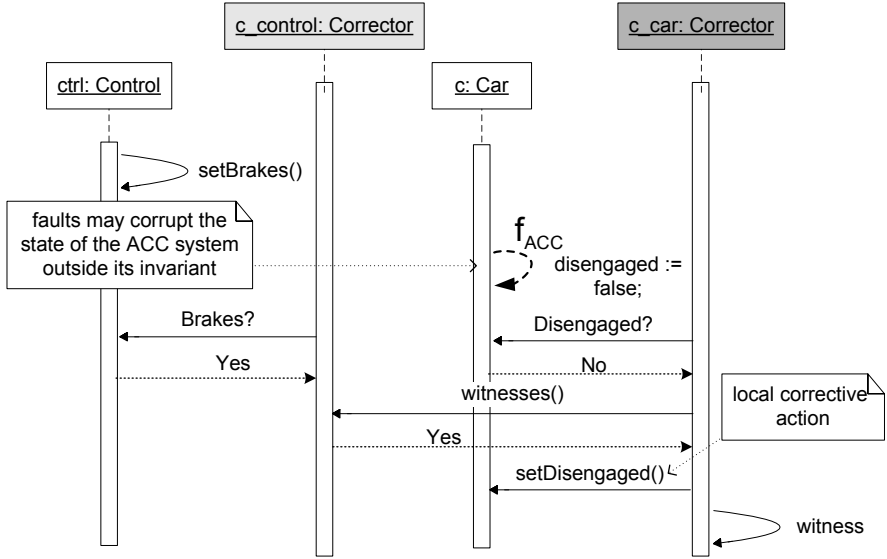


Fig. 11. The behavior of the sequential corrector applied to the ACC system

ACC Example: The effect of f_{ACC} faults on the corrector pattern applied to the ACC system is that faults may corrupt the value of the witness predicates $Z_{control}$ and Z_{car} to false. The progress of the corrector element $c_{control}$ guarantees that the corrector pattern will recover to its invariant $Z_{car} \Rightarrow Z_{control}$ if Z_{car} holds and $Z_{control}$ has been falsified by faults.

Remark. In this section, we only considered the application of an instance of the corrector pattern for the ACC system. While the ACC example is small, the number of the elements of an instance of the corrector pattern cannot go beyond the number of system components. Moreover, for nonmasking fault-tolerance, it is often the case that only one instance of the corrector pattern should be instantiated to model the correction of the violations of system invariant, which does not hinder the scalability of our approach. Moreover, even though composing a corrector pattern with the functional model of an FIS system may add a layer of complexity, the modularity provided by the corrector pattern facilitates the management of such complexity. (Other pattern-driven methods may also suffer from this additional layer of complexity introduced by pattern instantiation.)

6 Generating Promela Code and Automated Analysis

In order to enable rigorous analysis of modeling artifacts in model-driven development of fault-tolerant systems, we generate formal specifications of UML models and use model checkers for detecting the inconsistencies between fault-tolerance and functional requirements. Towards this end, we extend the Hydra

UML formalization framework [24] to generate the formal specifications of the UML models of faults and FTSs in the Promela modeling language [25]. Hydra [24] is a generic framework for generating formal specifications from UML diagrams. Promela is a language for modeling concurrent and distributed programs in the model checker SPIN [25]. The syntax of Promela is based on the C programming language. A Promela model comprises (1) a set of variables, (2) a set of (concurrent) processes modeled by a predefined type, called *proctype*, and (3) a set of asynchronous and synchronous channels for inter-process communications.

Hydra uses a set of mapping rules (see Figure 12 for a high-level summary) to translate the entities in a UML metamodel to the entities in a Promela metamodel. For example, Hydra translates each UML object to a *proctype* in Promela. Thus, each element c_i of the corrector pattern is formalized as a separate process that is concurrently executed with the processes that represent UML functional objects. The transitions of the state diagram of each object are formalized either as message-passing actions or as regular assignment actions of the corresponding process in Promela. The inter-object associations are formalized as message exchange channels in Promela.

<i>UML Metamodel Entity</i>		<i>Promela Metamodel Entity</i>
Object	→	proctype
Instance variable	→	Variable
Association	→	Channel
Generalization	→	Duplicated proctype
State	→	State block
Composite State	→	proctype
Concurrent Composite State	→	Concurrent proctypes
Transition	→	Messages/Assignments

Fig. 12. An excerpted set of formalization rules in Hydra [24]

Extending Hydra for fault formalization. In UML state diagrams, we distinguish fault transitions from regular transitions by defining a **Fault** stereotype [29]. The extended Hydra treats the transitions of a fault-type f differently than other transitions in that it integrates the transitions of f (modeled in different state diagrams) in a separate process **Fault_f** in Promela that is concurrently executed with all other processes. The actions of **Fault_f** are all non-deterministic atomic actions that can be either variable assignment actions or send/receive operations on communication channels (already defined in the Promela model of the system). For example, a message loss fault is modeled as an action that removes a message (or the acknowledgement of a message) from a channel. As another example, a fault action may non-deterministically change the value of a Boolean variable from true to false or vice versa. Such a formalization is advantageous in that the resulting Promela model modularizes fault transitions and separates them from the functional part of the Promela specifications so that the effect of faults on system behaviors can easily be simulated and analyzed.

Analysis. We use the SPIN model checker to simulate and verify the Promela specifications generated by Hydra. Moreover, we visualize the results of checking Promela models in UML state/sequence diagrams. For example, while verifying the UML model of an FTS against the correction requirements, we may find counterexamples that represent the inconsistencies of the corrector pattern and the functional objects. To analyze such inconsistencies, we use SPIN to generate the counterexamples and use Theseus [26] to visualize each step of the SPIN counterexample in UML state/sequence diagrams. Such a visualization of counterexamples facilitates the analysis and refinement of UML models.

ACC Example: In the formalization of the UML model of the ACC system, five proctypes are generated corresponding to the Control, Car and Radar objects and the corrector elements $c_{control}$ and c_{car} . Fault formalization results in the generation of a Fault_ACC proctype that, when executed, may non-deterministically set the values of Car.disengaged, $Z_{control}$ and Z_{car} to false. To verify the nonmasking fault-tolerance of the candidate model (i.e., the composition of the UML model and the corrector pattern), we first verify the invariant \mathcal{I}_{ACC} as an assertion, without including the Fault_ACC proctype in the generated Promela model (i.e., model in the absence of faults). The corresponding LTL property is specified as $\Box(\mathcal{I}_{ACC})$, which was verified in the absence of faults. We also verify that, in the absence of faults, the candidate model does not deadlock. This ensures that the corrector pattern does not interfere with the functional model in the absence of faults. Afterwards, we verified the progress (denoted $\Box(X_{ACC} \Rightarrow \Diamond Z_{car})$) and the convergence (denoted $\Diamond(\Box X_{ACC})$) of the corrector pattern while including the Fault_ACC proctype in the Promela model in order to ensure that the corrector pattern is itself nonmasking fault-tolerant. The reachability of the invariant \mathcal{I}_{ACC} is also ensured by the convergence of the corrector pattern; i.e., the candidate model eventually recovers to its invariant. In the verification of the correction requirements, the Theseus [26] visualization tool highlighted a set of safety-violating transitions in the state diagram of the Control object that would reach a state where Control.Brakes was false, but $Z_{control}$ had remained true. This was a counterexample illustrating how the safeness of $c_{control}$ would be violated. Since $c_{control}$ is instantiated as a detector, we had to modify the model so that this inconsistency is resolved. Towards this end, we added some actions that would atomically set $Z_{control}$ to false if the brakes were no longer applied. Notice that, in this case, resolving the inconsistencies of the corrector pattern and the functional model required a change in the behavior of the functional model. Such modifications illustrate how the addition of fault-tolerance concerns may require some changes in functional requirements.

7 Related Work

In this section, we discuss related work for modeling and analyzing error recovery. Several approaches [45, 46] exist for modeling and analyzing dependability aspects, most of which focus on system availability and reliability without

providing a reusable artifact for specifying error recovery. For example, Lopez-Benitez [45] presents a technique based on stochastic Petri nets for modeling and analyzing local and global system availability in the presence of node and communication link failures. Huszerl and Majzik [47] generate stochastic Petri nets from UML state charts in order to provide quantitative measures for comparing different redundancy management strategies against crash failures. Bondavalli *et al.* [46] present an approach for dependability analysis in both structural and behavioral UML models based on an intermediate Petri net model generated from UML diagrams. While they also model faults as timed transitions in Petri net models and generate a tool-independent intermediate Petri net model from UML diagrams, their approach for modeling fault-tolerance is based on exception handling and replication, whereas the corrector pattern provides an abstract reusable modeling artifact, which can be refined to a fault-tolerance design mechanism (e.g., exception handling).

In error recovery based on exception handling [48, 49, 50, 51], the focus is on the design of systems that tolerate exceptional conditions by systematic exception resolution. For example, Xu *et al.* [48] formally model exception handling in distributed systems and use coordinated atomic actions [52] to provide a distributed mechanism for exception resolution. Garcia and Beder and Rubira [50, 51] separate the concern of exception handling from functional concerns by introducing a meta-level architecture that captures the logic of exception handling in concurrent and distributed systems. While their approach provides a set of patterns for designing different tasks involved in exception handling, no measures are provided for ensuring the fault-tolerance of exception handlers and for verifying the interaction between error recovery and functional concerns in concurrent systems.

In addition to the above approaches, several formal models for error recovery exist in the literature [4, 42, 53, 37, 54] that provide a foundation for automated analysis of error recovery. Arora and Gouda [4] introduce the notion of convergence that presents a generic point of view of recovery in the presence of different types of faults. Based on Arora and Gouda's work [4], Arora and Kulkarni [42] show that a wide range of legacy fault-tolerance mechanisms can be captured by two basic fault-tolerance components, namely detectors and correctors, upon which we have developed two fault-tolerance analysis patterns [27]. Belli and Grosspietsch [53] provide a hybrid formal framework for modeling and specifying fault-tolerance against erroneous inputs and design flaws, where they use Petri nets for hierarchical specification of concurrent systems and regular expressions for specifying low-level system actions. Magee and Maibaum [54] use modal action logic to specify and verify fault-tolerance in component-based systems, where they adopt a state-based model in partitioning the system state space to the set of normal and abnormal states. Aforementioned approaches provide formal frameworks for specifying and analyzing fault-tolerance concerns, whereas the corrector pattern provides a semi-formal means for capturing and specifying fault-tolerance concerns in earlier stages of the system development lifecycle.

In summary, the corrector pattern provides a design-independent abstraction for capturing the requirements of error recovery before any design decision is made. Such an abstraction simplifies the task of modeling as the focus is on identifying constraints (i.e., correction predicates) that should be satisfied by a fault-tolerant system independent of what design mechanism is used to realize recovery. Moreover, the use of the corrector pattern enables modular specification and analysis of recovery requirements, which in turn simplifies the traceability of recovery from requirements analysis to design and implementation phases. In addition to providing a means for early modeling of recovery, we are investigating the application of techniques for the addition of fault-tolerance [37] in automatic specification and instantiation of the corrector pattern in UML state diagrams.

8 Conclusions and Future Work

In this paper, we introduced an object analysis pattern, called the *corrector* pattern, for modeling and analyzing nonmasking fault-tolerance, where a nonmasking fault-tolerant program guarantees to recover from error conditions to a set of legitimate states (called invariant). Instances of the corrector pattern are added to the UML model of a system to create the UML model of its fault-tolerant version. The corrector pattern also provides a set of constraints for verifying the consistency of functional and fault-tolerance requirements and the fault-tolerance of the corrector pattern itself. We extended McUmbler and Cheng's UML formalization framework [24] to generate formal specifications of the UML model of fault-tolerant systems in Promela [25]. Subsequently, we used the SPIN model checker [25] to detect the inconsistencies between fault-tolerance and functional requirements. To facilitate the automated analysis of nonmasking fault-tolerance, we employed the Theseus visualization tool [26] that animates counterexample traces and generates corresponding sequence diagrams in terms of the UML model elements. Even though in this paper we presented only the corrector pattern for specifying nonmasking fault-tolerance, we have also developed a companion *detector* pattern [27] for modeling failsafe fault-tolerance, where a failsafe fault-tolerant system guarantees safety even when faults occur. The use of the detector and corrector patterns simplifies and modularizes fault-tolerance concerns and helps to separate the analysis of functional and fault-tolerance concerns, while providing a means to analyze their mutual impact. As an extension of this work, we are investigating the application of a synthesis tool that we have previously developed (called Fault-Tolerance Synthesizer [38]) in automating the identification of the fault-span, scenarios with faults, and the instantiation of the corrector pattern.

Acknowledgements

The authors greatly appreciate the feedback from Heather Goldsby and comments from the anonymous reviewers for ADS.

This work was partially sponsored by NSF grants EIA-0000433, EIA-0130724, CDA-9700732, CCR-9901017, CNS-0551622, CCF-0541131, CAREER CCR-0092724, ONR grant N00014-011-0744, DARPA Grant OSURS01-C-1901, Air Force Research Lab under subcontract MICH 06-S001-07-C1, Siemens Corporate Research, a grant from the Michigan State University's Quality Fund, and a grant from Michigan Technological University.

References

1. Campbell, R.H., Randell, B.: Error recovery in asynchronous systems. *IEEE Transactions on Software Engineering* SE-12(8) (1986)
2. Douglass, B.P.: *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks and Patterns*. Addison-Wesley, Reading (1999)
3. Gomaa, H.: *Designing Concurrent, Distributed, and Real-Time Application with UML*. Addison-Wesley, Reading (2000)
4. Arora, A., Gouda, M.G.: Closure and convergence: A foundation of fault-tolerant computing. *IEEE Transactions on Software Engineering* 19(11), 1015–1027 (1993)
5. Demirbas, M., Arora, A.: Convergence refinement. In: *International Conference on Distributed Computing Systems*, pp. 589–597 (2002)
6. Randall, B.: System structure for software fault-tolerance. *IEEE Transactions on Software Engineering*, 220–232 (1975)
7. Cristian, F.: Exception handling and software fault-tolerance. *IEEE Transactions on Computers*, C-31(6) (1982)
8. Schneider, F.B.: Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys* 22(4), 299–319 (1990)
9. Elnozahy, E.N., Alvisi, L., Wang, Y.-M., Johnson, D.B.: A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys* 34(3), 375–408 (2002)
10. Saridakis, T.: A system of patterns for fault-tolerance. In: *The 7th European Conference on Pattern Languages of Programs (EuroPLoP)*, pp. 535–582 (2002)
11. UML profile for modeling quality of service and fault tolerance characteristics and mechanisms (2002), <http://www.omg.org/docs/ptc/04-06-01.pdf>
12. France, R., Georg, G.: An aspect-based approach to modeling fault-tolerance concerns. Technical Report 02-102, Computer Science Department, Colorado State University (2002)
13. Tichy, M., Schilling, D., Giese, H.: Design of self-managing dependable systems with uml and fault tolerance patterns. In: *Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems (WOSS)*, Newport Beach, CA, pp. 105–109 (2004)
14. Tkatchenko, M., Kiczales, G.: Uniform support for modeling crosscutting structure. Appeared in AOM Workshop held in conjunction with AOSD (2005)
15. Arora, A.: A foundation of fault-tolerant computing. PhD thesis, The University of Texas at Austin (1992)
16. Ilic, D., Troubitsyna, E.: Modeling fault tolerance of transient faults. In: *Proceedings of Rigorous Engineering of Fault-Tolerant Systems*, pp. 84–92 (2005)
17. Laibinis, L., Troubitsyna, E.: Fault tolerance in use case modeling. In: *the Workshop on Requirements for High Assurance Systems* (2005)
18. Rubira, C.M.F., de Lemos, R., Ferreira, G.R.M., Castor Filho, F.: Exception handling in the development of dependable component-based systems. *Software Practice and Experience* 35, 195–236 (2005)

19. Shui, A., Mustafiz, S., Kienzle, J., Dony, C.: Exceptional use cases. In: Briand, L.C., Williams, C. (eds.) *MoDELS 2005*. LNCS, vol. 3713, pp. 568–583. Springer, Heidelberg (2005)
20. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. *Communications of the ACM* 17(11) (1974)
21. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, Reading (1995)
22. Fowler, M.: *Analysis Patterns: Reusable Object Models*. Addison-Wesley, Reading (1997)
23. Konrad, S., Cheng, B.H.C., Campbell, L.A.: Object analysis patterns for embedded systems. *IEEE Transactions on Software Engineering* 30(12), 970–992 (2004)
24. McUmbler, W.E., Cheng, B.H.C.: A general framework for formalizing UML with formal languages. In: the proceedings of 23rd International Conference of Software Engineering, pp. 433–442 (2001)
25. Holzmann, G.: The model checker SPIN. *IEEE Transactions on Software Engineering* 23(5), 279–295 (1997)
26. Goldsby, H., Cheng, B.H.C., Konrad, S., Kamdoun, S.: A visualization framework for the modeling and formal analysis of high assurance systems. In: *Proceedings of the ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS)*, Genova, Italy, pp. 707–721 (2006)
27. Ebnenasir, A., Cheng, B.H.C.: A framework for modeling and analyzing fault-tolerance. Technical Report MSU-CSE-06-5, Computer Science and Engineering, Michigan State University, East Lansing, Michigan (January 2006)
28. Ebnenasir, A., Kulkarni, S.S.: Hierarchical presynthesized components for automatic addition of fault-tolerance: A case study. In: the extended abstracts of the ACM workshop on the Specification and Verification of Component-Based Systems (SAVCBS), Newport Beach, California (2004)
29. Booch, G., Rumbaugh, J., Jacobson, I.: *The Unified Modeling Language User Guide*. Addison-Wesley, Reading (1999)
30. Kulkarni, S.S., Arora, A.: Automating the addition of fault-tolerance. In: *Proceedings of the 6th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pp. 82–93 (2000)
31. Fischer, M.J., Lynch, N.A., Peterson, M.S.: Impossibility of distributed consensus with one faulty processor. *Journal of the ACM* 32(2), 373–382 (1985)
32. Kulkarni, S.S., Ebnenasir, A.: Enhancing the fault-tolerance of nonmasking programs. In: *Proceedings of the 23rd IEEE International Conference on Distributed Computing Systems (ICDCS)*, pp. 441–449 (2003)
33. Gries, D.: *The Science of Programming*. Springer, Heidelberg (1981)
34. Tiwari, A., Rueß, H., Saïdi, H., Shankar, N.: A technique for invariant generation. In: Margaria, T., Yi, W. (eds.) *ETAPS 2001 and TACAS 2001*. LNCS, vol. 2031, pp. 113–127. Springer, Heidelberg (2001)
35. Varghese, G.: Self-stabilization by local checking and correction. PhD thesis, MIT/LCS/TR-583 (1993)
36. Avizienis, A., Laprie, J.C., Randell, B., Landwehr, C.: Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing* 1(1) (2004)
37. Ebnenasir, A.: *Automatic Synthesis of Fault-Tolerance*. PhD thesis, Michigan State University (2005)
38. Ebnenasir, A., Kulkarni, S.S.: FTSyn: A framework for automatic synthesis of fault-tolerance, <http://www.cs.mtu.edu/~aebnenas/research/tools/ftsyn.htm>

39. Kulkarni, S.S., Ebnenasir, A.: Complexity issues in automated synthesis of failsafe fault-tolerance. *IEEE Transactions on Dependable and Secure Computing* 2(3), 201–215 (2005) (to appear)
40. Mittal, N., Garg, V.K.: On detecting global predicates in distributed computations. In: *Proceedings of the 21st IEEE International Conference on Distributed Computing Systems (ICDCS)*, Phoenix, Arizona, USA, pp. 3–10, (April 2001)
41. Garg, V.K., Waldecker, B.: Detection of strong unstable predicates in distributed programs. *IEEE Transactions on Parallel and Distributed Systems* 7(12), 1323–1333 (1996)
42. Arora, A., Kulkarni, S.S.: Detectors and Correctors: A theory of fault-tolerance components. In: *IEEE International Conference on Distributed Computing Systems (ICDCS)*, pp. 436–443 (May 1998)
43. Emerson, E.A.: *Handbook of Theoretical Computer Science: Chapter 16, Temporal and Modal Logic*. Elsevier Science Publishers B.V., Amsterdam (1990)
44. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: *Proceedings of the 21st International Conference on Software Engineering (ICSE99)*, Los Angeles, CA, USA, pp. 411–420 (1999)
45. Lopez-Benitez, N.: Dependability modeling and analysis of distributed programs. *IEEE Transactions on Software Engineering* 20(5), 345–352 (1994)
46. Bondavalli, A., et al.: Dependability analysis in the early phases of UML-based system design. *International Journal of Computer Systems Science and Engineering* 5, 265–275 (2001)
47. Huszerl, G., Majzik, I.: Modeling and analysis of redundancy management in distributed object-oriented systems by using uml statecharts. In: *27th Euromicro Conference*, pp. 200–207 (2001)
48. Xu, J., Romanovsky, A., Randell, B.: Coordinated exception handling in distributed object systems: From model to system implementation. In: *Proceedings of the 18th IEEE International Conference on Distributed Computing Systems*, pp. 12–21. *IEEE Computer Society Press*, Los Alamitos (1998)
49. Beder, D.M., Randall, B., Romanovsky, A., Snow, C.R., Stroud, R.J.: An application of fault-tolerance patterns and coordinated atomic actions to a problem in railway scheduling. *ACM SIGOPS Operating System Review* 34(4) (2000)
50. Garcia, A.F., Beder, D.M., Rubira, C.M.F.: A unified meta-level software architecture for sequential and concurrent exception handling. *The Computer Journal*, *British Computer Society* 44(6), 569–587 (2001)
51. Beder, D., Rubira, C.: A meta-level software architecture based on patterns for developing dependable collaboration-based designs. In: *Proceedings of the second Brazilian workshop on fault-tolerance* (2000)
52. Xu, J., Randell, B., Romanovsky, A.B., Rubira, C.M.F., Stroud, R.J., Wu, Z.: Fault tolerance in concurrent object-oriented software through coordinated error recovery. In: *FTCS*, pp. 499–508 (1995)
53. Belli, F., Grosspietsch, K.E.: Specification of fault-tolerant system issues by predicate/transition nets and regular expressions-approach and case study. *IEEE Transactions on Software Engineering* 17(6), 513–526 (1991)
54. Magee, J., Maibaum, T.: Towards specification, modelling and analysis of fault tolerance in self managed systems. In: *Proceedings of the 2006 international workshop on Self-adaptation and self-managing systems*, pp. 30–36 (2006)

Architectural Fault Tolerance Using Exception Handling

Rogério de Lemos

Computing Laboratory
University of Kent, UK
`r.delemos@kent.ac.uk`

Abstract. When building dependable systems by integrating untrusted software components that were not originally designed to interact with each other, it is inevitable the occurrence of architectural mismatches related to assumptions in the failure behaviours. These mismatches if not prevented during system design have to be tolerated during run-time. This paper presents an architectural abstraction based on exception handling for structuring fault-tolerant software systems. Exception handling has been used effectively for incorporating fault tolerance into software systems. The proposed architectural abstraction transforms untrusted software components into idealised fault tolerant architectural elements (iFTE), which clearly separate the normal and exceptional behaviours, in terms of their internal structure and interfaces. An advantage of this architectural abstraction is that it can be instantiated into both components and connectors. Moreover, the proposed abstraction clearly facilitates system structuring, and the analysis of exception propagation, which can make the overall system quite complex if exceptions, and their respective handlers, and not properly incorporated into system design. The feasibility of the proposed approach is evaluated in terms of a simple case study.

1 Introduction

Fault tolerance aims at delivering correct service despite the presence of faults [2]. A fault tolerant system should be well-structured to ensure that the extra software does not add to the complexity of the system, and improves the overall system dependability [13]. The architecture of a software system is an abstraction of its actual structure. The identification of the system structure early in its development process allows abstracting away from system details, thus assisting the understanding of broader system concerns [18].

The architecture of a software system can be seen as a set of connected components, their externally visible properties and their relationships [6]. Consequently, software architectures are usually described in terms of its components – which represent computation units, connectors – which encapsulate the interaction between components, and their configuration – which characterizes the topology of the system in terms of the interconnection of components via connectors [12] [18].

Fault tolerance aims to avoid system failure via error detection and system recovery at run-time [2]. At the architectural level, error detection relies on

monitoring mechanisms, or probes, for detecting erroneous states at the interfaces of architectural elements or in the interactions between these elements. On the other hand, system recovery aims, first, to eliminate erroneous states from the system – known as error handling, and second, to reconfigure the system architecture for isolating those architectural elements that might have caused the erroneous states – known as fault handling. Architectural abstractions offer a number of features that are suitable for the provision of fault tolerance and error confinement [9]. Architectures also provide a global system perspective that enables high-level interpretation of system faults, thus facilitating their identification. The separation between computation and communication/coordination, which enforces modularisation and information hiding, facilitates error detection, confinement, and system recovery. The architectural configuration, which imposes structural constraints, helps to identify anomalies in the system structure. Explicit system structuring facilitates the introduction of mechanisms such as program assertions, pre- and post-conditions, and invariants that enable the detection of potential erroneous architectural states. Architectural changes for supporting fault handling during system recovery can include the addition, removal, or replacement of components and connectors, modifications to the configuration or parameters of components and connectors, and alterations in the component/connector network's topology.

This paper presents an architectural abstraction for structuring fault tolerant software systems based on exception handling. This abstraction known as the idealised fault tolerant architectural element (iFTE) provides the means for incorporating error detection and error handling into software architectures. This abstraction was previously introduced [7], and in this paper, in addition of defining an improved version of the iFTE, we also provide more details of the architectural abstraction, such as, the definition of the iFTE in terms of an architectural description language, the description on how exceptions are propagated among architectural elements, and the definition of a formal model for validating the internal and external behaviours of the iFTE against pre-identified normal and exceptional behavioural scenarios. The rest of the paper is organized as follows. In Section 2, we describe an architectural abstraction in terms of its interfaces and its internal structure, and how it enables the propagation of exceptions at the architectural level. Section 3 presents an embedded system case study, which demonstrates the feasibility of the proposed approach. Related work is presented in Section 4. Finally, the last section presents some concluding remarks and directions for future research.

2 Idealised Fault Tolerant Architectural Element (iFTE)

Exception handling has shown to be an effective mechanism for incorporating fault tolerance into software systems [5]. It allows to structure systems in such a way that exceptional and the normal behaviour can be kept separate. The idealised fault-tolerant component is a structuring concept that makes it easy to identify what parts of a system have what responsibilities for trying to cope with which sorts of faults [1].

The incorporation of exception handling at the architectural level has been suggested as a valuable mechanism for error handling, if properly incorporated into the system structure [10]. It allows one to reason about the software fault tolerance

properties at a higher level of abstraction by properly assigning exceptional behaviour responsibilities among the architectural components and connectors of a software system.

The architectural abstraction being advocated for the structuring of fault-tolerant systems is the idealised fault-tolerant architectural element (iFTE) [7]. The iFTE enforces the principles associated with the concept of the idealised fault-tolerant component [1], and includes the following responsibilities:

1. detection of errors in the architectural elements or their interactions;
2. raising and handling of internal exceptions associated with the detected errors;
3. handling of exceptions that were raised externally by other architectural elements;
4. masking of internal or external exceptions by means of redundant resources;
5. propagation of exceptions, internal or externally raised, that cannot be masked.

Similar to the idealised fault-tolerant component, the architectural approach being introduced advocates the complete separation on how architectural elements should deal with their normal and abnormal behaviours. An advantage of the proposed architectural abstraction is that it can be instantiated into both idealised fault-tolerant architectural components and connectors, whose interactions are governed by exception communication rules:

1. idealised fault-tolerant architectural component - responsible for preventing internal errors from propagating to the rest of the system by handling them as internal exceptions, and constraining the error behaviours by signalling them as external exceptions;
2. idealised fault-tolerant architectural connector - responsible for resolving potential conflicts between exceptions signalled by the collaborating components, preventing the propagation of errors caused mismatches by handling them as internal exceptions, and constraining these errors by signalling them as external exceptions.

The idealised fault-tolerant architectural element (iFTE) is a specialisation of the peer-to-peer style [6]. In this architectural style any element can interact with other elements for providing services to them or requesting their services. In the proposed architectural abstraction communication between architectural elements is a request/reply interaction, connectors are first class architectural elements that coordinate the interactions between components, and provided and required exceptional interfaces are included for enforcing the behaviour of the idealised fault-tolerant component [1].

2.1 iFTE: Architectural Abstraction

The general model of an iFTE is shown in Figure 1. The iFTE has four types of external interfaces, and these are clearly partitioned into normal and abnormal (exceptional) behaviour:

1. `I_iFTE_PS` defines an access point for the (fault-tolerant) services provided by the iFTE (PS – provided services);
2. `I_iFTE_PE` defines an access point where iFTE signals its external exceptions (PE – provided exceptions);

3. **I_ifTE_RS** specifies services required by the iFTE for implementing its normal behaviour or handling exceptions (**RS** – required services);
4. **I_ifTE_RE** specifies the external exceptions that the iFTE is able to handle (**RE** – required exceptions).

These interfaces can be instantiated according to the different services that are provided and required by the iFTE. For example, a component that provides several services is expected to have an independent set of interfaces, in terms of services and exceptions, for each of the services it provides.

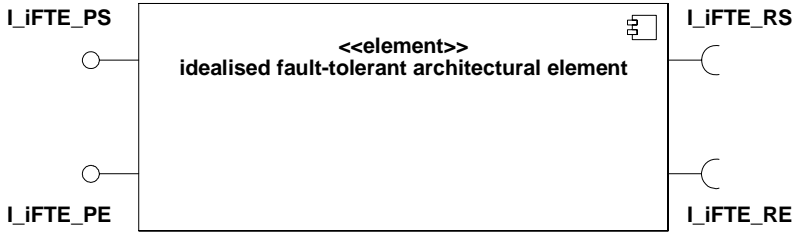


Fig. 1. The idealised fault-tolerant architectural element (iFTE)

The Architecture Analysis & Design Language (AADL) [16] is used for representing the iFTE architectural abstraction, as shown in Figure 2. The ports are defined under **features**, while the permitted connections between the different ports are defined under **flows**. Each interface is partitioned into two parts for representing the input and output ports of the interface. For example, **I_ifTE_PS_i** represents the input of a provided services interface of the iFTE, while **I_ifTE_PS_o** represents its output.

```

system ifte_abstraction
  features
    I_ifTE_PS_i: in event data port  Service;
    I_ifTE_PS_o: out event data port  Service;
    I_ifTE_PE_o: out event data port  Exception;
    I_ifTE_RS_i: in event data port  Service;
    I_ifTE_RS_o: out event data port  Service;
    I_ifTE_RE_i: in event data port  Exception;
  flows
    Ret_Ser_a: flow path I_ifTE_PS_i -> I_ifTE_PS_o;
    Sig_Exc_a: flow path I_ifTE_PS_i -> I_ifTE_PE_o;
    Req_Ser_b: flow path I_ifTE_PS_i -> I_ifTE_RS_o;
    Ret_Ser_b: flow path I_ifTE_RS_i -> I_ifTE_PS_o;
    Sig_Exc_b: flow path I_ifTE_RS_i -> I_ifTE_PE_o;
    Ret_Ser_c: flow path I_ifTE_RE_i -> I_ifTE_PS_o;
    Sig_Exc_c: flow path I_ifTE_RE_i -> I_ifTE_PE_o;
end ifte_abstraction;

```

Fig. 2. The AADL model of an iFTE

There are seven different relationships that can be established between the interfaces of an iFTE. After a services request is made through `I_iFTE_PS_i`, the iFTE may respond in three different ways: `Ret_Ser_a` - returns normal services through `I_iFTE_PS_o`, `Req_Ser_b` - it requests external services through `I_iFTE_RS_o`, or `Sig_Exc_a` - it signals either an interface or an internal exception through `I_iFTE_PE_o`. After a request for external services is made through `I_iFTE_RS_o`, four behaviours are possible. If the external architectural element returns a normal service through `I_iFTE_RS_i`: `Ret_Ser_b` - the iFTE returns a normal service through `I_iFTE_PS_o`, or `Sig_Exc_b` - the iFTE signals an exception through `I_iFTE_PE_o`. If the external architectural element signals an exception through `I_iFTE_RE_i`: `Ret_Ser_c` - the iFTE returns normal services through `I_iFTE_PS_o`, or `Sig_Exc_c` - the iFTE propagates an exception through `I_iFTE_PE_o` in case is not able to handle the external exception.

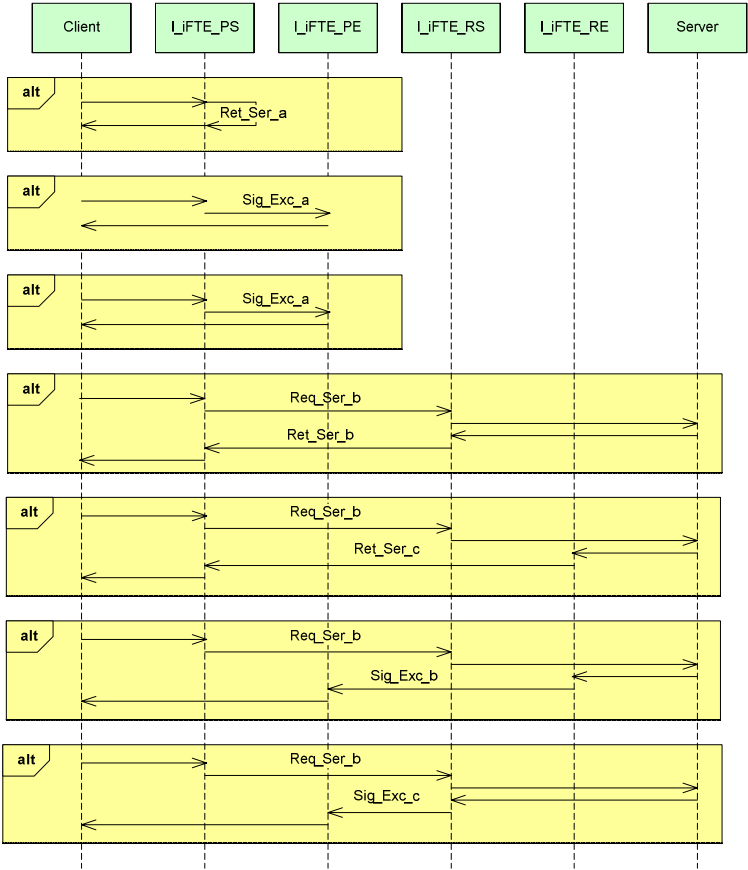


Fig. 3. The operational profile of an iFTE

From the above relationships between the interfaces of an abstract representation of an iFTE, we can identify seven different scenarios that represent the operational profile of an iFTE. These seven scenarios are described in Figure 3 in terms of an UML sequence diagram in which each scenario is alternative sequence of events.

2.2 iFTE: Detailed Design

The detailed design of an iFTE is shown in Figure 4, and it contains five architectural elements:

1. **Normal component** implements the normal behaviour of the iFTE, and it can be associated with an existing component or system;
2. **Abnormal component** handles the exceptions raised by the Normal component, and those propagated from the environment of the iFTE;
3. **Provided component** acts like a bridge between the provided services of the iFTE and its environment. It manages the provided interfaces of the iFTE by providing the required services, detecting interface exceptions from requests made to the iFTE, and signalling exceptions when the Abnormal component is not able to handle the exception;
4. **Required component** acts like a bridge between the required services of the iFTE and its environment. It manages the required interfaces by requesting services from other architectural elements, and detecting exceptional conditions raised by components with which the iFTE interacts;
5. **Coordinator connector** coordinates the interaction between the four internal components of an iFTE;

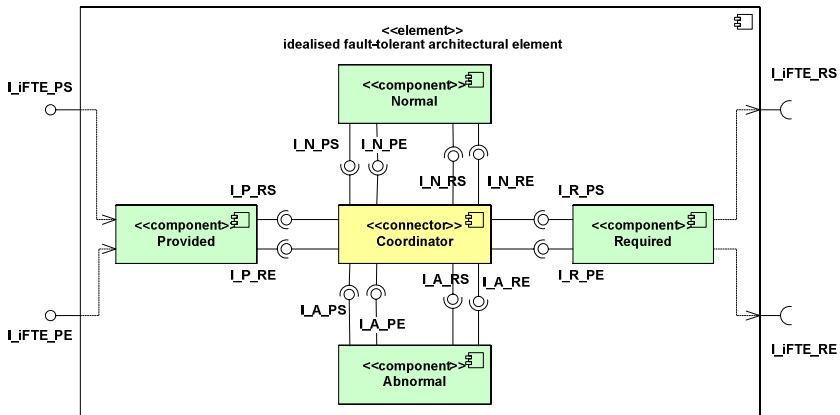


Fig. 4. General model of the idealised fault-tolerant architectural element (iFTE)

The internal architectural elements of the iFTE interact through internal interfaces, and these interfaces also enforce the separation between normal and exceptional behaviours. Between Provided (Required) and the Coordinator, there are two interfaces, one, representing the request (provision) of services, and the other the

propagation of exceptions. Between Normal and the Coordinator, there are two sets of interfaces representing essentially the access points for the provided (I_N_PS and I_N_PE) and required (I_N_RS and I_N_RE) services: through the first set of interfaces the Normal component provides its services and raises exceptions, and through the second type of interfaces the Normal receives requested services or external exceptions. The same type of interfaces exists between the Abnormal and Coordinator (I_A_PE). If an exception is raised by the Normal component, this exception is signalled through interface I_N_PE to the Coordinator, which propagates the exception to the Abnormal component through I_A_RE. If the Abnormal is able handle the exception, it informs the Provided component through the I_A_PS, otherwise it propagates an exception to Provided through I_A_PE. From the above description of the internal structure of an iFTE, it can be observed that the general notion of having an architectural element with four interfaces can be recursively applied to the architectural elements of the iFTE. Each of the four architectural elements listed above have got four types of interfaces for each type of service that an iFTE provides and requires.

In the following, the main architectural elements that make up an iFTE are presented in more detail.

2.2.1 Provided and Required Components

The Provided and Required components are very similar in their internal behaviours, except for the interface exception signalled by the Provided, so in the following we focus on the Provided component. These components in addition of managing the interfaces of the iFTE, they are also responsible for acting like integrators that remove architectural mismatches that might occur when integrating different architectural elements. For example, they would be responsible for changing the type of exception being propagated for avoiding potential mismatches between exceptions and their respective handlers, since the latter might be different depending on the context.

```

system Provided
features
  I_P_PS_i: in event data port  Service;
  I_P_PS_o: out event data port  Service;
  I_P_PE_o: out event data port  Exception;
  I_P_RS_i: in event data port  Service;
  I_P_RS_o: out event data port  Service;
  I_P_RE_i: in event data port  Exception;
flows
  Sig_Exc_a: flow path I_P_PS_i -> I_P_PE_o;
  Req_Ser_b: flow path I_P_PS_i -> I_P_RS_o;
  Ret_Ser_b: flow path I_P_RS_i -> I_P_PS_o;
  Sig_Exc_b: flow path I_P_RE_i -> I_P_PE_o;
end Provided;

```

Fig. 5. The AADL model of the Provided component

The **Provided** component is described in Figure 5 in terms of AADL. There are four different relationships that can be established between the interfaces of the **Provided** component. A request for services made through `I_P_PS_i`, can either signal an interface exception through `I_P_PE_o` (`Sig_Exc_a`), or be forwarded to the **Normal** component through `I_P_RS_o` (`Req_Ser_b`). The request service returns through interfaces `I_P_RS_i` and `I_P_PS_o` (`Ret_Ser_b`), or an exception has been propagated through interface `I_P_PE_o`, and signalled through interface `I_P_RE_i` and (`Sig_Exc_b`).

2.2.2 Normal Component

The behaviour of the **Normal** component is presented in Figure 6. There are six different relationships that can be established between the interfaces of the **Normal** component. When a service is requested by the **Coordinator** through `I_N_PS_i`, three possible internal behaviours are possible: `Ret_Ser_a` in which normal services are returned through `I_N_PS_o`, `Sig_Exc_a` in which an exception is signal through `I_N_PE_o`, and `Req_Ser_b` in which **Normal** has to request through interface `I_N_RS_o` an external service in order to be able to provide the services being required. If the external architectural element returns normal service through `I_N_RS_i`, then two behaviours are possible. Either **Normal** returns the normal service through `I_N_PS_o`, represented by `Ret_Ser_b`, or an internal exception might be raised, which causes the **Normal** to signal an exception through the interface `I_N_PE_o`, represented by `Sig_Exc_b`. Finally, in case the external architectural element signals an exception trough `I_N_RE_i`, this exception has to be propagated to the **Abnormal** component through the interface `I_N_PE_o`, which is represented by `Sig_Exc_c`.

```

system Normal
  features
    I_N_PS_i: in event data port  Service;
    I_N_PS_o: out event data port  Service;
    I_N_PE_o: out event data port  Exception;
    I_N_RS_i: in event data port  Service;
    I_N_RS_o: out event data port  Service;
    I_N_RE_i: in event data port  Exception;
  flows
    Ret_Ser_a: flow path I_N_PS_i -> I_N_PS_o;
    Sig_Exc_a: flow path I_N_PS_i -> I_N_PE_o;
    Req_Ser_b: flow path I_N_PS_i -> I_N_RS_o;
    Ret_Ser_b: flow path I_N_RS_i -> I_N_PS_o;
    Sig_Exc_b: flow path I_N_RS_i -> I_N_PE_o;
    Sig_Exc_c: flow path I_N_RE_i -> I_N_PE_o;
end Normal;

```

Fig. 6. The AADL model of the **Normal** component

2.2.3 Abnormal Component

The **Abnormal** component is represented in Figure 7. There are seven different relationships that can be established between the interfaces of the **Abnormal** component. Exceptions are signalled to the **Abnormal** component through the

interface `I_A_RE_i`. If an exception is received, it can either be handled and the success is notified through `I_A_RS_o` to the Normal component (`Ret_Ser_a`), or the exception is propagated to the Provided component through the `I_A_PE_o` interface (`Sig_Exc_a`). If Abnormal requires services either from the Normal component or from another external architectural element, then the request is made through the `I_A_RS_o` interface (`Req_Ser_b`). The return of this request can either be the requested service via `I_A_RS_i` (`Ret_Ser_b`), or an exception via `I_A_RE_i` (`Sig_Exc_b`). In case the Abnormal component requires further external services then a request is made through the `I_A_RS_i` interface (`Req_Ser_c`). The final relationship is related to the situation in which an exception is propagated via `I_A_RE_i` for the Abnormal to recover the iFTE to an error free state (`Sig_Exc_d`).

The final architectural element of the iFTE is the **Coordinator** connector, which is responsible for coordinating the exchange between the iFTE internal components. This connector encapsulates all the complexity associated with the iFTE, which includes the request and return of services between the different components, and the propagation of exceptions.

```

system Abnormal
features
  I_A_PS_i: in event data port  Service;
  I_A_PS_o: out event data port  Service;
  I_A_PE_o: out event data port  Exception;
  I_A_RS_i: in event data port  Service;
  I_A_RS_o: out event data port  Service;
  I_A_RE_i: in event data port  Exception;
flows
  Ret_Ser_a: flow path I_A_RE_i -> I_A_PS_o;
  Sig_Exc_a: flow path I_A_RE_i -> I_A_PE_o;
  Req_Ser_b: flow path I_A_RE_i -> I_A_RS_o;
  Ret_Ser_b: flow path I_A_RS_i -> I_A_PS_o;
  Sig_Exc_b: flow path I_A_RS_i -> I_A_PE_o;
  Req_Ser_c: flow path I_A_RS_i -> I_A_RS_o;
  Sig_Exc_d: flow sink I_A_RE_i;
end Abnormal;

```

Fig. 7. The AADL model of the Abnormal component

It is worth noting that in the context of some applications, in particular service oriented architectures the idealised fault-tolerant architectural element (iFTE) does not necessarily need to be represented as a self-contained element. In such architectural configurations, a single interface of the Normal component can be accessed by several other architectural elements, and the Normal component might have other interfaces through which it can be accessed. In those architectural configurations, that contain components and connectors that are not based on the idealised fault-tolerant architectural element (iFTE), additional mechanisms should be provided for dealing with their exceptional behaviour. In the presence of such non-fault-tolerant architectural elements, the separation of behaviours between normal and exceptional behaviours should nevertheless be enforced at the level of the

architectural configuration. This separation enhances the software understandability and maintainability, which also contributes positively to its dependability.

2.3 Exception Propagation

This section presents how exceptions are propagated among architectural abstractions (iFTEs), and the context in which they should be handled. In an architectural configuration, exceptions are propagated between components and connectors, or vice-versa, and they can be handled either in the context of components, or connectors.

This work follows previous work on the propagation of exceptions in object-oriented designs that were based on cooperative actions [8]. In this paper, objects are replaced by components and cooperations by connectors, but both components and connectors can be represented as iFTEs. This presentation will be based on the architectural configuration of Figure 8, which contains three components and a connector of the type iFTE. The connector (`conn_X`) defines the cooperation between the three components (`comp_A`, `comp_B`, `comp_C`). There is nothing in particular regarding this configuration, which is a generic configuration that allows illustrating in simple terms the exception propagation between architectural elements – the principles illustrated here can be applied to other architectural configurations involving iFTEs.

In order to maintain the architectural integrity, internal exceptions are propagated through interface of the architectural element either as declared or undeclared architectural exceptions [4]. For the idealised fault-tolerant architectural connector, this may require to translate the type of an external exception being propagated. The objective of this translation is to deal with potential mismatches that might exist between collaborating idealised fault-tolerant architectural components. With such approach, internal exceptions that are raised inside components and connectors are encapsulated by their architectural interfaces.

2.3.1 From Components to Connectors

If the normal part of a component raises an internal exception after a service has been requested this exception should be treated by the component's abnormal part. According to Figure 8, `comp_B` signals an internal exception that should be handled in the context of the component (①). For that, it should use locally available resources or request external resources in a way that is transparent to the rest of the system – this scenario characterises the component context in handling exceptions. However, if the component's abnormal part cannot handle the exception, then this exception should be propagated to the connector, or connectors, managing the component's interactions (exception ② is propagated to `conn_X`).

From the perspective of component's environment, the component is responsible for handling exceptions that are propagated from other components or connectors with which collaborates, and propagate exceptions that it cannot handle locally. By partitioning the component's structure into normal and abnormal ensures that no other behaviour at the component's interface is allowed except for the normal and exceptional behaviours that are specified in terms of what is provided and what is required.

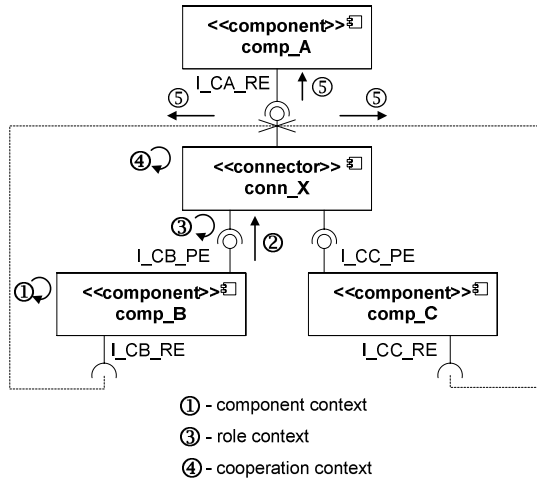


Fig. 8. Propagation and handling of exceptions

2.3.2 From Connectors to Components

When a connector receives an exception propagated from one of its collaborating components, it attempts first to handle the exception at the context of the role played by that component, before handling this exception at the collaboration context. The handling exceptions at the context of roles is beneficial compared with handling of exceptions at the collaborations level because the latter might require more sophisticated and complex means for making sure that all the collaborating components are error free. For example, in Figure 8, when **comp_B** propagates the exception to **conn_X**, the exception is first handled at the role context (③) before being propagated to the collaboration context (④). If the exception cannot be handled at the connector level, then it has to be propagated to all collaborating components (⑤). When this happens, it is the responsibility of the collaborating components to handle these exceptions individually. Different from the forward propagation scenario between components and connectors, such as the exception (②) in Figure 8, this backward propagation essentially notifies the collaborating components that they might be in an erroneous state, and that they should recover from it.

Another scenario in which exceptions have to be handled at the context of the connector is when an internal exception is raised because there was a violation of the collaborative behaviour associated with the connector (internal exception ④, in Figure 8). As before, either the connector handles locally the exception, or propagates the exception to the collaborating components for them to take the necessary corrective actions (exception that is propagated by **conn_X** to **comp_A**, **comp_B** and **comp_C**).

Although the collaborating components should not be aware of the additional behaviour introduced by a connector and its respective exceptions, the exceptions being propagated should be meaningful for the collaborating components. An example of such exception might be a failure exception notifying one of the collaborating components that the requested service cannot be provided. The same

happens when the connector receives an exception from one of the collaborating components. Depending on its handlers, either the connector can treat this exception locally, or request the collaboration from other components for handling the exception.

2.3.3 From Connectors to Connectors

Since connectors may not be able to interconnect directly with other connectors, the propagation of exceptions has to be made through the components that play roles in the different connectors. For example, Figure 9 shows a particular scenario in which the cooperation related to connector `conn_Y` is nested within a cooperation associated with connector `conn_X`. In case `comp_C` fails, which provides two different services for the two connectors – assuming independent services, an exception (①) is propagated to `conn_Y`. If this connector is not able internally to handle the exception, it propagates forward the exception (②) to the collaborating component (`comp_B`). If `comp_B` is not able to treat the exception, it propagates the exception (③) to `conn_X`. Again, if this connector cannot deal with this exception, it should propagate backward the exception (④) to the collaborating components.

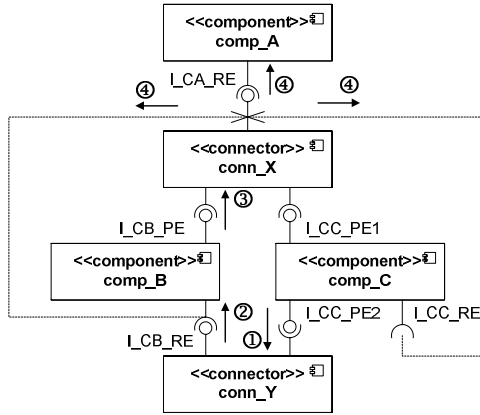


Fig. 9. Exception propagation in nested cooperations

2.4 Verification of the iFTE

The general model of the iFTE, in terms of its components, connectors and their interactions was modelled in UPPAAL using extended timed automata [11], where the interactions between architectural elements were assumed to be blocking request/reply, and represented as synchronous channels. In the behavioural modelling of architectural elements, each of the provided and required interfaces is partitioned into two places for representing input and output of the ports – similar to the AADL model. For example, `I_iFTE_PS_i` represents the input of the provided services interface of the iFTE, while `I_iFTE_PS_o` represents its output. In order to simplify the UPPAAL modelling of the iFTE abstraction and its detailed design the backward propagation of exceptions were not modelled.

The verification of the iFTE was performed in two parts, first, as an architectural abstraction in terms of the behaviour of the iFTE interfaces, and then in terms of the behaviour of the architectural elements that implement an iFTE.

2.4.1 iFTE: Architectural Abstraction

The behaviour of an iFTE is represented as an extended timed automaton that captures the relations between the four types of interfaces, as shown in Figure 10. In this diagram, the places **Client** and **Server** represent two other architectural elements interconnected to the iFTE, the other places connected to these two represent the interfaces of an iFTE, and the committed places represent internal states of the iFTE – these are directly related to the flows specified on the AADL model of the iFTE, presented in Figure 2. The annotated transitions on the iFTE represent the interactions between the iFTE and the **Client** and **Server**. In order to simplify the UPPAAL modelling of the architectural abstraction, the backward propagation of exceptions was not considered.

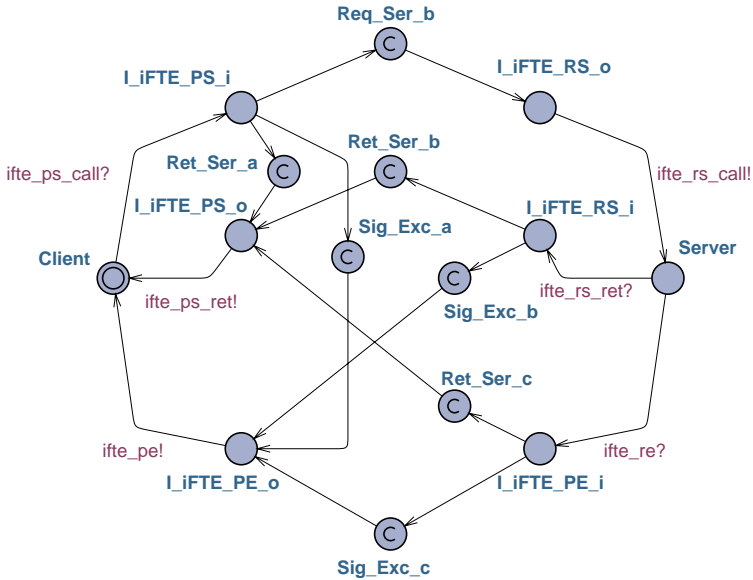


Fig. 10. An extended timed automaton model of an iFTE

For analysing the correctness of the iFTE architectural abstraction, a small architectural configuration consisting of three architectural elements: a **Client**, **Server** and iFTE, were modelled using UPPAAL. This configuration was model checked to ascertain the inexistence of deadlocks, and to confirm the proper flow of services requests and propagation of exceptions.

The same iFTE model was employed to verify the propagation and handling of exceptions on the architectural configurations shown in Figures 8 and 9. The outcome of the analysis has confirmed the proper flow of service requests and the forward propagation of exceptions.

2.4.2 iFTE: Detailed Design

For verifying the detailed design of an iFTE, the same configuration consisting of three architectural elements, employed above, was used. However, instead of using an architectural abstraction for representing an iFTE, the iFTE is represented in terms of its internal architectural elements. Again, a single provided and required services are considered, however more services could have been considered in the model, however these should be assumed to be independent from each other.

The extended timed automata model of the internal architectural elements of the iFTE follow their respective specifications made in AADL. In the following, only the Normal and Abnormal components are presented in more detail.

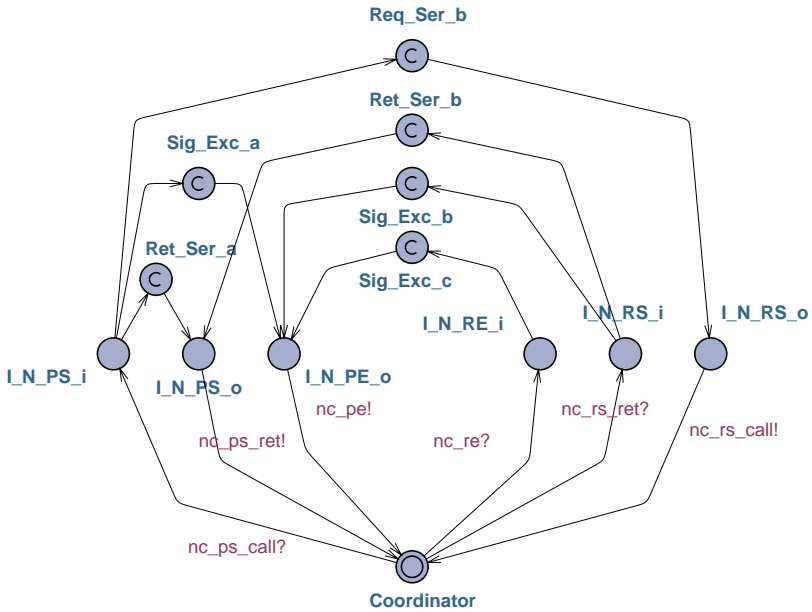


Fig. 11. An extended timed automaton model of Normal component

The modelling of the Normal component is shown in Figure 11 follows from the modelling of the iFTE, described above. The place Coordinator represents the internal connector Coordinator, the four interfaces of Normal are partitioned in terms of inputs and outputs, and the committed places represent the internal states of the Normal component, which is consistent with the AADL representation of Figure 6. The labelled transitions between the Coordinator and the places representing the Normal interfaces are synchronisation channels that capture the interaction between the Coordinator connector and the Normal component. In the following, we describe some of its behaviours. The Coordinator requests a service through *nc_ps_call?*, which might have three possible outcomes: the Normal returns the service (*Ret_ser_a*) through channel *nc_ps_ret!*, the Normal signals an exception (*Sig_Exc_a*) through channel *nc_pe!*, or an external service (*Req_ser_b*) is

requested through `nc_rs_call!`. Once the requested service returns through `nc_rs_ret?`, there are two possible outcomes. Either **Normal** returns a service (`Ret_Ser_b`), or an exception (`Sig_Exc_b`). In case the service requested returns an exception (`nc_re?`), the **Normal** propagates that exception to the **Abnormal** component

The modeling of the **Abnormal** component follows closely that of the **Normal** component, so in the following, we focus on the internal states of the **Abnormal** component. From the **Coordinator**, the **Abnormal** receives an exception (`ac_re?`) originated either from the **Normal** component, or from an external service required by the **Abnormal**. The handling of this exception can follow three possible paths: `Ret_Ser_a` - **Abnormal** handles the exception and returns a service to the **Coordinator** (`ac_ps_call!`), `Req_Ser_b` - **Abnormal** request additional service either from the **Normal** or another external architectural element (`ac_rs_call!`), and `Sig_Exc_a` - **Abnormal** is not able to handle the exception and propagates the exception (`ac_pe!`). When the **Abnormal** receives the requested service (`ac_rs_ret?`): handles the original exception and returns to normal service (`Ret_Ser_b`), propagates the exception because is not able to handle it (`Sig_Exc_b`), or requests additional services (`Req_Ser_c`).

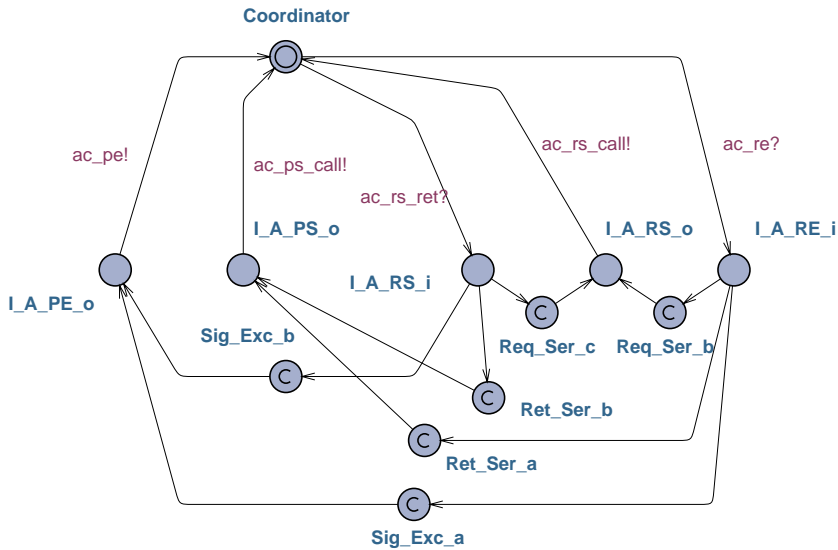


Fig. 12. An extended timed automaton model of Abnormal component

For analysing the correctness of the detailed design of an iFTE, the architectural configuration previously employed was used. Instead of the model of the architectural abstraction, the detailed model of an iFTE was employed. The analysis has shown that the configuration was deadlock free, and that the propagation of exceptions across the internal architectural elements of an iFTE was according to specification of the architectural abstraction.

3 Case Study

The example that has been chosen is a simplified version of the control system for a mining environment [19]. The extraction of minerals from a mine produces water and releases methane gas to the air. In addition to extracting minerals, the mining control system is used to drain water from the sump, and to remove air from the mine when the methane level becomes high. The mining control system consists of three subsystems for extracting minerals, for controlling the level of water in the sump, and for controlling the level of methane in the mine. When the water reaches a high level, the pump is turned on and the sump is drained until the water reaches a low level. A water flow sensor is able to detect the flow of water in the pipe. However, the pump is situated underground, and for safety reasons it must not start, or continue to run, when the amount of methane in the mine exceeds a safety limit. For controlling the level of methane, there is an air extractor controller that monitors the level of methane inside the mine, and when the level is high an air extractor is switched on to remove air from the mine. The whole system is also controlled from the surface via an operator console that should handle any emergencies raised by the automatic system.

3.1 Architectural Representation

The architectural representation of the mining control system is shown in Figure 13. In this representation, components and connectors are represented as stereotyped UML2.0 components, and the links between the architectural elements are represented as dependencies. It is assumed that in this system all the architectural elements are iFTEs, except for the four sensors (AirFlow, MethaneHigh, WaterLow, WaterHigh).

In this architectural configuration, there are three controllers implemented as idealised fault tolerant connectors: **MineralExtractorController**, **AirExtractorController**, and **PumpController**. Each controller is responsible for dealing with the normal behaviour of the system, and handling any exceptions that are propagated by the components. Depending on the state of the sensors (MethaneHigh, WaterLow, WaterHigh), one of the controllers will be always activated: in normal conditions – the water level and the concentration of methane are low, the **MineralExtractorController** is activated, when the water level is high and the concentration of methane low the **PumpController** is activated, and when the concentration of the methane is high the **AirExtractorController** is activated. In case there is a failure in one of the architectural elements that cannot be handled by the system, it is the responsibility of the **MineralExtractorController** to notify the key elements of the architectural configuration that such a failure has occurred. In the configuration of Figure 13, this is done through backward exception propagation from the **MineralExtractorController** interface `I_MEC_PE`.

3.2 Exception Propagation

In order to exemplify the flow of exceptions, in the following, we consider the case in which the **AirExtractor** fails. The propagation of exceptions is represented in a simplified architecture of the mining control system, as shown in Figure 14.

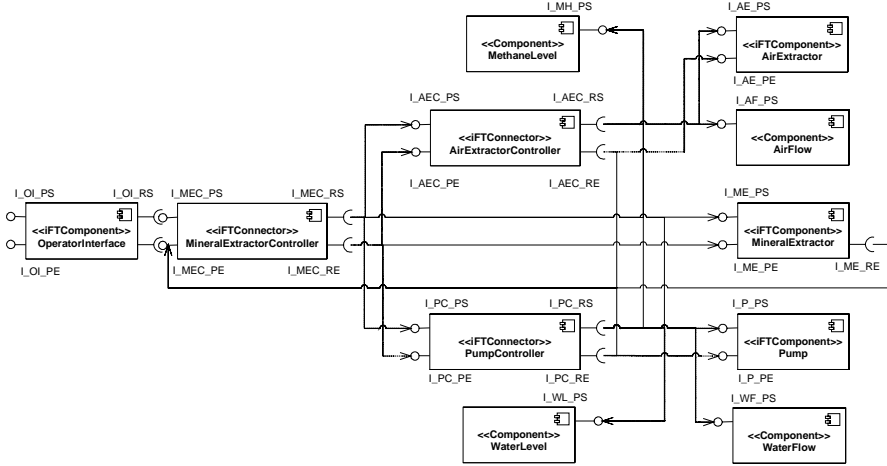


Fig. 13. Architectural configuration of the mining control system

When an error is detected inside the **AirExtractor**, an internal exception is raised (①) and locally handled. If **AirExtractor** is not able to handle this exception, it propagates an exception to **AirExtractorController** (②). Again this component attempts to handle this exception at the role context (③), but if it fails, it propagates the exception to **Controller** (④). Since the concentration of methane is high and the **AirExtractor** has failed, there is nothing that **Controller** can do, except to propagate an exception among its collaborating architectural elements (⑤). Upon receiving this exception, the **MineralExtractor**, the **PumpController** and the **AirExtractor Controller** should shut down their activities, and the **OperatorInterface** should raise an alarm for the operator to take the appropriate measures.

3.3 Evaluation

Aiming to analyse the request of services and the propagation of exceptions on architectural configuration of iFTEs, the mining control system, as represented in Figure 13, was modelled using extended timed automata templates used for modelling the iFTE architectural abstraction. Although the model did not include the internal architectural elements of an iFTE, it was enough to represent the basically functionality of the mining control system. Using model checking, it was verified that the architectural configuration was free of deadlocks, and all the system states were reachable. Based on this preliminary analysis, the model was validated by checking whether there was a proper invocation of services and propagation of exceptions according to predefined scenarios. For example, the exception propagation represented in Figure 14 was demonstrated to occur. This analysis was performed by simulating the model, and make sure that no alternatives traces existed that would lead to non-expected propagation scenario. This has shown that the architectural configuration was able to tolerate some faults that might occur in the system. However, this analysis was not exhaustive due to limitations in the modeling

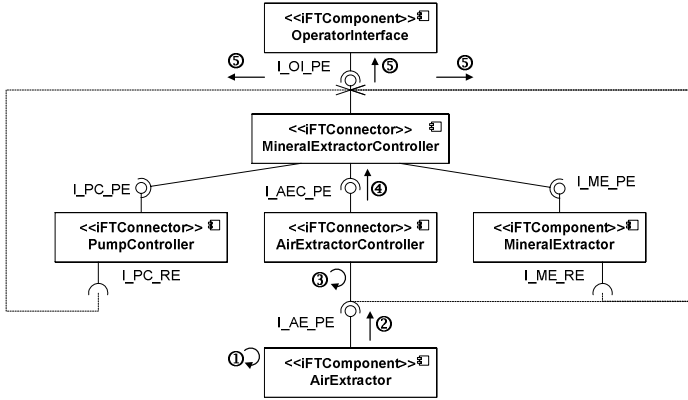


Fig. 14. Representation of an exceptional scenario

technique, which is not flexible enough to allow the incremental development of the models in order to represent a wide range of failures.

The complete analysis of the impact of using the iFTE when architecting fault tolerant systems is beyond the scope of this paper. However, brief comments can be made from the experience of using the architectural abstraction. As already mentioned in the Introduction the separation between computation and coordination enforces modularisation, which facilitates the process of decomposing the different architectural elements in terms of normal and exceptional behaviour. Hence, one of the advantages of using iFTE when architecting fault tolerant is its ability to be instantiated to both components and connectors. However, this modularisation, which is helpful for achieving a good structuring for the purpose of error confinement and handling, has its disadvantages. The performance of the system might be affected because more components are involved for the provision of the same services, but this is the price that has to be paid for enforcing the separation between normal and exceptional behaviour. Concerning scalability, this is more a limitation of exception handling than the architectural abstraction being proposed. Since exception handling for the provision of fault tolerance is an application dependent technique, it depends on the characteristics of the application and the availability of redundancies to determine how many exceptions, and their respective handlers, are necessary for a system. The use of architectural abstractions like the iFTE tends to facilitate the design of systems which are more complex in nature, which in our case, means to have more exceptions to handle or propagate in order to tolerate faults.

4 Related Work

In the context of error handling, although exceptional handling at the architectural level has been reported in the literature, most of the contributions still consider the component to be the place where exceptions should be handled. An example of such contribution is the idealised C2 component (iC2C), based on the idealised fault-tolerant component, which has been proposed for structuring software systems

compliant with the C2 architectural style [10]. This is also case of architectural approaches based on Catalysis [14] and DUALY [15]. A more general strategy for exception handling is the integration of two complementary strategies, a global exception handling strategy for inter-component composition, and a local exception handling strategy for dealing with errors in reusable components [4]. An important issue when dealing with exceptions at the architectural level is techniques for analysing the flow of exceptions. The Aereal framework allows specifying rules to be associated with the propagation of exceptions, and checking for violations of these rules [3]. However, when considering collaborating components, components alone might not be able to handle exceptions. Instead, components might need to collaborate with other components for handling a particular failure scenario, in the same way that components need to collaborate to deliver a specified correct service.

5 Conclusions

In this paper, we have presented an architectural approach, based on exception handling, for structuring fault-tolerant software. In this approach, the architectural elements are partitioned into normal and exceptional parts, thus promoting a clear separation of concerns on how errors are detected, and how they should be handled. The proposed idealised fault-tolerant architectural element (iFTE) has been represented using AADL and verified using model checker UPPAAL. The overall feasibility of the proposed architectural abstraction, and its respective detailed design, has been evaluated in the context of simple case study, the mining control system.

Although the idealised fault-tolerant architectural element (iFTE) has shown to be effective in obtaining well-structured systems by promoting error confinement, one major limitation concerning this concept is its failure assumptions, which might be difficult to enforce considering the inherent complexity of some software components. For example, when a software component fails, how to enforce the only output it produces is a failure exception? It is clear that when devising abstractions for structuring software systems at the architectural level, more realistic failure assumptions have to be considered, and currently work is in progress that considers different fault-tolerant architectural abstractions that are able to enforce more realistic failure assumptions. Concerning the verification of the architectural abstraction, more appropriate formal techniques, like the B method and CSP, are being investigated that would allow to perform a more thorough analysis in exception. Regarding the representation of the proposed architectural solution using an architectural description language, work has already started in representing the iFTE in terms of the AADL Error Model Annex [17], which complements the description capabilities of the AADL core language.

Overall, since exception handling is an application dependent solution, special care has to be taken for avoiding increasing system complexity unnecessarily because any undesirable circumstance might be considered an exception. Moreover, since the handling of exceptions rely on rollback and rollforward techniques for error recovery, if these are not properly implemented, complexity might also increase. These issues can be considered a weakness when devising solutions that can be applied to a wide range of fault-tolerant software applications. For that, validation means that include

regression testing and fault injection are being integrated to a development approach based on the iFTE. These provide additional assurances that the final system code is an actual implementation of its architectural representation.

Acknowledgements

The author would like to thank Patrick H. S. Brito, Fernando Castor Filho, Paulo Asterio de C. Guerra, and Cecília Mary F. Rubira for their contributions to this paper.

References

- [1] Anderson, T., Lee, P.A.: *Fault Tolerance: Principles and Practice*. Prentice-Hall, Englewood Cliffs (1981)
- [2] Avizienis, A., Laprie, J.-C., Randell, B., Landwehr, C.: Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing* 1(1), 11–33 (2004)
- [3] Castor Filho, F., Brito, P.H.S., Rubira, C.M.F.: A Framework for Analyzing Exception Flow in Software Architectures. In: *Proceedings of the ICSE 2006 Workshop on Architecting Dependable Systems (WADS)*. St. Louis, MI, USA. May 2005. pp. 21–27 (2005)
- [4] Castor Filho, F., de C Guerra, P.A., Pagano, V.A., Rubira, C.M.F.: A Systematic Approach for Structuring Exception Handling in Robust Component-Based Software. *Journal of the Brazilian Computer Society* 3(10) (2005)
- [5] Cristian, F.: Exception Handling. *Dependability of Resilient Computers*. Anderson, T., (ed.) BSP, pp. 68–97 (1989)
- [6] Clements, P., et al.: *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, Reading (2003)
- [7] de Lemos, R., de C. Guerra, P.A., Rubira, C.: A Fault-Tolerant Architectural Approach for Dependable Systems. *IEEE Software (Special Issue on Software Architectures)*, 80–87 (2006)
- [8] de Lemos, R., Romanovsky, A.: Exception Handling in a Cooperative Object-Oriented Approach. In: *Proceedings of the 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'99)* Saint Malo, France. May 1999, pp. 3–13 (1999)
- [9] Gacek, C., de Lemos, R.: Architectural Description of Dependable Software Systems. In: Besnard, D., Gacek, C., Jones, C.B. (eds.) *Structure for Dependability: Computer-Based Systems from an Interdisciplinary Perspective*, pp. 127–142. Springer-Verlag, London, UK (2006)
- [10] de C. Guerra, P.A., Rubira, C., de Lemos, R.: A Fault-Tolerant Software Architecture for Component-Based Systems. In: de Lemos, R., Gacek, C., Romanovsky, A. (eds.) *Architecting Dependable Systems*. LNCS, vol. 2677, pp. 129–149. Springer, Heidelberg (2003)
- [11] Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL in a Nutshell. *International Journal on Software Tools for Technology Transfer* 1(1–2), 134–152 (1997)
- [12] Perry, D.E., Wolf, A.L.: Foundations for the Study of Software Architectures. *SIGSOFT Software Engineering Notes* 17(4), 40–52 (1992)

- [13] Randell, B.: System Structure for Software Fault Tolerance. *IEEE Transactions on Software Engineering* 1(2), 220–232 (1975)
- [14] Rubira, C.M.F., de Lemos, R., Ferreira, G.R.M., Castor Filho, F.: Exception Handling in the Development of Dependable Component-Based Systems. *Software-Practice and Experience* 35(3), 195–236 (2005)
- [15] Di Ruscio, D., Muccini, H., Pelliccione, P., Pierantonio, A.: Towards Weaving Software Architecture Models. In: ECBS, Joint Meeting of the 4th MBD and 3rd MOMPES. Potsdam, Germany, March 2006 (to appear)
- [16] SAE-AS5506 Architecture Analysis and Design Language. Society of Automotive Engineers (SAE) (2004)
- [17] SAE-AS5506/1 SAE Architecture Analysis and Design Language (AADL) Annex, vol. 1 Annex E: Error Model Annex. International Society of Automotive Engineers. Warrendale, USA (June 2006)
- [18] Shaw, M., Garlan, D.: *Software Architectures: Perspectives on an Emerging Discipline*. Prentice-Hall, Inc. Upper Saddle River, NJ (1996)
- [19] Sloman, M., Kramer, J.: *Distributed Systems and Computer Networks*. Prentice Hall, Englewood Cliffs (1987)

Model-Centric Development of Highly Available Software Systems*

Rick Buskens¹ and Oscar Gonzalez²

¹ Lockheed Martin Advanced Technology Laboratories

rbuskens@atl.lmco.com

² Bell Laboratories, Alcatel-Lucent

ojgonzale@alcatel-lucent.com

Abstract. In today's rapidly evolving marketplace, the ability to quickly build and deploy new systems is an increasingly critical factor in a company's success. For certain domains, such as telecommunications, it is taken for granted that systems will be highly available, with expectations of "5 9s" or even higher availability, translating to five minutes or less downtime per year. However, building highly available systems is generally very challenging, and becoming even more challenging as the systems increase in complexity. High availability (HA) middleware solutions partially address this challenge by providing common HA services that system developers can use. However, developers still need to spend significant effort integrating their systems with the HA middleware. In this paper, we present the Aurora Management Workbench (AMW) as a solution to the integration problem. AMW is an HA middleware and tools for building highly available distributed software systems. It is unique in its approach for developing highly available systems: developers focus only on describing key architectural abstractions of their system as well as system high availability needs in the form of a model. Tools then use the model to generate much of the code needed to integrate the system with the AMW HA middleware, which also uses the model to coordinate and control HA services at run-time. This paper describes our approach and our initial successes using it to develop commercial telecom systems.

Keywords: high availability, middleware, model-centric software development.

1 Introduction

Developing highly available distributed software systems is a challenging task, and becoming even more challenging as system size and complexity continue to rise. To succeed at this task, software system developers must not only intimately understand the domain of the systems they build, they must also be experts in high availability.

High availability (HA) middleware aims to relieve some of this complexity by providing reusable foundational building blocks, or services, for reliability. ISIS [4] and Horus [26] are two early examples of such middleware; ARMOR [17] is another

* This work was done while the author was at Bell Laboratories.

more current example. Systems such as DOORS [8], Eternal [24] and AQuA [10] are CORBA-based HA middleware solutions that contributed to the development of the FT-CORBA [13] specification. Work in progress by the Service Availability Forum (SAF) [31] aims to standardize HA middleware APIs that are platform-independent. There are also a number of commercial HA middleware solutions [9,12,14]. All HA middleware solutions require run-time configuration to be specified – e.g., the assignment of software components to processing nodes, heartbeat monitoring time intervals and monitoring periods, etc. – through a combination of manually-generated configuration files and/or graphics user interface tools.

Leveraging existing HA middleware solutions requires significant design and development work in order to integrate the HA middleware with a software system of interest. A critical requirement is that the object model of the software system must align with the object model of the HA middleware. The impact of numerous application-specific issues on high availability must also be understood, such as how dependencies and interactions among software components constrain how high availability features may be used. Refer to Fig. 1. In the figure, the HA run-time infrastructure and HA middleware library code are provided by an HA middleware vendor. To “glue together” the software system (application) with the HA middleware requires application-specific HA code to be written by a developer. Typically, the amount of such code for any sizeable software system is large – on the order of many tens or even hundreds of thousands of lines of code. Furthermore, once a system has been built to be highly available, it is vitally important that the resulting system can be thoroughly tested to ensure it meets its fault tolerance needs.

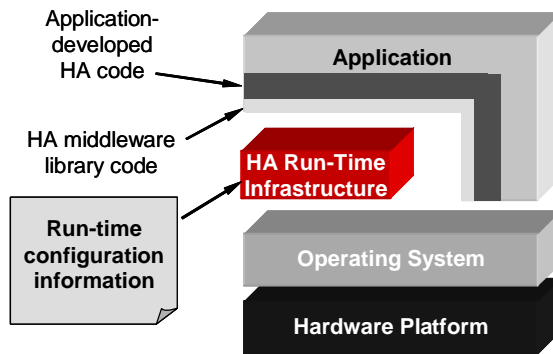


Fig. 1. Typical Software System Leveraging HA Middleware

In the commercial world, this often leads to a dilemma we refer to as the “developer’s paradox”. Developers may want or need to build reliability into their systems. However, market forces typically pressure developers to incorporate as many functional features into their systems as possible, for it is these features that drive revenue for their products. Time that is spent building reliability into a system is therefore time not spent developing revenue-generating features.

To address this critical challenge, the objective of the work presented herein is to make it as simple as possible to build and test highly available software systems. We

achieve this by getting developers to describe their application and its reliability needs in an abstract and simple way. Code generation tools translate these abstract descriptions into much of the code necessary to integrate application software with an HA middleware to provide overall system reliability. The abstract descriptions also specify behavioral properties that must be met at run-time. We refer to the overall approach as *model-centric development* for building highly available systems. Model-centric development based on the UML [34] has been around for years, and papers have been published on using the UML to model dependability concerns [27]. Our work differs significantly from this work in that our models are at an even higher level of abstraction than the design-level models supported by the UML. Techniques to support dependability modeling and analysis of distributed object-oriented applications designed in compliance with the FT-CORBA specification have also been developed [20]. Our work does not address dependability modeling and analysis. The main contributions of our work are: (1) model abstractions of distributed systems and high availability that replace traditional hand coding of software; and (2) an HA middleware and code generation tools that leverage the model to simplify application integration. We are not aware of any other work that focuses on this particular integration problem. While our solution contains several novel reliability features, discussion of these features is *not* the focus of this paper, unless such features are enabled by our approach. We focus instead on presenting the key concepts that enable a separation of concerns so that developers can describe many of their HA needs in a declarative way rather than through detailed design models or actual code.

The remainder of this paper is organized as follows. Section 2 examines key issues faced by developers when building highly available software systems. Section 3 briefly describes the general area of model-centric development and how we intend to apply it to address high availability. Section 4 discusses the Aurora Management Workbench (AMW), our model-centric approach towards developing highly available software systems. We briefly describe our solution implementation in Section 5, and summarize our experiences of applying this solution in practice in Section 6. Future work is presented in Section 7. Summary and conclusions are presented in Section 8.

2 The Challenges of Building Highly Available Systems

Building highly available systems poses numerous challenges. In this section, we discuss what we believe are four of the most important challenges. Each challenge is discussed both generally and in the context of integrating with an HA middleware. For convenience, the discussion focuses on software systems.

2.1 Significance of System Architecture

System architecture has a critically large impact on the set of issues that must be addressed in order to construct a highly available system. Consider two implementations of the same software system (application) in an environment where only crash failures occur. One implementation consists of a single process P that maintains state S (Fig. 2(a)). For simplicity, suppose we design P to be resilient to failures by having P checkpoint its state S every time S changes. Fig. 2(b) shows a

second implementation that consists of two interdependent communicating processes, A and B . State S in the single process implementation is now partitioned into states S_A and S_B . To design this second implementation to be resilient to failures requires managing the communication channels between A and B , including re-establishing the channels after process failures, coordinating the checkpoints of S_A and S_B so that a consistent snapshot of the current overall system state is taken, and re-synchronizing S_A and S_B when either A or B fails and is recovered. Additionally, testing of a larger number of scenarios is necessary in order to confirm correct behavior under failures.

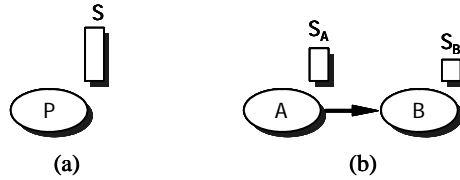


Fig. 2. Different Architecture Realizations for Same Problem

As the above example illustrates, more complex system architectures require more issues to be addressed when building highly available systems as compared to less complex system architectures, with a commensurate increase in effort required to achieve high availability. Practical systems today are significantly more complex than the simple examples presented above. Furthermore, a simple system architecture that meets application functional requirements is often in direct conflict with the more complex architecture that is necessary to meet system scalability, performance, and availability requirements. Unfortunately, it is the system developer who must deal with the added complexity. High availability middleware solutions available today offer limited assistance in dealing with these kinds of architectural issues.

2.2 Initialization and Run-Time Upgrades

Almost without exception, published research ideas on fault tolerance are described with respect to a system in operation. In practice, developers spend significant time addressing initialization and run-time upgrades. While fault tolerance techniques that are applied during normal system operation can usually also be applied during system initialization or upgrades, often the unique behavior of initialization or upgrades will allow for special, more efficient, techniques to be used.

Initialization refers to starting up a system and bringing it to a point where the system can begin to perform its function. Any and all state information is derived from typically static configuration data (e.g., stored in databases, configuration files, etc.) plus initialization activities involving interaction among components (of the system being initialized and/or other systems). Checkpointing during initialization can often be avoided, as any state lost due to failures can usually be easily re-derived. For large systems, intelligent failure handling during initialization is important both to permit successful initialization of portions of the system and to keep initialization times low in spite of unexpected failures [25].

Run-time upgrades result in a change to the run-time configuration of a system while the system is operational. Examples of run-time upgrades include adding a component (hardware or software) to a system or replacing a component with a newer version. For systems requiring very high availability (e.g., 5 9's or higher), upgrades must be performed without disrupting ongoing operation. Because normal operational state continues to change while the system is being upgraded, simple rollback recovery techniques cannot be easily applied when failures occur during an upgrade.

Support for initialization and run-time upgrades, along with fault detection and recovery during these system states, must also integrate seamlessly with fault detection and recovery support during normal operation. Significant portions of these capabilities are application-specific; hence the burden of ensuring these capabilities are performed properly rests on the shoulder of the developer.

2.3 Expertise Required

Constructing highly available systems requires significant expertise in fault tolerance. Developers must thoroughly understand not only the system components they are responsible for developing and the relationship of these components with others in the system, they must also understand techniques for fault detection, isolation and recovery, state preservation, etc., and how interdependencies among system components affect the way in which these techniques must be realized to meet system availability needs.

Developers that use an HA middleware must map their desired approaches for providing high availability to calls to the HA middleware APIs. To maximize flexibility of the HA middleware, these APIs tend to implement basic, low-level services that can be composed to provide higher-level, more complex HA services. The developer is responsible for constructing the appropriate higher-level services from the basic building blocks. This usually translates into a significant amount of coding for the developer (more on this in Section 6).

The HA middleware must also be configured for run-time operation. This run-time configuration must be kept consistent with the API usage in the code to ensure proper operation, made more difficult by the fact that developers writing system code are typically not the same as those who specify the run-time configuration. Mismatches between API usage and the run-time configuration can lead to incorrect or even inconsistent and unexpected behavior of the HA middleware at run-time.

2.4 Testability

Another significant challenge in building highly available systems is in testing them to ensure they perform correctly. Thorough testing is necessary. To ensure correct behavior requires that failures be injected to trigger HA operation at key points during system operation. For any practical system, this a monumental task. Ideally, testers need to be able to control the behavior of the HA run-time infrastructure and how it interacts with the rest of the system. Today's HA middleware solutions offer limited assistance with this task.

3 Model-Centric Development for High Availability Services

Model-centric development offers significant promise as a technique for building software systems more quickly and easily than by traditional methods [30]. The basic principle is that important system characteristics are captured via abstractions that form a model of the system. The model then acts as input for code generation.

Commercial tools such as IBM's Rational Software Architect [16] and Telelogic's Rhapsody [33] are two examples of model-driven development tools for software developers. These tools allow developers to compose high-level models from foundational model primitives, where systems comprise software components that implement state machines and interact through various communication mechanisms.

A second class of model-centric development tools aims to provide high-level model abstractions customized for specific domains. Rather than allow developers to build models by using and composing low-level primitives, *domain-specific modeling languages* (DSMLs) define model elements that capture the essential high-level model components of the specific domain. Researchers at Vanderbilt University have taken this a step further and developed the Generic Modeling Environment (GME) [19] that provides a DSML for developing other DSMLs.

To address many of the challenges of building highly available software systems described in Section 2, we have developed a more holistic approach to building such systems through the use of models. In particular, we have developed a domain-specific modeling language for modeling system high availability characteristics. Interestingly, a sizeable number of the key model elements capture architectural elements of software systems. In addition to the novelty of the modeling language itself, our approach offers two additional unique characteristics:

1. Our domain-specific modeling language incorporates both architectural attributes and run-time behavioral attributes, making it easier for developers to ensure that these two sets of attributes are well-matched, and enabling tools to be developed to check for inconsistencies in the specification of such attributes;
2. Our approach is the first we know of that uses model-centric development to address the *integration* of a software system with an HA middleware.

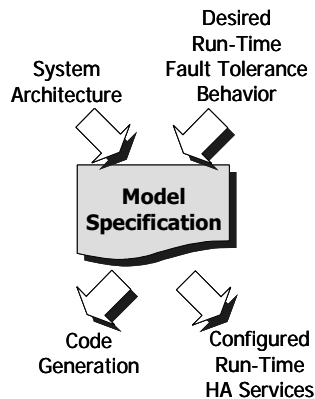


Fig. 3. Model-centric approach to developing highly-available systems

Fig. 3 captures the essence of our approach. Models developed using our domain-specific modeling language capture both architectural and run-time behavior attributes of the software system being developed. These models then act as the *single source* for both code generation, reducing programmer effort and easing integration of the HA middleware with the system being developed, as well as the configuration of run-time HA services. The next section discusses our solution in detail.

4 Aurora Management Workbench

The Aurora Management Workbench (AMW) is our implementation of a model-centric approach to building highly available systems [6]. AMW consists of a modeling language for specifying highly available systems, code generators that generate code from the model, and a high availability middleware – including run-time entities – that coordinate initialization, fault detection and recovery, and run-time upgrade procedures using the model. An overview of AMW is presented next.

4.1 Overview

Refer to Fig. 4. Compared with the typical software system that utilizes an HA middleware (Fig. 1), AMW's configuration specification – i.e., the model – contains *build time* information used for code generation purposes. In addition, because the complete model is available to the HA middleware rather than being embedded in the code written by system developers, the HA run-time infrastructure is able to provide additional HA services that have traditionally been considered application-specific. With this approach, functional code traditionally written to capture application-specific fault tolerance behavior is replaced with high-level specifications of desired behavior. That is, model abstractions replace traditionally hand-written code. While developers must still write application-specific code to support high availability, the

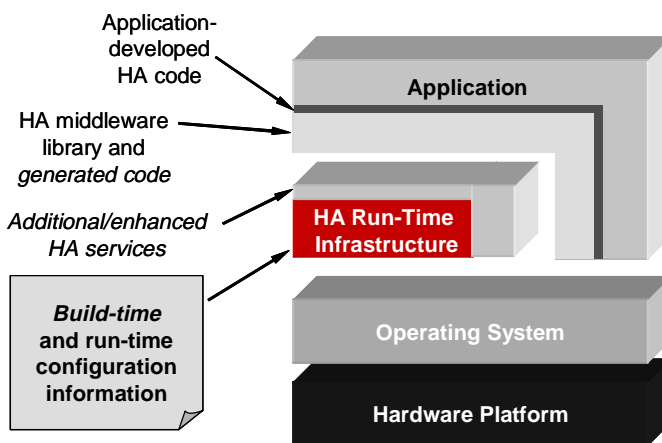


Fig. 4. AMW approach to developing highly available systems

quantity of such code written is significantly reduced and focuses only on application-specific needs that cannot be captured in the high-level models.

AMW provides its own implementation of an HA middleware. In keeping with our objective of making it as simple as possible for developers to build highly available systems, AMW aims for as much transparency to the developer as possible. Existing HA middleware solutions do not permit the same level of transparency for the various HA services, including management of inter-component communication channels (Section 4.5) and transparent fault injection (Section 4.4) to name a few. Even with code generation from model specifications, leveraging existing HA middleware solutions would require significantly more work for system developers than is required with AMW. We now discuss key aspects of AMW in more detail.

4.2 Model Abstractions: System Architecture

System architecture is defined as the structure of the system, defined in terms of system elements, any externally visible properties of the elements, and the relationship among the elements [1]. Fig. 5 pictorially captures the key system architecture elements of the AMW model. The key system architecture elements are:

- *Node* or *processor*. This type of system element hosts the executing software entities of the system. Nodes may be heterogeneous in terms of processing capacity, memory, disk space, operating system, or any other attributes. During operation, nodes contain zero or more *capsules*.
- *Process* or *capsule*. This system element is the smallest unit that can be started from within an operating system shell. In a UNIX environment, a capsule is equivalent to a UNIX process. Capsules contain one or more *software components*.
- *Software component* or *server*. This system element is a logical entity to be treated as the smallest software unit for initialization, recovery, upgrades, etc. A software component supports one or more communications *interfaces* and interacts with other software components through message passing.
- *Interface*: a communications port into a software component. In our model, each component has one *AMW interface* and zero or more *application interfaces*.
- *Communication link*, or simply *link*: a channel that supports communication between two components. Specifically, a communication link allows one component to send messages to a specific interface of another component.
- *Interdependency*: describes an application-specific relationship between software components. We have identified three types of dependencies that impact the ordering of initialization, recovery, and run-time upgrade events among system elements. These dependency types are: communication dependencies, data dependencies and relationship dependencies [25].

To facilitate ease of use and to ease integration of high availability services into individual system software components, AMW components are comprised of three parts: a *management* part provided by AMW, an *application* part provided by the developer, and a *linkage* part also provided by the developer (Fig. 6). The management part provides most of the high availability support infrastructure needed by a software component. It shields all interaction involving the system component

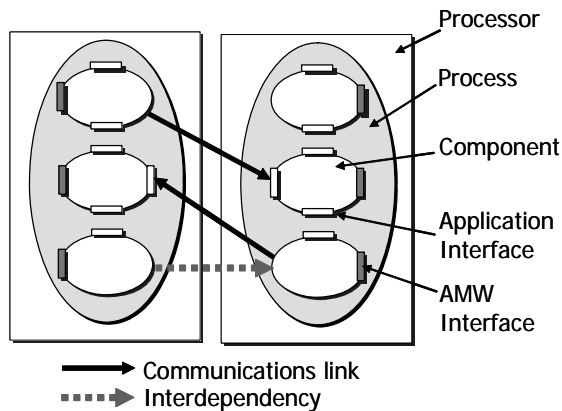


Fig. 5. Key AMW System Architecture Elements

and the AMW run-time infrastructure, and provides some high availability services that are completely transparent to the system component itself. The application part implements the functional capabilities of the system. The linkage part hooks the management and application parts together by providing any system-specific support for high availability. Typically, the linkage part is only a very small amount of code.

In AMW, we further distinguish the *logical* architecture of the system (e.g., the types of components that may reside within certain types of capsules) from the *physical*, or *deployment*, architecture of the system (e.g., how many capsules of a particular type are actually instantiated, which nodes are they instantiated on, etc.). A key benefit of our deployment architecture model is that decisions traditionally made at design time and often embedded in application code are now captured in a model that allows the HA middleware to make these decisions at run-time. One illustrative example is assigning a capsule to run on one of a set of nodes based on the availability and load conditions of all suitable nodes. We know of no other high availability middleware that supports this capability today. Furthermore, the UML does not yet support such dynamic deployment models.

Fig. 7 presents a simple example of a partial architectural specification that illustrates the logical architecture of an example system. The figure shows two types, or *classes*, of capsules, CC1 and CC2, that each contains different types, or *classes*, of components/servers. CC1 contains components of type A and B, and CC2 contains components of type C. Components of type A have no explicit application interfaces (i.e., there is no direct way to send a message to these components). Components of types B have two communications interfaces, BIf1 and BIf2. Similarly, components of type C have two communications interfaces, CIf1 and CIf2. Components of type A communicate with components of type B through communications interface BIf1 and with components of type C through communications interface CIf2. Similarly, components of types B and C interact with each other through communications interfaces BIf2 and CIf1. Finally, components of type A have dependencies on components of type B; components of type B have dependencies on components of

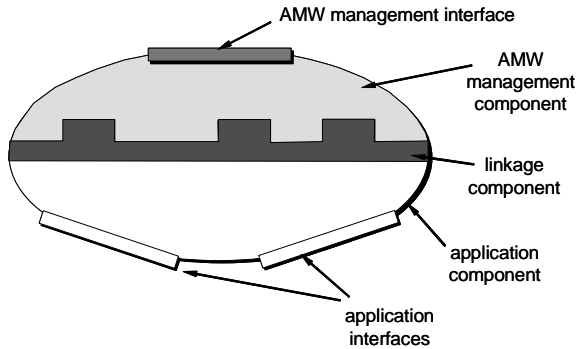


Fig. 6. Detailed AMW component model

type C. (We discuss how these interdependencies affect run-time operation in Section 4.5.) The specification in the right portion of the figure captures what is graphically depicted in the left hand side of the figure.

Developing the model specification of the physical system architecture is almost identical to the procedure followed to capture the logical system architecture. Two key differences are that each specification of an instance of an element must reference, or derive from, the corresponding type specification and that there must be a specification for each instance of an element. For example, given the logical specification in Fig. 7, suppose that the physical architecture required one capsule of type CC1 containing two components of type A and one component of type B as well as two capsules of type CC2 each containing one component of type C. The corresponding physical specification, which forms an augmentation to the logical specification of Fig. 7, is shown in Fig. 8. Note that attributes defined as part of the corresponding class specifications – e.g., the links and interfaces specifications as well as the dependency information – are automatically inherited by the instance specifications. (These inherited attributes can be overridden, if desired.)

With traditional HA middleware solutions, application developers must write code or develop executable scripts to address application-specific fault tolerance behavior (e.g., to implement fault escalation policies). With AMW, application-specific behavior is captured as high-level declarative policies that are interpreted by the AMW run-time infrastructure and translated into coordinated fault management activities between AMW and the application. This eliminates much of the coding normally required by developers. Run-time behavioral policy specifications in AMW support the following fault management activities:

- *Fault detection.* Behavior attributes such as monitoring frequency are specified in the model to aid in detecting software failures.
- *State preservation.* The AMW model supports specifications related to state preservation (i.e., checkpointing). These specifications are used by AMW so that AMW can coordinate recovery actions. Note that coordination of these recovery actions has traditionally been the responsibility of the application. Our approach enables this coordination to be performed mainly by the HA infrastructure, simplifying application development.

- *Fault isolation/fault containment.* Knowledge of the communication links among application components allows AMW to isolate faulty components efficiently. Declaration of failure groups allows AMW to treat a group of components atomically from a failure perspective.
- *Fault recovery.* Examples of behavior attributes that are used during recovery include replication strategies, component criticality and interdependencies.
- *Fault escalation.* AMW provides declarative support for complex, hierarchical, multi-component, system-wide (i.e., cross-node) fault escalation policies; no coding is required on the part of the developer to implement this application-specific behavior. While we are aware of HA middleware solutions that support declarative fault escalation policies, such policies are typically non-hierarchical or are confined to a single node [14].

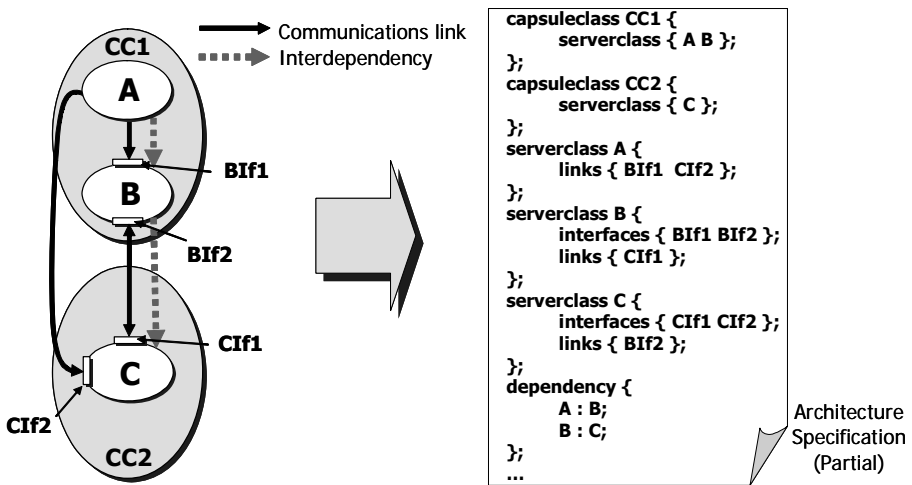


Fig. 7. Example AMW logical architecture specification (partial)

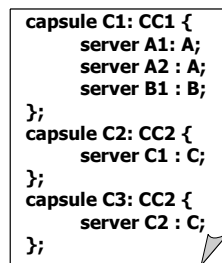


Fig. 8. Example AMW physical architecture specification (partial)

Fig. 9 provides an example specification capturing desired run-time behavior in the AMW model, specifically focused on failure groups and declaratively specifying fault escalation policies. The logicalGroup model construct defines a group of software

process components containing those elements specified in the members model construct (in this case, the group consists of only a single member). The fault escalation strategy, identified by the escalationStrategy model construct, defines a policy for fault escalation for members of the logical group. In this particular instance, the fault escalation strategy consists of two components: a leaky bucket strategy followed by a migration strategy. The leaky bucket strategy specifies that member capsule1 should be restarted on a processor when it fails, unless it fails more than three times in a 100-second window, in which case the node on which capsule1 was running when this failure threshold is reached should be rebooted (indicated by the auxiliaryAction). At this point, the escalation strategy specifies that capsule1 should be migrated to run on a different processor. The strategy further specifies that at most three processor migrations are permitted.

Note that traditional highly available systems would require the above policy specification to be implemented within the system code itself and would require several hundred or more lines of code to be implemented. Furthermore, a change to the policy would require modification of this code. With AMW, a change to the policy simply requires changing the specification; no changes to the AMW run-time infrastructure or the system itself are needed.

AMW further supports declaring fault tolerance behavior not only for traditional run-time operation, but also for initialization and run-time upgrades, where special considerations may warrant special behavior.

4.3 Code Generation

A key aspect of model-centric software development is code generation from the model descriptions. Code generation in AMW provides much of the needed software for integrating an application with the AMW HA middleware. It not only reduces developer effort but also results in improved code quality over manual efforts.

Fig. 10 illustrates the basic operation. System architectural specifications plus specifications of desired run-time behavioral policies are combined to form a

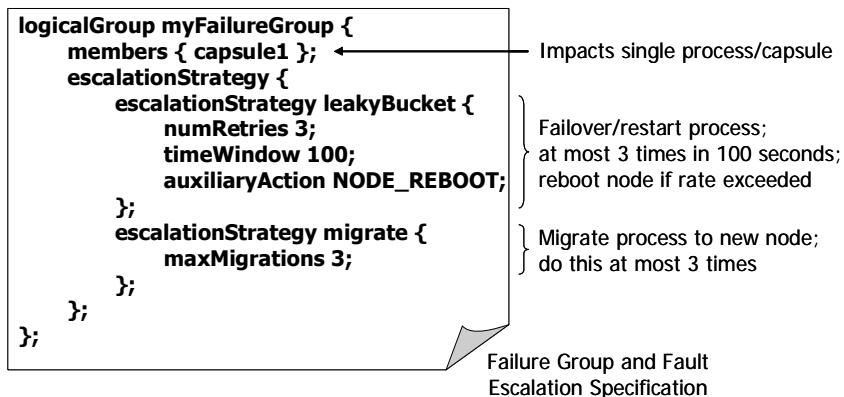


Fig. 9. Example AMW run-time behavior specification

configuration file. This configuration file acts as input to AMW's code generation. Code generation produces code that implements capabilities that handles much of the integration of system software components with the AMW HA middleware, a number of which are application-specific. Once AMW code generation is complete, the work that remains for the developer is system-specific functional behavior (i.e., behavior that is not related to high availability operation) as well as a small amount of activity required by the developer to complete the integration of system software components with the AMW HA middleware.

AMW has two code generators. The first of these utilizes system architecture and fault tolerance behavior specifications of our model to generate highly integrated code impacting initialization, fault detection and recovery, and run-time upgrades:

- For each component, a set of tables that hold strongly typed communication handles used to communicate with other components. Developers thus use structured messages for component interaction, versus unstructured byte streams.
- For each component, a management interface between the component and our HA middleware run-time system. This includes a messaging infrastructure, hidden from the developer that is customized to application-specific HA needs.
- For each component that performs checkpointing, a set of strongly typed APIs for saving, deleting, and restoring checkpoints. The strong typing of the checkpointing APIs simplifies application developer effort, eliminating the need for developers to write conversion routines that convert structured data to unstructured data. These APIs completely hide the details of where the checkpoint data is stored. Control of the checkpoint destination (either file or backup component) is managed by the AMW run-time infrastructure, simplifying developer effort.
- For each capsule, code that coordinates component creation and instantiation, including thread management, event processing, etc.

In addition, the generated code contains hooks for transparent fault injection, to enable controlled testing of high availability services during system operation. As the emphasis is on testing HA infrastructure operation, fault injection capabilities are focused around message exchanges between HA-enabled system components and the AMW HA middleware. Specifically, messages can be lost/dropped or delayed and, in some cases, corrupted. These capabilities apply both to incoming messages (those sent by the AMW HA middleware to the system component) and to outgoing messages (those sent by the system component to the AMW HA middleware).

The second code generator takes an augmented model of the system that also includes descriptions of component messaging interfaces (messages plus messaging content/parameters) then generates actual messaging interfaces for each component, skeleton implementations of the message handlers for the component, a skeleton implementation of the component tied to the corresponding interfaces, plus makefiles needed to build the skeleton versions of executable system entities.

In short, the two code generators together enable the construction of an executable skeleton system from high-level specifications of the system, its architecture, and its fault tolerance needs. To fully exercise the fault tolerance behaviors of the skeleton system, the developer must add code to integrate the application with the AMW

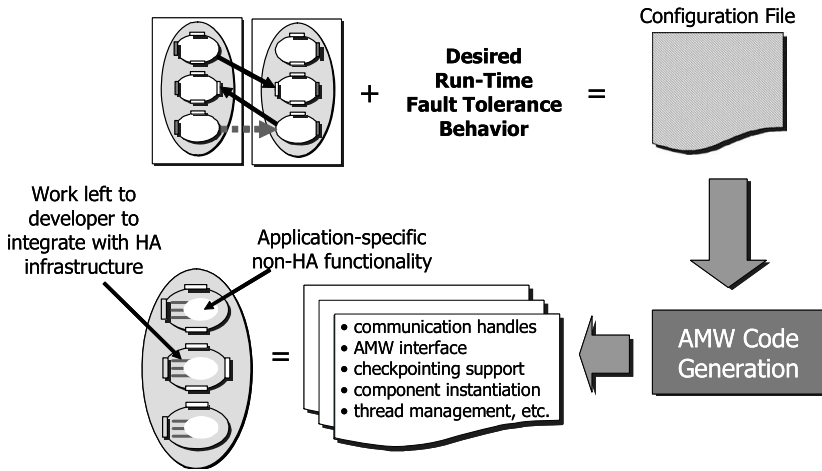


Fig. 10. AMW code generation

run-time infrastructure. This integration effort varies across applications, but in general is minimal. Specifically, developers are responsible for:

- Code that instantiates and terminates components. As the components themselves are system-specific, they require system-specific actions for creation and destruction. Specific component creation actions might include allocating resources, such as memory. Specific component destruction actions might include freeing of those resources. Note that AMW is responsible for coordinating the creation and destruction of these components.
- Code that maintains system-specific component-level data structures. Invocation of these maintenance routines is triggered by AMW as part of initialization, shutdown, failover and auditing activities.
- Invocations of checkpointing APIs, for state preservation purposes. While AMW provides application-specific APIs for checkpointing, based on model specifications of data types to be checkpointed, developers must currently determine what data to checkpoint and when to perform the checkpoints.

4.4 Model-Driven Run-Time Services

As discussed previously, the AMW model supports declarative specifications of desired run-time fault tolerance behavior. These specifications are used by the AMW run-time infrastructure and replace application-specific code that would otherwise be written by developers. By separating the specification of run-time fault tolerance behavior from application code, AMW enables different specifications of run-time behavior to be used for different instances of the same application. For example, one instance of an application may run in an environment where minimal fault tolerance is required; the same application operating in another environment may require more stringent fault tolerance.

Compared with other approaches, our model enables advanced behavior to be handled by the AMW run-time infrastructure, which operates at a finer granularity of control than existing solutions [9, 12, 14], while still not requiring explicit coding by developers. In particular, high availability issues related to application architecture are more insulated from application code. This allows AMW to manage traditional application-independent activities, such as starting and stopping capsules, as well as other application-dependent activities traditionally handled within application code. Some examples include:

- Inter-component communication channels are automatically managed by AMW. This includes setting up the communication channels between communicating components *before* the components are initialized, monitoring the status of the communication channels, and automatically triggering the setup of new channels as a part of failure recovery or run-time upgrades. To achieve this without developer involvement requires separating component *creation* from component *initialization*. AMW first creates components, then sets up inter-component communication channels, then triggers component initialization.
- Component and capsule monitoring performed by AMW to detect entity crashes and hangs requires no coding by developers. Detecting application-specific failures is achieved by overwriting a callback function with application-specific tests.
- The AMW run-time infrastructure manages the destination to which data is checkpointed (e.g., file or backup component). To the application developer, checkpointing is location transparent. A component checkpoints relevant data at relevant times or upon relevant events; AMW transparently directs the checkpointed data to a corresponding file or backup component. When failures associated with the checkpoint destination occur, AMW automatically executes recovery actions to restore the checkpoint destination, then coordinates with the component taking the checkpoints to trigger a “dump” of the checkpoint data.
- Assignment of capsules to nodes is performed at run-time by AMW. Traditional HA middleware solutions require that one specify *exactly* the set of nodes where a capsule *should* execute. AMW specifications describe where a capsule *can* execute. At run-time, AMW assigns a capsule to one of the suitable nodes based on current conditions of the nodes – e.g., whether they are operational or not and how much the nodes are already loaded.
- Often the most complex parts of building highly available systems are fault handling and fault escalation. In AMW, *system-wide* fault escalation is driven by declarative policies in our model and requires *no coding* by developers. The AMW run-time infrastructure interprets these policies during fault handling. To resolve a recurring fault, AMW escalates from the lowest to the highest impact recovery action. Fig. 9 showed an example model specification for fault escalation.
- AMW facilitates the reconfiguration of a distributed system during operation to reflect a controlled change in available resources. The support provided by AMW allows an application to *create* and *completely integrate* new application software components and additional processing nodes into a running system. Removal of software components and processing nodes is also supported. This capability avoids downtime that might otherwise be required to perform these operations and is the starting building block for more complex services such as software upgrades.

4.5 The AMW Run-Time Infrastructure

The responsibility of the AMW run-time infrastructure is to manage all high availability related aspects of the software system in operation. It consists of two main components, shown in Fig. 11:

- *Configuration manager* (ConfigMgr). The ConfigMgr manages the run-time configuration of software system components, including the processor on which a capsule executes, the assignment of application components to capsules, the fault tolerance characteristics of capsules and components, component interdependencies, etc. In addition, the ConfigMgr also coordinates all run-time HA related activities, including initialization, reconfiguration after failures, and run-time configuration upgrades. It also receives failure notifications and correlates the notifications to perform fault isolation. Each software system will have one active ConfigMgr. To handle failures of the ConfigMgr itself, each software system may have any number of optional standby ConfigMgrs.
- *Element manager* (ElementMgr). ElementMgr starts, stops, and monitors software system entities, and reports detected failures to the ConfigMgr. One ElementMgr resides on each processor in the system.

Additionally, AMW provides an interface that allows it to interact with the “external” world (i.e., entities outside of the AMW operational infrastructure). This permits externally-generated messages and events to be sent to the AMW run-time infrastructure to control its operation, inject failures, trigger run-time upgrades, etc. It also allows external entities to register for and receive AMW events. This is useful for OA&M applications. Importantly, this interface enables AMW to integrate and co-exist with other high availability solutions (a unique feature of AMW, and especially important when dealing with legacy applications).

We intended to leverage an existing high availability middleware and run-time infrastructure in our efforts, rather than developing our own. However, the model-centric approach, combined with our fine-grained object model plus our aim to make high availability as transparent as possible to software developers, required greater capability to be provided by the run-time infrastructure than existing solutions would allow. In particular, a critical requirement of the run-time infrastructure is that it can understand and use the AMW model.

Support for fault injection in the run-time infrastructure is also included. In addition to supporting the injection of message delays and losses and some forms of message corruption, failures of the AMW run-time infrastructure itself (e.g., ConfigMgr, ElementMgr) can be injected.

4.6 Addressing the Challenges

Section 2 presented four key challenges faced by developers building highly available systems. AMW addresses each of the challenges. Specifically:

- *Addressing the complexity of system architecture.* Key components in the AMW model are architectural specifications, introduced for the explicit purpose of simplifying developer effort to address high availability issues directly related to

architectural decisions. Code generation from the AMW model then provides a substantial portion of the code needed to integrate system components with the AMW high availability infrastructure.

- *Addressing the complexity of system architecture.* Key components in the AMW model are architectural specifications, introduced for the explicit purpose of simplifying developer effort to address high availability issues directly related to architectural decisions. Code generation from the AMW model then provides a substantial portion of the code needed to integrate system components with the AMW high availability infrastructure.
- *Addressing initialization and run-time upgrades.* As already described, AMW handles not only fault detection and recovery of a software system during normal operation, it also provides initialization support and support for run-time configuration upgrades, as well as fault tolerance services during initialization and run-time upgrades. Furthermore, AMW's initialization, fault detection and recovery and run-time configuration upgrades capabilities are tightly integrated.
- *Reducing the level of expertise required.* The architectural and run-time behavior specifications in the AMW model replace traditional hand-written code to implement high availability capabilities for system software. In addition, the high-level nature of the model abstractions makes them easy to learn and use.
- *Addressing the concerns of testability.* Code generated by AMW contains hooks for fault injection. The AMW run-time infrastructure is instrumented with fault injection hooks as well. Collectively, this enables testing of a wide variety of scenarios where failures may occur in practice. While the focus is naturally on testing high availability services in the context of a software system, extremely fine-grained control is provided (e.g., drop the first initialization message received by system component X, then delay delivery of the response to the second initialization message by 30 seconds).

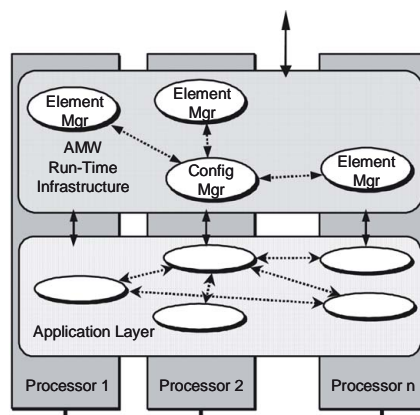


Fig. 11. AMW run-time infrastructure components

5 AMW Implementation

AMW is being used in Alcatel-Lucent's field-deployed products. It exposes a C++ API for application-specific integration with the AMW HA middleware and a CORBA API for application messaging. The AMW code generators and run-time infrastructure comprise 250K source lines of highly modular C++ code. The modeling language is text-based, as examples in the previous sections show.

While the focus of this paper is on the use of high-level models to minimize or replace traditional hand-coding of software in order to make a software system highly available, we now briefly details of some aspects of the AMW implementation:

- *Failure assumptions.* Processors and processes fail only by crashing, and do not generate spurious events before or during a crash ("fail silent"). Application components fail either by crashing or by not responding (e.g., due to deadlocks, etc.). Inter-processor communication links do not fail. Alternatively, the communications infrastructure between processors contains redundant links (dual-link solutions are readily available in hardware today). Any number of processors, processes and software components, including AMW run-time infrastructure components, may fail simultaneously. Failure detection is accurate. These assumptions have proven sufficient to date in situations where AMW is used. Since these failure assumptions are captured internally in AMW (i.e., they are not exposed to developers through the AMW model), they can easily be extended.
- *Failure detection.* Application component failures are detected through periodic callbacks into the component. A positive response to the callback indicates a fault-free component; a negative response or no response after a configurable timeout indicates a faulty component. The callback may invoke a number of application-specific tests, such as data structure audits, as appropriate for the component. Process failures are detected via a periodic heartbeat messaging mechanism between AMW management components residing within the process (refer to Fig. 6) and the AMW ElementMgr residing on the same processor as the component. A positive heartbeat response indicates a fault-free process; a negative response or no response indicates a faulty component. Processor failures are detected via a periodic heartbeat messaging mechanism between AMW ElementMgrs residing on the different processors. An adaptive algorithm based on [3] determines which ElementMgrs monitor each other. ElementMgrs perform local fault isolation and event correlation, then report the failures to the ConfigMgr whereupon recovery is initiated. The specific detection techniques are inherent in the current implementation of AMW. However, the period and timeout values in both the callbacks and heartbeating are configured in the AMW model. These values may be changed dynamically without altering a single line of application code.
- *Replication strategies.* AMW supports three replication strategies, captured at the component level in the AMW model (the impact on process replication is automatically derived from component level replication strategies). Replication strategy *no recovery* instantiates and initializes a component at system startup but does not recover the component if it fails for any reason. This configuration is

useful when a component is needed only for first-time initialization purposes (e.g., a database containing initial configuration data for the component). For the *restart* replication strategy, a single, active instance of a component is instantiated and initialized at system startup and recovered (by restarting the component) if it fails. Specifying an *active-standby* replication strategy causes instantiation and initialization of two instances of the component at system startup, housed in different processes executing on different processors. If the primary instance fails, the backup instance is promoted to become the new primary instance. If either the primary or backup instance fails, a new backup instance will be started. Resynchronization of in-memory checkpoint data will be triggered automatically by AMW. Replication strategies are specified *external* to the component itself. That is, the component contains no code that assumes any particular replication strategy. The model captures all unique characteristics of each replication strategy so that the AMW run-time infrastructure can configure the application to achieve the desired behavior. At run-time, the AMW ConfigMgr exchanges messages with the application component to trigger appropriate behavior.

- *State preservation.* Checkpoints may be taken of entire data structures or of specific elements of larger data structures. Checkpoints may be taken periodically or on an event-driven basis. A novel capability of our code generation approach is that checkpoints are *location-transparent*. The AMW run-time infrastructure determines where to store a component's checkpoint data (e.g., at a corresponding backup component, a file, etc.). Calls to AMW's checkpoint APIs in application code then automatically route the checkpoint data to the AMW-determined location without application knowledge. This simplifies application development effort to support checkpointing, as application components need not concern themselves with deciding where to store checkpoint data nor deal with issues related to failures of these storage locations. It also eases application-specific failure recovery activities, as the AMW run-time infrastructure coordinates state resynchronization across components.
- *Run-time reconfiguration.* Today's implementation of AMW supports processors, processes and application components to be added or removed during operation, through interaction with the AMW ConfigMgr. What distinguishes AMW from all other implementations of HA middleware solutions is that, in the case of adding a new entity, AMW coordinates the complete integration of the new entity into the running system. This includes, where appropriate, automatically instantiating active and standby instances of the component, automatically setting up a checkpoint location for the active component, automatically setting up communication channels between the new component and existing components, automatically triggering new component initialization, automatically promoting appropriate components to play an active role, and automatically registering components to be monitored for failure detection. Traditionally, all of the above activities are performed manually and through code hand-written specifically for the component. Using AMW, all of these activities are coordinated automatically. In addition, the reconfiguration activity is robust to failures.

6 Results

This section presents results of experiments and real systems that use AMW. We first compare the effort required to develop a simple highly available client-server application using AMW versus a commercially available HA middleware. We then briefly describe a real system that uses AMW and provide quantitative data to help assess the benefits provided by AMW. Finally, we discuss and quantify the minimal time required to develop a skeletal new application to be made highly available.

6.1 Example 1: Developing a Simple Client-Server Application

Our first example illustrates the effort needed to build a simple, highly-available client-server application using AMW. The application consists of two client components residing in the same process and one server component that resides in another process. The server provides a simple service: a counter is incremented and its value is returned for each client request. Clients issue requests to the server periodically. The clients and server are configured as active-standby. Checkpointing between the active and standby servers preserves counter values across failures. The clients must not be started unless the server is active. Fault detection of the clients and server and their encapsulating processes is achieved through periodic monitoring and process termination. See Fig. 12 for the application architecture.

For this application, the AMW tools generate 1213 lines of code – 590 lines of code for the clients and 623 lines of code for the server. Only *sixteen* lines of code were hand-written to implement the above service (see Fig. 13). Six lines of client-side code were hand-written to queue a periodic timer, send a request to the server, and print the counter value returned by the server. Ten lines of server-side code were hand-written to initialize and increment the counter, print the counter value after each increment, and checkpoint the counter value. The entire application, including all high availability and fault injection support, was developed in less than two hours.

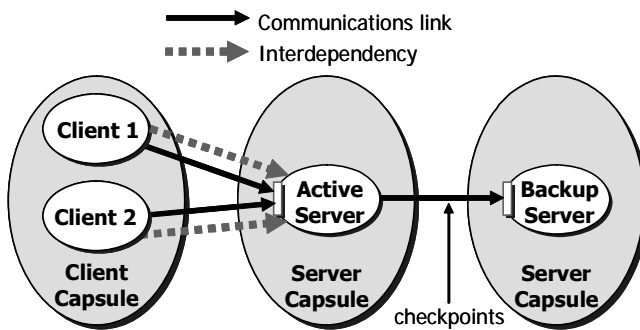


Fig. 12. Architecture of simple client-server application

The same client-server application was written using a commercially available HA middleware and took two weeks to develop. A total of 1030 lines of hand-written code were needed to implement the application and integrate it with the commercial HA middleware. This did not include any hooks to support fault injection.

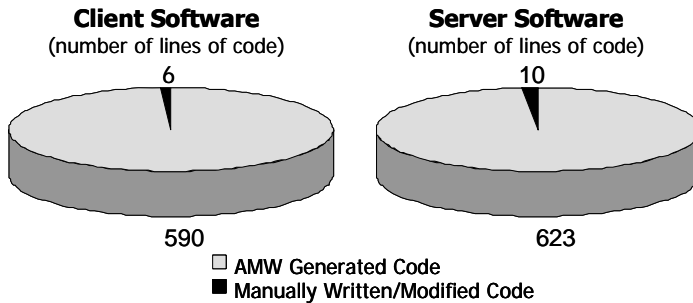


Fig. 13. AMW generated versus hand-written code

6.2 Example 2: Telecommunications Call Processing

AMW is used in field-deployed telecommunications call processing systems. Telecommunications call processing systems, also referred to as switching systems, primarily setup and teardown call sessions between interested parties; these sessions may involve voice-only calls, data-only calls, or a combination of both. A telecommunications call processing network architecture contains functional elements that provide call control, access control, service control, and location registration functions (for wireless networks). Call processing scenarios refer to various groupings of such functions coordinated through sequences of signaling messages. Call processing software must process each call request within a few hundred milliseconds [23], and the entire system should not be out of service for more than a few minutes per year [36].

One such telecommunications call processing systems using AMW consists of several million lines of code comprising a few thousand software components and many tens of thousands of communication links executing on tens of processors. Roughly 150K source lines of code, spread across hundreds of source code files and makefiles, are generated by AMW tools to integrate the product with the AMW HA middleware. This translates into significant development savings (10-15 man-years of effort) versus writing the code manually.

6.3 Thirty Minutes to a Running Skeleton Highly Available System

Rapid application development (RAD) aims to produce prototypes of systems quickly, albeit typically with reduced features and reduced scalability, and to iterate on these prototypes to add capabilities over time [21]. Typical RAD tools (e.g., Borland's C++ Builder [5] and Microsoft Visual Studio [22]) leave high availability as a concern that must be addressed by developers. AMW's tools and run-time infrastructure help to address high availability issues much sooner in development, where problems are much easier to identify and fix. Our experience shows impressively that a skeleton of a complete system can be *operational* in less than thirty minutes after the model specification is developed. Developers can then incrementally add functionality to the system. System testers have immediate access to a running system where configured fault tolerance behaviors and the HA middleware itself can be tested, even before the application programs are designed.

7 Future Work

Numerous directions for future work remain, as there are still many challenges to overcome. Below is a sampling of exciting research directions:

- *Support for richer model abstractions.* Richer model abstractions would further simplify developer effort to build highly available systems. An ambitious goal is to incorporate high-level availability objectives in the model (e.g., expressing a desire for “5 9s” availability), then have the AMW run-time infrastructure automatically identify and maintain a running configuration of the software system to achieve the high-level objectives.
- *Simplifying evolution of legacy software to leverage AMW.* The cross-cutting nature of high availability concerns makes it difficult to add high availability to an existing software system. Similarly, evolving a highly available software system to use a new high availability infrastructure such as AMW is equally challenging. Source code transformation technology (e.g., as in [2,35]) and aspect-oriented programming [18] can address this evolution through significant automation.
- *Simplifying integration of third-party software.* To preserve the integrity of third-party software that is integrated into a larger system, minimal changes (ideally, none) are made to the third-party software to preserve its integrity, in spite of the fact that overall system availability is usually compromised. Where source code is available, program transformation and aspect-oriented programming techniques are viable and promising alternatives to improve availability of third-party software.
- *Incorporating AMW tools into an integrated development environment (IDE).* IDEs such as IBM’s Eclipse [11] offer developer productivity improvements by providing an integrated environment in which developers can design their systems and develop and test their code. Incorporating AMW model specification and code generation tools as plug-ins into this framework will enhance the ability of this valuable infrastructure for building highly available systems.
- *Support for additional programming languages.* Support for additional programming languages, such as Java or C#, enables widespread use of AMW in different application domains, including those where systems are developed using multiple programming languages.
- *Integrating AMW with modeling tools.* Architectural modeling tools, such as Kansas State University’s Cadena [15,7], enable developers to build formally verifiable models of the system architecture. Vanderbilt University’s system execution modeling (SEM) research [32] allows simulation of component behavior before the component implementation is complete. Integrating AMW with these modeling tools permits examination of high availability system characteristics at the model level without requiring execution in an operational environment.
- *Support for additional fault tolerance techniques.* Additional fault detection and recovery techniques, such as customizable fault detectors as supported in ARMOR [17] and compiler-based techniques [29], will enrich AMW capabilities.
- *Contributing to standards.* We are looking into standardization of AMW’s rich set of high availability services and novel model-based means to provide these services in forums such as the Service Availability Forum (SAF) [31].

8 Summary and Conclusions

Developing highly available distributed software systems is a challenging task. Four issues contribute significantly to this challenge: system architecture, the need to seamlessly integrate fault detection and recovery services with initialization and run-time upgrade services, the high level of fault tolerance expertise needed by system developers to build highly available systems, and the need to thoroughly test a system to ensure it meets fault tolerance requirements. While high availability middleware solutions help in building highly available software systems by avoiding the need to redevelop common high availability services, they do not address all of these issues.

To address the above issues, we developed the Aurora Management Workbench (AMW), which takes a model-centric approach to building highly available software systems. AMW consists of a novel modeling language for specifying characteristics of software systems important for high availability, novel code generators that generate code from the model, and a novel high availability middleware, including run-time entities, that use the model to coordinate initialization, fault detection and recovery, and run-time upgrade procedures. Architectural specifications are significant and critical elements of the model. Fault injection hooks are provided in both the AMW run-time infrastructure software and the generated code that enable fine-grained testing of high availability behavior “out of the box”.

With AMW, developers capture a system’s architecture and its high availability needs using the model language elements. AMW’s code generators use the model specification to generate much of the code required to integrate a software system with AMW’s high availability middleware. Run-time infrastructure components also use the model specification to realize desired operational fault tolerance behavior. While developers must still write some code to complete the integration of their software with AMW, the effort required to do so is typically minimal and primarily consists of manipulating application-specific state information in response to startup, shutdown and failure recovery events.

AMW was evaluated in an experimental setting. In this setting, a simple highly available client-server application was developed in only a few hours using AMW and required less than twenty lines of hand-written code. The same application took approximately two weeks to develop using a commercial high availability middleware and required more than one thousand lines of hand-written code.

AMW is used in large, deployed telecommunications products consisting of several million lines of code comprising a few thousand software components and many tens of thousands of communication links executing on tens of processors. This clearly demonstrates that a model-centric approach to HA software development is effective in building highly available large and complex distributed software systems.

Acknowledgements. The success of AMW in practice is due to the heroic efforts of the team that took a research prototype of the ideas captured in this paper and transformed it into a hardened asset suitable for use in real products. We thank this team tremendously for their efforts.

References

1. Bass, L., Clements, P., Kazman, R.: *Software Architecture in Practice*, 2nd edn. SEI Series in Software Engineering (2003)
2. Baxter, I., Pidgeon, C., Mehlich, M.: DMS: Program Transformations for Practical Scalable Software Evolution. In: *Proceedings of the 2004 International Conference on Software Engineering*, Scotland, UK (May 2004)
3. Bianchini, R., Buskens, R.: Implementation of On-Line Distributed System-Level Diagnosis Theory. *IEEE Transactions on Computers* 41(5) (1992)
4. Birman, K.: ISIS: A System for Fault Tolerance in Distributed Systems. Technical Report TR 86-744, Department of Computer Science, Cornell University, Ithaca, NY (April 1986)
5. Borland C++ Builder, <http://www.borland.com/us/products/cbuilder/index.html>
6. Buskens, R., Sabnani, K.: *Towards Rapid Development of Configurable, Reliable, and Scalable Wireless Applications*, PIMRC (2000)
7. Childs, A., Greenwald, J., et al.: CALM and Cadena: Metamodeling for Component-Based Product Line Development. *IEEE Computer* 39(2) (2006)
8. Chung, P., Huang, Y., Yajnik, S., et al.: DOORS – Providing Fault Tolerance of CORBA Objects. In: *IFIP International Conference On Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, The Lake District, England (1988)
9. Clovis Software, <http://www.clovis.com>
10. Cukier, M., Ren, J., Sabnis, C., et al.: AQUA: An Adaptive Architecture that Provides Dependable Distributed Objects. In: *Proceeding of the IEEE 17th Symposium on Reliable Distributed Systems (SRDS-17)*, West Lafayette, IN (1998)
11. Eclipse Integrated Development Environment (IDE), <http://www.eclipse.org>
12. Enea, <http://www.enea.com>
13. Ericsson, Eternal Systems, et al.: FT-CORBA. Joint Revised Submission. OMG TC Document orbos/99-12-19, OMG, Framingham, MA (1999)
14. GoAhead Software, <http://www.goahead.com>
15. Hatcliff, J., Deng, W., et al.: Cadena: An Integrated Development, Analysis, and Verification Environment for Component-Based Systems. In: *Proceedings of the 2003 International Conference on Software Engineering*, Portland, Oregon (2003)
16. IBM Rational Software Architect, www.ibm.com/software/awdtools/architect/swarchitect/index.html
17. Iyer, R.K., Kalbarczyk, Z., Whisnant, K., Bagchi, S.: A Flexible Software Architecture for High Availability Computing. In: *Proceedings of the High-Assurance Systems Engineering Symposium*, Washington D.C. (1998)
18. Kiczales, G., Lamping, J., et al.: Aspect-Oriented Programming. In: *Proceedings of the 1997 European Conference on Object-Oriented Programming*, Jyväskylä, Finland (1997)
19. Ledeczi, A., Maroti, M., et al.: The Generic Modeling Environment. Workshop on Intelligent Signal Processing, Budapest, Hungary (May 2001)
20. Majzik, I., Huszcl, G.: Towards Dependability Modeling of FT-CORBA Architectures. In: *Proceedings of the 4th European Dependable Computing Conference* (2002)
21. Martin, J.: *Rapid Application Development*, Macmillan Publishing Co. Inc. (1991)
22. Microsoft Visual Studio, msdn.microsoft.com/vstudio
23. Modarressi, A.R., Skoog, R.A.: Signaling System No. 7: A Tutorial. *IEEE Communications Magazine* 28(7) (1990)
24. Moser, L.E., Melliar-Smith, P.M., et al.: The Eternal System: An Architecture for Enterprise. In: *Proceeding of the 3rd International Enterprise Distributed Object Computing*, Mannheim, Germany (1999)

25. Ren, Y., Buskens, R., Gonzalez, O.: Dependable Initialization of Large-Scale Distributed Software. In: Proceedings of the 2004 International Conference on Dependable Systems and Networks (2004)
26. van Renesse, R., Maffeis, S., Birman, K.: Horus: A Flexible Group Communication System. Communications of the ACM 39(4) (1986)
27. Rodrigues, G., Rosenblum, D., Emmerich, W.: A Model Driven Approach for Software Systems Reliability. In: Proceedings of the 26th International Conference on Software Engineering (2004)
28. Royce, W.: Managing the Development of Large Software Systems. In: Proceedings of IEEE WESCON, vol. 26(8) (1970)
29. Roy-Chowdhury, A.: Manual and Compiler Assisted Methods for Generating Fault-Tolerant Parallel Programs, PhD thesis, University of Illinois at Urbana-Champaign (1996)
30. Schmidt, D.: Model-Driven Engineering. IEEE Computer 39(2) (2006)
31. Service Availability Forum, <http://www.saforum.org>
32. Slaby, J.M., Baker, S., Hill, J., Schmidt, D.C.: Applying System Execution Modeling Tools to Evaluate Enterprise Distributed Real-Time and Embedded System QoS. In: Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA '06), IEEE Computer Society Press, Los Alamitos (2006)
33. Telelogic Rhapsody, www.telelogic.com/Products, www.ilogix.com/sublevel.aspx?id=53
34. Unified Modeling Language, <http://www.uml.org>
35. Waddington, D.G., Yao, B.: High Fidelity C++ Code Transformation. In: Proceedings of the 5th Workshop on Language Descriptions, Tools and Applications (LDTA) (April 2005)
36. Zorpette, G.: Keeping the Phone Lines Open. IEEE Spectrum 26(6) (1989)

An Outline of an Architecture-Based Method for Optimizing Dependability Attributes of Software-Intensive Systems

Lars Grunske¹, Peter Lindsay¹, Egor Bondarev²,
Yiannis Papadopoulos³, and David Parker³

¹ School of Information Technology and Electrical Engineering
ARC Centre for Complex Systems,
The University of Queensland,
4072 Brisbane (St.Lucia), Australia
grunske,pal@itee.uq.edu.au

² Eindhoven University of Technology, System Architectures and Networking Group
5600 MB, Eindhoven, The Netherlands
e.bondarev@tue.nl

³ Department of Computer Science, University of Hull
Hull HU6 7RX, U.K.

y.i.papadopoulos@hull.ac.uk, d.j.parker@dcs.hull.ac.uk

Abstract. Dependability requirements such as safety and availability often conflict with one another making the development of dependable systems challenging. It is not always possible to design a system that fulfils all of its dependability requirements and consequently, it is necessary to identify conflicts early in the development process and to optimize the architectural design with regard to dependability and cost. This paper first provides an overview of fifteen different approaches to optimizing system designs at an architectural level. Then an abstract method is proposed that synthesises the main points of the different approaches to yield a generic approach that could be applied across a wide variety of different system attributes.

1 Introduction

Complex mission- and safety-critical systems play a vital role in many application domains, including air traffic control, railway signalling, and healthcare. The design and development of such systems is challenging because systems and software engineers need to deal with a large number of dependability requirements, such as safety, availability and reliability, while keeping life-cycle costs as low as practicable. We define *dependability* of a software-intensive system as its “ability to deliver service that can justifiably be trusted” [1].

As Avizienis et al. [1] point out, for all but the simplest systems “dependability is typically interpreted in a relative, probabilistic sense: . . . due to the unavoidable presence or absence of faults, systems are never totally available, reliable, safe, or secure”. Dependability requirements often conflict with one another [2], making trade-offs necessary. For example, when faults are detected in a system, a shutdown may be required

to prevent escalation of hazardous conditions; the decision of whether to trigger a shut-down involves a trade-off between *safety* (absence of catastrophic consequences on the user and the environment) and *availability* of the system (the system's "readiness for correct service" [1]).

The architectural design phase is critical in system development: decisions made in this phase have a significant impact on the cost and quality of the final system [3,4]. System dependability is a particularly important cost driver, with design decisions regarding the levels of performance, integrity and fault tolerance required in a system having a significant impact on its development, deployment and maintenance costs. Changes to the architectural design late in its development are typically very costly because of the amount of rework required, and it can lead to long delays. For these reasons it is important that architectural design decisions are made carefully, and it is critically important that system architects are provided with methodologies and techniques that enable potential designs to be evaluated against system dependability requirements.

A number of techniques are now in widespread use for evaluating different aspects of system dependability at the level of architectural design models [3]. For example, Fault Tree Analysis [5] is widely used in industry to quantitatively predict the likelihood of hazardous system failures from assumptions about component failure rates, and thus to assess the safety of a system design. Markov Analysis [6] is often used to predict system availability and *reliability* (the system's "continuity of correct service" [1]) based on assumptions about component failure rates. Queuing Networks [7] are often used to predict system timing performance based on assumptions about input rates and component processing time. Of course, prediction of component performance is still a matter of research for many technologies with prediction of the nature and number of faults in software components being particularly troublesome. Nevertheless, despite these limitations, software and systems engineers do manage to develop systems with reasonably predictable dependability attributes. The challenge is how to combine these different techniques into a single evaluation framework that enables informed trade-offs between different system attributes, including dependability and cost. More specifically, to what extent is it possible to automate the analysis, and to derive architectural designs that are optimal with respect to the different dependability attributes that are under consideration (remembering that a single optimal solution may not be possible).

This paper reviews 15 different methods for optimising various aspects including dependability, cost and performance. The focus is on comparing the methods from five different viewpoints: the optimisation objectives pursued by each method; the heuristic and meta-heuristic techniques used for searching in large design spaces; the architectural models used for evaluation of dependability attributes; the techniques used for multi-objective optimisation; and the intended applications. We contend that enough commonalities exist among the different approaches to enable us to abstract and define a generic method for architecture-based dependability optimisation. From this, specific approaches can be derived through a tailoring process to suit particular optimisation problems and applications. The development of a conceptual framework that integrates the different approaches into a single generic approach is this paper's main contribution to research. The derived generic optimisation method and its tailoring process

summarise and capture, we hope, some of the useful common experience that has been generated by research in this field during the last 20 years.

The rest of this paper is organized as follows: Section 2 contains the review of dependability optimisation approaches. The common elements of a generic method are described in Section 3 while the method itself is specified in section 4. Section 5, describes the tailoring process for the abstract method and Section 6 discusses limitation of the current approach. Finally, in section 7, we draw conclusions and outlines the future prospects of this work.

2 Overview of Current Dependability Optimization Approaches

There is a long established body of work on optimising hardware architectures in terms of reliability and cost using exact mathematical methods such as dynamic programming [8,9] and integer programming [10]. The aim of this work has been the optimisation of architectures via automatic selection of components and appropriate fault tolerant schemes to meet reliability and cost requirements. Whilst this work has shown that exact techniques can calculate guaranteed optimal solutions, they often do so by imposing constraints on the design that are undesirable in real systems. For example, they typically use multiple identical copies of a component for redundancy where in practice however it would be better to use diverse components or technologies, to protect against common-mode failures. Similarly, exact methods typically make overly simplistic assumptions about the nature of architectures, such as that they consist simply of components connected in parallel or in series.

The rest of this section outlines 15 approaches from the literature over the last two decades concerned with trade-off analysis and/or optimisation on architectural design models. Although much of this work has focused on reliability and cost optimisation, some of the approaches concern other criteria related to dependability and performance, including safety, resource usage and timing performance.

Approach 1. To the best of our knowledge, the first use of a meta-heuristic, in this case a Genetic Algorithm (GA), for reliability optimisation is described by Coit et al in [11]. The GA was used to maximise reliability, given cost and weight constraints, and minimise cost, given reliability and weight constraints, on 33 variations [9] of the *Redundancy Allocation Problem (RAP)* as defined by Fyffe et al [8]: that is, the problem of deciding on the configuration and degree of replication of components in a system. A penalty function based on degree of infeasibility (i.e., the number of constraints not satisfied) and search time ensured that the GA was able to proceed through the infeasible region, which was found to be the most efficient route to the optimum solution. To allow for the stochastic nature of the GA, the experiments described in the paper were run 10 times for each problem with the best solution being kept. Components could be mixed flexibly within functional unit subsystems: this feature enabled the GA to find solutions with higher reliability in 27 out of 33 problems than when exact methods such as those in [8,9] were used. In the remaining problems, 4 of the solutions were identical and in 2 the solutions gave a slightly lower reliability.

Approach 2. In his PhD [12], Nicholson describes an approach that addresses reliability but also focuses on the real-time aspects of the architecture. The approach involves two phases. The goal of the first phase is to find an optimal architecture topology which includes decisions about appropriate redundancies. The second phase aims at finding optimal deployment architectures, which specify how software components are mapped to real-time tasks and how these real-time tasks are then mapped to hardware nodes. Nicholson argues that this distinction is necessary since the two phases involve optimization problems with different complexities. As a result, Genetic Algorithms are applied in the first phase and Simulated Annealing is used as a search heuristic in the second phase. The optimisation parameters used include worst case reaction times (WCRT), reliability metrics (e.g. mean time to failure -MTTF), resource usage metrics, and production costs. The overall approach is implemented in the tools X-Alloc and X-Topmeter and is used with success on the case study of a computer assisted braking system.

Approach 3. Ant Colony Optimisation (ACO) is a meta-heuristic optimisation method that was inspired by the action of real ant colonies. Liang and Smith [13] present an approach to solving the Redundancy Allocation Problem (see Approach 1 above) using a specific ant colony system, where each ‘ant’ corresponds to a particular system configuration. Mimicking the use of pheromones to establish a positive feedback loop, a particular configuration path is chosen with greater probability as the number of times it is chosen increases. This leads to the quick identification of solutions that satisfy the criteria. An adaptive penalty function is used to accommodate the constraints and determine the amount of pheromone on the trail. Ant mutation is used to discourage local optima convergence and good paths are enhanced by creating duplicates of the current best solution. One shortcoming of the Ant system is that it may be sensitive to its parameters and the method with which trail updates are carried out. When compared to the Coit and Smith GA [11] it produced good solutions with low variation over the multiple runs, however the GA produced ‘better’ solutions with higher reliability results.

Approach 4. Thiele et al. [14,15] present an approach for design space exploration and architecture optimisation for Network Processor Architectures. The approach is based on models for packet processing tasks, a description of workload streams entering a system, and a specification of a feasible space of architectures including computation and communication resources. For each architectural alternative, the model data is statically analysed by Real-Time Calculus [14] in order to evaluate performance. The design alternatives are the subject for the multi-objective optimization performed by the SPEA2 framework [15]. This framework enables architecture optimisation using genetic algorithms.

Approach 5. Palermo et al. [16] introduce a design space exploration framework for parameterized embedded System-on-Chip (SoC) architectures. The framework targets the power-consumption/response-time optimisation problem for embedded devices. It uses the following three architecture generation and evaluation algorithms: Random Search Pareto (RSP), Pareto Simulated Annealing (PSA) and Pareto Reactive Tabu Search (PRTS). The authors claim that the use of these algorithms reduces search efforts by three orders of magnitude. The assessment of the relevant quality attributes is provided

by simulation and dynamic profiling of the target systems. The framework is illustrated by the case study of a GSM encoding application.

Approach 6. Another technique for the design space exploration of embedded System on ChiP (SoC) systems has been developed by Palesi et al. [17]. This technique focuses on the architecture optimisation with respect to the power/latency trade-offs. For architecture generation and evaluation, it reuses the genetic algorithms implemented by the SPEA2 framework [15]. The instrumentation for architecture mutations is based on parameterizations of the hardware IP blocks.

Approach 7. TSRAP [18] is a system that uses a Tabu Search algorithm on the Fyffe et al [8] formulations of the Redundancy Allocation Problem (RAP), maximising reliability whilst keeping cost and weight within constraints. It is argued that Tabu search suits the inherent neighbourhood structure of the RAP and its deterministic action reduces its sensitivity to search parameters such as the initial solution. TSRAP starts from a random feasible configuration and evaluates every possible single point change to that configuration. The evaluation uses a penalty function that modifies the optimisation goal with a penalty based on the degree of constraint violation allowing the search to progress through the infeasible region. The best solution of the current iteration is chosen, provided it is not Tabu, and the process is repeated. In order to prevent the search from settling in local optima each move is recorded in a Tabu list and may not be used for future moves whilst it remains on the list. This continues until the maximum number of iterations, without an improvement to the best feasible solution found, is reached. Working on the same example problem, TSRAP is found to be more efficient than the GA used in Coit and Smith [11] because each move makes only a single point change to one subsystem which can be re-evaluated independently from the rest of the system. In the example problem this leads to approximately 14 times fewer evaluations. TSRAP also found superior solutions with greater consistency and less variability.

Approach 8. Pareto frontier based algorithms may allow a true multi-objective search but often pursue a very large number of solutions which may be unmanageable. Kulturel-Konak et al. [19] seek to overcome this by using a Multinomial Tabu Search to identify the Pareto frontier before passing this through a pruning method that reduces the set using user specified guidelines. The proposed Multinomial Tabu Search operates in similar way to the Tabu Search described in [18] except that at each iteration one of the multiple objectives is randomly selected to be the optimisation goal. Solutions are compared, and if applicable, added to a list of non dominated solutions. In order to improve diversity in this list, when a threshold for iterations without updating the list is reached, a non-dominated solution is selected as the current move and the tabu list is emptied before continuing. Once the Pareto frontier is identified a pruning method based around a Monte Carlo simulation is used to significantly reduce the frontier according to priorities set by the user, thus providing a manageable set of trade-off solutions for the designer.

Approach 9. A generic framework for architecture optimisation has been proposed by Künzli et al. in [20]. This modular design space exploration framework allows the use of various optimisation techniques depending on the problem domain. The framework provides the following multi-objective optimisation techniques: black-box optimisation, randomized search and problem-dependent search. The multi-objective evaluation

module of the framework deploys the concept of Pareto-dominance. The framework is based on the PISA (platform and programming language independent interface for search algorithms) protocol that specifies a problem-independent interface between the search/ selection strategies on one hand and the problem domain-specific estimation and variation operators on the other. The framework is illustrated by a simple application example where the design is optimized for the most efficient cache architecture.

Approach 10. Papadopoulos and Grante [21] describe an approach to multi-objective optimization of safety critical systems with application on automotive designs. The goal is to find optimal tradeoffs among safety and reliability (treated as separate objectives) and cost in the design of such systems. A GA which promotes population diversity [22] is used to progressively improve a Pareto set of non-dominated solutions that represent different tradeoffs among the parameters of the optimisation. The approach departs from earlier work in that the safety and reliability model (i.e. a set of system fault trees) is automatically synthesised from an engineering model that is augmented with information about component failures. It also moves beyond the classical “success-failure” model introducing a failure scheme in which components can exhibit more than one failure mode which include not only the loss but also the commission of functions as well as value and timing failures.

Approach 11. Fredriksson et al. [23] describe a single-objective optimization approach that is targeted towards hard real-time systems. The goal is to find an optimal allocation from components to tasks. Genetic algorithms are chosen as the optimization strategy, where each gene represents a component and contains a reference to the task it is assigned. Each allocation produced by the GA is evaluated by a fitness function, which sums up memory consumption on the stack and CPU overhead. Both parameters are determined using a basic scheduling analysis algorithm based on parameters such as worst-case execution times or required stack usage that are attached to each component in the system.

Approach 12. Grunske [24] describes a method that focuses on improvement of reliability within given cost and weight constraints. The method uses a multi-objective optimisation strategy that is implemented by a simple evolutionary algorithm which progressively improves a set of Pareto optimal solutions. Reliability is evaluated using Reliability Block Diagrams, which are generated separately for each function delivered by the system, based on the components that are needed to perform this function. The approach is illustrated on a case study of a satellite control system and two of its main functions.

Approach 13. A process for architecture optimisation of real-time component-based software systems has been proposed by Bondarev et al. [25]. The process aims at design-time prediction of performance, and resolving performance trade-offs. Evaluation of properties is based on (a) models of individual components, (b) synthesis of the models into an executable system model, and (c) simulation of the generated system model. Pareto principles are employed to identify un-dominated architectures. Extensive visualisation support helps the designer to identify bottlenecks in the un-dominated architectures and further optimise them. Architecture variations are created

Method & Reference	Application Area/Case Study	Optimization Approach/Strategy	Dependability Evaluation Models	Dependability Improving Measures
Coit et al. [11]	Reliability Design, Redundancy Allocation Problem	Genetic algorithm, Single optimisation goal with constraints	Reliability Block Diagrams (RBD), Simple Weight and Cost Estimations	Component substitution & replication
Nicholson [12]	Safety-critical, Real-time Systems/Computer Assisted Braking System	Multi-Objective Optimisation/Genetic Algorithms and Simulated Annealing	Reliability Block Diagrams (RBD), Schedulability Analysis	N-Version Programming, Redundancy, Task Allocation
Liang and Smith [13]	Reliability Design, Redundancy Allocation Problem	Ant system, Single optimisation goal with constraints	Reliability Block Diagrams (RBD), Simple Weight and Cost Estimations	Component substitution & replication
Thiele et al. [14]	Network Processor Architectures	Multi-Objective Optimisation/Genetic Algorithms	Real-Time Calculus (RTC): Analytical approach employing workload and resource curves	Hardware Topology, SW/HW Mapping
Palermo et al. [16]	Embedded systems on SoC platforms/GSM encoding application	RSP, PRTS and PSA Algorithms/ power-vs-delay trade-off	Simulation and dynamic profiling	Hardware parameters: number and size of cache, ALU, multiplier and memory blocks
Palesi et al. [17]	Embedded systems on SoC platforms	Genetic Algorithms of SPEA2 framework/ power-vs-delay trade-off	Simulation	Hardware parameterizations
Kulturel-Konak et al. [18]	Reliability Design, Redundancy Allocation Problem	Tabu Search, Single optimisation goal with constraints	Reliability Block Diagrams (RBD), Simple Weight and Cost	Component substitution & replication
Kulturel-Konak et al. [19]	Reliability Design, Redundancy Allocation Problem	Multi-Objective Optimisation, Multinomial Tabu search with Pareto pruning	Reliability Block Diagrams (RBD), Simple Weight and Cost	Component substitution & replication
Künzli et al. [20]	Generic	Multi-Objective Optimisation/Multiple randomized-search algorithms	Performance run-time profiling (used in case-study)	Task allocations, Hardware Topology, SW/HW Mapping, Parameters
Papadopoulos and Grante [21]	Safety Critical Systems / Automotive design	Multi-Objective Optimisation/Genetic Algorithms	Automatically constructed Fault Trees, Simple Weight and Cost Estimations	Component substitution & replication
Fredriksson et al. [23]	Real-time Systems/ Generic Case Study	Single-Objective Optimisation/Genetic Algorithms	Schedulability Analysis, Analysis of the Memory Consumption	Component to Task Allocation
Grunske [24]	Mission-Critical Embedded Systems/Satellite Control Application	Multi-Objective Optimisation/Evolutionary Algorithms	Reliability Block Diagrams (RBD), Simple Weight and Cost Estimations	Component Replication (Two Channel Redundancy, Triple Modular Redundancy)
Bondarev et al. [25]	Real-time component-based software systems/ car radio navigation system	Manual, guided by results of architecture analysis tool	Task synthesis from component model and task simulation	Software components, SW/HW mapping, parameters of hardware blocks and hardware topology
Pimentel et al. [26]	Embedded systems on heterogeneous platforms	Multi-Objective Optimisation	Trace transformations, Kahn Process Networks, Simulation	Task re-allocations, Hardware Parameters and Topology, SW/HW Mapping
Livolsi et al. [27]	Component-Based Systems/ Financial System Case Study	Multi-Objective Optimisation/Evolutionary Algorithms		Hardware Parameters and Topology

Fig. 1. Overview of architecture optimization methods

by the changing of software components, hardware/software mapping and hardware topology. The process is illustrated by a case study on a car radio navigation system.

Approach 14. Pimentel et al. [26] describe a SESAME framework for exploration and optimisation of embedded system architectures at multiple abstraction levels. The framework deploys both analytical and simulation methods for evaluation of quality attributes. Analytical methods are fast and work with high levels of abstraction, while simulation methods are more detailed and require detailed system specifications. The framework focuses on predicting performance-related quality attributes and exploration of heterogeneous multiprocessing systems with respect to these quality attributes. The performance evaluation methods used are the following: Kahn Process Networks, trace transformations and simulation. The SPEA2 framework [15] is used for multi-objective architecture optimisation. The approach is illustrated by a Motion-JPEG encoder case study.

Approach 15. Livolsi et al. [27] propose a guided architecture-based design optimisation technique for component-based systems. The technique enables exploring possible architectures by repeatedly applying evolutions to initial architectures, with the quality attributes of each architecture being evaluated throughout. The found quality attributes provide feedback that guides the designer to the next iteration. An optimisation iteration cycle contains three modules: Corrector, Effector and Evaluator. The Corrector module

takes initial architectures and system requirements as an input and creates generation guidelines for the Effector module. The latter applies evolutionary algorithms to generate a new population of architectural alternatives. The Evaluator module takes the new population, assesses its quality attributes and performs multi-objective evaluation using Pareto-optimal principles. The evaluation results are sent to the Corrector module for the next iteration. This optimisation technique is supported by the ABACUS software toolkit.

Figure 1 provides a summary of our review. The table defines the application area, optimisation strategy, evaluation models and dependability improving measures used for each approach.

3 Basic Elements of a Dependability Optimization Approach

There are three elements that seem to be shared by all dependability optimisation approaches: a set of evaluation models, a set of improvement measures and a set of optimisation strategies. A discussion of these three elements follows:

3.1 Dependability Evaluation/Prediction Based on Architecture Specifications

All of the approaches employ a model of the system architectural design, at some level of abstraction, that enables dependability attributes to be evaluated quantitatively. It is outside the scope of this paper to discuss how well design-time dependability evaluation methods can predict the dependability of the system when it is eventually put into operation: we simply assume the methods are well established and reliable. Many if not all of the methods mentioned here are certainly well established, and have been shown to improve system dependability.

Evaluation methods can be divided into two categories: analytical and simulation-based. Analytical methods (eg [14,26]) apply formal mathematical theories, such as process algebra, state machines, Petri Nets and Queuing Networks to predict values of dependability or more general Quality Attributes (QAs). By contrast, simulation-based approaches (eg [16,17,25]) emulate the behaviour of the designed system by executing its architecture models.

Analytical methods have a number of benefits, such as being able to discover worst- and best-case boundary cases. For trivial systems, they provide results fast, but for complex systems the required computation normally increases exponentially or worse [28], which limits their use in large industrial cases.

Simulation-based techniques may require a considerable amount of computation to simulate system execution, but often the amount of time required grows linearly with the system complexity, which makes them good candidates for evaluation of large systems. Because they step through the execution of the system model, they can provide useful insights into the system architecture, such as task interleaving and synchronisation, and bottlenecks that might arise. However, like system testing, simulation is limited to evaluating particular runs of the system, and so may miss problematic cases. Cost estimation is typically done analytically.

Most of the evaluation methods are constrained to specific domains of software architectures (i.e. component-based, object-oriented, streaming or event-based architectures). Moreover, these methods, with some exceptions, require a specific type of software architecture to be deployed. For instance, the framework presented in [25] works only for ROBOCOP, CORBA and Koala architectures. This happens mainly due to the fact that tuning an approach to a specific type of architecture allows simplifying specification (model) structures, reducing prediction-error rates, and making the approach learning-curve less steep.

Many of the conventional methods work for embedded systems generally, independent of particular application domains. For example, the RTC method [14] has been successfully applied to medical, consumer electronics, automotive and control systems.

It is important to note that the prediction accuracy resulted from the available methods is heavily dependent on the accuracy of data specified in their input models: the phrase “garbage in - garbage out” perfectly holds here. That issue imposes strict requirements for well-established process of model specification and maintenance when using the method.

Examples of evaluation techniques addressing specific quality attributes are the following: reliability evaluation is addressed in [29], [30] and [31]; timing performance is addressed in [7] and [32]; task latencies and resource usage are tackled in [25] and [14]; and safety is dealt with in [33] and [34].

3.2 Dependability Improving Measures

Optimisation methods employ a variety of measures for improving the dependability of an architecture. An example of these measures is the integration of redundancy mechanisms in order to improve reliability [35]. Another quality improving measure is the reassignment of software elements to another hardware platform in order to avoid common cause failures and to reduce the workload of a specific hardware platform. These measures can be seen as transformations that take an architecture specification $ArchSpect_i$ as input and produce a new and hopefully improved version of the architecture specification $ArchSpect_{i+1}$. A transformation may rewrite any part of the initial architecture specification. It may change the deployment view, the structure view or the behaviour view. It may replace certain system components or it may change how system functionality will be allocated to them. To be generally applicable to different systems and different types of architecture specification the transformations must be specified in an abstract way. Since most architecture specifications are represented as graphs, graph transformations are one possible specification formalism of such transformations at an abstract level [36].

In the context of the optimisation of system architectures, transformations that respect system requirements (i.e., preserve system functionality and satisfy associated constraints) and increase specific dependability attributes are especially interesting. Such transformations are similar to refactorings at the code level, hence the use of the name architectural. To ease the selection of a transformation, each architectural refactoring needs in addition to the transformation specification a set of pre-conditions and goals/post-conditions. Pre-conditions describe restrictions in order to ensure the correct application of a transformation rule. Examples of these preconditions include

constraints that disallow the repeated application of the same refactoring to one component or applications that would violate system level design constraints, such as mechanical or economical constraints. Goals describe the desired outcomes of a transformation rule application, e.g., this refactoring will improve the reliability of the system. However, the desired outcome can't be guaranteed, because whether or not the goal is achieved often depends on the context in which the refactoring is applied. As an example the goal of the triple modular redundancy pattern, where three copies of a component are used in conjunction with a two-out-of-three voter, is to reduce the possibility that a failure of a single component has an affect on the correct behaviour of the system. Consequently, this refactoring should be used to increase reliability properties. However, if the reliability of the replicated components is higher than the reliability of the introduced voter component an application has exactly the opposite effect. In this case, the reliability of the overall system would be reduced, since failures of the voter component would decrease the system's reliability.

Some refactorings may improve some dependability attributes but decrease others: for example, modular redundancy can improve reliability but may degrade performance.

Based on a repeated selection and application of architecture transformations a search graph can be created, where nodes describe possible architecture specifications and directed edges between nodes represent the application of a transformation rule. This graph is also known as the design space [27].

3.3 Optimization Strategy/Design Space Exploration

In general, the optimal design of dependable systems is a complex combinatorial problem. Using the terminology introduced in the previous section, the goal of locating optimal *Arch.Spec_i* requires the exploration of a typically large non-continuous design space. Such exploration is prone to premature convergence to local optima when automated search techniques are applied.

The naïve approach to such exploration is to conduct a random search. In this approach, solutions are selected at random from the search space and the best solution yet found is kept. This technique relies on computational speed to potentially obtain solutions quicker than a human designer through brute computational force. Naturally, such an approach has very poor scalability and there is no guarantee of good solutions. A human designer may be able to use experience and accumulated knowledge to infer good design solutions but this is often limited by the time available to try alternative solutions. As systems increase in size so a combinatorial explosion expands the search space to potentially infinite size. The problem of massively increased search space size is one that affects all search techniques and is commonly tackled by introducing a cost function and by putting bounds on maximum cost to be considered. For example a reduction in search space size could be achieved by limiting the maximum number of redundant components allowed in a subsystem. Even with constraints, though, the potential for combinatorial explosion remains.

To tackle that problem a number of heuristics, which include simulated annealing, Tabu search, genetic and ant colony-based algorithms have been applied to the problem. Each of these techniques is stochastic in that whilst they have a random element, the search is guided providing better than random performance. In their simple form,

each of these heuristics can readily tackle single-objective optimisations and through the use of penalty functions can incorporate multiple constraints. However, the use of constraints to manage multi-objectivity presupposes knowledge of where the optimum solution for a given problem lies. Even if this information is available, it can often be tricky to fine-tune the penalty weighting for constraint violations. It is to overcome these shortcomings and provide a true multi-objective search that the concept of Pareto optimality has been worked into heuristic techniques. A Pareto optimal set contains individual solutions that are un-dominated (Solution A dominates solution B if A outperforms B on every dependability attribute of interest) by other solutions in the search space. They represent a selection of trade-offs that can be presented to the designer to ultimately choose the best for a specific application.

Simulated annealing (SA) is a technique that mimics the process in which the crystals in heated metals are formed as they cool. Recent work [37] has included the use of SA in multi-objective optimisations. Individual rather than population based, the algorithm starts from a random initial solution which is then mutated. There are several factors affecting whether the mutated result is kept and these are its dominance relative to the previous position and an acceptance function that is composed of the solutions fitness and the current temperature. If the new solution dominates the old then it is accepted as the current solution. If the new solution is dominated by the current solution then an acceptance function determines whether the move is accepted. Early in the search, the high temperature in the analogy allows a random search through the dominated region. In the final case where neither the current or new solution is dominant the new solution is always accepted. This avoids stagnation and encourages search in the middle region of the Pareto set. SA for multi-objective search has an inherent weakness over population-based algorithms such as genetic algorithms as it is individual based and can only search one area of the search space at a time. Further research in this relatively new area will include parallelising the algorithm and making use of populations.

There is a greater body of work making use of Genetic algorithms (GAs) to solve multi-objective problems [15,38,39]. GAs are based on the evolutionary processes found in nature and unlike SA maintain a population of potential solutions. These are randomly chosen with a selective pressure to choose individuals with a higher fitness to be mutated and combined to form a new generation of potential solutions. The methodology does not guarantee finding an optimum but has been shown to be an effective way of finding at least near-optimal solutions to particular problems. GAs provide strong global search whilst having relatively weak local search characteristics. Care must also be taken to reduce convergence to local optima. [39] provides a good example of an approach to multi-objective optimisation using GAs, where an un-dominated population and a working set of potentially un-dominated solutions are both maintained during evolution. Individuals are selected from the un-dominated set to be altered through mutation and crossover operators. The fitness function encourages parents to be selected from un-crowded regions of the Pareto estimated set to ensure the trade-off set is evenly distributed and no one area is too concentrated. Weaknesses in this approach involve the tuning of parameters such as the maximum size of the non-dominated population. In order to balance performance with effectiveness the size of the

non-dominated population is capped which can potentially lead to good solutions being lost depending on the size and nature of the Pareto set.

Hybrid approaches such as [40] suggest the mixing of the use of GAs and SA to overcome the complementary weaknesses in each. Where SA 'wastes' computation early in the search identifying areas containing good solutions GAs can make use of their population search to quickly identify promising areas. The SA is then well placed to provide a thorough local search of these Areas, a traditional weakness of GAs.

Tabu search (TS) is another search technique that is individual rather than population based. Where SA randomly picks a change to the current individual, TS evaluates all possible single point changes to the current individual and selects the best move from these. A list of Tabu moves encourages the search of new areas and discourages the reversal of moves by temporarily preventing the selection of previously made moves. A noted weakness of TS is a tendency to search only a small proportion of the entire search space unless specific diversification measures are taken. [19] uses a modified Tabu search to identify the complete (or large) Pareto frontier. They note that, whilst the Pareto set offers a selection of trade-offs, the sheer number of individuals in that set, potentially infinite, makes choosing solutions from within that set unmanageable. To overcome this they use a Monte Carlo simulation-based pruning algorithm to significantly reduce the size of the Pareto set whilst maintaining the basic properties of the frontier.

It seems clear that despite progress in various areas, none of the techniques offers an overwhelming advantage in performing multi-objective search and the techniques that use combinations of methods to overcome the weaknesses of individual algorithms are desirable.

4 The Abstract Method at a Glance

This subsection describes an abstract method for multi-objective design space optimisation of software-intensive systems. This method has been synthesized from the above-mentioned techniques proposed by industry and academia and specifies the process of designing and optimising architectures with contradicting requirements on the multiple quality attributes (QA).

Figure 2 depicts the abstract method in terms of an iterative design workflow. The workflow is represented by logical blocks (actors and data types) and arrows (actions and data-relations). The workflow diagram has two constraints: (a) only an actor can initiate a certain action and, (b) an actor cannot be a source of a data-relation. The input to the workflow is a set of functional requirements and constraints. In the design space identification phase, the Architect creates an initial population of architectures based on the system requirements and his/her own experience and intuition. The population of architectures forms a *solution space* [27]. Each of the architectures may be characterized by a set of quality attributes (examples of these in different dependability domains are given above). To predict values of the quality attributes, the architecture becomes input to a specific *QA Analyser* tool. This QA Analyser determines (with some level of accuracy) the values of the relevant quality attributes that form a so-called *quality space* [27]. The available techniques that can be used for such QA Analyser, are outlined in subsection 3.1.

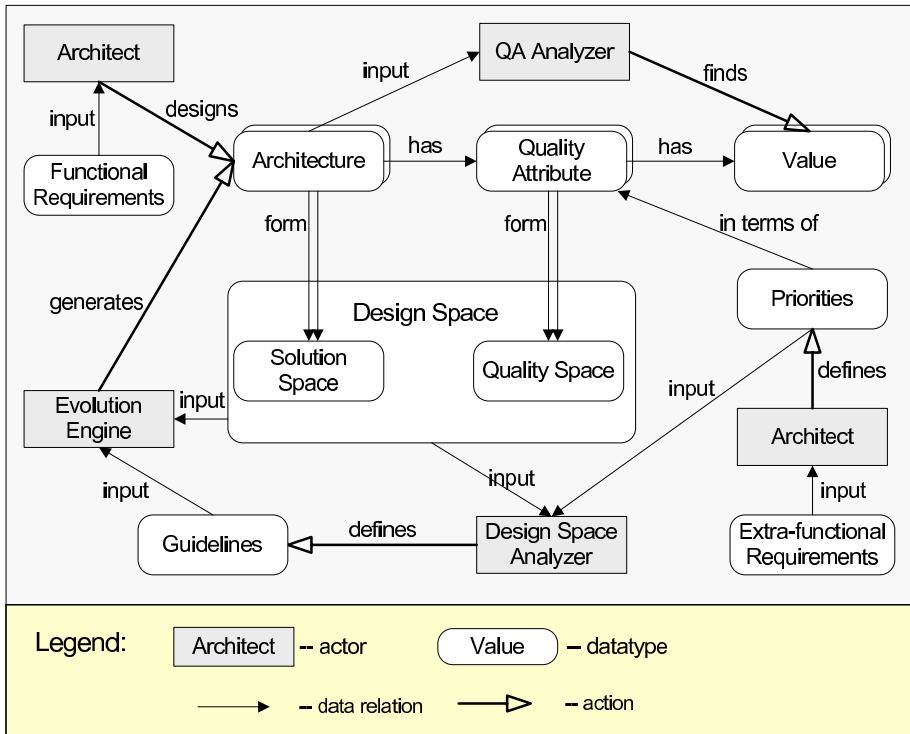


Fig. 2. An abstract method for multi-objective architecture optimisation of software-intensive systems

The solution space and quality space form the *design space* of a particular iteration. The design space can be represented by a multi dimensional diagram locating architectures along QA-related axes according to their predicted QA values.

The next workflow iteration involves extending the architecture population. At this phase, the architect defines *priorities* between multiple quality attributes, based on the dependability requirements. Normally, one dependability requirement addresses one quality attribute. In the case of conflicting requirements, the architect consults project stakeholders to identify the right priorities. As an example, QA priorities can be specified as weighted cost functions. The architect sets the priorities at the beginning of the optimisation process and may tune the priority values during next iterations.

A more advanced way to prioritize contradicting quality attributes is to employ *utility functions* [41]. In particular, utility functions allow dealing with the situations when a value of a quality attribute comes close to the required value, but still the corresponding requirement cannot be met. If the utility function for this quality attribute is set to be low, the architecture can be accepted for further consideration.

These priorities (utility functions), together with the current design space (population of architectures including its quality attributes) are the input to a *Design Space Analyser*

tool. This tool plays a crucial role in the whole architecture optimisation process. The main functions of the Design Space Analyser are as follows:

1. **Evaluation of the Current Design Space and Finding the Set of Non-dominated Architectures.** The evaluation process can be based either on the Pareto-frontiers methods [42], or on the various cost function approaches [43]. In case of the Pareto-frontiers methods, the evaluation identifies the set of non-dominated architectures. If a cost function (e.g. weighted sums or products) were used, the result would be one specific order that reflects the fulfilment of the different optimisation objectives.
2. **Identification of Weak Points in the Design of the Non-dominated Architectures.** This step requires some knowledge on how certain design decisions influence values of quality attributes. The initial data for extracting such knowledge can be obtained from the QA Analyser output (e.g., task behaviour timelines or the hardware resource load imposed by tasks). Based on this input, the Design Space Analyser tool can identify weak points or bottlenecks in the design of non-dominated architectures. An example of this function follows. Imagine, a system has strict requirements on the response time of a certain task. The QA Analyser tool finds/predicts the response time value by simulating the execution of the architecture. The Design Space Analyser acquires the simulation timeline of the task and analyses the bottlenecks in the system that led to the response time increase. Examples of such bottlenecks are: low buffer capacity or network bandwidth, high processor load by other tasks, task blocking and deadlock.
3. **Identification of Guidelines for Generation of Next Architecture Populations.** Having the data about weak points and bottlenecks in the architecture designs and knowing how each particular bottleneck influences the architecture dependability, the Design Space Analyser creates *guidelines* for generating of a new set of architectures. The guidelines may specify: (a) a generation algorithm for the next iteration, (b) variable elements in the architecture (e.g. measures to improve quality), (c) value diapasons for these variable elements. For instance, the guidelines may specify for a system that processor frequency and buffer capacity could be reduced up to 50 per cent of their current values without any consequences for high-priority QAs. At the same time, the guidelines may point out that the system bus bandwidth causes signal delays and should be increased.

The guidelines that are generated by the Design Space Analyser, together with the available design space, form an input for an *Evolution Engine*. The Evolution Engine is responsible for generation of new architecture populations. Possible working principles for the Evolution Engine are described in subsection 3.3.

The newly generated architecture population is sent to the QA Analyser to determine their quality attribute values. Once the QA values are found, the architect may take a decision to stop the architecture optimisation process, or to continue with next iteration. The criteria for the stop-decision are the following: (a) convergence point achieved - no enhancement seen in comparison to previous iterations; (b) requirements satisfaction - all relevant requirements are met; and (c) no time available for next iterations.

The one important feature of this generalized iterative method is that different techniques for QA analysis and architecture generation can be applied during the same process. We call this feature - *cascaded optimisation*. Cascaded optimisation allows

varying the analysis and architecture generation techniques depending on the current situation. For instance, once the Design Space Analyser notices that local search provided by Simulated Annealing algorithm comes close to local optima, then it generates and sends a guideline to the Evolution Engine to change from local to global search algorithms (from Simulated Annealing to Genetic Algorithm) in order to explore the wider space.

5 Tailoring the Abstract Method

The abstract method for dependability optimisation, as presented in the previous section, has been extracted from several existing approaches and consequently there are already several successful instantiations of the method. However, one question remains: how to apply the abstract method in a new application area within a new context. In this section we provide some guideline for tailoring the abstract method towards a specific problem.

To tailor the abstract method, the system architect must select, for each of the basic elements, an appropriate representation that fits their needs. The first choice the system architect must make is to identify a set of dependability evaluation methods for the QA Analyser, in order to determine the quality of an architecture specification. This selection must be based on the dependability requirements that are of interest for the system stakeholders. Once the set of relevant dependability requirements have been identified, an appropriate dependability-attribute specific evaluation method must be chosen for each of the requirements. This selection depends on the required accuracy of the prediction, the existing information about the dependability attributes of the architectural elements, and the budgeted evaluation time. Additionally, most dependability evaluation methods are based on specific assumptions that must hold in order to obtain correct evaluation results. Moreover, some evaluation methods only work in one specific architecture style, e.g. the pipes-and-filters style. As a result, the selection of an appropriate dependability evaluation method is a difficult task, where no generic recipe can be given. However, to select an appropriate dependability evaluation method, we advise consulting the literature surveys that compare different evaluation methods in specific dependability domains. Examples of such surveys are [7,32] for performance, [29] for reliability and [33] for safety.

The second element of the abstract dependability evaluation method, i.e. the dependability improving measures that are used in the design space analyser, must be chosen in accordance with the required dependability attributes. Currently, several pattern catalogues have been developed, that focus on one specific dependability domain. As an example, Grunske [44] describes patterns to improve the safety of a system and Saridakis [45] focuses on improving the fault tolerance of an architectural design. These pattern catalogues can guide the system architect to select relevant dependability improvement measures for the specific problem. Additionally, the selection and application of these measures/patterns should be guided by an initial analysis of the weaknesses of the architecture specification. As an example to improve the reliability of a system a sensitivity analysis [46] that identifies the components with the most influence on the system reliability could be used. For safety, single points of failures can be identified by a Minimal Cutset Analysis [33].

The third element, the Optimization Strategy/Design Space Exploration Technique, is the hardest element to select. This is because the selection should be made based on the solution landscape, which is often not known prior the optimisation. Despite this problem, different optimisation strategies have different performance characteristics. Furthermore, these characteristics depend on strategy-specific parameters, like the number of elements in each generation of an evolutionary algorithm. The survey of Ehrgott and Gandibleux [47] is recommended, to help select the most appropriate optimisation strategy and its parameters. Additionally, a general introduction to a variety of optimisation strategies and heuristics is given in Reeves' book on modern heuristics [48].

To have a deeper understanding in the tailoring process of the abstract method, we will consider two examples. The first example is the satellite control system as used in Approach 12 [24]. The problem in this case is to improve the availability of the system, while keeping the cost and the weight of the system as low as possible. Only one quality improving measure, namely introducing redundancy, was chosen. Restricting the number of quality improving transformations also restricts the design space. Consequently, the result of the optimisation could be improved by selecting other transformations, e.g. transformations that detect and mask faults [45]. As the optimization strategy, evolutionary algorithms are used. Choosing this option has been proven successful, since the evolutionary algorithm was able to find the Pareto optimal solution in the limited design space after a few iterations. However, this does not imply that the selection would also be appropriate for a complex design space. To evaluate the candidate architectures in the evolutionary algorithm, Reliability Block Diagrams (RBD) have been selected. RBDs assume the components fail independently and that a component can only be in two states: either the component functions as specified, or it has failed. Both assumptions lead to an oversimplified evaluation model and consequently a more sophisticated evaluation model like Component Fault Trees [49] or Failure Propagation Tables [34,50] would be more appropriate.

The second example for the tailoring process of the abstract method is the design case of a JPEG decoder presented in [51]. The goal was to increase system performance and robustness while keeping the system cost at the lowest possible rate. Simulations of architecture specification models were used as the dependability evaluation method. The used simulation [52] returns various performance metrics, like worst-case latency, throughput and processor usage. The method also allows calculating robustness in a way described below. The designer increases environmental-, platform-, or user-workload rate on a system, slightly deviating from the worst-case scenarios specified for the system. These "worse than worst-case" scenarios are also simulated and the predicted performance values are compared to the values obtained for worst-case scenarios. The deviation rate characterizes the *system robustness* - the ability of a system to provide correct services under unspecified overload conditions.

The following dependability improvement measures were chosen: Adjusting the capacity of a hardware blocks (frequency, bandwidth, memory size), changing the topology of a hardware architecture, and changing the mapping (deployment) of software components onto hardware nodes. In principal, other transformations could be also selected, namely replacing software components providing the same services and changing a topology of component assembly. Their application could have enhanced the

quality of the resulting solution. The example did not use any of the above-mentioned automatic optimisation strategies. The set of solutions were generated by architects. We believe that the design case would benefit from utilisation of appropriate architecture generation techniques (e.g. Simulated Annealing or Tabu Search) and consequently the approach would benefit from the abstract method presented in this paper.

After reviewing these two examples, it becomes clear that the tailoring process of the abstract method involves many decisions that could significantly influence the outcome of the dependability optimisation. However, by providing the abstract method and the guidelines in this section, the number of false or suboptimal decisions could be reduced.

6 Current Limitations of Architecture Based Dependability Evaluation and Optimisation Methods

There are several known limitations for architecture-based dependability evaluation and optimisation methods. In this section, we would like discuss the most important ones.

Inability to find all Pareto optimal solutions. The design spaces of most real-world applications are complex, if not infinite. As a result, a complete exploration of the design space is infeasible. To perform a guided search, heuristics, like simulated annealing, genetic algorithms or Tabu search are used in the abstract method. These heuristics help to search complex design spaces; however, there is no guarantee that globally optimal solutions will be found. Often these heuristics only produce suboptimal or locally optimal solutions, yet these solutions are often better than the original non-optimised architecture specification.

Inaccurate dependability evaluation results due to inaccurate information about dependability attributes of components. The outcomes of dependability evaluations at an architectural level are always *estimations* of the real dependability attributes of the system in operation. The reason for this is that most parameters used for dependability evaluation, like the failure rate of a software component, cannot be quantified exactly. There is always a degree of uncertainty. Furthermore, due to the early stage of development it is often unclear if the evaluation result quantifies the dependability of the real system after it has been built. The reason for this inaccuracy is that components have not yet been built, and consequently only (educated) guesses can be made about relevant dependability metrics of these components. Furthermore, if components are purchased off-the-shelf, then the dependability evaluation relies on the correctness of the dependability parameters provided by the component vendors [53].

Inaccurate dependability evaluation results due to limitation of the evaluation model. Another source of inaccurate dependability evaluation results are the limitations of the dependability evaluation models and methods themselves. These models and methods make assumptions in order to reduce the complexity of the dependability evaluation. As an example Gokahale [54] has identified several weaknesses of current reliability evaluation models. These limitations include modelling limitations such as the inability to model failure dependencies between components, the need for the model to have Markov properties, and the inability of current reliability evaluation methods to handle concurrent execution of components.

Problems with formalising and automating architecture transformations. Another requirement of fully automated architecture optimisation approaches is the need for formal and automatable architecture transformations. This limits the selection of the architecture transformations to only the class of dependability improving measures that can be formally specified and applied without intervention of the system architect. However, some transformations need additional information that must be provided by the system architect in order to apply the transformation. Examples are transformations that add additional components into the architecture. These components must be formally specified and basic component-based dependability metrics must be added to allow evaluations of the system's dependability attributes after the transformation.

To conclude this section, there are currently several limitations of standard dependability optimisation approaches. The knowledge of these limitations is important for system architects that want to use the abstract method presented in this paper for their own project, since they apply also to almost any instantiation of the abstract method.

7 Conclusion and Future Work

Over the last 20 years, the design of software intensive systems has shifted towards tightly integrated system architectures, which are characterized by extensive sharing of information and hardware. In such architectures, a large number of shared processors and communication channels allow a large number of potential configuration options at design time and a large number of potential reconfigurations at runtime. Architecture-based assessment technologies can answer questions regarding the dependability of individual configurations, whether they are schedulable or reliable for example. However, technological support is also needed in addressing the issue of a global dependability optimisation, i.e. a method that can aid the development of a hardware and software architecture that will guarantee optimal use of resources in the context of the given dependability and cost constraints.

In this paper, we performed a comparative review of methods that have recently been proposed to address variants of this complex architecture optimisation problem. Because of the sheer computational complexity of the problem and the consequent impracticality of using exact optimization techniques, only heuristic and meta-heuristic techniques have been considered here. These include approaches that use genetic algorithms, hill climbing, simulated annealing, Tabu search, and ant algorithms to solve various formulations of the architecture optimisation problem. Our review suggests that none of these methods consistently outperforms the others, either in the quality of derived solutions or the computational effort needed and that, in general, performance largely depends on the formulation of the problem and the shape of the landscape of the potential design space. However, in general, it appears to be the case that the various formulations of the problem exhibit many local optima which suggests that a global heuristic such as a Genetic Algorithms can generally improve the quality of the solutions reached by reducing the probability of the search getting stuck in local optima. Using a hybrid combination of a global heuristic with local search techniques also seems to provide a successful strategy, which further improves the quality of solutions and reduces computational effort. This is indeed an area where current research is largely

focused and where we expect to see some significant results in the future. These developments, in conjunction with developments on parallel search algorithms (e.g. parallel GAs) as well as progress on parallel and GRID computing, create possibilities for major advances in the optimisation of dependable systems in the near future.

Despite the wide diversity of the methods that we reviewed, significant commonalities were identified to enable us to define an abstract, generic, *meta-method* for architecture-based dependability optimisation from which specific approaches can be tailored. Although only a skeleton of this meta-method and its tailoring process have been provided in this paper, our long term aim is refinement of this approach and development of tools that can support its application and reuse across different applications. Such a meta-method could guide developers in the optimisation of dependability-critical systems by providing a general reusable framework which will enable additional functionality (e.g. new heuristics) to be included or specific actions to be taken to suit particular formulations of the optimisation problem.

References

1. Avizienis, A., Laprie, J.C., Randell, B., Landwehr, C.: Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput.* 1, 11–33 (2004)
2. Clements, P.C., Kazman, R., Klein, M.: *Evaluating Software Architectures: Methods and Case Studies*. Addison Wesley Longman, Reading (2001)
3. Bass, L., Clements, P., Kazman, R.: *Software Architecture in Practice*, 2nd edn. Addison-Wesley, Reading (2003)
4. Grunske, L.: Early quality prediction of component-based systems—a generic framework. *Journal of Systems and Software* 80, 678–686 (2007)
5. Vesely, W.E., Goldberg, F.F., Roberts, N.H., Haasl, D.F.: *Fault Tree Handbook*. U. S. Nuclear Regulatory Commission (1996)
6. IEC (International Electrotechnical Commission): IEC 61165: Application of Markov techniques (1995–2003)
7. Balsamo, S., Marco, A.D., Inverardi, P., Simeoni, M.: Model-Based Performance Prediction in Software Development: A Survey. *IEEE Transactions on Software Engineering* 30, 295–310 (2004)
8. Fyffe, D.E., Hines, W.W., Lee, N.K.: System reliability allocation and a computational algorithm. *IEEE Transactions on Reliability* 17, 64–69 (1968)
9. Nakagawa, Y., Miyazaki, S.: Surrogate constraints algorithm for reliability optimisation problems with two constraints. *IEEE Transactions on Reliability* 30, 175–180 (1981)
10. Ghare, P.M., Taylor, R.E.: Optimal redundancy for reliability in series system. *Operations Research* 17, 838–847 (1969)
11. Coit, D.W., Smith, A.E.: Reliability optimization of series-parallel systems using a genetic algorithm. *IEEE Transactions on Reliability* 35, 535–544 (1996)
12. Nicholson, M.: *Selecting a Topology for Safety-Critical Real-Time Control Systems*. PhD thesis, Department of Computer Science, University of York (1998)
13. Liang, Y.C., Smith, A.E.: An ant system approach to redundancy allocation. In: Angeline, P.J., Michalewicz, Z., Schoenauer, M., Yao, X., Zalzal, A. (eds.) *Proceedings of the Congress on Evolutionary Computation*, pp. 1478–1484. IEEE Press, Los Alamitos (1999)
14. Thiele, L., Chakraborty, S., Naedele, M.: Real-time calculus for scheduling hard real-time systems. In: *Proc. IEEE International Symposium on Circuits and Systems (ISCAS)*, Geneva, Switzerland, pp. 101–104 (2000)

15. Zitzler, E., Laumanns, M., Thiele, L.: SPEA2: Improving the Strength Pareto Evolutionary Algorithm. In: Giannakoglou, K., Tsahalis, D., Periaux, J., Papailou, P., Fogarty, T., (eds.) EUROGEN 2001, Evolutionary Methods for Design, Optimization and Control with Applications to Industrial Problems, Athens, Greece pp. 95–100 (2002)
16. Palermo, G., Silvano, C., Zaccaria, V.: A flexible framework for fast multi-objective design space exploration of embedded systems. In: Chico, J.J., Macii, E. (eds.) PATMOS 2003. LNCS, vol. 2799, pp. 249–258. Springer, Heidelberg (2003)
17. Givargis, T., Palesi, M.: Multi-objective design space exploration using genetic algorithms. In: Proceedings of the 10th International Symposium on Hardware/Software Codesign (CODES-02, pp. 67–72. ACM Press, New York (2002)
18. Kulturel-Konak, S., Coit, D.W., Baheranwala, F.: Reliability optimization of series-parallel systems using a genetic algorithm. *IIIE Transactions* 45, 254–260 (2006)
19. Kulturel-Konak, S., Smith, A.E., Coit, D.W.: Pruned pareto-optimal sets for the system redundancy allocation problem based on multiple prioritized objectives. *Journal of Heuristics* (2006) (inPrint)
20. Künzli, S., Thiele, L., Zitzler, E.: Modular design space exploration framework for embedded systems. *IEE Proceedings - Computers and Digital Techniques* 152, 183–192 (2005)
21. Papadopoulos, Y., Grante, C.: Evolving car designs using model-based automated safety analysis and optimisation techniques. *Journal of Systems and Software* 76, 77–89 (2005)
22. Andersson, J., Wallace, D.: Pareto optimization using the struggle genetic crowding algorithm. *Engineering Optimization* 34, 623–643 (2002)
23. Fredriksson, J., Sandström, K., Åkerholm, M.: Optimizing Resource Usage in Component-Based Real-Time Systems. In: Heineman, G.T., Crnković, I., Schmidt, H.W., Stafford, J.A., Szyperski, C.A., Wallnau, K. (eds.) CBSE 2005. LNCS, vol. 3489, pp. 49–66. Springer, Heidelberg (2005)
24. Grunske, L.: Identifying "good" architectural design alternatives with multi-objective optimization strategies. In: Osterweil, L.J., Rombach, H.D., Soffa, M.L. (eds.) 28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20–28, 2006, pp. 849–852. ACM Press, New York (2006)
25. Bondarev, E., Chaudron, M.R.V., de With, P.H.N.: A process for resolving performance trade-offs in component-based architectures. In: Gorton, I., Heineman, G.T., Crnkovic, I., Schmidt, H.W., Stafford, J.A., Szyperski, C.A., Wallnau, K.C (eds.) CBSE 2006. LNCS, vol. 4063, pp. 254–269. Springer, Heidelberg (2006)
26. Pimentel, A.D., Erbas, C., Polstra, S.: A systematic approach to exploring embedded system architectures at multiple abstraction levels. *IEEE Trans. Computers* 55, 99–112 (2006)
27. Livolsi, D., O'Neill, T., Leaney, J., Denford, M., Dunsire, K.: Guided architecture-based design optimisation of CBSs. In: ECBS 2006, pp. 247–258. IEEE Computer Society Press, Los Alamitos (2006)
28. Gritzalis, S., Spinellis, D., Georgiadis, P.: Security protocols over open networks and distributed systems: formal methods for their analysis, design, and verification. *Computer Communications* 22, 697–709 (1999)
29. Goseva-Popstojanova, K., Trivedi, K.S.: Architecture-based approach to reliability assessment of software systems. *Perform. Eval* 45, 179–204 (2001)
30. Reussner, R.H., Schmidt, H.W., Poernomo, I.: Reliability prediction for component-based software architectures. *Journal of Systems and Software – Special Issue of Software Architecture – Engineering Quality Attributes* 66, 241–252 (2003)
31. Hamlet, R.G., Mason, D.V., Woit, D.M.: Theory of software reliability based on components. In: Proceedings of the 23rd International Conference on Software Engineering, ICSE 2001, Toronto, Ontario, Canada, 12–19 May 2001, pp. 361–370. IEEE Computer Society Press, Los Alamitos (2001)

32. Becker, S., Grunske, L., Mirandola, R., Overhage, S.: Performance prediction of component-based systems – a survey from an engineering perspective. In: Reussner, R., Stafford, J.A., Szyperski, C.A. (eds.) *Architecting Systems with Trustworthy Components*. LNCS, vol. 3938, pp. 169–192. Springer, Heidelberg (2006)
33. Grunske, L., Kaiser, B., Papadopoulos, Y.: Model-driven safety evaluation with state-event-based component failure annotations. In: Heineman, G.T., Crnković, I., Schmidt, H.W., Stafford, J.A., Szyperski, C.A., Wallnau, K. (eds.) *CBSE 2005*. LNCS, vol. 3489, pp. 33–48. Springer, Heidelberg (2005)
34. Papadopoulos, Y., McDermid, J.A., Sasse, R., Heiner, G.: Analysis and synthesis of the behaviour of complex programmable electronic systems in conditions of failure. *Int. Journal of Reliability Engineering and System Safety* 71, 229–247 (2001)
35. Laprie, J.C.(ed.): *Dependability: basic concepts and terminology*. Springer, Heidelberg (1992)
36. Grunske, L.: Formalizing architectural refactorings as graph transformation systems. In: 6th International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD05), pp. 324–329. IEEE Computer Society Press, Los Alamitos (2005)
37. Nam, D., Park, C.H.: Multiobjective Simulated Annealing: A Comparative Study to Evolutionary Algorithms. *International Journal of Fuzzy Systems* 2, 87–97 (2000)
38. Horn, J., Nafpliotis, N., Goldberg, D.E.: A Niched Pareto Genetic Algorithm for Multiobjective Optimization. In: *Proceedings of the 1st IEEE Conference on Evolutionary Computation*, IEEE World Congress on Computational Intelligence, Piscataway, New Jersey, vol. 1, pp. 82–87. IEEE Service Center, Los Alamitos (1994)
39. Knowles, J.D., Corne, D.W.: Approximating the Nondominated Front Using the Pareto Archived Evolution Strategy. *Evolutionary Computation* 8, 149–172 (2000)
40. Yim, J.S., Kyung, C.M.: Datapath layout optimisation using genetic algorithm and simulated annealing. *IEE Proceedings - Computers and Digital Techniques* 145, 135–141 (1998)
41. Walsh, W.E., Tesauro, G., Kephart, J.O., Das, R.: Utility functions in autonomic systems. In: *1st International Conference on Autonomic Computing (ICAC 2004)*, New York, USA, 17–19 May 2004, pp. 70–77. IEEE Computer Society Press, Los Alamitos (2004)
42. Mattson, C.A., Messac, A.: Pareto frontier based concept selection under uncertainty, with visualization. *Optimization and Engineering* 6, 85–115 (2005)
43. Zanchettin, C., Ludermir, T.B.: The influence of different cost functions in global optimization techniques. In: *Proc. 9th Brazilian Symposium on Neural Networks (SBRN'06)*, Los Alamitos, CA, USA, pp. 17–31. IEEE Computer Society Press, Los Alamitos (2006)
44. Grunske, L.: Transformational patterns for the improvement of safety properties in architectural specifications. In: *Proceedings of The Second Nordic Conference on Pattern Languages of Programs (VikingPLOP 03)*, Bergen, Norway (2003)
45. Saridakis, T.: A system of patterns for fault tolerance. In: *Proceedings of the EuroPlop (2002)*
46. Gokhale, S.S., Trivedi, K.S.: Reliability prediction and sensitivity analysis based on software architecture. In: *13th International Symposium on Software Reliability Engineering (ISSRE 2002)*, Annapolis, MD, USA, November 12–15, 2002, pp. 64–78. IEEE Computer Society Press, Los Alamitos (2002)
47. Ehrgott, M., Gandibleux, X.: A Survey and Annotated Bibliography of Multiobjective Combinatorial Optimization. *OR Spektrum* 22, 425–460 (2000)
48. Reeves, C.R.: *Modern Heuristic Techniques for Combinatorial Problems*. John Wiley & Sons, New York (1995)
49. Grunske, L., Kaiser, B.: Automatic generation of analyzable failure propagation models from component-level failure annotations. In: *5th International Conference on Quality Software (QSIC 2005)*, Melbourne, September 19–20, 2005, pp. 117–123. IEEE Computer Society Press, Los Alamitos (2005)

50. Papadopoulos, Y., Parker, D., Grante, C.: Automating the failure modes and effects analysis of safety critical systems. In: *Int. Symposium on High-Assurance Systems Engineering (HASE 2004)*, pp. 310–311. IEEE Computer Society Press, Los Alamitos (2004)
51. Bondarev, E., Chaudron, M.R.V., de Kock, E.A.: Exploring performance trade-offs of a jpeg decoder using the deepcompass framework. In: *WOSP '07: Proceedings of the 6th international workshop on Software and performance*, pp. 153–163. ACM Press, New York, USA (2007)
52. Bondarev, E., Chaudron, M.R.V., de With, P.H.N.: Compositional performance analysis of component-based systems on heterogeneous multiprocessor platforms. In: *EUROMICRO '06: Proceedings of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications*, Washington, DC, USA, pp. 81–91. IEEE Computer Society Press, Los Alamitos (2006)
53. de Castro Guerra, P.A., Romanovsky, A.B., de Lemos, R.: Integrating COTS software components into dependable software architectures. In: *6th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2003)*, Hakodate, Hokkaido, Japan, May 14–16, 2003, pp. 139–142. IEEE Computer Society Press, Los Alamitos (2003)
54. Gokhale, S.S.: Architecture-based software reliability analysis: Overview and limitations. *IEEE Transactions on Dependable and Secure Computing* 4, 32–40 (2007)

A Distributed Monitoring System for Enhancing Security and Dependability at Architectural Level

Paola Inverardi and Leonardo Mostarda

Dip. di Informatica, Università di L'Aquila, Coppito 67100, L'Aquila, Italy
{inverard,mostarda}@di.univaq.it

Abstract. In this work we present the DESERT tool that allows the automatic generation of distributed monitoring systems for enhancing security and dependability of a component-based application at architectural level. The DESERT language permits to specify both the components interfaces and interaction properties in term of correct components communications. DESERT uses these specifications to generate one filter for each component. Each filter locally detects when its component communications violate the property and can undertake a set of reaction policies. DESERT allows the definition of different reaction policies to enhance system security and dependability. DESERT has been used to monitor applications running on both mobile and wired infrastructures.

1 Introduction

In this work we present the DESERT tool that allows the automatic generation of distributed monitoring systems for enhancing security and dependability of a component-based application at architectural level.

In our system model we assume a set of black-box components that interact with each other by exchanging messages. A message encodes information about the type of communication, i.e. a request or a reception, the kind of service and its parameters and the (returned) data. This architectural level model has shown to be flexible enough to model several types of distributed systems and communication patterns. For instance, in [1] we model mobile sensors applications. In this case components are sensor devices and communication is achieved by means of send and receive asynchronous invocations. In [2] we have modeled CORBA middleware based applications. In this case we have CORBA components that communicate by means of different types of service invocations (i.e., asynchronous, synchronous and deferred synchronous invocations).

At the architectural level we define an *anomalous component* as one that interacts with the remaining components in order to subvert the 'correct' system behavior. Anomalous component interactions can have different origins and their detection constitutes the basis to provide different functionalities of the system. For instance, anomalous interactions can originate by a malicious component that exploits other components vulnerabilities (see [3] for an extended survey).

In this case the component detection constitutes the basis to build an Intrusion Detection System (IDS)[4,5,6,7,8] to enhance the system security functionality. In the field of dependable computing, anomalous components interactions can be a consequence of architectural mismatch [9] and/or of components faults that lead to system failure. In this case the detection mechanism can be the basis for error detection and system recovery [10]. In the field of performance evaluation, anomalies on components interactions can be a consequence of degraded response time, thus detection mechanisms can be used to provide reconfiguration mechanisms.

Today's monitoring tools [11] are a viable solution to detect anomalous components interactions. They are tools that: (i) gather information about applications, (ii) interpret the gathered information; (iii) respond appropriately (i.e. they can undertake different reaction policies). A monitoring system can be characterized by the functionalities it provides. Modern monitoring tools are used to increase security, dependability and performance (see Section 2 for a detailed survey). Moreover they can be part of the target system therefore they can add new system behaviors.

In this work we present the DESERT tool [12,2,1,13] that allows the generation of distributed monitoring systems for component based applications.

The monitoring definition is obtained starting from a DESERT program written in the DESERT definition language. The DESERT program contains both an interfaces descriptions part and a global automaton one (Figure 1 part 2). The interfaces descriptions part is obtained by means of an interface description language that permits to describe each component of the system in terms of its name and the services that it requires/provides. The global automaton part can contain different state machines (that in the following will be referred to as interaction properties) that are described by means of a DESERT state machine definition language. A state machine describes the correct messages exchange among components (i.e. the correct component communications). As we are going to see in Section 3.2, the state machines can model complex communication patterns but they are not suitable to describe temporal properties (i.e. the DESERT approach is appropriate for applications without timeliness requirements).

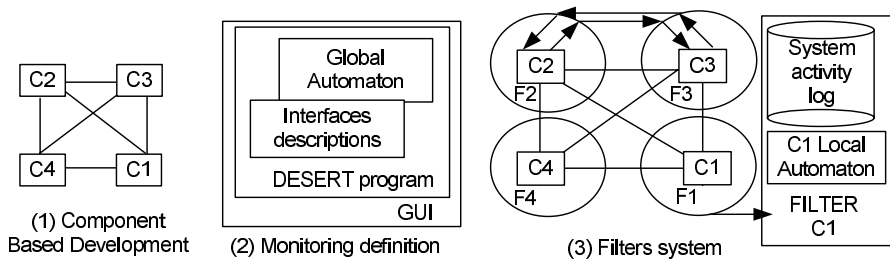


Fig. 1. DESERT phases

The DESERT program is a means to define a logically centralized monitoring tool. This monitoring tool: (i) gathers all messages exchanged among components; (ii) checks the messages consistency with respect to the property defined by the global automaton; (iii) in case of mismatches undertakes a set of reaction policies. Different reaction policies can be defined to provide different functionalities of the system. For instance, when a component misbehaves for malicious purposes isolation of this component can be the correct reaction to enhance the system security. In case the component performs anomalous interactions as a consequence of a fault, recovery reaction can improve the system dependability.

The implementation of the logically centralized monitoring system can pose problems of security, reliability and performance. Furthermore, already existing legacy distributed systems could not allow the addition of a new component which monitors the information flow in a centralized way.

To overcome these problems DESERT automatically decomposes the global automaton in a set of local automata that are assigned one for each component. A local automaton constitutes the basis to build a filter that is interposed between its component and the environment (see Figure 1 part 3). The filter captures all incoming/outgoing component messages and uses the local automaton to locally detect violation of the policy expressed by the global automaton. In other words, the filters taken as a whole system constitute a monitoring system that is “equivalent” (see [12,13] for a formal description) to the centralized one.

The DESERT tool implements both a front-end and a set of back-ends. The former, starting from the DESERT program and a component name (e.g. *C2*), produces the platform independent specification of the *C2* local automaton. The latter translates the *C2* local automaton specification in a specific filter implementation. For instance for distributed applications where the components communicate by using the CORBA middleware we have implemented a CORBA back-end. This back-end automatically produces a new CORBA component (the filter) that is interposed between the component communications and the environment.

We point out that the novelty of this work is the description of the DESERT definition language and the overview of the DESERT distribution basic steps. In fact in [12,13] we primarily focus on the proofs of correctness and completeness of the distribution process while in [2] and in [1] we only sketch how the approach can be suitable for different areas (i.e., enforcement and security respectively).

The paper is organized as follows. In the next section we present an overview of the monitoring system technology and we summarize the contribution of our architectural level monitoring technology. Section 3 introduces our monitoring definition language, in particular, Section 3.1 describes our interface description language and Section 3.2 the state machine specification language. Section 4 shows how interfaces and state machine can be used to define a logically centralized monitoring system. Section 5 summarizes the different reaction policies that can be set to generate different monitoring systems for different areas of applications. Section 6 sketches the generation of the distributed implementation of the logically centralized monitoring system. Section 7 describes different

case studies in which we have applied our monitoring approach. We show how our reaction policies can be tuned in order to output different monitoring systems used for different functionalities of the system. Finally, Section 8 provides conclusive remarks and future work.

2 Monitoring Tools at Glance: Concepts and Terminology

Monitoring systems have been around since the 1960s. Originally they were conceived with a centralized structure and used to debug centralized systems. Today's monitoring systems monitor distributed applications and themselves have a distributed architecture. Generally speaking, a monitoring system can be defined as tools that: (i) gathers system information; (ii) interprets the gathered information; (iii) after interpretation can undertake a set of reaction policies. Monitoring provides a solution for areas of growing concerns: lack of dependability, security and performance enhancement and tools to support distributed applications.

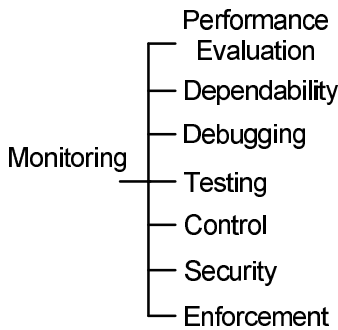


Fig. 2. Monitoring uses

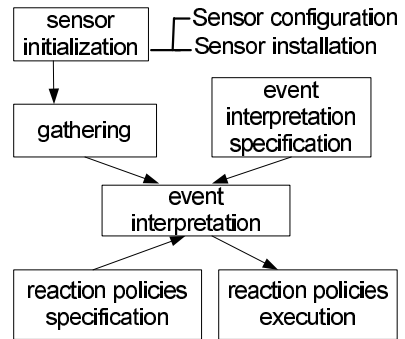


Fig. 3. Monitoring activities

In Figure 2 we show the primary uses of monitoring tools. Dependability includes cases in which interpretation involves detection of system errors and reaction policies are undertaken to enhance the system fault tolerance [11]. Performance evaluation includes detection of the system response time degradation. In this case reactions can include system reconfiguration, dynamic program tuning and on-line steering. Security involves the interpretation of information in order to detect attacks. In this case interpretation of information can be performed using a set of predefined intrusion signatures and/or the correct system behavior definition. The former are referred to as misuse detection systems [8,14,15,16] while the latter are referred to as anomaly detection systems [17,7,18,5,19,12,4]. Debugging and testing employs monitoring techniques to extract data values from an application being tested. Control includes cases in which the monitoring implements part of system functional requirements. Finally, enforcement [20] is the case in which the monitor interprets information in order to ensure desired system behaviors.

Despite the various monitoring system utilizations their definition involves the following standard activities: (i) sensor initialization; (ii) gathering; (iii) event interpretation specification; (iv) event interpretation; (v) reaction policies specification; (vi) reaction policies execution (see Figure 3). Each activity can be performed either by the user or by the monitoring system itself.

In order to describe the above activities in the following we introduce some basic concepts common to all monitoring systems.

Information arrives to the monitoring tools in the form of *events*. Events can regard the system states, interactions among system parts and system activities. We point out that events can be related to different layers of abstraction, i.e., hardware-level, process level and application level. A *sensor* is the monitoring element that locally gathers events. Sensor automatically sends events when they occur or it is the monitoring system itself that can request them.

Sensor initialization (see Figure 3) includes configuration and installation. Configuration is carried out by deciding what events a sensor will gather and the definition of additional sensor capabilities, e.g., local conditions checking. Sensor installation is carried out by placing the sensor code at the correct location. This is usually performed through code instrumentation¹ or conceiving the sensor as an external observer that sniffs all communications among system parts.

Gathering is the activity in which sensors collect events and forward them to the monitoring system. The gathering can be either off-line or in-line. The former is characterized by the fact that the gathering code uses the resources of the target system. The latter is characterized by the fact that the gathering code uses resources separated from the target system ones.

Event interpretation is the heart of the monitoring system, where the monitoring system interprets events. Event interpretation is achieved using the event interpretation specification that is usually defined by means of errors conditions description and/or description of correct system behaviors. Event interpretation can be categorized either as synchronous or asynchronous. Asynchronous is when events are interpreted after system execution. Synchronous when the system is suspended until event interpretation.

Reaction policies execution can take place after event interpretation and its implementation must comply with the reaction policies specification. Reaction policies specification are a consequence of the monitoring system uses. Earlier monitoring systems were only involved in logging and tracing reactions. Nowadays monitoring systems embed complex reactive utilities that can be undertaken after event interpretation.

Generally speaking, reaction policies can be either non-intrusive or intrusive.

Non-intrusive reaction policies do not affect the program behavior except for execution speed and program size. Logging and tracing are examples of these monitoring system reactions. Logging can be performed to record violations of the correct system behavior. Tracing can be viewed as a high-level logging utility

¹ Instrumentation requires code access and can be manually performed by the user or automatically by code analysis.

that records all sequences of interactions resulting in the anomalous system behavior.

Intrusive reaction policies affect, to some degree, the state, the configuration and/or the execution of the system (see [10] for an extended survey). For instances intrusive reaction policies are: (i) termination; (ii) shunning; (iii) re-configuration; (iv) rollback; (v) rollforward. Termination refers to the monitoring system ability to terminate part (or the whole) system execution. Shunning is the case in which the monitoring system denies the traffic generated by a specific source or a set of sources. Reconfiguration can physically alter the location or functionality of network or system elements. For instance, in the field of dependability, reconfiguration can be useful to switch in spare components or reassign tasks among non-failed components. In the case of security reconfiguration can be used to isolate the attackers. Rollback brings the system back to a previous saved state. Rollforward brings the system, in a new 'safe' state.

In this paper we focus on monitoring systems that use formal specifications for event interpretation (in the following referred to as specification-based monitoring systems). Different names, that depend on the monitoring system uses, can refer to a specification-based monitoring system. In the field of security they are referred to as specification-based and anomaly-based IDSs [5,12,4]. In the field of dependability and correctness checking they can be referred to as software-fault monitors [11] and enforcement mechanisms [20], respectively. Despite these various uses, a specification-based monitoring system is usually interpreted as a tool that takes an application and a specification of software properties and checks that the execution meets the properties, i.e., that the properties hold for the given execution. A property can describe the correct communication among system parts, the correct system states and the correct system activities. A specification language is a language that is used to describe properties.

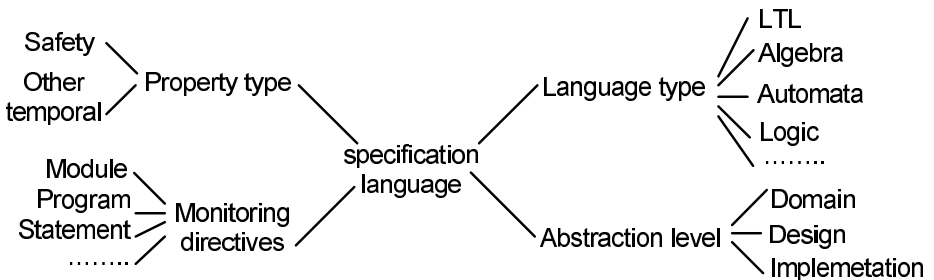


Fig. 4. Specification language categorization

In Figure 4 we show the elements that can characterize a specification language: (i) the language type; (ii) the abstraction level; (iii) property type; (iv) the monitoring directives. The type of language used to define a property can be based on algebra, automata, logic and so on. The abstraction level refers to the support that the language provides in order to specify the property and the knowledge about the domain, the design and the implementation of the system.

For instance, a language that provides support to specify properties for CORBA middleware would be classified domain-based. A language that allows the specification of properties in implementation independent fashion would be design based. Finally, properties that involve statements and variables of a system have to be defined by means of implementation-dependent language.

Two types of properties can be specified: safety and temporal ones. A safety property expresses that something bad never occurs. A temporal property includes progress and bounded liveness [11]. Monitoring directives specify that a property can be evaluated at different levels, i.e., program, module, statement and so on.

The DESERT tool allows the automatic generation of distributed specification-based monitoring systems for enhancing security and dependability in distributed black-box components applications. The black-box nature of the components imposes that our events can be observable messages exchanged among them. The DESERT language allows the definition of both the system model and the correct system behavior. The system model is provided by means of an interface description language. This language permits to describe each component of the system in terms of its name and the services that it requires/provides. The correct system behavior is provided by means of a global automaton that describes the correct messages exchange among components (i.e. interaction properties). As described in [20] automata can describe safety and bounded liveness properties.

A monitoring system, based on the global automaton, gathers all messages exchanged among components and verifies that such messages do not violate the policies expressed by the global automaton. Our monitoring system is off-line since it does not use the resources of the target system (in particular it is implemented as an external observer). Moreover, it is synchronous since messages are delivered only after interpretation.

Architectural level monitoring permits to obtain implementation-independent description, however DESERT provides the basic mechanisms to add design and implementation details into the properties descriptions. Moreover, as we describe in Section 5 DESERT allows the definition of reaction policies tailored for security and dependability purposes.

3 The DESERT Definition Language

In this section we describe the DESERT definition language that permits to specify both the system model and the global automaton. The system model specifies the components interfaces descriptions.

3.1 Components Interfaces Descriptions

This part is composed of a set of *component interface declarations*. A component interface declaration is composed of: (i) the component name; (ii) a list of services description.

A service description is either of the form `!serviceName(Parameters).returnType` or `?serviceName1(Parameters1).returnType1`. The former declares that

the component is a client of the service *serviceName* having the formal parameters *Parameters* and the returned value type *returnType*. The latter declares that the component provides the service *serviceName1* to the environment.

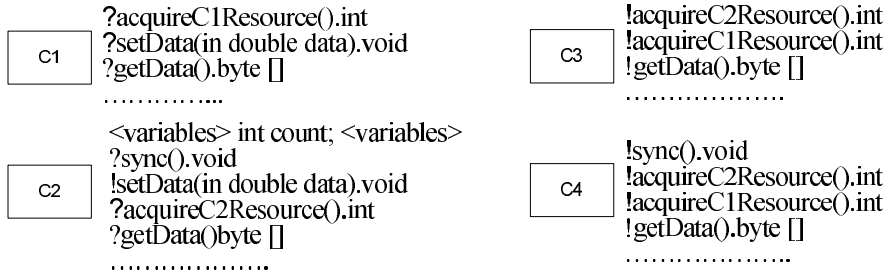


Fig. 5. Components interfaces

The *Parameters* is constituted by a sequence of the type $T_1 D_1 X_1, T_2 D_2 X_2, \dots, T_n D_n X_n$, with $n \geq 0$ ($n = 0$ means an empty sequence of parameters). T_i is a label that can take one of the following values: *in* and *out*. *in* specifies that the parameter X_i is an input service parameter (i.e., it must be provided to the service). *out* specifies that X_i is an output service parameter (i.e., it contains an output after the service execution). D_i is the X_i domain and defines the set of values that the parameter can take.

Suppose that a component is a client (server) of the service *!serviceName* (*Parameters*). *returnType* (*?serviceName1(Parameters1).returnType1*). The label *returnType* (*returnType1*) defines the set of values that the component client (server) can receive (send) back after the *serviceName* (*serviceName1*) synchronous service invocation. We point out that the type of service invocation (i.e. synchronous and asynchronous) is specified by means of a label that is prefixed to the service declaration. Optionally, a user can add variables declarations in the components interfaces descriptions.

In Figure 5 we sketch part of the case study that refers to a cooling water pipe distributed industrial application (see [2] for a detailed description). It concerns the monitoring of messages exchanged among a set of components that collect and correlate data on the amount of water that flows in different water pipes. This water is used to cool industrial machinery. The water pipes are monitored by the server components *C1* and *C2* that interact with *Programmable Logic Controllers* (PLCs) in order to obtain the data related to each water flow. The clients *C3* and *C4* request services on the servers in order to write/read the water flow data.

C1 can receive incoming requests of the *?acquireC1Resource().int* and *?setData(in double data).void* services in order to allow the exclusive access to the Area 1 data and to manually set its local data, respectively. *C2* can receive incoming requests of the *?acquireC2Resource().int* and *?sync().void* services in order to allow the exclusive access to the Area 2 data and to accept a synchronization request, respectively. Moreover, *C2* can require the *!setData(in double*

data). *void* service to *C1* in order to send its data to *C1*. In particular *C2* sends its data after the reception of a *sync()* request. *C3* and *C4* are client of the services exposed by the component *C1* and *C2*. Notice that we relate the variable *count* to the component *C2*. We point out that this variable is used in the state machine definition, i.e., it is part of the monitoring system definition.

The goal of the overall application is to ensure consistency on the water flows data that are used for billing purposes. To this extent we define a state machine.

3.2 The Global Automaton

After the components interfaces descriptions, the DESERT program has to describe a state machine that defines the correct components communications.

In the following we introduce some notation useful to describe our automata. Let us suppose that the *C* interface declaration defines a service of the type *!serviceName*(*T₁ D₁ X₁, ..., T_n D_n X_n*).*returnType* (i.e., *C* is a client of the *serviceName* service) and *S* exports the service *?serviceName*(*T₁ D₁ X₁, ..., T_n D_n X_n*).*returnType* (i.e., *S* is a server of the *serviceName* service). The notation *C c, x*; is used to declare two instances (i.e., *c* and *x*) of the component *C*. The symbol '*' denotes an unknown type of component (see case study of Section 7 for examples).

In the following we describe the events that we monitor at architectural level. Let us consider the declarations *C c*; and *S s*; :

- *!serviceName* (*X₁, ..., X_n*)*_c_s* defines an event that can be observed when *c* performs the *serviceName* (*X₁, ..., X_n*) service invocation on *s*. It is worth noticing that when the invocation is executed the parameter *X₁ ... X_n* are suitable instantiated by *c*.
- *?serviceName* (*X₁, ..., X_n*)*_c_s* defines an event that can be observed when *s* performs the *serviceName* (*X₁, ..., X_n*) service-receive invocation on *s*.
- If the *serviceName* service is synchronous we can define the symmetrical invocation *!serviceName_s_c* and *?serviceName_s_c*, i.e., the answer to the *serviceName* service observed on the server and client side, respectively.

The receive invocation *?serviceName*(*X₁, ..., X_n*)*_c_s* has associated the default variables *?serviceName.X_i*, with $1 \leq i \leq n$, each of them contains the *X_i* parameter value when the invocation is performed at the *s* server side. The send invocation *!serviceName*(*X₁, ..., X_n*)*_c_s* has associated the default variables *!serviceName.X_i*, with $1 \leq i \leq n$, each of them contains the *X_i* parameter value when the invocation is performed at the *c* server side. If the *serviceName* service is synchronous then the default variable *!serviceName* (*?serviceName*) contains the *serviceName* returned value sent (received) by *s* (*c*). We point out that all invocations that include the symbol * define the same variables (see [13] for details).

Let *p* be one of the above invocations. Each transition of the global automaton can be labeled with a piece of information of the form *p*[*P*]{*code*}. The label

[P] is a predicate that the invocation p must verify, the field {code} a piece of code executed when the transition is performed. In particular, as we are going to see in section 7, a predicate is a means to avoid attacks/faults caused by invocations with malformed formats (e.g., sql injection and buffer overflow) while the code can be executed to perform more complex checks based on the component variables.

A state machine describes the system traces that an external observer should see in the case of correct components interactions. In particular, these traces are related only to the services invocations defined inside the global automaton alphabet.

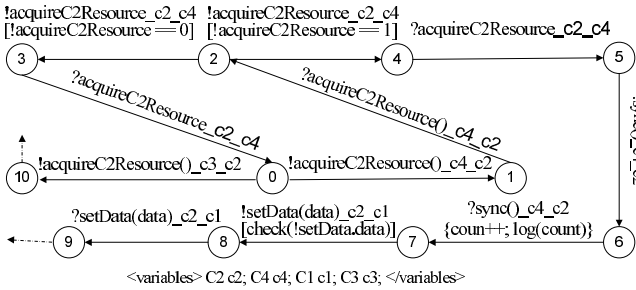


Fig. 6. Global automaton



Fig. 7. Simple property

In Figure 6 we show a portion of the state machine declaration that is related to our case study [13,2]. In this case we have the component instance $c1$ of the type $C1$, $c2$ of the type $C2$, $c3$ of the type $C3$ and $c4$ of the type $C4$. We can observe that in the state 0 both clients $c3$ and $c4$ have the possibility to perform an *acquireC2Resource()* service invocation on the server $c2$. In the case that $c4$ performs the invocation the state machine moves from state 0 to 1. We point out that this invocation is observed at the $c4$ client side. In state 1 this invocation is observed at the $c2$ server and the global automaton state can be changed from 1 to 2. From the state 2 two possible transitions exit. One models the $c2$ *acquireC2Resource* service response when its returned value is equal to 0. This is expressed by the condition *!acquireC2Resource == 0* (e.g., this is the case of server busy). The other one models the $c2$ *acquireC2Resource* service response when its returned value is equal to 1. If the latter is applied then the state machine moves to state 4, where the $c2$ services can be provided to the client $c4$. Notice that each time the service *sync()* is received by $c2$ the related variable *count* is updated. Moreover, the state machine does not include the description of the service *!getData().byte[]* and other components services (see Figure 5). In Figure 7 we show a more simple policy in which concurrent services invocations are allowed. This is represented by multiple transitions that enter and exit from the same state.

We remark that different global automata can define different security policies and can be used to concurrently monitor the components. However, in the remaining we focus on a single automaton since all results can be easily extended to multiple concurrent automata.

In the following we sketch strengths and weaknesses of the DESERT definition language.

The DESERT language flexibility allows the definition of monitoring systems in different contexts. For instance, in the context of CUSPIS project [21] user services are implemented by a client (e.g. c) that performs invocations on finite set of servers s_1, \dots, s_n . Servers can interact with each other and with further components. In this case a global system view is needed to ensure the correctness of interactions scattered over several components. We have defined a *server side* policy that characterizes client sessions in terms of both servers received invocations (e.g. $?serviceName(Parameters)_{c_s_i}$) and servers performed invocations (e.g. $!serviceName(Parameters)_{s_i_s_j}$). In the case of wireless sensor networks (see [1] for details), the automaton can contain only invocations of the form $!serviceName(listOfParameters)_{c_s}$ (i.e., a client side policy). This is the case in which mobile devices (e.g. sensors) send asynchronous service invocations to unknown servers, therefore, we have to write client side policies.

If the state machine describes all possible services invocations we may incur in the usual state explosion problem. However, the crucial property commonly only interests a subset of the global system behavior. Moreover, it is worth notice that the global automaton does not allow the definition of temporal constraints so that DESERT is a tool unsuitable to model interactions with timeliness requirements.

4 A Logically Centralized Monitoring System

A DESERT user defines the global automaton at the level of system integration, i.e., she defines the correct components communications by having a global system view. From the global automaton perspective it is easy to derive a logically centralized monitoring system. The logically centralized monitoring system does not have a concrete counterpart but its knowledge makes it easy to understand the filters system generation (i.e., the distributed monitoring system implementation) that is hidden by the DESERT tool. Therefore for the sake of simplicity we will describe the monitoring use and the reaction policies referring to a centralized approach. In Section 6 we show how the distributed implementation of the centralized approach is automatically generated.

In the following we use the case study of Section 3 to describe all actions that the centralized monitoring system can undertake, i.e., (i) send invocation acceptance; (ii) buffering action; (iii) receive invocation acceptance; (iv) forwarding action; (v) anomaly detection action.

The logically centralized monitoring system has a buffer used to store the components invocations it captures. Suppose that the monitoring system picks the invocation $!acquireC2Resource()_{c4_c2}$ up from its local buffer and the global

automaton of Figure 6 is in state 0. Then the monitoring system can 'accept' this invocation since there is a 0-exiting transition labeled with it and there is not a predicate to be satisfied. Accept means that the monitoring system buffers the invocation `?acquireC2Resource()_c4_c2`² and changes the automaton state to 1 (in the following this monitoring system activity will be referred to as *send invocation acceptance*). In state 1 there is the possibility that the monitoring system picks an invocation `!acquireC2Resource()_c3_c2` up from the buffer (i.e., the client `c3` requires the access to the same `c2` resources). This invocation cannot be accepted since there is not a 1-exiting transition labeled with it. Therefore, the monitoring system checks the existence of a state reachable from 1 where the following conditions hold: (i) there is an exiting transition t labeled with the invocation `!acquireC2Resource()_c3_c2`; (ii) the predicate related to the transition t is satisfied (in our case 0). In this case the monitoring system puts the invocation back to process it later (*buffering action*). By continuing our example, the monitoring system can pick the invocation `?acquireC2Resource()_c4_c2` up from the buffer and accepts it by means of the 1-exiting transition. In this case the monitoring system forwards the `acquireC2Resource` invocation to the server `c2` and changes the automaton state to 2 (*receive invocation acceptance*). Since the service `acquireC2Resource` is synchronous the monitoring system has to wait for the result `?acquireC2Resource` to put it in the buffer. In any automaton state invocations that are not described in the global automaton are forwarded without any check, e.g. all `getData()` service invocations are forwarded, (*forwarding action*). An anomalous component invocation is detected when: (i) the invocation cannot be accepted in the current automaton state (e.g. q) and in any state reachable from q ; (ii) the invocation was buffered and not consumed after a finite amount of time³; (iii) the invocation is related to services not present in the interfaces definitions (*anomaly detection action*).

After an anomaly detection action our centralized monitoring system can undertake a reaction policy. In the next section we describe all different reaction policies that can be set by means of the DESERT tool. Different reaction policies allow the generation of monitoring tools for different areas, e.g. security, dependability and enforcement (see Section 2).

5 The DESERT Reaction Policies and the Application Areas

In Figure 8 we show all reaction policies that can be set in order to generate monitoring systems for different uses.

² The buffering is needed since it is the global automaton that can (or cannot) define when the related receive invocation must be delivered. This is strictly related to whether or not the receive invocation is defined inside the global automaton alphabet.

³ The amount of time must be chosen by the user. In particular it can be assigned at 'the global automaton level' (i.e, all invocations must be consumed after a fixed amount of time) and/or to each single invocation.

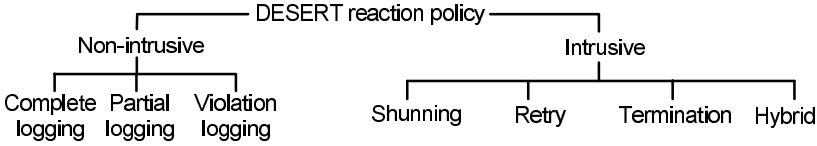


Fig. 8. DESERT reaction policies

The *logging* reaction policy permits to obtain a monitoring system that does not affect the system behavior. Different levels of logging can be set up. The *complete logging* allows the logging of all messages exchanged among components⁴. The *partial logging* permits to log all messages exchanged among components that belong to the global automaton alphabet. The *violation logging* permits to log all messages exchanged among components that violate the property expressed by the global automaton. For instance, the logging reaction policies can support off-line and on-line testing. In the former case complete logs can be produced and analyzed in order to detect traces that violate the case tests. In the latter case the monitoring system can produce a violation log in which all traces violating the global automaton policy are recorded. In other words we can test the run time system traces with respect to the global automaton ones.

The *Shunning* reaction policy can be partial and complete. In the partial shunning the monitoring system logs the information details of any anomalous message m that violates the property. When m has been logged, the monitoring discards m and does not deliver it to the receiver. In the complete shunning the monitoring system registers the sender of the message m and denies all future messages send by it, i.e., the monitoring system isolates the component that performed the violation. The shunning reaction policies can be used in the field of security [1] in order to isolate the component that attacks the system. However, it can generate components anomalous behaviors as a consequence of no returned value to them, therefore shunning cannot be applied in order to enhance the system fault tolerance.

In the *retry* reaction policy the monitoring system discards and logs any message m that mismatches the correct behavior. After m has been discarded the monitoring returns an *error* to the component that has sent m . An *error* is a value that a component recognizes either as an exception or a failure condition. The retry reaction policy can be a means to improve the system fault tolerance. The error detection mechanism is provided by our monitoring tool that detects a component misbehavior. Moreover, the error value returned can be a simple recovery mechanism to let the component try again. It is worth noticing that while the shunning policy can be always applied without having any type component knowledge the retry reaction requires that the component explicitly declares a handled returned error value.

⁴ We log for each invocation: time, service name, parameters or returned value, sender and receiver.

Termination refers to the monitoring system ability to terminate the components generating the anomalous behavior. This reaction can be followed by a reinitialization phase in which components can be configured and restarted. Notice that in this case the monitoring system must have a mechanism to stop and restart a component execution.

Hybrid reaction policies include the case in which different reaction policies are assigned to different invocations. For instance, the shunning policy can be associated to each invocation related to services that are critical for the system security. The retry policy can be used for invocations that can be performed after the correct system login, i.e., this policy can be used to recover authorized components.

We can observe that both retry and shunning policies produce a monitoring system that acts like an enforcement mechanism (EM). As defined in [20] enforcement mechanisms compare a formal specification with the system steps. When there is a violation of the formal specification an EM can either terminate the system execution or replace an unacceptable execution step with an acceptable one.

6 The Distribution Process

The implementation of the logically centralized monitoring system is not practical in systems composed by a large number of distributed components and of interactions properties, where the parsing efficiency, scalability and failure can become relevant issues. Moreover, already existing legacy distributed systems could not allow the addition of a new component which monitors the information flow in a centralized way. The DESERT solution is an algorithm to automatically distribute the logically centralized monitoring system (i.e., the 'centralized' DESERT program) on each component of the system. It performs this generation by decomposing the global automaton in a set of local automata that are assigned one for each component of the system. A local automaton constitutes the basis to build a filter that locally monitors its component communications. The set of filters taken as a whole system constitutes a distributed monitoring system "equivalent" to the central one.

In Figure 9 we show the basic components of the DESERT tool that allow the generation of the monitoring system implementation. A graphical user interface allows the description of both components interfaces and state machines. These descriptions are stored in XML format.

The front end is composed of the following components: the local automaton generator, the parser and the semantic controller. The local automaton generator component takes in input the XML file and a component name (e.g. *C2*). It forwards the XML file to the parser and semantic component that performs all syntax and semantic checks, respectively. In the case that there are not errors the local automaton generator generates the XML specification of the *C2* local automaton. We remark that this process can be performed locally on the host where *C2* resides on.

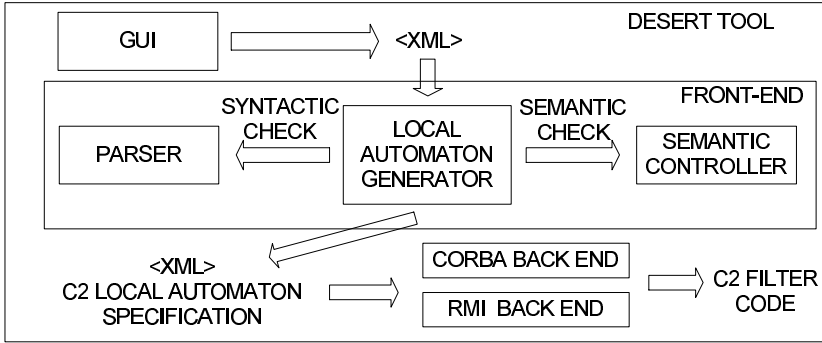


Fig. 9. The DESERT C2 filter generation

The C_2 local automaton (in the following denoted with A_{C_2}) is part of the global automaton enriched with transitions labeled with synchronization messages (see Figure 10) that in the following will be referred to as dependencies messages. These transitions are applied by the filter C_2 to send (receive) information to (from) other filters. Dependencies allow the simulation of the centralized monitoring system. In [12,13] we show all formal proofs, we discuss the overhead introduced by such synchronization messages and we show how it does not constitute a problem since they are small in size (i.e., they are integer). Moreover, in [13] we also point out that both the time required to exchange the synchronization messages and their parsing can slow the application response time (i.e., the DESERT tool enhance security and dependability issues at the expense of the system response time).

The C_2 local automaton specification is platform independent and may be translated into different filter implementations. For instance for distributed applications where the components communicate by using the CORBA middleware we have implemented a CORBA back-end. This back-end automatically produces a new CORBA component (the filter) that is interposed between the component communications and the environment. The filter exposes all services that the component requires and provides to the environment (see Figure 10). The entire process of filter generation is polynomial on the global automaton size. We point out that the filters work at the middleware level therefore we do not require components source code.

In the following we sketch the local automata generation and we discuss the filters actions. For the sake of presentation we introduce some notation. We use the notation $q' = \delta(q, p)$ to denote a global automaton rule that exits from the state q , enters in q' and is labeled with the invocation p . We denote with $q' = \delta_C(q, p)$ the same rule projected on the A_C local automaton. We denote with $k(q, q')(p)$ an integer that uniquely identifies the global automaton rule $q' = \delta(q, p)$. We use $P(q, q')(p)$ ($C(q, q')(p)$) to denote the predicate (code) related to the rule $q' = \delta(q, p)$.

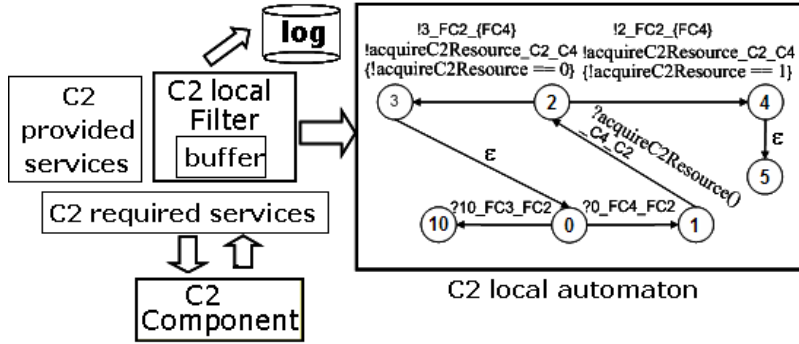


Fig. 10. The behavior of the run-time filters

Local automata are generated by performing two phases: local automata generation and dependencies generation.

In the local automata generation phase each rule of the global automaton is projected on a local automaton. Suppose that the global automaton defines the rule $q' = \delta(q, p)$ and p is an invocation locally observed on the component C . Then this phase adds the rule $q' = \delta(q, p)$, the predicate $P(q, q')(p)$ and the code $C(q, q')(p)$ to the A_C local automaton. In other words, looking at the global automaton, interactions that happen locally on a component C are projected on A_C . For instance in Figure 10 we show the A_{C2} local automaton related to our case study. It is worth noticing that it contains only rules labeled with invocations locally observed on the component $C2$.

The local automata obtained after this phase are not sufficient to realize the correct monitoring. A local automaton A_C can be constituted by disconnected sub-automata. The filter FC cannot be able to choose the right sub-automaton. Moreover, given a sub-automaton it cannot establish the next one. Our solution is to enrich local automaton with dependencies information and to link the sub-automata with ϵ -moves.

A dependency can be of the form $!k(q_1, q_2)(p1)_FC_ \{FC1, \dots, FCn\}$ and $?k(q_3, q_4)(p3)_FCi_FCj$. The former (outgoing dependency) is always related to the A_C rule $q_2 = \delta_C(q_1, p1)$ and is used by the filter FC to inform the filters $FC1, \dots, FCn$ that it has applied such local rule. The latter (incoming dependency) is used by the filter FCj to receive the integer $k(q_3, q_4)(p3)$ sent by the filter FCi .

The dependencies generation phase is used to add transitions labeled with dependencies to the local automata. In the following we sketch the different sub-phases that compose the dependency generation.

In the first sub-phase the dependencies generation considers each state q of the global automaton that is exited by transitions projected on different local automata. Suppose that q is exited by the transitions $q_i = \delta(q, p_i)$, with $1 \leq i \leq n$, that are projected on the n different local automata AC_i . In this case the dependencies generation phase considers each automata AC_i and relates the outgoing

dependency $!k(q, q_i)(p_i)_FCi_ \{FC1, \dots FCn\}$ to its rule $q_i = \delta_{Ci}(q, p_i)$, with $q \neq q_i$. Moreover, the phase considers each filter FCj , with $j \neq i$, and adds to A_{Cj} the rule $q_i = \delta_{Cj}(q, ?k(q, q_i)(p_i)_FCi_FCj)$. Suppose that the local automata of the filters $FC1, \dots FCn$ are in state q and the filter FCi applies the A_{Ci} rule $q_i = \delta_{Ci}(q, p_i)$. In this case FCi has to parse the outgoing dependency related to this rule, i.e., it sends the integer $k(q, q_i)(p_i)$ to the filters $FC1, \dots FCn$. Each filter FCj , with $j \neq i$, can accept the integer by applying the transition labeled with the related incoming dependency (i.e., $q_i = \delta_{Cj}(q, ?k(q, q_i)(p_i)_FCi_FCj)$). In other words, dependencies ensure that filters synchronize with each other so that exactly one q -exiting transition, labeled with an invocation, is accepted. This validates the constraint imposed by the global automaton. In the case that different filters, at the same time, want to apply a q -exiting rule a leader election can be performed to elect the one that will apply its local rule. We point out that synchronization among filters is required only when the states of the applied rules are different.

In the second sub-phase the dependencies generation considers each rule $q' = \delta(q, p)$, with $q \neq q'$, projected on a filter FC and all filters $FC1, \dots FCn$ where a q' -exiting rule has been projected. The phase relates to the rule $q' = \delta(q, p)$ the dependency $!k(q, q')(p)_FC_ \{FC1, \dots FCn\}$ and for each local automaton of the filter FCi , with $1 \leq i \leq n$, defines the rule $q' = \delta_{Ci}(q, ?k(q, q')(p)_FC_FCi)$. Each filter FCi applies this dependency when FC has applied the rule $q' = \delta(q, p)$ and parsed the related dependency $!k(q, q')(p)_FC_ \{FC1, \dots FCn\}$. In this way the filters $FC1, \dots FCn$ synchronize to the state q' and the ordering imposed by the global automaton is respected, i.e., q' -exiting rules can be applied only after the q -exiting rule is applied.

Finally, ε -moves can be added to each local automaton in order to correctly link eventually disconnected states.

The FC -filter activities are similar to the ones of the logically centralized monitoring system. It checks that both C local invocations and incoming dependencies verify the policy defined by the local automaton. It has a buffer where it can store all C -local invocations and all incoming dependencies. Moreover, it can undertake all reactions policies defined by the logically centralized monitoring system. In the following we sketch the FC filter activities by assuming that its A_C local automaton is in state q .

Suppose that the filter FC picks the invocation $!servicName(parameters)_C_S$ from its buffer, such invocation labels a q -exiting transition and verifies the related predicate. Then FC updates the A_C state, forwards the invocation to the filter FS and parses the dependencies (if any) related to such rule (*send invocation acceptance*).

Suppose that the filter FC picks the invocation $?servicName(parameters)_S_C$ up from its buffer, such invocation labels a q -exiting transition and verifies the related predicate. Then FC updates the A_C state, forwards the invocation *serviceName* to component S and parses the dependencies (if any) related to such rule (*receive invocation acceptance*). We point out that when the service is

synchronous the filter waits for the service answer and puts it back in its local buffer.

The filter forwards without any check, invocations that are not defined inside the global automaton alphabet (*forwarding action*). It puts back in the buffer, invocations that cannot be accepted in the current state q , but can be accepted in a state reachable from q (*buffering action*). It accepts each incoming dependency that labels a transition exiting from the current automaton state (*incoming dependency acceptance*). Finally, FC locally detects the anomalous interactions when: (i) the invocation cannot be accepted in the current automaton state (e.g. q) and in any state reachable from q ; (ii) the invocation was buffered and not consumed after a finite amount of time; (iii) the invocation is related to services not present in the interfaces definitions; (iv) an incoming dependency cannot be accepted (*anomaly detection action*).

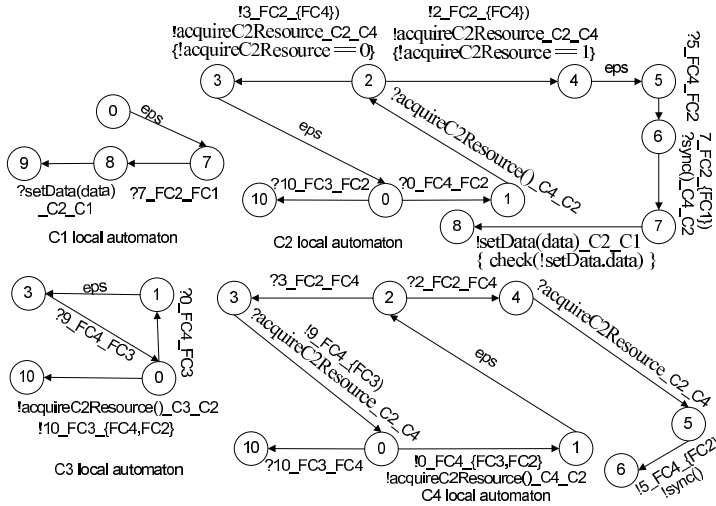


Fig. 11. The local automata

In Figure 11 we show part of the local automata related to our case study where we denote with eps an ε move. When the distributed monitoring system starts, all local automata are in state 0. Filter $FC3$ and $FC4$ synchronize so that exactly one of them performs its local invocation. Suppose that $FC4$ gains the right. Under this assumption $FC4$: (i) sends the integer 0 to the filters $FC3$ and $FC2$; (ii) sends the message $\text{acquireC2Resource}()$ to the filter $FC2$; (iii) changes the local automaton state to 1. Both filters $FC2$ and $FC3$ receive the dependency 0 and move to the state 1. We remark that any $FC3$ invocation has to be buffered so that mutual exclusion is ensured. We can observe that the service $\text{!setData(data)}_C2_C1$ can be provided after the chain of invocations $\text{!acquireC2Resource}()_C4_C2$, $\text{?acquireC2Resource}()_C4_C2$, $\text{!acquireC2Resource}()_C2_C4$, $\text{!sync}()$, $\text{?sync}()$ is performed.

We remark that the local automata generation and the filters generation is hidden to the user by the DESERT tool. The user has to describe the centralized specification (i.e., system model and global automaton) and apply the DESERT tool in order to generate the filters system. As it is shown in [13,12] the filters simulate the logically centralized monitoring system. Filters realize a peer-to-peer monitoring that enhances security and fault tolerance w.r.t. the centralized implementation. This is consequence of the fact that a distributed implementation has not a single point of vulnerability. Moreover, when a filter fails unrelated filters can continue their activities.

7 The Case Studies

In this section we show different case studies where we have applied the DESERT tool. These case studies are related to different applications that run on both mobile and wired infrastructures.

In the following we sketch how DESERT can be applied to enhance the security in a component based application.

Component based software development (CBSD) aims to build a system from existing components. In contrast to traditional development, where system integration is often a marginal aspect, component integration is the centrepiece of CBSD. Developers have to face problems of components adaptation and ensure an acceptable security and dependability level. It is a widely accepted fact that components integration problems cannot be always addressed at development time. Components can be poorly documented so that the integration developers can make mistakes in the integration process. Components can contain bugs or malicious code, therefore security flaws are introduced. Components can be employed not exactly in the contexts for which they are intended, therefore, faults are introduced at the integration level. A component may have more functionalities than the developers know about and so he/she cannot understand the implications of introducing the component inside the system (see [3,9] for an extended survey).

Unsolved integration problems often result in the possibility of anomalous components interactions⁵. The DESERT tool can be used to generate monitoring systems providing a further layer that enhances the security of the integration code.

For instance in the case study presented in this paper we have monitored the components to ensure the consistency of water flow data. In this case study components are written in java so that tools to obtain the source code can be

⁵ Notice that very often it is not possible to establish the nature of the component anomalous interactions. As it is described in [10] an external observer cannot distinguish when a component is interacting in anomalous way as a consequence of malicious intents or internal fault. However, from our point of view we can deal with such violations with different reaction policies. In particular, when the security is a crucial aspect we can isolate anomalous components. In the case that the fault tolerance must be enhanced we can recover such components.

used. This permits to analyze the components logic and produce rogue implementations that exploits bugs and overcomes the static security measures. In the simulation phase rogue clients were produced in order to: (i) exploit the components vulnerabilities; (ii) perform unauthorized access; and (iii) simulate race conditions. Furthermore, malicious clients were used to obtain fake water flow data. Local filters were able to discover such anomalous behavior (see [13] for details), apply the DESERT shunning reaction (see Section 5 for details), isolate the attackers and alert the system manager.

In [2] we have used DESERT to automatically assemble a set of components. In this context, one of the main goals is to compose and eventually adapt loosely coupled independent components to make up a system [22,23]. Building a distributed system from reusable or COTS components introduces a set of problems, mainly related to compatibility and communication. Often, components may have incompatible or undesired interactions. One widely used technique to deal with these problems is to use adaptors. They are additional components interposed between the components forming the system that is being assembled. The intent of the adaptors is to moderate the communication of the components in a way that the system complies only to a specific behavior. In [2] we use the *SYNTHESIS* tool to produce a global automaton specification (i.e. a centralized adaptor) that forces the system to exhibit only a set of *safe* or *desired* behaviors. For example, the adaptor forces the system to exhibit only the subset of deadlock-free and/or explicitly specified wanted behaviors. Such specification is automatically distributed and implemented by the DESERT tool by using the retry policy. In this case the monitoring system acts like an enforcement mechanism that ensures the policy described by the global automaton. The retry policy is used to enhance the system fault tolerance since it allows the anomalous components to continue their execution.

In [1] we have used DESERT to provide intrusion detection facilities in the the CoP protocol [24]⁶. CoP is a protocol used for routing on mobile wireless sensor networks (WSNs).

In the following we summarize attacks that can be undertaken in wireless sensor networks (see [25] for an extended survey).

1. *Compromised Node*: Due to an external intervention, a sensor may be compromised and can be used to subvert the correct WSN behavior.
2. *False Node*: Additional fake nodes could be thrown in the sensed area sending false data or blocking the passage of true data.
3. *Node Malfunction or Outage*: A node in a WSN may malfunction and generate inaccurate or false data or it could just stop functioning hence compromising used paths.
4. *Message Corruption*: Attacks against the integrity of a message occur when an intruder inserts itself between the source and the destination and modifies the contents of a message.

⁶ The research was partially funded by the European project COST Action 293, "Graphs and Algorithms in Communication Networks" (GRAAL). Preliminary results contained in this paper appeared in the [1].

5. *Denial of Service*: A denial of service attack may take several forms. It may consist in jamming the radio link or it could exhaust resources or misroute data.

Generally speaking, state machines can be used to face the above attacks (see [8,12,4] for details). For instance, messages corruption can be avoided by means of the predicates that define the correct message format. Denial of service can be detected by bounding the number of messages in each automaton path. Moreover, the automaton paths permit to describe the correct ordering among invocations. In the following we show how the DESERT tool has been used to address some of the above attacks by means of the CoP protocol [24]. To this extent we have enhanced the DESERT definition language with invocations that can contain the component type names. Suppose that C and S are two different types of components and their interfaces are defined by using the DESERT notation (see 3 for details). The invocation $!serviceName(X_1, \dots, X_n)_{-C-S}$ defines that one of the possible instances of the component C is sending the $serviceName$ asynchronous invocation to the component instances of the type S . The invocation $?serviceName(X_1, \dots, X_n)_{-C-S}$ defines that a set of instances of the component S can receive the asynchronous invocation of the $serviceName$ service.

In the field of location-awareness and clustering protocols like CoP, we model a mobile WSN by a set of sensors $AH = \{s_1, s_2, \dots, s_k\}$. Let $L \subseteq AH$ be a subset $\{l_1, l_2, \dots, l_m\}$ of sensors identifying the clusterhead of a given protocol P . In other words, the sensors in L characterize a set of areas $Ar = \{Ar_1, Ar_2, \dots, Ar_m\}$ (clusters) where each area Ar_i represents the portion of the sensed area where the corresponding l_i plays the clusterhead role. There can be different roles according to P , let $R = \{C_1, C_2, \dots, C_n\}$ be the set of roles. Each C_i has associated an interface that characterizes all messages sent/received by sensors playing that role. It is worth noticing that in this case role is used as synonymous of component type.

In Figure 12 we show the four types of roles that each sensor can play and the corresponding interfaces according to the described CoP protocol. Considering an area denoted with A_{VGN} we define the Out-range, the In-range and the Clusterhead roles that are played by sensors residing in it, and the Extern role representing the sensor playing the clusterhead role inside an adjacent A_{VGN} area.

The role Out-range models a sensor s located in a position that is inside the A_{VGN} but at a distance greater than ds (ds is a natural number) from the A_{VGN} center. This role defines an interface composed by the $?pos(double\ x, double\ y).void$ and $!send(double\ x, double\ y, int\ dest, char[]\ msg).void$ asynchronous services. $?pos(double\ x, double\ y).void$ specifies that the sensor s can receive the incoming message $pos(double\ x, double\ y)$ used to set up its initial position. The parameters $double\ x$ and $double\ y$ are suitably instantiated with the coordinates of s . $!send(double\ x, double\ y, int\ dest, char[]\ msg).void$ specifies that s can send the message msg towards the sink $dest$. The parameters x and y are instantiated with the current position of s and $dest$ is an integer that denotes a sink.

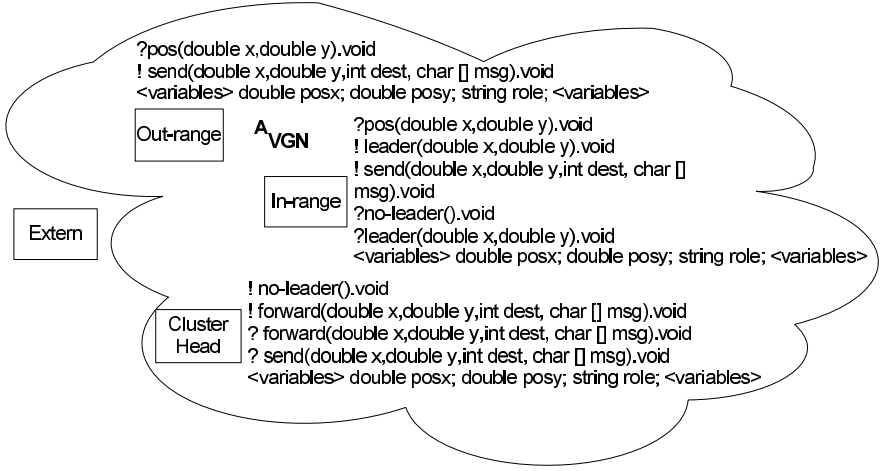


Fig. 12. The CoP roles

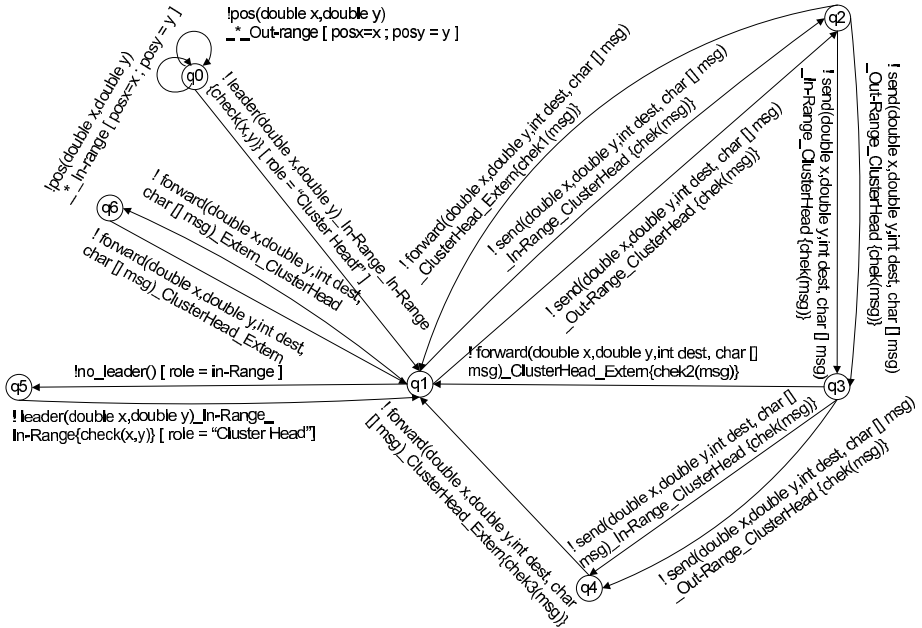


Fig. 13. Global Automaton

The role In-range models a sensor s located in a position inside an A_{VGN} and at distance at most ds from the corresponding center. This role adds to the Out-range role the following services: $!leader(double\ x, double\ y).void$, $?no-leader().void$ and $?leader(double\ x, double\ y).void$. The service $!leader(double\ x, double\ y).void$ specifies that the sensor s can send the message $leader(double\ x, double\ y)$ in order to become clusterhead. The parameters $double\ x$ and $double\ y$ are suitably instantiated with the coordinates of s . $?no-leader().void$ is implemented by s in order to accept the notification sent by the clusterhead when it leaves its role. $?leader(double\ x, double\ y).void$ is used by s to receive the notification of a sensor s' that requires to be clusterhead. The parameters $double\ x$ and $double\ y$ are suitably instantiated with the coordinates of s' .

The role Clusterhead is played by a sensor s providing the forward of messages towards the right sink. This role defines the following services: $!no-leader().void$, $?forward(double\ x, double\ y, int\ dest, char\ []\ msg).void$, $!forward(double\ x, double\ y, int\ dest, char\ []\ msg).void$ and $?send(double\ x, double\ y, int\ dest, char\ []\ msg).void$. The $!no-leader().void$ service specifies that the sensor s can send the asynchronous message $no-leader()$ to the environment. This message is sent by s in order to leave its clusterhead role due to its movement or to its draining battery. $?forward(double\ x, double\ y, int\ dest, char\ []\ msg).void$ implements the service used by s in order to receive the message msg . This message is forwarded by a clusterhead s' that resides in an area surrounding the one of s . The parameters x and y denotes the position of the clusterhead s' and $dest$ encodes the sink. The service $!forward(double\ x, double\ y, int\ dest, char\ []\ msg).void$ is used by s in order to forward the message msg towards the sink $dest$. The parameters x and y denotes the position of s . The service $?send(double\ x, double\ y, int\ dest, char\ []\ msg).void$ is used by s in order to receive a message msg sent by a sensor s' residing inside the A_{VGN} . The parameters x and y denotes the position of s' and $dest$ is an integer that denotes a sink.

The role Extern models one of the clusterheads surrounding the current A_{VGN} .

All roles have associated the real numbers x and y used to store the current position of the sensor and the string $role$ that encodes the current role played by the sensor.

Starting from the description of the CoP protocol we now point out some basic properties that should be guaranteed in order to obtain a fair behavior of the protocol.

1. For each area A_{VGN} there must be at most one sensor playing as clusterhead.
2. When a finite amount of data has been collected by a clusterhead, it must be forwarded in the correct direction.
3. A clusterhead that changes its status to normal sensor due to a movement or because of the draining battery has to forward all the collected messages before its movement.
4. All messages forwarded by a clusterhead have to be received by the clusterhead of the adjacent VGN area.

5. When a clusterhead leaves its role a new sensor (if any in the area) has to take its role.

We formalize these properties by defining a state machine that will be given in input to our tool in order to produce the distributed “patch” for the sensors participating in the CoP protocol.

Figure 13 shows the Global Automaton related to the sensors based system of Figure 12. This automaton defines the correct sequences of messages inside each AV_{GN} . At the beginning (state $q0$) all the sensors are informed about their positions⁷. According to their position, each sensor sets its local variable *role*. The In-Range sensors candidate themselves to become leader. Once the ClusterHead has been elected, the system moves to state $q1$ and the real interaction can start. This transition, in practice, realizes property 1. Property 2 is realized by the path $q1, q2, q3$. In this example we fixed the “finite amount of data” by means of a maximum of three collected messages after which the clusterhead necessarily forwards them. Property 3 is realized by means of transition $q1, q5$, in fact, if the system state is $q1$ there are no messages stored in the clusterhead. When data is forwarded, it is received by the Extern role, i.e., some clusterhead of another AV_{GN} on the way to the specified sink.⁸ And this realizes property 4. Finally, property 5 is valid by means of transition $q1, q5$. From $q5$, in fact, a new ClusterHead must be elected before any other communication can occur.⁹ Note that, when a ClusterHead receives a forward (transition $q1, q6$), it necessarily has to forward it (transition $q5, q1$).

Concerning the predicates, $check1(msg)$ is used to verify the correct format of the message msg forwarded by the ClusterHead. This predicate permits to check that msg is not a buffer overflow attack. The $check2(msg)$ is similar to the above one, however it adds a test verifying that msg is equivalent to the compression of the two messages previously received by the ClusterHead. The predicate $check(x, y)$ verifies that the leader is at distance at most ds from the AV_{GN} center.

We have used the DESERT tool to automatically generate a filter for each sensor. Each filter is constituted by a few lines of code installed in the sensors. This realizes a distributed system that locally detects violation of the sensors interactions policies and is able to minimize the information sent among sensors in order to discover attacks across the network.

⁷ It is worth notice that the position is informed by means of the invocation *pos* that exits from the state 0. In particular this invocation is performed by a component not modeled inside the system (i.e., the satellite component) therefore we use the symbol ‘*’ in the sender field.

⁸ Note that, in our example, while a clusterhead is collecting messages (i.e., the system is either in $q2$ or $q3$ or $q4$), it is not allowed to receive a forward. This, in fact, can happen only at $q1$. In order to not waste messages, this means that, according to the scheduling at the MAC layer, there is some time that is a priori set up. During such a time a clusterhead can wait for other messages without incurring in any forward.

⁹ Again, in order to not waste forward messages we may think of a buffer for the In-Range roles in which a forward is temporarily stored till a new ClusterHead is elected.

In [1] we show how our method affects the performance of the CoP protocol. The experiments are performed running the powered protocol over hundreds of random instances of mobile WSNs. We show the overhead in terms of consumed energy and in terms of performed instructions by the filtered sensors. The experiments also show the estimated percentage reduction of the network lifetime respect to the original CoP protocol.

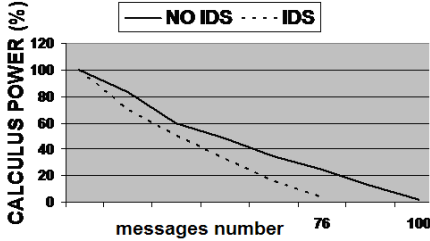


Fig. 14. Average of the residual computational power depending on messages exchanged inside an $AVGN$

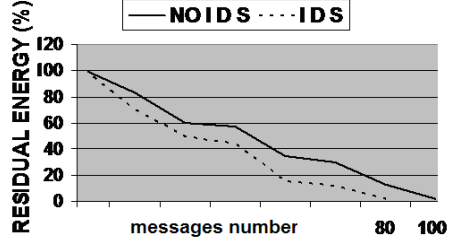


Fig. 15. Average of the residual energy depending on messages exchanged inside an $AVGN$

In Figure 14 we show the lifetime of the system inside an $AVGN$. Considering each kind of message of the sensors as a different set of instructions, we show the overhead in terms of percentage of computational power loss. The cost of ensuring the normal protocol behavior in terms of number of instructions is increased, on average, around 24%. Notice that transmission/receptions operations are much more expensive than local computations. According to the consumption values expressed in [26,24], transmitter and receiver electronics consume an equal amount of energy per bit, namely $5nJ/bit$. While the energy to support the signal above some acceptable threshold against power attenuation caused by the distance is just $100pJ/bit/m^2$.

In Figure 15 we show that, on average, the percentage of the draining of the sensors batteries inside an $AVGN$ is increased by around 20%.

Concerning the detection of attacks we detect any behavior that violates the property expressed by the global automaton. We use the DESERT shunning policy in order to isolate the attacker. In particular each malicious node is isolated by its filter and by the filters of all surrounding nodes¹⁰.

8 Conclusions and Future Works

In this work we presented the DESERT tool that allows the automatic generation of distributed monitoring systems for enhancing security and dependability at architectural level.

¹⁰ When the filter of a sensor is compromised other surrounding filters detect and discard the anomalous invocation. Moreover, they observe with each other in order to send exactly one alert towards the sink.

An architectural level definition language permits to specify both the system model and the correct system behavior. The system model is provided by means of an interface description language. The correct system behavior is provided by means of state machines. These 'centralized' specifications are used by the front-end and the back-end of the DESERT tool in order to generate a distributed monitoring system implementation. The monitoring system is constituted by one filter for each component that locally detects violation of the global specification.

DESERT has been used for applications running on both mobile and wired infrastructures.

In future work we are developing more complex detection and recovery mechanism. It can happen that the retry reaction policy causes a condition in which the same component retry several times as a consequence of the same condition error. Moreover, very often the component generating the anomalous behavior does not always correspond to the source that triggered the error¹¹. In both cases we exploit the integration level view provided by the global automaton. We analyze the state reached in the global computation so that we can identify the sources of errors and we can recover several distributed components.

References

1. Inverardi, P., Mostarda, L., Navarra, A.: Distributed IDSs for enhancing security in mobile wireless sensor networks. In: IEEE International Workshop on Pervasive Computing and Ad Hoc Communications (IEEE PCAC'06), IEEE Computer Society Press, Los Alamitos (2006)
2. Inverardi, P., Mostarda, L., Tivoli, M., Autili, M.: Automatic synthesis of distributed adaptors for component-based system. In: Proceedings of the 21st Automated Software Engineering (ASE) Conference (2005)
3. Lindqvist, U., Jonsson, E.: A map of security risks associated with using cots. *Computer* 31, 60–66 (1998)
4. Orset, J.M., Alcalde, B., Cavalli, A.: An EFSM-based intrusion detection system for ad hoc networks. In: Peled, D.A., Tsay, Y.-K. (eds.) ATVA 2005. LNCS, vol. 3707, Springer, Heidelberg (2005)
5. Ko, C., Ruschitzka, M., Levitt, K.: Execution monitoring of security-critical programs in distribute system: A specification-based approach. *IEEE* (1997)
6. White, G.B., Fisch, E.A., Pooch, U.W.: Cooperating security managers: A peer-based intrusion detection system. *IEEE Network* (1996)
7. Stillerman, M., Marceau, C., Stillman, M.: Intrusion detection for distributed applications. *Communications of the ACM* (1999)
8. Eckmann, S.T., Vigna, G., Kemmer, R.A.: Statl: An attack language for state-based intrusion detection. *Journal of Computer Security* 10, 71–104 (2002)
9. de Lemos, R., Gacek, C., Romanovsky, A.: Architectural Mismatch Tolerance. In: *Architecting Dependable Systems*. LNCS, vol. 2677, pp. 175–196. Springer, Heidelberg (2003)
10. Avizienis, A., Laprie, J., Randell, B., Landwehr, C.: Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transaction on Dependable and Secure Computing* 1, 11–33 (2004)

¹¹ In this case a severe reaction policy would terminate several components.

11. Delgado, N., Gates, A.Q., Roach, S.: A Taxonomy and Catalog of Runtime Software-Fault Monitoring Tools. *IEEE Transactions on Software Engineering* 30, 859–871 (2004)
12. Inverardi, P., Mostarda, L.: A distributed intrusion detection approach for secure software architecture. In: Morrison, R., Oquendo, F. (eds.) *EWSA 2005. LNCS*, vol. 3527, pp. 168–184. Springer, Heidelberg (2005)
13. Mostarda, L.: Distributed detection systems for secure software architectures, Ph.D, Thesis in computer Science, University of L'Aquila (2006)
14. Porras, P.A., Neumann, G.P.: Event monitoring enabling responses to anomolous live disturbances. *Proc. of 20th NIS Security Conference* (1997)
15. Snapp, S.R., Dias, J.B.G.V., Goan, T., Heberlein, L.T., Ho, C., Levitt, K.N., Mukherjee, B., Smaha, S.E., Grance, T., Teal, D.M., Mansur, D.: Dids (distributed intrusion detection system) - motivation architecture and early prototype. In: *Proc. 14th National Security Conference* vol. 1, pp. 361–370 (1997)
16. Vigna, G., Kemmerer, R.A.: Netstat: a network-based intrusion detection system. *Journal Computer Security* 7, 37–71 (1999)
17. Javitz, H.S., Valdes, A.: The nides statistical component description and justification. Technical report - Columbia University (1994)
18. Vaccaro, H., Liepins, G.: Detection of anomalous computer session activity. In: *Proc. of the 1989 Synopsium on Security and privacy*, pp. 280–289 (1989)
19. Sen, K., Vardhan, A., Agha, G., Rosu, G.: Effecient decentralized monitoring of safety in distributed system. In: *ICSE* (2004)
20. Schneider, F.B.: Enforceable security policies. *ACM Trans. on Information and System Security* 3, 30–50 (2000)
21. European Commision 6th Framework Program - 2nd Call Galileo Joint Undertaking: Cultural Heritage Space Identification System (CUSPIS), <http://www.cuspisproject.info>
22. Crnkovic, I., Larsson, M.: Building reliable component-based Software Systems. Artech House, Boston, London (2002)
23. Szyperski, C.: Component Software: Beyond Object-Oriented Programming. Addison-Wesley, Reading (2004)
24. McCann, J.A., Navarra, A., Papadopoulos, A.A.: Connectionless Probabilistic (CoP) routing: an efficient protocol for Mobile Wireless Ad-Hoc Sensor Networks. In: *IPCCC* (2005)
25. Perrig, A., Stankovic, J., Wagner, D.: Security in wireless sensor networks. *Commun. ACM* 47, 53–57 (2004)
26. Heinzelman, W., Chandrakasan, A., Balakrishnan, H.: Energy-Efficient Communication Protocols for Wireless Microsensor Networks. In: *Proc. of the Hawaiian Int. Conf. on Systems Science* (2000)

Architecting Dynamic Reconfiguration in Dependable Systems

Antônio Tadeu A. Gomes¹, Thais V. Batista², Ackbar Joolia³, and Geoff Coulson³

¹ Laboratório Nacional de Computação Científica (LNCC)

Av. Getúlio Vargas 333, 25651-075 Petrópolis-RJ, Brazil

² Universidade Federal do Rio Grande do Norte (UFRN)

Departamento de Informática

Campus Universitário – Lagoa Nova, 59072-970 Natal-RN, Brazil

³ Computing Dept, Infolab21, Lancaster University

Lancaster LA1 4WA, UK

atagomes@lncc.br, thais@ufrnet.br, joolia@comp.lancs.ac.uk,
geoff@comp.lancs.ac.uk

Abstract. The need for dynamic reconfiguration is a complicating factor in the design of dependable systems, as it demands from software architects both rigour and planning. Although recent research has shown that systematic and integrated “specification-to-deployment” environments are promising approaches to architecting dependable systems, few proposals have yet considered dynamic reconfiguration, and then only in specific situations. In this paper, we propose a generic approach to supporting dynamic reconfiguration in dependable systems. The proposed approach is built on our view that dynamic reconfiguration in such systems needs to be causally connected *at runtime* to a corresponding high-level software architecture specification. In more detail, we propose two causally-connected models: an architecture-level model and a runtime-level model. Dynamic reconfiguration can be applied either through an architecture specification at the architecture level, or through reconfiguration primitives at the runtime level. Both foreseen and unforeseen reconfigurations are supported. We discuss the issues involved in handling these two types of reconfiguration at both levels and the mapping between them. We also discuss an implementation of our approach that evaluates its main benefits.

Keywords: dependable systems, dynamic reconfiguration, specification-to-deployment environments.

1 Introduction

Computer systems are an integral part of everyday life. Therefore, dependability—along with its various dimensions (availability, reliability, safety, integrity, maintainability, and security)—has become a key issue in the design of such systems. Until recently, the concerns associated with dependability have been mostly on system design and implementation, and less attention has been paid to the need for the

dynamic reconfiguration of dependable systems. However, this is set to change. For example, network systems design and autonomic computing are two application areas in which dependability and dynamic reconfiguration must be taken into account together. Modern network systems, such as access and backbone routers, have strong requirements for both 24x7 operation and managed software evolution (*e.g.* to allow network operators to dynamically deploy new QoS/ resource management and security strategies) [11]. And in autonomic systems such as self-organized sensor networks [2], minimising the degree of human-manned reconfiguration is the central tenet.

The need for dynamic reconfiguration is a complicating factor in building dependable systems (irrespective of the reasons why dynamic reconfiguration is necessary), as it demands from software architects both appropriate rigour and planning for adaptability. Recent research has shown that systematic and integrated “specification-to-deployment” environments are promising approaches to architecting dependable systems; but few proposals have yet considered dynamic reconfiguration, and then only in particular application areas such as embedded systems [36], or have focused on specific aspects such as adaptive fault recovery [15]. More specifically, previous work in this area has not yet considered a more comprehensive integration (and possible extension) of classical software architecture concepts, such as architectural styles [13] (we discuss the concepts of software architecture and architectural style in Section 2.2 below), and the most recent developments in reconfigurable runtime technologies, such as the adoption of reflective or aspect-oriented programming techniques [23,24].

In this paper, we propose just such a generic approach to architecting dynamic reconfiguration in dependable systems. The proposed approach is built on our view that dynamic reconfiguration needs to be causally connected *at runtime* to a corresponding high-level architecture specification. In more detail, we propose two causally-connected models, as depicted in Fig. 1: a *runtime-level model* and an

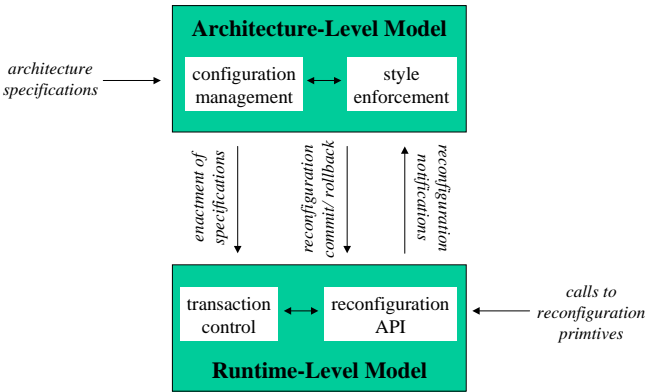


Fig. 1. The architecture- and runtime-level models

architecture-level model (Fig. 1 also depicts some of the services—e.g. the configuration management service—that we discuss later in the paper). Dynamic reconfiguration can be applied either through reconfiguration primitives at the runtime level or through architecture specifications at the architecture level—using, for instance, general-purpose architecture description languages (ADLs) or domain-specific languages (DSLs). Moreover, we argue that both foreseen and unforeseen reconfigurations should be supported.

Our previous publications [3,20] have focused on the design and implementation of our approach using two mature technologies—the OpenCOM reflective component runtime [10] and the Acme ADL [14]. By providing a less technology-grounded overview of our approach, we aim in this paper at thinking forward about the more stringent requirements for dynamic reconfiguration in dependable systems. The remainder of the paper is structured as follows. The key elements of our approach to dynamic reconfiguration in dependable systems at the runtime and architecture levels (and causality relationships between them) are presented in Section 2. The issue of handling foreseen and unforeseen reconfigurations at both levels is then discussed in Section 3. Sections 4 and 5 summarise our implementation efforts to date, including a detailed case study, and is therefore an up-to-date account of our previous work [3,20]. Section 6 is devoted to a discussion of related work, and Section 7 provides concluding remarks.

2 Dynamic Reconfiguration at the Runtime and Architecture Levels

The essential elements of our approach are summarised in the the following list (see also Fig. 1) and discussed in detail in the following two sub-sections:

- *reconfiguration primitives* – used to initiate reconfiguration requests at the runtime level;
- *reconfiguration notification service* – used to support causal connection between the runtime and architecture levels by enabling the former to notify the latter of any changes arising from calls to the reconfiguration primitives;
- *reconfiguration transaction service* – used to atomically group sequences of calls to the reconfiguration primitives and also to enable rollback in aborted reconfigurations;
- *mapping from architecture-level concepts to runtime-level concepts* – used to formalise the correspondences between the two levels;
- *configuration management service* – used to map architecture level reconfiguration specifications onto sequences of calls to the reconfiguration primitives;
- *style enforcement service* – used to embody and enforce rules and constraints on reconfigurations specified at the architecture level.

2.1 Runtime Level

A runtime defines a deployment and execution environment for software systems. Runtime-level models are becoming quite sophisticated in their capabilities for dynamic reconfiguration, providing (among other services) primitives that allow inspecting and adapting the structure of a system (*i.e.* its configuration) and its behaviour. Examples of such primitives—referred to as **reconfiguration primitives** in this paper—include addition, removal, and replacement of software elements, (dis)connection between such elements, (dis)closure of element interfaces, etc. Such reconfiguration primitives can be used to introduce dependable functionality in the system.

To motivate the functionality that should be provided at the runtime level, consider a dependable web-based client-server application. In a very simple configuration, two servers (primary and secondary) are involved. If the primary server is no longer available, the client is automatically switched to the secondary server. For this runtime-level reconfiguration to be causally connected with the architecture level, the runtime should provide a **reconfiguration notification service** to inform the architecture level about the calls to the reconfiguration primitives that are involved in the client switching.

The runtime should also be able to manage complex reconfiguration operations, such as the client switching reconfiguration in the aforementioned example. This specific reconfiguration operation involves two related calls to reconfiguration primitives: a disconnection from the primary server, followed by a connection to the secondary one. For such cases we argue that the runtime-level model should provide a **reconfiguration transaction service**, whose aim is twofold: (i) to provide the architecture level with a view of multiple calls to reconfiguration primitives as generating a single reconfiguration notification (an example of this is provided in Section 2.2); and (ii) to keep track of the calls to the reconfiguration primitives involved in a reconfiguration operation so that a rollback can be done if such operation is invalidated at the architecture level. This service is therefore essential for a proper dependability support as it prevents the system to be in an invalid state both at architectural and runtime levels.

Finally, a runtime must preserve the safety and integrity of a dependable system even within single calls to reconfiguration primitives, especially where multithreading is employed. To understand this, consider again the client switching reconfiguration. In this situation, we need to suspend all threads issuing new requests from the client to the primary server, and wait for the completion of the currently outstanding requests before actually disconnecting these elements.

2.2 Architecture Level

Software architecture modelling plays an important role in the development of dependable software. An architecture specification uses high-level representations such as ADLs or DSLs to embody information about a system's configuration while abstracting away details about the internal implementation of its elements. Such a specification allows the software architect to reason about structural properties early in the development process, fostering dependable and extensible designs. Typically,

an architecture specification also describes some principles governing the system's design and evolution. In our view, architectural specifications and styles must also consider and enforce dependability requirements. Architectural styles [13] are the most common way to describe such principles in software architecture modelling. These are defined in terms of formally-described software element types, as well as rules that govern the composition of such elements. For instance, a pipeline style might comprise a basic set of pipe and filter element types and rules that permit only non-circular compositions of such elements in a configuration. A configuration following a specific style can thus be seen as an instance of such style at the runtime level.

In our view, *three* main issues should be addressed at the architecture level to allow its causal connection with the runtime level.

First, there must be a precise (complete) **mapping from architecture-level concepts to runtime-level concepts**. Surprisingly, this is a largely overseen requirement in the literature [1].

Second, the architecture level should be able to handle architecture specifications not only during system deployment (*i.e.* an initial configuration that is set up from 'scratch' based on a complete specification) but also over a currently running system (in which case the current specification will typically be a delta from the original configuration). Getting back to our example of a dependable web based client-server application, imagine that the primary-backup configuration can no longer cope with the load from the clients. The system architect then decides to replace the primary server with a more sophisticated implementation that is able to detect an imminent overload and redirect new client calls to the secondary server. This is a typical situation in which the system architect might enact a "partial" specification. We argue that, in such cases, the system architect could be provided with a **configuration management service** that would map (either full or partial) architecture specifications onto reconfiguration primitives to be effected at the runtime level. Moreover, partial specifications should allow for both architectural *construction* and *destruction* (the latter to correctly represent element removals and disconnections at the runtime level).

Thirdly and finally, architectural styles should be extensively used at the architecture level as a means of constraining reconfigurations in running systems. For instance, it would be interesting to associate our dependable web-based application with a runtime impersonation of the client-server style, which would forbid invalid reconfigurations such as clients with dangling connections. In this case, the figure of a **style enforcement service** comes into play, handling style-specific rules governing reconfigurations. A style enforcement service would receive reconfiguration notifications from the runtime level and check whether any of the corresponding calls to reconfiguration primitives have violated the architectural rules defined in the corresponding style. Note that the definition of architectural rules depends on the target functionality domain—*e.g.* a layered style and a client-server style each have their specific constraints—and can vary considerably. To avoid specific controller implementations for each style of interest, a *generic* style enforcement service should be devised. This controller would be configurable with regard to the rules it can enforce. More specifically, *validation scripts* would be used as runtime

representations of architectural rules. Such scripts (implemented, for example, as code in an interpreted language, or as finite state machines) could be loaded to (or unloaded from) the style enforcement service during runtime. “Running” styles could therefore be evaluated and refined/fixed without compromising the availability of the dependable system—as long, of course, as the running configuration did not violate the rules of the modified style.

It is important to bear in mind that a call to a reconfiguration primitive at the runtime level may be part of a complex reconfiguration operation, implementing some dependable functionality, and therefore it is not sufficient for the style enforcement service to check each call in isolation. To understand this, consider a system that follows a pipeline style (*i.e.* a system in which non-contiguous configurations and ‘cycle’ connections are disallowed), such as a software-based router. In a dependable implementation, this router could adapt to new network conditions by inserting a new element (*e.g.* a new packet scheduler) in the middle of the pipeline. This reconfiguration operation could be regarded as valid from the point of view of the currently defined architectural rules. Nevertheless, inserting such an element might involve sequences of distinct calls to reconfiguration primitives in the configuration. For example: (i) the addition of the new element, (ii, iii) disconnections between the two to-be-neighbour elements, and (iv) new connections between these two elements and the new one. Any of the first three calls would appear invalid if treated in isolation. Hence the need for the style enforcement service at the architecture level interacting with the transaction service at the runtime level. The way such interaction is accomplished depends, however, on the type of reconfiguration in question; this issue is discussed in the next section.

3 Handling Foreseen and Unforeseen Reconfigurations

Dynamic reconfiguration can be classified as either *programmed* or *ad-hoc*, according to the moment at which it is specified. Programmed reconfiguration [37] is specified at design time while ad-hoc reconfiguration is unpredictable at design time but can occur at runtime. While the former is important for automatic reconfiguration, the latter is suitable for software maintenance. In this sense, programmed reconfiguration is related to the concept of self-organizing architectures [16,29].

The example (in Section 2) of a dependable web based client-server application illustrates a common situation in which both types of reconfiguration can co-exist at the architecture level. First, consider the simpler configuration involving two servers with automatic client switching on failure. This situation can be foreseen at design time, so an important requirement at the architecture level is to provide constructs to specify such a situation as a programmed reconfiguration. More specifically, to support such type of reconfiguration the architecture-level model must provide *predicate-action* constructs to specify the situations that trigger reconfigurations and which reconfigurations must take place. Now consider the case of the system architect replacing the primary server with the more sophisticated implementation that, on overload, redirects new client requests to the secondary server. This situation has not been predicted at design time; therefore, it is interesting that ad-hoc reconfiguration—in

terms of enacting partial specifications—be available at the architecture level as well. In this case, the style enforcement service is called to analyse the reconfiguration specified in the partial specification, and if the architectural rules are not violated, the system representation at the architecture level is updated and the related reconfiguration operation is executed atomically at the runtime level by the transaction service. Note that, in this situation, the style enforcement service can make the transaction service completely aware of the beginning and end of the reconfiguration operation.

Ad-hoc reconfiguration can be also effected directly at the runtime level. For instance, dependability requirements identified at runtime can be inserted in the application using ad-hoc reconfiguration. The notification service described in Section 2.1 is then crucial to keep the architecture level consistent with the runtime configuration, and vice-versa.

Architecture-level consistency can be attained through an additional *configuration update service*, which would provide causal connection in the runtime-to-architecture direction by updating the system representation at the architecture level. Although this can be of interest to provide full causal connection in both directions—*e.g.* for the purpose of architecture analysis [1]—we expect that in practice most systems will employ *either* architecture *or* runtime level ad-hoc reconfigurations, but not both simultaneously. More specifically, we believe that runtime level ad-hoc reconfiguration is more likely to happen in lower-level system environments—*e.g.* those driven primarily by OS events—whereas architecture level ad-hoc reconfiguration seems more suitable for higher-level functionality. In Section 4 we discuss an implementation of our approach that provides full causal connection in the architecture-to-runtime direction only. This mechanism guarantees that dependable conditions and statements of the architectural level are enforced at the runtime level.

Runtime-level consistency, on the other hand, can be provided through the combination of the runtime-level transaction service and the architecture-level style enforcement service, as in the case of architecture level ad-hoc reconfiguration. In contrast to what happens in the architecture level, however, the style enforcement service cannot make the transaction service aware of which calls to reconfiguration primitives start and finish the reconfiguration operation at the runtime level. Ideally, the transaction service should be also transparent (at least selectively) to someone calling reconfiguration primitives at this level, thus allowing the seamless adoption of our approach in legacy systems.

4 The Plastik Architecture

To experiment with our approach, we have been developing an architecture called Plastik, which extends and causally integrates two existing technologies. The runtime-level technology is based on the OpenCOM reflective component model [10] and the associated notions of reflective meta-models and component frameworks. The architecture-level technology is based on the Acme ADL [14] and the associated

Armani first-order predicate logic (FOPL) constraint language [26]. Both of these technologies are presented below, with a particular emphasis on the model extensions provided by Plastik to underpin the concepts discussed in the previous sections.

4.1 The Runtime-Level

We have chosen OpenCOM as the basis of Plastik's runtime-level model because of its high performance and good support for dynamic reconfiguration in comparison with related technologies [6]. Besides, it has been successfully used over the past few years for building reconfigurable software in a variety of domains [4,11]. Fig. 2 provides a pictorial view of the fundamental entities of OpenCOM's model: *components* (language-independent units of functionality and deployment), *interfaces* and *receptacles* (units of service provision and consumption respectively), *bindings* (runtime associations between interfaces and receptacles), and *capsules* (containers offering runtime kernel services such as (un)loading, instantiating and destroying components, and (un)binding interfaces and receptacles). Other first-class entities in the model—omitted in Fig. 2 for clarity—are *caplets* (nested sub-scopes within capsules), *loaders* (pluggable caplet-specific loading mechanisms), and *binders* (pluggable intra- and inter-caplet binding mechanisms). These provide OpenCOM-based software with capsule distribution transparency, thus playing a crucial role in facilitating the uniform deployment of such software in a wide range of platforms from standard PCs, to resource-poor PDAs, to OS-bare embedded systems, to high-speed network processor hardware.

Above those fundamental entities, the notions of *reflective meta-models* [9] and *component frameworks* (CFs) [34] are extensively employed in OpenCOM to manage and constrain dynamic reconfiguration. The reflective meta-models provide OpenCOM with 'low-level' facilities to programmatically inspect, adapt, and extend diverse system aspects (e.g. configuration topology, call interception and thread management) at runtime. These facilities thus offer openness and flexibility to system developers. CFs, on the other hand, are targeted at system- and application-level functionality. CFs are tightly-coupled sets of components (usually deployed as composites) that work together to address a focused domain of functionality. When deployed as composites, CFs can be also encapsulated in 'outer' CFs. Importantly, CFs in OpenCOM can incorporate policies that determine to what extent the CF can

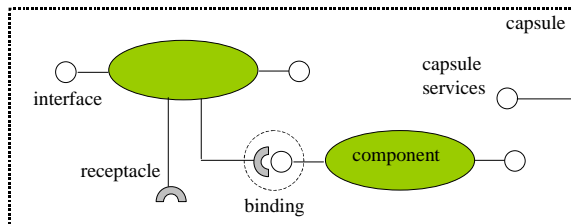


Fig. 2. Fundamental entities of OpenCOM's model

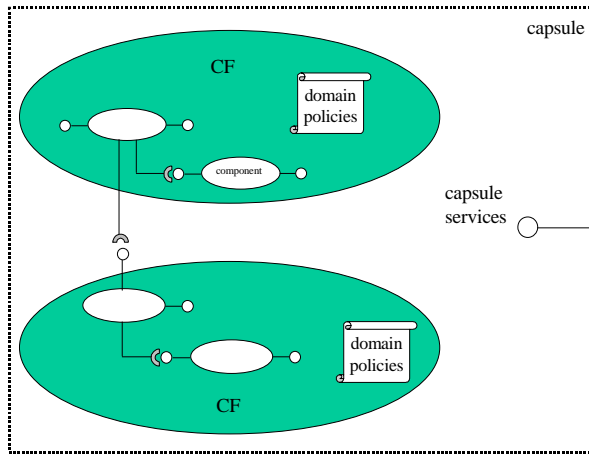


Fig. 3. Component frameworks (CFs) in OpenCOM

be reconfigured at runtime. Fig. 3 illustrates the notion of CFs in OpenCOM. As will be seen, CFs are central in our approach to causally connecting the runtime- and architecture-level models.

Reconfiguration transaction service. This is implemented in OpenCOM by per-CF proxy components—*i.e.* each CF is given an internal, dedicated proxy for the runtime kernel services, as illustrated in Fig. 4. Such a component addresses three issues. First, it provides *inter-CF isolation*; a proxy component prevents other components within one CF from interfering with other CFs because such components cannot access the kernel services directly. Second, proxy components allow for *deferred reconfigurations*; they can defer actual calls to reconfiguration primitives that make up a reconfiguration operation.¹ Deferring such an operation allows a style enforcement service to validate it at the architecture level before the proxy actually commits it (*i.e.* forwards the deferred calls to the actual kernel services), with the possibility of rollback on failure. For all this to be possible the style enforcement service is explicitly notified about pending transactions—refer to the Plastik implementation of the reconfiguration notification service described below. Third, proxy components offer a local *registry service*, which allows arbitrary meta-data—expressed in terms of <key,value> pairs—to be attached at runtime to any OpenCOM element. These are read and potentially modified by any component within the CF (including the proxy).

Reconfiguration notification service. Plastik implements this service based on OpenCOM’s interception meta-model [9]. This allows a programmer to interpose

¹ As mentioned in Section 3, in an ideal scenario the runtime should allow for “implicit” reconfiguration transactions. In our current implementation, however, the proxy component includes transaction primitives for systems to explicitly scope multiple calls to reconfiguration primitives.

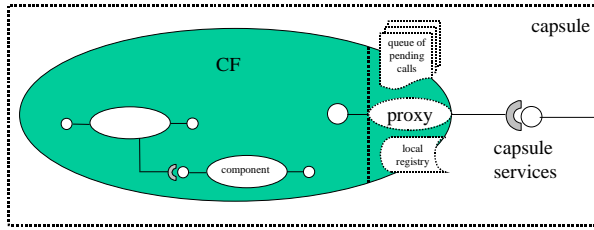


Fig. 4. Per-CF reconfiguration transaction service in OpenCOM

interceptors (small software modules) at bindings between interfaces and receptacles. For the reconfiguration notification service, interceptors are associated with the interfaces provided by proxy components. This permits that actions such as reconfiguration operations and updates of registry entries within a CF be sensed and notified to interested parties. To receive such notifications, interested parties must subscribe to per-CF *notifier* components that manage which subscribers are interested in which notifications. The following subsection provides an example of this service usage at the architecture level.

Safe reconfiguration primitives. Work in this area is not yet complete in the context of the Plastik architecture. However, ongoing work on it has progressed in the context of a related OpenCOM project [31], and we are currently integrating this with Plastik.

4.2 The Architecture Level

We have chosen Acme as the basis of Plastik's architecture-level model because it offers sufficient generality to straightforwardly describe a variety of system structures. Besides, it comes with tools that provide a good basis for designing and manipulating architectural specifications and generating code.

Plastik employs the standard Acme constructs: *components* and *connectors* (elements of computation and interaction), *ports* and *roles* (interfaces of components and connectors respectively), *attachments* (associations between ports and roles), *representations* (internal decompositions of components/ connectors), *properties* (annotations on other Acme constructs), *systems* (configurations of components and connectors), and *families* (architectural styles). Plastik also employs Armani expressions embedded in Acme family specifications to define invariants (architectural rules) over Acme systems.

Fig. 5 provides an example of the specification (and an associated graphical representation) of a simplified software-based router in Acme (the RouterInst system). The example also illustrates the use of the family construct as a means of specifying the main component types involved in a typical router (the Classifier, Forwarder and Scheduler types) and an architectural rule on the pipeline composition of such components (the validTopology rule).

```

Family Router extends PlastikMF with {
  Property Type IPPacket = Record
    [Type: string; IPHdr: string; Payld: string];

  Component Type Classifier: PlastikMF = {
    Port i : ProvidedPort = new ProvidedPort extended with {
      Property packet: IPPacket; }; };
    Port o1, o2 : RequiredPort = new RequiredPort;
  };

  Component Type Forwarder: PlastikMF = {
    Port i : ProvidedPort = new ProvidedPort;
    Port o : RequiredPort = new RequiredPort;
  };

  Component Type Scheduler: PlastikMF = {
    Port i1, i2 : ProvidedPort = new ProvidedPort;
    Port o : RequiredPort = new RequiredPort;
    Property algorithm: string = "FIFO";
  };

  Connector Type connPath: PlastikMF = {
    Role src : ProvidedRole = new ProvidedRole;
    Role snk : RequiredRole = new RequiredRole;
  };

  Rule validTopology = Invariant
    Forall c1,c2:Component in sys.Components |
      (reachable(c1,c2) =>
        (((satisfiesType(c1, Classifier) AND
          satisfiesType(c2, Forwarder))
        OR
          (satisfiesType(c1, Forwarder) AND
            satisfiesType(c2, Scheduler)))
        AND
          (!(satisfiesType(c1,Classifier) AND
            satisfiesType(c2, Scheduler)))))
        AND
          !reachable(c1,c1); // no cycles allowed...
    }; // end family spec

  System RouterInst = new Router extended with {
    Component cls = new Classifier;
    Component fwd = new Forwarder;
    Component sch = new Scheduler;
    Connector CtoF, FtoS = new connPath;

    Attachments {
      cls.o1 to CtoF.src;
      CtoF.snk to fwd.i;
      fwd.o to FtoS.src;
      FtoS.snk to sch.i1;
    };
  }; // end system spec

```

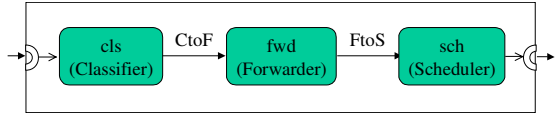


Fig. 5. Router description in Acme

Mapping architecture-level concepts to the runtime level. We address this first issue by defining a set of Acme-to-OpenCOM correspondences, which are shown in Table 1. The system- and representation-to-CF correspondences are central: Acme families naturally correspond to CF specifications as domain-specific units of reusable and dynamically reconfigurable functionality. Notice that, for system-to-CF correspondences, it is always assumed that software systems in OpenCOM are themselves contained in (outermost) CFs. Although somehow restrictive in general, we consider this characteristic particularly interesting for dependable systems. The one-to-one mapping between Acme connectors and OpenCOM components arises from the possibility of easily supporting ‘rich’ connectors (such as SQL statements or RPC channels) as inter-capsule bindings. Acme properties are mapped onto registry entries in the CF proxy component. For the mappings involving OpenCOM interfaces and receptacles, a more precise semantic proved to be necessary on the corresponding Acme constructs. The proposed solution to this was to devise a generic PlastikMF family that defines port and role types identifying units of service provision (ProvidedPort, ProvidedRole) and consumption (RequiredPort, RequiredRole). The rest of the correspondences are intuitively clear from Table 1.

Table 1. Acme-to-OpenCOM correspondences

Acme	OpenCOM
system/ representation	CF
component/ connector	component
port/ role	interface/ receptacle
attachment	binding
property	registry entry (proxy)

Enacting partial specifications and programmed reconfigurations. Plastik extends the classic Acme constructs with the following additional ones:

- an *on-do* construct: this allows software architects to express runtime conditions (using Armani expressions) under which programmed reconfigurations (specified in extended Acme) should take place in an architecture. This construct is used to map dependability requirements identified at specification time to the correspondent programmed reconfiguration operations.
- *detach* and *remove* constructs: these allow software architects to dismantle parts of an architecture by deleting attachments or destroying existing components, connectors or representations.
- a *dependencies* construct: this allows software architects to express runtime dependencies among components/ connectors. This construct itself intrinsically supports dependability as it avoids that a reconfiguration operation leads the system to an inconsistent state where the dependencies between the architectural elements are not available. This can be thought of as a ‘syntactic sugar’ to the on-do construct; it is useful when the runtime condition is a component or connector instantiation and the programmed

configuration should subsequently support MPLS forwarding as well as IPv4 forwarding. Furthermore, the standard FIFO scheduler component is to be replaced by a priority scheduler so that MPLS packets have precedence over IPv4 packets.

Configuration management service. Although conceptually at the architecture level, this service is implemented in Plastik at the runtime level as per-CF *configurator components*. Such components interpret specifications in (extended) Acme to reconfiguration operations that are effected at the runtime level, according to the Acme-to-OpenCOM correspondences presented above. The first task accomplished by a configurator is to initialise its CF with a full specification of an Acme system or representation. This involves establishing the initial state of the CF in terms of loading, instantiating and binding the necessary OpenCOM components, and ensuring that the rest of the runtime machinery (including the other Plastik services) is instantiated. Having established the runtime environment, the configurator processes the Acme family specifications and the on-do and dependencies statements related with the system/ representation, generating scripts to be deployed on Plastik’s style enforcement service (see below). Family specifications are translated to validation scripts, as mentioned in Section 2. Similarly, on-do and dependencies statements are translated to runtime representations called *programmed reconfiguration scripts*. In the case of such statements, the corresponding runtime conditions are registered in the notifier component as events of interest to the style enforcement service. Fig. 7 illustrates these ideas.

Style enforcement service. As depicted in Fig. 7, this architecture-level service is also implemented at the runtime level as per-CF components called *policers*. Such

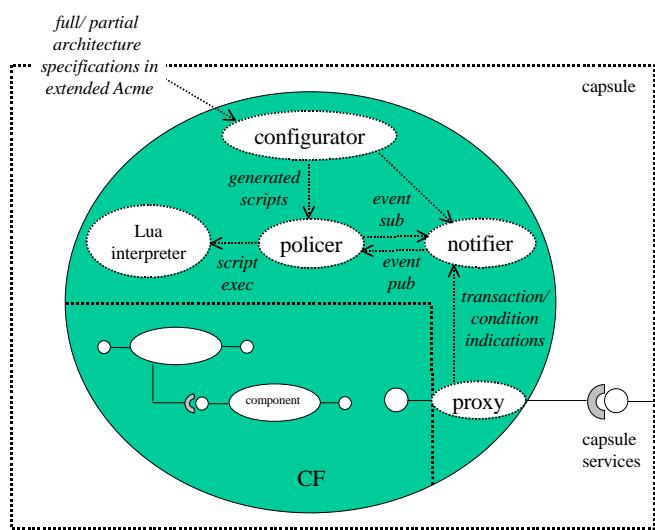


Fig. 7. Per-CF configuration management and style enforcement services in OpenCOM

components host both validation and programmed reconfiguration scripts. Whenever a proxy notifies a policer about a pending transaction (see Subsection 4.1 for a discussion of transactions), the latter executes the validation scripts to ascertain the correctness of the corresponding reconfiguration operation, committing or aborting it accordingly. Similarly, programmed reconfiguration scripts effect reconfiguration operations in response to notifications about runtime conditions of interest.

Both validation and programmed reconfiguration scripts are uniformly implemented as programs in the interpreted language Lua [19]. We have chosen Lua because it provides fast code interpretation, high flexibility, and a small footprint (around 100k). This last feature is particularly important since Plastik is designed to be potentially deployable in resource-poor platforms.

5 Case Study

To evaluate our approach, we have experimented with the architectural specification of a telemedicine system called AToMS (AMI Teleconsultation and Monitoring System) [18]. This system aims at fostering the ubiquitous adoption of pre-hospital drug treatment for patients with acute myocardial infarction (AMI) in Brazil. Crucially, only an specialist (cardiologist) can diagnose an AMI patient as eligible to drug treatment, based on the result of an eletrocardiogram (ECG) and the patient's clinical state. The AToMS system helps emergency teams (located at the place the first assistance is delivered—*e.g.* in an ambulance) and specialists (remotely located at a teleconsultation center) to exchange information about the patient and decide on his/her eligibility for promptly receiving the drug treatment. In this system, emergency teams carry portable devices communicating with ECGs that collect monitored data from the patient. The ECG data is sent to the teleconsultation center together with information about the patient's clinical state through a wireless network (*e.g.* GPRS, WiFi mesh, or WiMAX). The teleconsultation center puts the emergency team promptly in contact with available cardiologists, which can then remotely decide on the applicability of drug treatment based on the data provided to them.

5.1 Architecture Baseline

Fig. 8 presents a pictorial representation of the AToMS software architecture. A partial description of such architecture in Acme is given in Fig. 9. As can be seen from these figures, the system follows a pub-sub style at its outermost level. The ETSubsys and VCSSubsys components (representing the emergency teams and the teleconsultation center, respectively) are internally composed as client-server subsystems. Within ETSubsys, ETCli components represent 'rich' client applications running on the portable devices held by the emergency teams, whereas the ETSrv component represents a web service. These components communicate with each other through an asynchronous SOAP connector deployed over a wireless medium. Within

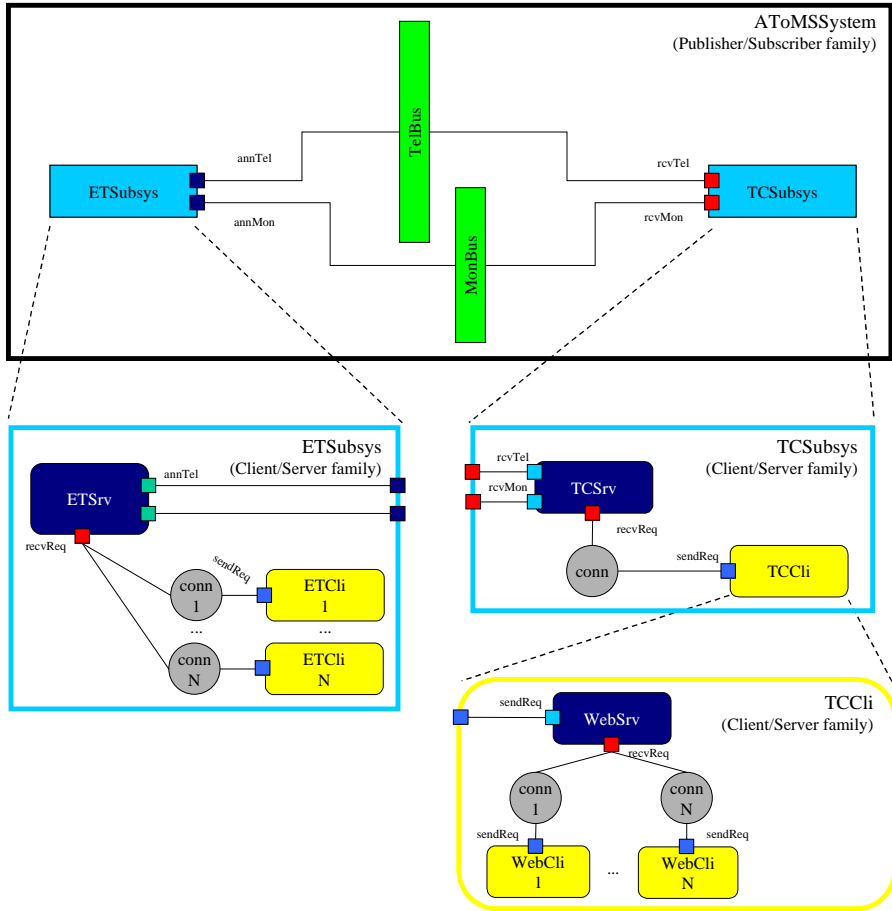


Fig. 8. Pictorial representation of the AToMS architecture

TCSys, TCCLI components represent web applications accessed by cardiologists through standard web browsers (synchronous HTTP), whereas the TCSrv component represents another web service. Teleconsultation requests/replies and monitoring notifications are exchanged asynchronously between the two aforementioned web services through two message bus connectors (TelBus e MonBus), which will be typically deployed over a wired network.

5.2 Dimensions of Dependability

The AToMS system presents various dependability dimensions, many of which can be considered at the architecture level. Some of these are described below.

```

System ATOMSSys = new PubSubFam extended with {
  Component ETSubsys = new ParticipantT extended with {
    Ports { annTel, annMon: p_announce = new p_announce; };
    Representation rep = new CliSrvFam extended with {
      Component ETClil: ClientT = new ClientT;
      Component ETSrv: ServerT = new ServerT extended with {
        Ports{ annTel; };
      };
      Connector connl: CConnT = new CConnT extended with {
        Property protocol: string = "SOAP";
      };
      Attachments { ETClil.sendReq to connl.cliSide;
                    ETSrv.recvReq to connl.srvSide; ...};
    };
    Bindings { ETSrv.annMon to ETSubsys.annMon;
              ETSrv.annTel to ETSubsys.annTel; };
  }; //end of ETSubsys spec.
  Component TCSubsys = new ParticipantT extended with {
    Ports{ rcvTel, rcvMon: p_receive = new p_receive; };
    Representation rep = new CliSrvFam extended with {
      Component TCcli: ClientT = new ClientT extended with {
        Representation rep = new CliSrvFam extended with {
          Component WebClil: ClientT = new ClientT;
          Component WebSrv: ServerT = new ServerT extended with{
            Ports{ sendReq; };
          };
          Connector connl: CConnT = new CConnT extended with {
            Property protocol: string = "HTTP";
          };
          Attachments { WebClil.sendReq to connl.cliSide;
                        WebSrv.recvReq to connl.srvSide; ...};};
        Bindings { WebSrv.sendReq to TCcli.sendReq; };};};
      Component TCSrv: ServerT = new ServerT extended with {
        Ports { rcvTel; rcvMon; };
      };
      Connector conn: CConnT = new CConnT extended with {
        Property protocol : string = "SOAP";
      };
      Attachments { TCcli.sendReq to conn.cliSide;
                    TCSrv.recvReq to conn.srvSide; };
    };
    Bindings { TCSrv.rcvTel to TCSubsys.rcvTel;
              TCSrv.rcvMon to TCSubsys.rcvMon; };
  }; //end of TCSubsys spec.
  Connector TelBus: EventBusT = new EventBusT extended with {
    Roles { pubTel: r_publisher = new r_publisher;
           subTel: r_subscriber = new r_subscriber; };
  };
  Connector MonBus: EventBusT = new EventBusT extended with {
    Roles { pubMon: r_publisher = new r_publisher;
           subMon: r_subscriber = new r_subscriber; };
  };
  Attachments { ETSubsys.annTel to TelBus.pubTel;
                ETSubsys.annMon to MonBus.pubMon;
                TCSubsys.rcvTel to TelBus.subTel;
                TCSubsys.rcvMon to MonBus.subMon; };
};

```

Fig. 9. ATOMS description in Acme

First, in an ideal situation the applications running on the portable devices should be able to communicate with a teleconsultation center irrespective of the emergency team's current location. Therefore, to enhance its robustness, the AToMS system must be able to use many different wireless technologies (*e.g.* WiFi in an indoor setting such as a shopping center, or GPRS outdoors). Moreover, detecting an available technology should be transparent to the emergency team—the same should also apply to the switching between technologies during operation. Clearly, such scenario calls for the dynamic reconfiguration of the system in terms of triggering smart handovers between different wireless networks. In Plastik, such reconfiguration can be foreseen at the architecture level, as depicted in Fig. 10. This figure builds on the specification provided in Fig. 9 by describing an on-do statement, and associated remove and dependencies statements. Also, a new Acme family (WirelessCliSrvFam) inheriting from the client-server style is defined as a means of specifying a connector type with specific wireless-related properties. The intended effect of the specification in Fig. 10 is that on detecting that a WiFi connection cannot be hold, the associated connector instance is replaced by another one that implements a GPRS connection.

```

Family WirelessCliSrvFam extends CliSrvFam with {
  Property Type RFSPropT = Enum { Excellent, VeryGood, Good
                                Low, VeryLow, Disconn};
  Connector Type WCSConnT extends CSConnT with {
    Property technology : string = "WiFi"; //default tech.
    Property RFS : RFSPropT;
  };
};

System AToMSSys = new PubSubFam extended with {
  Component ETSubsys = new ParticipantT extended with {
    ...
    Representation rep = new WirelessCliSrvFam extended with {
      ...
      Connector conn1: WCSConnT = new WCSConnT extended with {
        Property protocol: string = "SOAP"; // initially WiFi
      };
      ...
      on (conn1.technology == "WiFi" AND
          conn1.RFS in {VeryLow, Disconn})) do {
        remove conn1; //components are detached automatically
        Connector conn1: WCSConnT = new WCSConnT extended with {
          Property protocol: string = "SOAP";
          Property technology : string = "GPRS";
          dependencies {
            Attachments { ETClil.sendReq to conn1.cliSide;
                          ETSrv.recvReq to conn1.srvSide; };
          };
        };
      };
      ...
    };
  };
};

```

Fig. 10. Specification of a dynamically reconfigurable wireless connector in Plastik

It is important to bear in mind that many exceptional situations may happen in the AToMS system which could not be foreseen by the system architect. For instance, suppose that within ETSubsys a GPRS connection is currently in use between a client and the server. Because of its current location, the client experiences some intermittence in its GPRS connection, but no other wireless technology is available therein for a handover. An alternative solution could thus be to apply an ad-hoc reconfiguration—for instance, by dropping in a retransmission module to work over the lossy connection. To guarantee that the client interface does not change, and that the retransmission of messages in case of failure occurs transparently to it, it is interesting that such module be inserted into the wireless connector, as part of the communication protocol, or into the client, as an application-level plug-in.

As ETSubsys follows the client-server style, the aforementioned ad-hoc reconfiguration can be constrained so that the new module can be inserted into the wireless connector or the client component, but not in between them. The architectural rule in the client-server style which constrains such reconfiguration is depicted in Fig. 11. This rule mandates that if two components are connected, one must be a client and the other one must be the server.

In Plastik, the rule depicted in Fig. 11 is mapped to a validation script in Lua, as illustrated in Fig. 12. Such script is to be hosted by the policer component in the CF implementing ETSubsys. Functions prefixed by *rtk* and *rmm* (shown in bold font in the figure) are implemented by the Lua interpreter as a set of operation invocations on the OpenCOM runtime kernel services and reflective meta-models respectively. The validation script is called in the context of a transaction (identified by parameter ‘*tid*’), which is scoped by the proxy component in the same CF as the policer. More details about the mappings between Acme rules and Lua validation scripts can be found in [20].

5.3 Discussion

As it currently is, Plastik was able to cover the major dependability issues that arose in the presented case study. Nevertheless, some aspects have been identified for future investigation.

The first one is that the on-do construct does not allow dealing with sets of elements in a system or representation specification in a flexible way. Although the condition expressions in on-do statements can use the ‘*forall*’ and ‘*exists*’ Armani quantifiers, in the case of Fig. 11 the only possible way to describe the intended

```

Rule ClientServerConnection = Invariant
  Forall c1:component in self.components
    Forall c2:component in self.components
      connected(c1, c2) =>
        (declaresType(c1, ClientT) AND
         declaresType(c2, ServerT)) OR
        (declaresType(c1, serverT) AND
         declaresType(c2, ClientT))

```

Fig. 11. Client-server style rule in Acme

```

function lvs_ClientServerConnection(tid)
  local forAllExp_1 = true
  for _, c1 in ipairs(rmm.enumInstances(tid))
    local forAllExp_2 = true
    for _, c2 in ipairs(rmm.enumInstances(tid))
      local boolExp = true
      if rmm.connected(c1, c2) then
        local c1cat = rtk.getProp(tid, c1, "Category")
        local c2cat = rtk.getProp(tid, c2, "Category")
        boolExp =
          ((c1cat == "Client" and
            c2cat == "Server") or
           (c1cat == "Server" and
            c2cat == "Client"))
      end --if
      if not boolExp then
        forAllExp_2 = false; break
      end --if
    end -for
  end -for
  if not forAllExp_2 then
    forAllExp_1 = false; break
  end -if
end -for
return forAllExp_1
end --function

```

Fig. 12. Validation script in Lua

programmed reconfiguration was to replicate the on-do statement for each connector in ETSubsys.

A second point is that there is currently no adequate support in Plastik for managing dynamic reconfiguration in distributed systems. For the programmed reconfiguration in Fig. 10 to be possible, the ETSubsys component had to be mapped onto a widely-distributed capsule. Although the degree of capsule distribution is implementation dependent in OpenCOM, the general intention is that capsules be relatively tightly-coupled so that centralised state can be held [10]. For widely-distributed systems, inter-capsule bindings (e.g. using RPC or pub-sub or streaming) are preferred. Currently, Plastik does not support the explicit management of such type of bindings.

6 Related Work

There has been a good amount of research over the past few years on specification-to-deployment environments, although much of it has focused on particular application areas or specific aspects. Little of it, however, has comprehensively addressed dynamic reconfiguration issues, which in our belief is crucial to enabling the architecture of dependable systems. In the following paragraphs we survey relevant research, classifying the work in two main categories: *dynamic software architecture techniques* and *reconfigurable runtime technologies*.

Dynamic software architecture techniques. Aster [5,22] is a development environment providing an ADL with an embedded property sub-language that supports the architectural specification of object-oriented distributed systems and their non-functional requirements. Based on an application specification, an Aster tool selects appropriate middleware objects and integrates them with the described application. Aster supports ad-hoc reconfiguration at the architecture level, but does not support any kind of programmed reconfiguration. Furthermore, Aster focuses on high-level functionality and cannot realistically be used for architecting systems in stringent environments such as the networking examples considered above.

Mae (Managing Architectural Evolution) [33] is an architectural evolution environment based on xADL [12]. Mae explicitly supports programmed reconfiguration by employing a configuration versioning mechanism (*i.e.* configurations are basically checked in and out to/ from a version space), but does not support ad-hoc reconfiguration.

The work in [30] presents an approach to runtime software evolution based on the C2 ADL [35] and focused on connector-centred reconfiguration. This work has similarities to our approach as regards providing some causal connection between architecture- and runtime-level models. Nevertheless, reconfigurations can be applied only at the architecture level and enforced only on connectors.

ArchWare [28] is a specification-to-deployment environment that also provides some causal connection between architecture- and runtime-level models. ArchWare strengths reside in its formal support at the architecture level, which is provided by a specific π calculus-based ADL, and its reflective capabilities at the runtime level, which are provided by an implementation of the *hyper-code* abstraction [21]. In contrast, we departed from two mature technologies (OpenCOM and Acme), leveraging both of them through the definition of precise inter-model mappings and a few extensions leading to the enablement of causally-connected reconfiguration.

Three other pieces of work on specification-to-deployment environments have bearing on our approach. First, in [7], architectural styles are treated basically as policies for system self-repair in the runtime infrastructure—*i.e.* the system is monitored during runtime so that style-prescribed reconfigurations are effected when style constraints are violated. Therefore, this system supports both programmed and ad-hoc reconfigurations. However, it does not consider the enactment of partial specifications directly provided by software architects during runtime—*i.e.* architecture-level ad-hoc reconfigurations are not considered. Second, in [15], an architectural runtime configuration management system is proposed, the focus of which is on adaptive fault recovery. The fundamental facility provided by this work, however, resides in enhancing adaptation *visibility* through visual tooling; this work relies on human judgement and intervention for determining whether a reconfiguration was valid or not. Third, work by Georgiadis *et al.* [16] discusses the feasibility of using architectural constraints as the basis for the specification and deployment of self-organising architectures at distributed execution environments. Finally, the work presented in [1] addresses the issue of precise mapping between architecture- and runtime-level models. In contrast to our approach, however, its primary contribution is to provide semi-automated incremental synchronization between architecture- and runtime- views, thus ensuring conformance between the architecture specification and the configuration of the running system.

Reconfigurable runtime technologies. FORMAware [27] is a reflective component-based framework that augments explicit architectural specifications with meta-information to constrain dynamic reconfiguration. FORMAware supports ad-hoc reconfiguration, which is managed by a transaction service; however, programmed reconfiguration is not supported. Another fundamental difference between FORMAware and our approach is that we adopt an ADL for the definition of architectural invariants, whereas FORMAware style rules are described as pieces of procedural code. This lowers its level of abstraction with respect to our approach.

Koala [36] is a component model that uses an ADL based on Darwin [25] to manage the complexity of software in consumer electronics products. Dynamic reconfiguration is, however, restricted to switching between pre-existing components based on statically defined conditions. Furthermore, there is no causal connection between the architecture- and runtime-level models.

Fractal [6] is a hierarchically-structured Java based component model which uses an XML-based ADL to specify the high level structure of an application, and which provides runtime reflective features to support dynamic reconfiguration. Nevertheless, Fractal has no support for the description of constraints at the architecture level, and again provides no causal connection between the levels.

Finally, Knit [32] is a component definition and linking language that promotes the wrapping and reuse of existing code (in C). One of its main features is minimising componentisation overhead; thus it is particularly useful for designing and implementing complex, low-level systems. Knit also provides some support for specifying and enforcing architectural rules, but it has nothing comparable to a fully general ADL. Also, in comparison with our approach, Knit is programming language-specific and does not support either ad-hoc or programmed reconfiguration.

7 Final Remarks

We have motivated and discussed an approach to dynamic reconfiguration in dependable systems. The approach can be seen as bearing on dependability in two distinct ways. First, it helps us view dynamic reconfiguration as a concern that goes very naturally with generally-applied approaches to software engineering for dependability—such as ADLs, model driven development (MDD) paradigms, and domain specific languages. Second, our approach is intrinsically internally dependable though its use of system elements such as the reconfiguration transaction service, and policing components that are automatically generated from architecture-level specifications.

Our approach is also highly generic in that it supports both programmed and ad-hoc reconfiguration at both the architecture and the runtime levels.

In future work we plan to evaluate the Plastik architecture in the context of real application scenarios. We are especially interested in applying the approach in the resource-poor and ‘primitive’ application environments (*e.g.* embedded systems and sensor networks) in which OpenCOM itself is routinely applied. This will be a strong test of the viability of our approach (and especially its runtime machinery).

We also plan to broadly evaluate the Plastik reconfiguration strategies using highly dependable (and possibly distributed) applications. We intend to consider dependable

scenarios where software reconfiguration is specified in the architecture model and reified in the OpenCOM runtime and also where the reconfiguration is triggered by the runtime level. In this way we can explore the power of the Plastik dynamic reconfiguration mechanisms to the realization of the self-organizing, self-repairing and self-healing support commonly demanded by dependable applications.

More generally, we plan to formalise the interface between the runtime and architecture levels of the architecture, and on this basis, to explore the wider application of our approach. For example, this will enable us to plug in different and independently-developed architecture levels that conform to specific MDD paradigms (*e.g.* UML-based models [8] or DSL-oriented specification frameworks [17]). We believe that such a research direction will contribute strongly to bringing dynamic reconfiguration more into the mainstream of dependable systems design and deployment.

References

1. Abi-Antoun, M., Aldrich, J., Garlan, D., Schmerl, B., Nahas, N., Tseng, T.: Improving System Dependability by Enforcing Architectural Intent. In: Workshop on Architecting Dependable Systems. St. Louis (MO), USA (2005)
2. Akyildiz, I.F., Su, W., Sankarasubramaniam, Y., Cayirci, E.: Wireless Sensor Networks – A Survey. *Computer Networks* 38, 393–422 (2002)
3. Batista, T., Joolia, A., Coulson, G.: Managing Dynamic Reconfiguration in Component-Based Systems. In: Morrison, R., Oquendo, F. (eds.) EWSA 2005. LNCS, vol. 3527, pp. 1–17. Springer, Heidelberg (2005)
4. Blair, G., Coulson, G., Grace, P.: Research Directions in Reflective Middleware: the Lancaster Experience. In: Proc. 3rd Workshop on Reflective and Adaptive Middleware (RM2004) co-located with Middleware 2004, Toronto, Ontario – Canada pp. 262–268 (2004)
5. Blair, G.S., Blair, L., Issarny, V., Tuma, P., Zarras, A.: The Role of Software Architecture in Constraining Adaptation in Component-Based Middleware Platforms. In: IFIP/ACM International Conference on Middleware. Hudson River Valley (NY), USA (2000)
6. Bruneton, E., Coupaye, T., Stefani, J-B.: Recursive and Dynamic Software Composition with Sharing. In: International Workshop on Component-Oriented Programming. Malaga, Spain (2002)
7. Cheng, S-W., Garlan, D., Schmerl, B., Sousa, J.P., Spitznagel, B., Steenkiste, P., Hu, N.: Using Architectural Style as the Basis for Self-Repair. In: Working IEEE/IFIP Conference on Software Architecture, Montreal, Canada (2002)
8. Costa, P., Coulson, G., Mascolo, C., Picco, G.P., Zachariadis, S.: The RUNES Middleware – a Reconfigurable Component-Based Approach to Networked Embedded Systems. In: Annual International Symposium on Personal Indoor and Mobile Radio Communications. Berlin, Germany (2005)
9. Coulson, G., Blair, G.S., Clarke, M., Parlavantzas, N.: The Design of a Highly Configurable and Reconfigurable Middleware Platform. *ACM Distributed Computing Journal* 15, 109–126 (2002)
10. Coulson, G., Blair, G.S., Grace, P., Joolia, A., Lee, K., Ueyama, J.: OpenCOM v2: A Component Model for Building Systems Software. In: IASTED Software Engineering and Applications. Cambridge (MA), USA (2004)

11. Coulson, G., Blair, G.S., Hutchison, D., Joolia, A., Lee, K., Ueyama, J., Gomes, A.T.A., Ye, Y.: NETKIT: A Software Component-Based Approach to Programmable Networking. *ACM SIGCOMM Computer Communications Review* 33, 55–66 (2003)
12. Dashfof, E.M., van der Hoek, A., Taylor, R.N.: A Highly-Extensible, XML-Based Architecture Description Language. In: *Working IEEE/IFIP Conference on Software Architecture*, pp. 28–31. Amsterdam, The Netherlands (2001)
13. Garlan, D., Allen, R.J., Ockerbloom, J.: Exploiting Style in Architectural Design. In: *SIGSOFT Symposium on the Foundations of Software Engineering*. New Orleans (LA), USA (1994)
14. Garlan, D., Monroe, R., Wile, D.: Acme: Architectural Description of Component-Based Systems. In: Leavens, G.T., Sitaraman, M. (eds.) *Foundations of Component-Based Systems*, pp. 47–68. Cambridge University Press, Cambridge (2000)
15. Georgas, J.C., van der Hoek, A., Taylor, R.N.: Architectural Runtime Configuration Management in Support of Dependable Self-Adaptive Software. In: *Workshop on Architecting Dependable Systems*. St. Louis (MO), USA (2005)
16. Georgiadis, I., Magee, J., Kramer, J.: Self-organising Software Architectures for Distributed Systems. In: *First Workshop on Self-healing Systems*. Charleston, USA, pp. 33–38 (2002)
17. Gomes, A.T.A., Coulson, G., Blair, G.S., Soares, L.F.G.: A Component-Based Approach to the Creation and Deployment of Network Services in the Programmable Internet. Technical Report MCC-42/03, PUC-Rio, Brazil (2003)
18. Gomes, A.T.A., Ziviani, A., de Souza e Silva, N.A., Feijoo, R.A.: Towards a ubiquitous healthcare system for acute myocardial infarction patients in Brazil. In: *Pervasive Computing and Communications Workshops*. Pisa, Italy (2006)
19. Ierusalimsky, R., Figueiredo, L.H., Celes, W.: Lua – An Extensible Extension Language. *Software: Practice and Experience* 26, 635–652 (1996)
20. Joolia, A., Batista, T., Coulson, G., Gomes, A.T.A.: Mapping ADL Specifications to an Efficient and Reconfigurable Runtime Component Platform. In: *Working IEEE/IFIP Conference on Software Architecture*. Pittsburgh (MA), USA, pp. 131–140 (2005)
21. Kirby, G.N.C., Connor, R.C.H., Cutts, Q.I., Dearle, A., Farkas, A.M., Morrison, R.: Persistent Hyper-Programs. In: Albano, A., Morrison, R. (eds.) *Persistent Object Systems, Workshops in Computing*, pp. 86–106. Springer, Heidelberg (1992)
22. Kloukinas, C., Issarny, V.: Automating the Composition of Middleware Configurations. In: *IEEE International Conference on Automated Software Engineering*, Grenoble, France, pp. 241–244 (2000)
23. Kon, F., Costa, F., Campbell, R., Blair, G.S.: The Case for Reflective Middleware. *Communications of the ACM*. 45(6), 33–38 (2002)
24. Lagaisse, B., Joosen, W.: True and Transparent Distributed Composition of Aspect-Components. In: *Proc. ACM/IFIP Middleware 2006*, Melbourne, Australia (2006)
25. Magee, J., Dulay, N., Eisenbach, S., Kramer, J.: Specifying Distributed Software Architectures. In: *European Software Engineering Conference*. Sitges, Spain, pp. 137–153 (1995)
26. Monroe, R.T.: Capturing Software Architecture Design Expertise with Armani. Technical Report CMU-CS-98-163, Carnegie-Mellon University, USA (1998)
27. Moreira, R., Blair, G.S., Carrapatoso, E.: FORMAware – Framework of Reflective Components for Managing Architecture Adaptation. In: *International Symposium on Distributed Objects and Applications*. Rome, Italy (2001)

28. Morrison, R., Kirby, G., Balasubramaniam, D., Mickan, K., Oquendo, F., Cimpan, S., Warboys, B., Snowden, B., Greenwood, R.M.: Support for Evolving Software Architectures in the ArchWare ADL. In: Working IEEE/IFIP Conference on Software Architecture. Oslo, Norway, pp. 69–78 (2004)
29. Oreizy, P., Gorlic, M., Taylor, R., Medvidovic, N., et al.: An Architecture-Based Approach to Self-Adaptive Software. *IEEE Intelligent Systems* 14, 54–62 (1999)
30. Oreizy, P., Taylor, R.N.: On the Role of Software Architectures in Runtime System Reconfiguration. *IEE Proceedings* 145(5), 137–145 (1998)
31. Pissias, P., Coulson, G., Joolia, A.: Supporting Dynamic Reconfiguration in Multithreaded Component-Based Systems. Technical Report, Lancaster University, UK (2006)
32. Reid, A., Flatt, M., Stoller, L., Lepreau, J., Eide, E.: Knit: Component Composition for Systems Software. In: Symposium on Operating Systems Design and Implementation. San Diego (CA), USA, pp. 347–360 (2000)
33. Roshandel, R., van der Hoek, A., Mikic-Rakic, M., Medvidovic, N.: Mae – A System Model and Environment for Managing Architectural Evolution. *ACM Transactions on Software Engineering and Methodology* 3, 240–276 (2004)
34. Szyperski, C., Gruntz, D., Murer, S.: *Component Software – Beyond Object-Oriented Programming*, 2nd edn. Addison-Wesley, New York (2002)
35. Taylor, R.N., Medvidovic, N., Anderson, K.M., Whitehead, E.J., Robbins, J.E., Nies, K.A., Oreizy, P., Dubrow, D.L.: A Component- and Message-Based Architectural Style for GUI Software. *IEEE Transactions on Software Engineering* 22, 390–406 (1996)
36. van Ommering, R., van der Linden, F., Kramer, J., Magee, J.: The Koala Component Model for Consumer Electronics Software. *IEEE Computer* 33, 85–87 (2000)
37. Wermelinger, M.: Towards a Chemical Model for Software Architecture Reconfiguration. In: *IEE Proceedings - Software*, vol. 145, pp. 130–136 (1998)

Ecotopia: An Ecological Framework for Change Management in Distributed Systems

Tudor Dumitraş¹, Daniela Roşu², Asit Dan², and Priya Narasimhan¹

¹ ECE Department, Carnegie Mellon University, Pittsburgh, PA 15213, USA

² IBM T.J. Watson Research Center, Hawthorne, NY 10532, USA

tudor@cmu.edu, drosu@us.ibm.com, asit@us.ibm.com,
priya@cs.cmu.edu

Abstract. Dynamic change management in an autonomic, service-oriented infrastructure is likely to disrupt the critical services delivered by the infrastructure. Furthermore, change management must accommodate complex real-world systems, where dependability and performance objectives are managed across multiple distributed service components and have specific criticality/value models. In this paper, we present Ecotopia, a framework for change management in complex service-oriented architectures (SOA) that is ecological in its intent: it schedules change operations with the goal of minimizing the service-delivery disruptions by accounting for their impact on the SOA environment. The change-planning functionality of Ecotopia is split between multiple objective-advisors and a system-level change-orchestrator component. The objective advisors assess the change-impact on service delivery by estimating the expected values of the Key Performance Indicators (KPIs), during and after change. The orchestrator uses the KPI estimations to assess the per-objective and overall business-value changes over a long time-horizon and to identify the scheduling plan that maximizes the overall business value. Ecotopia handles both external change requests, like software upgrades, and internal changes requests, like fault-recovery actions. We evaluate the Ecotopia framework using two realistic change-management scenarios in distributed enterprise systems.

Keywords: Dynamic Change Management, Service Orchestration, Fault-Tolerant Architecture, Performability, Autonomic Computing.

1 Introduction

Enterprises demand highly available online systems and satisfactory service levels (*e.g.*, average response time) in the face of change. The kinds of changes that can occur are diverse, and can include recovery actions in response to failures, or upgrades due to new versions of software that become available. Current change-management strategies, for the most part, tend to execute a change request as soon as possible (*e.g.*, as soon as a fault is detected or an upgrade is requested), rather than looking for the best time to do so. The downtime (or the perceived lack of responsiveness/availability) due to change management can disrupt the performance expectations of services and have an adverse effect on business.

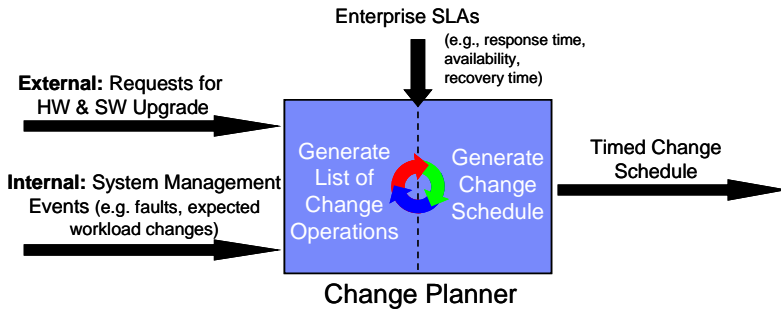


Fig. 1. Dynamic change management is likely to disrupt the critical services running in the IT infrastructure. Ecotopia handles changes based on both external requests (e.g., software upgrades) and events detected internally by the autonomic management infrastructure (e.g., faults) while taking into account their impact of the service-level agreements. The output is a timed schedule that seeks to wait for the most opportune time to apply each change operation and to maximize the enterprise business value.

Industry analysts indicate that "unmanaged change is one of the leading causes of downtime or missed service-level agreements (SLAs)." [1] Gartner Group states that "to address the 80 percent of unplanned downtime caused by "people failures," enterprises should invest in improving their change and problem management processes (to reduce downtime caused by application failures) and in automation tools, such as job scheduling and event management (to reduce downtime caused by operator errors)." [2] Thus, we hypothesize that it is more appropriate to *seek the most opportune time to execute the change operations in a distributed service-oriented infrastructure, based on the change's impact on the service-level objectives (e.g. response time, availability, and recovery time)*. Such an impact-sensitive change-management strategy aims to respect the overall performance and dependability guarantees of the running services, yet allowing the system to incorporate changes of various kinds.

Fig. 1 illustrates the main elements of the change-planning problem. In typical IT infrastructures, there are multiple kinds of change operations, originating from various sources. Some changes are planned in advance (e.g., deploying new applications, upgrading obsolete software, increasing the system capacity), and are derived from an *external* request for change (RFC). In other cases, changes are due to "firefighting" (i.e., mitigating the negative effects of unplanned situations), and are triggered by *internal* system-management events, e.g., faults or load surges. Change requests are characterized by a set of (partially) ordered change operations and by change objectives such as the deadline for implementing the change. The change-operation planner must produce a timed change-schedule for executing the changes and, in the process, must consider both the impact of the changes on all the relevant quality-of-service requirements as expressed by service-level objectives (SLOs), as well as the objectives of each change operation.

An SLO defines bounds and targets for a level-of-service metric (e.g., response time, recovery time, availability), called Key Performance Indicator (KPI). An SLO also has a specific *business value* metric (e.g., the penalties associated with a missed

change deadline or with a degraded performance) for gauging the utility of fulfilling the objective [3]. The change schedule must maximize the aggregated business value, associated with all of the enterprise's SLOs. This optimization must span a long time-horizon, to account for both transient effects that might occur during the change execution, and permanent effects that might persist after the change has been finalized.

The change planner must be "ecological" in nature, *i.e.*, it must assess the impact of the change on the environment and its SLOs by considering a number of factors: the inter-dependencies among various system components, the available prior knowledge of workload fluctuations or anticipated load surges during prime-time, as well as the degree of resource sharing across heterogeneous, off-the-shelf components that sometimes span independent administrative domains. In these environments, the high-level service objectives translate into component-level objectives that can be managed by component-specific configuration managers. For example, a workload manager prioritizes and routes the service requests by monitoring the response-time objectives, while a dependability manager primes backup nodes in anticipation of failures and performs recovery by monitoring the availability objectives. These managers use extensive, and sometimes proprietary, domain knowledge (*e.g.*, workload characteristics, resource-utilization models), and can perform sophisticated request classification, prioritization, monitoring and request routing [4].

As a result, we believe that the complexity and the distributed nature of objective-management in real-world systems makes it unfeasible for a fully centralized change-operation planner to directly assess the impact of change operations on each service KPI. Rather, the impact on service KPIs should be estimated by the component-specific managers that control these services. However, component-specific managers might not be able to directly assess SLO business values necessary for estimating the overall change-impact, either because they do not directly implement the enterprise SLO models or because the service spans multiple managers and administrative domains.

Building on this principle, we propose Ecotopia, a change-management framework that decouples the impact assessment (handled by multiple objective advisors, *e.g.*, performance and dependability advisors) from the change-operation scheduling (handled by a change orchestrator). The *orchestrator* builds the change-operation schedule and estimates its business value impact based on the service KPIs predicted by the *objective advisors*. The advisors are software components that incorporate the domain knowledge to answer "what-if" questions about service KPIs (such as performance and availability forecasts), given a description of the change operations and the timing properties associated with their execution. The orchestrator leverages the advisors' predictions to compute the per-objective and the aggregate business value, and to converge towards an optimal change-operation schedule through an iterative refinement process. The objective advisors themselves can be composite, third-party services.

The novel characteristics of the Ecotopia framework for orchestrating change-management operations are:

- Rich “what-if” interaction model that enables the use of fine-grained objective-advisor knowledge for an effective change scheduling decision. Our “what-if” model includes:
 - *Timeline of prediction points*: the advisors inform the orchestrator of the expected workload changes during the scheduling timeline. The orchestrator uses these guidelines to bootstrap the scheduling algorithms.
 - *Proactive actions*: the advisors can inform the orchestrator about specific actions that may improve the impact on KPIs during related change operations. The orchestrator can include these operations in the final schedule if they result in an improved overall business-value.
- Integrated management of both internal (*e.g.*, faults, workload changes) and external (*e.g.*, upgrades, capacity increases) changes. This approach is necessary because both types of changes affect a common pool of resources and services. Existing solutions [5, 6] assume different decision makers for the two types of changes.
- Complex business value functions for SLOs and change-request deadlines that can change along with the underlying enterprise service models, enabled by compliance with WS-Agreement standard [3]. Existing solutions support only priority-based models [5] or embedded, hard-coded utility functions [4, 7, 8].
- Optimization based on the long-term impact of change on performance and dependability objectives, accounting for both the time during and after execution of the change. Existing solutions consider only one of the two impact components (*e.g.*, [7] considers the impact during change execution, [5, 9] consider the impact after the change).

In Section 2 we compare Ecotopia with the state of the art in impact-aware change management. Section 3 describes the design of Ecotopia framework and Section 4 describes the current implementation. Section 5 presents two case studies of change management that we use to validate our architecture. Section 6 discusses the applicability of our ecological approach for realistic systems and outlines directions for future work.

2 Background

In their seminal paper, Segal and Frieder [10] identify a set of general requirements for any dynamic updating system: preserving program correctness (during and after the update), minimizing human intervention, supporting program restructuring and low-level program changes (*e.g.*, both implementations and interfaces), supporting distributed programs (communicating across mutually distrustful administrative domains), not requiring special-purpose hardware and not constraining the language and environment. Their survey illustrates that in general, research has focused on mechanisms for implementing change at different levels of granularity (*e.g.* replacing components, objects, procedures), rather than on impact assessment and coordination of distributed changes. Kramer and Magee [11] note that faults, as well as live upgrades, might have a disruptive effect on the functionality of a distributed system, and that the techniques to mitigate these problems could be combined in a unified

framework. For instance, a change-management system that totally separates the functional application concerns from the configuration management concerns (such as Kramer and Magee's Conic system), can provide a good basis for implementing fault recovery [11]. Conversely, an infrastructure built for fault-tolerance can provide a good basis for live upgrades because of the inherent redundancy [12, 13]. For example, a fault-tolerant CORBA system using the interception approach provides all the ingredients needed for dynamic change management of CORBA objects, including an interceptor (*i.e.*, the indirection layer needed when switching to a new version), replication mechanisms (for incrementally upgrading some replicas while others continue to provide service) and state extraction/restoration mechanisms (for maintaining consistency between versions) [12].

In the Ecotopia framework, we also adopt this unifying approach of considering both external (*e.g.*, software upgrades) and internal change requests (*e.g.*, operations needed to mitigate the effects of a fault). Additionally, the goal of our ecological framework is to manage the impact of change-management on the SOA environment (the running services and the existing resources). We assess this impact by asking and answering “what-if” questions about the outcome of the change operations. We assume some advance knowledge of the workload, as a running system has different behavioral profiles depending on the system load and the outcome of the changes will depend on the workload as well. Ecotopia tries to minimize the negative impact on the environment by using the answers to the “what-if” questions to determine the most opportune time to apply the changes, given the existing resources, the state of the running services and the workload.

2.1 Workload Prediction

Many workloads are characterized by a day-night periodicity [14]: the incoming request load increases during the day, with comparable peak request-rates from day to day, and decreases at nighttime to a very low baseline level. System administrators take advantage of this knowledge to over-provision the system for the highest expected loads [15] and to run maintenance activities (such as change management) during the night. There are also workloads with more complex patterns. The 1998 World Cup workload¹ [16] shows that the incoming load increases suddenly around game times, with lower peaks for the games played over a weekend. This trend is typical for sites dedicated to sporting events; this can be observed on Alexa.com² [17], by comparing statistics for two different sites covering the same event (*e.g.*, fl.com and fi-live.com): even if the peak loads are different, the access patterns are the same. On-line auction sites, such as ebay.com, exhibit similar load surges before the closing time of an auction. Furthermore, recognizable patterns of warnings and notifications that precede system events may facilitate the workload prediction [18, 19].

Ecotopia uses the ability to predict when the system is under high and low load for optimizing across multiple service-level objectives. For instance, an enterprise system

¹ This is the workload of a website dedicated the 1998 soccer World Cup in France. With 1.4 billion requests in the server logs, this is the largest web workload ever analyzed.

² Alexa is a tool for comparing statistics on the popularity and workloads of different websites.

may have two objectives: performance, expressed as average response time, and dependability, expressed as the expected recovery time after a system failure. After a fault (which, unlike a failure, does not completely disable the system), Ecotopia relies on knowledge of the workload to schedule the reconfiguration operations when the incoming load is low and avoid the penalties due to downtime during a busy period. Note that we do not assume that flash-crowd events (sudden load surges due to an unexpected increase in the site's popularity) are predictable; however, we show that exploiting irregular, but predictable workloads – such as the World Cup 98 trace [16] – allows Ecotopia to improve the scheduling of change operations when pursuing multiple objectives. Workload prediction is an optional part of the framework; Ecotopia's orchestrator can function with third-party advisors that answer “what-if” questions without providing workload predictions, *e.g.*, [8].

2.2 “What-if” Questions

Existing service-orchestration products [5, 20], perform resource arbitration between node groups by evaluating the impact just after the resource changes are enacted. While allowing the orchestration of distributed services [4], this approach is limited because it ignores the long-term impact of change management (*e.g.* interaction with expected workload change). The CHAMPS project [7] focuses on scheduling operations to satisfy external RFC deadlines. It develops a complex dependency-tracking framework and it formulates the scheduling problem as the optimization of a generic cost function given a set of constraints (representing the impact during change, *e.g.*, due to service unavailability), providing a *centralized* approach for both scheduling and impact analysis. Our work is based on the observation that centralized impact evaluation is not appropriate for complex enterprise environments.

The problem of optimizing business value in a decentralized manner has also been addressed in the context of autonomic management of storage systems. Hippodrome [9] refines the initial configuration of a storage system through an iterative process, using a performance model to estimate the throughput and capacity of a particular configuration. Like our framework, Hippodrome separates between optimization and impact assessment, although the interactions between the two components are more tightly integrated and is based on a proprietary protocol. We submit that for complex systems integrating multi-vendor components we need an open communication protocol, for instance based on Web Services. The K2 middleware [21] goes further in distributing the autonomic management functionality by eliminating the centralized decision-maker and allowing individual “allocation pools” to manage their own objectives. In K2, distributed decision algorithms determine the goal configuration and the allocation pools start moving in that direction; if conditions change part-way through reconfiguration, the system changes its direction without having to invalidate the previous plan. However, none of these systems consider the evolution in time of the KPIs and the long-term impact of their decisions which are necessary for avoiding system instability and minimizing the overall business impact.

Thereska et al. [8] define a “resource advisor” predicting the impact of data placement and encoding choices on performance. The advisor has a hierarchical design, based on several “what-if” modules (for predicting the CPU, network and disk delays and cache hit rates) that can be combined together for end-to-end KPI

predictions. Although it does not account for the detailed KPI evolution (it does not attempt to predict incoming request rates), the advisor continuously monitors the infrastructure and uses historical data to overprovision the system based on the peak loads observed. The authors report that prediction errors are less than 15% in most cases. This is an example of a third-party objective advisor that could be connected to the Ecotopia framework. Our orchestrator doesn't need to know the details of the performance models for storage systems; instead, it can use the "what-if" predictions to perform an ecological change management.

2.3 Timing the Application of Change Operations

The idea of waiting for the most opportune time to apply a change is widely accepted with respect to security patches for enterprise infrastructures. Beattie et al. [22] show that there is a sweet-spot for the time when security patches should be applied. Patches applied too early, without enough testing in the field, may introduce critical bugs or may conflict with local configurations. Patches applied too late leave the system exposed to security threats for an extended period of time. The authors argue that patching should be delayed until the risk of a security breach outweighs the risk of introducing bugs, and they develop a mathematical model for estimating the optimal time to apply a security patch.

Gorbenko et al. [23] tackle the problem of achieving high dependability of composite Web Services undergoing online upgrades of their components. They advocate running multiple versions of a service in parallel and using third-party interception middleware to switch to a new replica when the confidence in its correctness is sufficiently high. The "confidence in correctness" metric is computed based on comparing the responses from different versions of a service and using Bayesian inference to reason about future failure rates. This approach is the closest to our focus on the long-term impact of change operations, except that we use impact assessment across multiple service-level objectives and we use standard metrics, such as business value, for evaluating this impact.

Roşu et al. [24] introduce the approach of evaluating change plans based on actual SLO business values, which are computed by the orchestrator based on the service KPIs provided by objective advisors. Ecotopia extends this approach to a compressive "what-if" protocol appropriate for management of complex change requests. Other change orchestration solutions evaluate change plans in disconnection from the actual SLO of the enterprise, based on hard-coded utility models embedded in the resource advisors [4, 5, 8]. In [25], the change manager uses WS-Agreement specification to define business value parameters whereas the specification of the objective and business value functions is hard-coded in the orchestrator implementation. Neither of these approaches is appropriate for systems in which the objective and value models can evolve in time.

3 Design of an Ecological Change-Management Framework

A primary design goal for a change-management framework that targets distributed, service-oriented infrastructures is to make minimal assumptions about the kinds of

“knobs” that the various software components are prepared to expose to a change-management system for enabling the control of change impact. The key to achieving this goal is the separation of scheduling and impact analysis. In Ecotopia, these tasks are performed by different components, which may come from different providers.

Service orchestration refers to an executable business process that combines multiple services by defining their interactions dynamically, with the goal of aligning the behavior of the composite service with the business objectives [20]. Ecotopia contains an orchestration engine that queries multiple objective advisors for KPI predictions and combines their outputs into a change-operation schedule. The predictions are based on detailed domain knowledge of each system component, but this knowledge is not exposed outside the objective advisors. Instead, the advisors answer simple “what-if” questions [8] about the impact of concrete change operations on service KPIs, considering the workload and the tentative schedules of these operations. The orchestration is driven by the enterprise SLAs, which define methods for computing the business value [3] that corresponds to the predicted KPI values. The business value reflects the utility of a given change schedule, allowing us to compare schedules and make an “ecological” choice: considering the impact on the IT environment, we select the change schedule that minimizes the service-delivery disruptions and that maximizes the overall business value.

General assumptions. We assume that KPI predictions can be derived from some knowledge of future incoming loads, either because the workloads exhibit clear trends [14, 16], or because fluctuations are preceded by recognizable patterns of warnings and notifications [18, 19]. Furthermore, we assume that the execution times of all the change operations submitted to the Ecotopia orchestrator can be estimated and that services do not have hard real-time constraints (which is typical of enterprise systems).

3.1 Framework Components

Fig. 2 illustrates the main components and interactions in the Ecotopia framework. The *ChangeManager* receives high-level RFCs, decomposes them into finer-grained change operations and related dependencies, and forwards them to a centralized component called the *orchestrator*. The orchestrator receives the list of change operations and their execution constraints and generates a change plan through an iterative process. Distributed components called *objective advisors* analyze the impact of planned change operations; the orchestrator identifies the relevant advisors by querying the *SystemConfigurationDatabase*. The objective advisors represent the service managers in the infrastructure and can use manager-specific knowledge to estimate the impact of a change plan on the service KPIs. The orchestrator consumes these estimations and schedules the change operations with the goal of maximizing the overall business value. The interaction between the orchestrator and the advisors is based on the Web Services standard, which facilitates compatibility in a complex system with components built by different providers. The orchestrator sends the final schedule to the *ScheduleExecutor*, which triggers the change operations at the indicated times. The *ChangeManager* is analogous to the Task Graph Builder

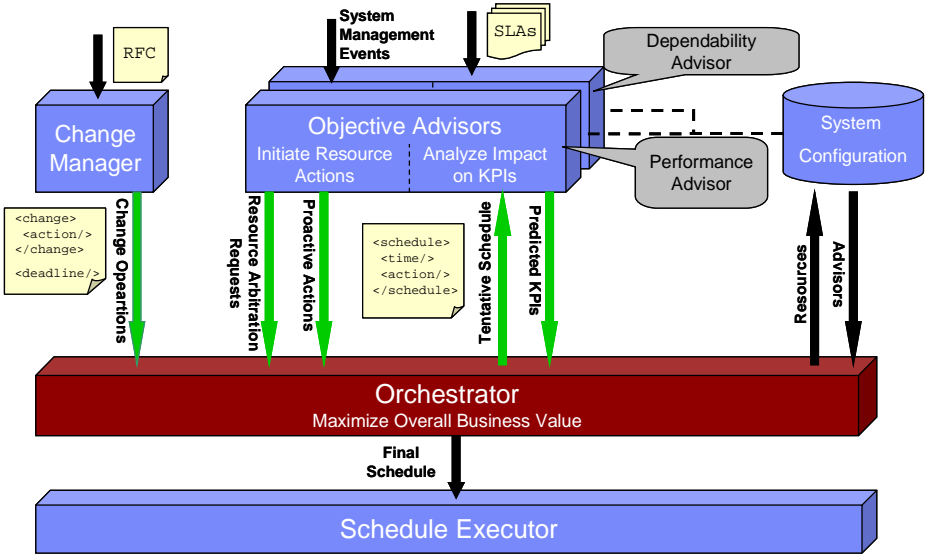


Fig. 2. Ecotopia’s distributed ecological architecture for change management separates the tasks of impact assessment (performed by the objective advisors) and change scheduling (performed by the orchestrator). The orchestrator receives requests for change, queries the objective advisors with “what-if” questions about the tentative change schedule and uses the answers to refine the schedule with the goal of maximizing business value. The “what-if” interactions are based on an open protocol that allows the integration of third-party objective advisors.

from [7], and the `ScheduleExecutor` is similar to the TIO Provisioning Manager [5]. In this paper, we focus on the orchestrator, the objective advisors and their interactions, which are novel.

Objective advisors. The objective advisors (e.g., performance and dependability advisors) exploit the functionality provided by the component-specific configuration managers. The advisors can be hierarchical and may span multiple administrative domains in order to manage end-to-end KPIs (in a similar manner to the resource advisor described in [8]). The Ecotopia advisors estimate the impact of observed, predicted, or scheduled events on a few service KPIs; for instance, we can define a performance advisor that predicts violations of the response-time objectives. The predictions do not depend on the actual enterprise business-value models, which are handled by the orchestrator.

The API of the advisors contains two functions, shown in see Table 1. `GetCurrentKPIs()` queries the KPI predictions if changes are not applied and it is used to assess the baseline for the change impact. `GetImpactKPIs()` retrieves the KPI predictions given a tentative change-operation schedule and is used to assess the impact the change schedule. These function invocations are synchronous (i.e., the requestor waits to receive the KPI predictions before proceeding). The reply includes the KPI predictions for the entire time horizon of the decision. This might span

multiple timeline points where the service KPIs change due to specific events such as expected workload changes or failures. These timeline points are called *prediction points*. The advisor reply includes one set of KPI predictions for each prediction point on the decision horizon. The replies can also suggest a set of *proactive actions* that are expected to improve the KPIs in conjunction with the change operations (*e.g.*, a “checkpoint database” action might reduce the recovery time). Proactive actions are included in the final change-operation schedule only if they improve the overall business value.

Orchestrator. The orchestrator is a resource broker and a change-operation planner. The orchestrator starts scheduling a group of change operations in two situations (see Table 1): (i) `InitiateChange()` indicates that a change sequence has been initiated, following a RFC; (ii) `InitiateResourceBrokering()` indicates that a predicted or observed infrastructure event (*e.g.*, a fault, a workload change) mandates a resource reassignment. All of these invocations on the orchestrator are asynchronous (*i.e.*, a response containing the schedule is not provided immediately). During the scheduling process, the orchestrator communicates with the objective advisors, asking “what-if” questions in order to assess the impact of tentative change-operation schedules on the future service KPI values.

Table 1. APIs of the Ecotopia framework components

Orchestrator	
<code>InitiateChange()</code> :	request for scheduling a group of change operations derived from an RFC.
<code>InitiateResourceBrokering()</code> :	request for reallocation of resources (<i>e.g.</i> nodes) to mitigate the impact of an event detected by the system management infrastructure (<i>e.g.</i> a hardware fault).
<code>ChangeSLA()</code> :	request for integration of SLA updates.
Objective Advisors	
<code>GetCurrentKPIs()</code> :	request for current KPI predictions for a given time interval, assuming no change applied (<i>i.e.</i> , only infrastructure events such as workload variation or node failures will occur).
<code>GetImpactKPIs()</code> :	request for KPI predictions over a given time interval for a schedule of change operations.

Based on the predicted KPIs, the orchestrator creates a tentative change-operation schedule and computes its overall business value (BV). The SLA defines service-level objectives based on the monitored KPIs (*e.g.*, a target for the average response-time) and associates a business-value function to each SLO (*e.g.*, a penalty for each request that misses the target). The orchestrator computes the overall BV for a particular state of the system by adding the business values of all the services and SLOs defined in

the service-level agreement. A change schedule will modify the overall BV by altering the state of the system and its monitored KPIs. When the orchestrator needs to choose among several alternative options for changing the system (*e.g.*, whether to include a proactive action in the schedule or not; all the possible times for scheduling a change operation), it uses the overall BV to select the best change-operation schedule. The overall BV reflects the utility of a change schedule and provides a way of comparing the effects of changes affecting multiple KPIs and SLOs.

The orchestrator is also invoked when an SLA has changed through `ChangeSLA()`, which indicates a modification in the overall business-value calculations. The orchestrator retrieves the new SLOs and the corresponding BV expressions and automatically updates its scheduling engine (more comprehensive mechanisms for managing SLAs updates are described in [24]). This is a reflexive hook allowing the orchestrator to update itself. In this case the change is applied immediately or at a specified time in the future, so it does not go through the scheduling process. New service-level agreements are typically defined in order to realign the business and IT objectives of the enterprise; therefore, the effect of the new SLAs must be reflected as soon as they are available.

The goal of change-operation scheduling is to maximize the business value for a certain time horizon. The Ecotopia orchestrator computes schedules for change-operation groups, which correspond to a request for change (RFC) or to a request for resource brokering. A schedule indicates when each individual change operation from the group will start executing. Using the overall business value, defined in the current SLAs, to compare different schedules, the orchestrator converges, through an iterative process, to the best feasible schedule.

3.2 “What-If” Interaction Protocol

The interaction protocol is at the heart of the Ecotopia framework. As shown in Fig. 2, a change sequence is initiated by the `ChangeManager` with the `InitiateChange()` function, or by an advisor with the `InitiateResourceBrokering()` function. The orchestrator initiates the “what-if” interaction by calling the `GetCurrentKPIs()` functions of each of the advisors to learn about their prediction points during the decision time horizon and to establish a baseline state for assessing the impact of the proposed schedules. Then the orchestrator creates and refines schedules through an incremental process. It invokes the `GetImpactKPIs()` functions on each of the advisors to acquire the KPI predictions necessary for assessing the impact of each of the proposed partial and complete schedules.

The orchestrator and the objective advisors exchange all the information about the current change group and change-operation schedule needed to assess the impact on the KPIs and to improve the schedule. Table 2 summarizes these parameters.

A *change operation* is defined by a name, a scope and a set of properties. The name is an enterprise-specific descriptor (*e.g.*, “Upgrade database software to version 10.0”) recognized by all of the related objective advisors and service managers. The scope identifies the resources (*e.g.*, “database node DB₁”) involved by the operation.

Table 2. Scheduling parameters

$CG(n, e_{1...n}, d_{1...n}, R, D)$	Change-operation group
n	Number of change operations in the group
e_i	Change operation
e_i'	Optional change operation
d_i	Duration of change operations e_i
$R(e_i, e_j)$	True if e_i must be executed before e_j
D	Deadline of the change group
m	Number of prediction points
Pp_k	Prediction point
T_H	Time horizon for scheduling and impact assessment
t_i	Time instant when change operation e_i is scheduled to begin.

The properties are a list of $\langle \text{name}, \text{value} \rangle$ pairs that describe operation characteristics such as the duration of executing the operation, the additional load imposed, etc. Change operations can be mandatory, such as the operations derived from an RFC, or optional, such as the resource-brokering operations. The scheduler can discard optional operations if they do not improve the business value. The set of operations in a group may expand during the scheduling process due to the proactive actions suggested by the objective advisors; in general, proactive actions can be considered optional.

Each change group defines a partial order among its constituent operations, indicating their precedence dependencies. A group may also specify a deadline for completing the execution of all its constituent operations and a business-value expression reflecting the penalty of late completion, which will be factored into the overall business value of the system to be maximized by the orchestrator. If the deadline information is missing, then the aggregated business value of the SLOs is the only criterion for selecting a schedule. A change-operation group can be preempted by the arrival of a group with a higher priority (e.g., if a previous change has damaged the system and needs to be rolled back).

The orchestrator uses the current KPI predictions as scheduling guidelines. The scheduler starts by invoking the `GetCurrentKPIs()` function of the objective advisors to retrieve the future variation of all the relevant KPIs due to infrastructure events (e.g., faults, workload surges) and changes that have already been scheduled. These prediction points indicate the time instants when the objective advisors expect the KPIs to change. After the scheduling of a change group is completed, the advisors add its impact on the infrastructure to the current KPI predictions.

To minimize the communication costs, the orchestrator might cache business-value information for partial schedules. Each unique schedule is tagged with an identifier (similar to a hash key), known to the orchestrator and advisors, and its related KPI predictions are saved. The orchestrator retrieves the predictions whenever it modifies the partial schedule by adding one or more change-operations, and thereby avoids repeating most of the computations.

4 Ecotopia Implementation

In this paper, we focus on the implementation of Ecotopia's orchestrator. The objective advisors rely on functionality provided by component-specific configuration managers [4, 5, 26, 27]. These managers encapsulate the extensive, and sometimes proprietary, domain knowledge (*e.g.*, workload characteristics, resource-utilization models), needed for assessing the impact of change operations on the service KPIs. For evaluating our framework, we have developed configurable emulators for the goal-advisors. We implement the orchestrator and the objective advisors as Web Services, which means that the orchestrator can interact with any third-party advisors that support the "what-if" interaction protocol described in Section 3.2.

4.1 Objective-Advisor Implementation

While the orchestrator is a centralized component, the objective advisors are distributed. Ecotopia uses an objective advisor for each SLO of each service defined the service-level agreement. For example, a performance advisor monitors the service to assess the response time, and a dependability advisor assesses the recovery time and the availability based on the amount of redundancy available in the current configuration. We implement the objective advisors in our framework in a hierarchical manner: as each service is composed of several other services, the advisor that corresponds to a top-level service queries several lower-level advisors corresponding to the component services. Every resource from the IT infrastructure is treated as a service: the network, the CPU, the disk, etc. have service-level objectives specifying the target for a set of KPIs, such as response time, throughput and recovery time.

The service composition and the mapping of services onto physical resources define a request queuing-path for each service. A change operation modifies this queuing path, either by altering its structure (*e.g.*, by defining a new service composition), or by modifying the parameters of the component queues (*e.g.*, by replacing a CPU with a faster one or by removing a replica from a load-balanced system). The advisors use this domain knowledge to answer "what-if" questions about service KPIs (such as performance and availability forecasts), based on the description of the change operations and the schedule.

The advisors corresponding to the primitive services contain analytical models of the corresponding resources and estimate the value of the KPIs based on the workload and configuration. For instance, the performance advisors estimate the response time of a primitive resource using the operational laws of queuing theory [28, 29], based on the incoming request rates and the known peak throughput of the resource. Higher-level advisors compute their KPI predictions by combining the outputs of the lower-level advisors along the corresponding queuing path. The composite queuing paths can be either sequential (*e.g.*, a request travels through a front-end, a local-area network and then a back-end) or parallel (*e.g.*, a load-balancer forwards the request to one of several servers for further processing). The parallel queuing paths do not necessarily have the same length; for instance, a request for a data item present in a proxy cache has a shorter path than a request that results in a cache miss and that

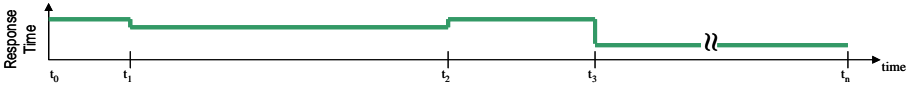


Fig. 3. A KPI (e.g., average latency) varies in time, depending on the workload and the system configuration. We represent this variation by a vector of $\langle t, KPI(t) \rangle$ pairs indicating the time when a KPI changes and the new value. This corresponds to a step function as shown in the figure.

needs to be forwarded to the application server for processing. The parallel queues have probabilities associated with each alternative path representing the percentage of requests that travel along those paths.

Our implementation is similar to the resource advisor described in [8]; in addition, we leverage workload predictions to estimate the long-term KPI variation. KPIs change in time; therefore, the advisors provide KPI estimations as time-varying functions $KPI(t)$. A KPI value is assumed to hold for a period of time, until some event causes the KPI to take another value. This means that $KPI(t)$ is a step function, as shown in Fig. 3. When replying to the invocation of `GetCurrentKPIs()`, the objective advisor will provide a list of pairs $\langle Pp_k, KPI(Pp_k) \rangle$, indicating the times (prediction points) Pp_k when the KPI is expected to change and the corresponding KPI values (see Table 2). `GetImpactKPIs()` returns a similar list, indicating the effect of the suggested change schedule on the KPIs, computed using the service queuing-path created by the change.

4.2 Orchestrator Implementation

The orchestrator generates change-operation schedules, which associate start times $t_1, t_2 \dots t_n$ with operations $e_1, e_2 \dots e_m$, respectively, which have the respective durations $d_1, d_2 \dots d_n$ (see Table 2). The schedule must comply with the partial ordering among operations and the group deadline D (if defined). During scheduling, the orchestrator queries the objective advisors for predictions of the impact on KPIs during the relevant time-horizon and uses these predictions to compute the overall business value and to refine the schedule. The time horizon T_H must be long enough to include the deadline D , but in general will be longer, in order to account for the KPI impact after the change has been executed. The aim of the scheduling process is to provide the best possible business value.

The orchestrator does not know the closed-form equation that yields the overall business value because part of this computation is performed inside the objective advisors, which act as black boxes for the orchestrator. In scheduling-theory terms, this means that the scheduling problem has an unknown objective function [30]. Given that the complexity of scheduling algorithms depends on their objective functions, it is impossible for us to reason about the complexity of our problem. Moreover, even if we had a closed-form expression for the business value, this would most likely be a non-regular objective function (a regular objective function is non-decreasing in the completion times of the change operations); there are few theoretical results for scheduling problems with non-regular objective functions. We therefore

focus on approximate scheduling algorithms that make the best effort to compute a solution close to the optimal schedule.

Business-value model. The SLO business values are functions that associate a dollar value with various levels of service provided by the system. A service-level objective defines a target for a particular KPI. A service may have multiple SLOs (some of these objectives may track a common KPI, *e.g.*, the target bounds for average latency and maximum latency), and each SLO has a business-value function. Since the KPIs change in time (see Fig. 3), the business values are also time-variable functions. At time t , a KPI value is $KPI(t)$ and the corresponding business value is: $BV_{SLO}(KPI(t))$. For each KPI that changes at times t_0, t_1, \dots, t_n , the business value for the time interval $[t_0, t_n]$ is computed using a weighted average:

$$BV_{SLO}([t_0, t_n]) = \frac{\sum_{i=0}^{n-1} BV_{SLO}(KPI(t_i))(t_{i+1} - t_i)}{t_n - t_0} \quad (1)$$

The business-value functions of different SLOs are designed to be additive. They are used for reasoning about the multiple impacts of various change operations and for selecting the best trade-offs. We add the business values of all the SLOs to compute the overall business value, which reflects the utility of the proposed schedule of operations:

$$BV_{All}([t_0, t_n]) = \sum_{All\ SLO_k} BV_{SLO_k}([t_0, t_n]) \quad (2)$$

Scheduling assumptions. In this paper, we make a few simplifying assumptions about our scheduling problem. First, we assume that all the operations in a change group are mandatory (there are no proactive actions). Second, we assume that all the change-operation groups have explicit deadlines. When not defined explicitly, the deadline can be fixed to the end of the time horizon for business-value evaluation; it makes no sense to schedule operations past this time horizon because we would not be able to see their impact on the business value. Third, the operations in a change group are totally ordered (*i.e.* an operation must complete before the next one can begin). While these assumptions are somewhat constraining, we believe that in practice there are many change-management situations that satisfy these constraints (we provide an example in Section 5).

Scheduling algorithms. The algorithms we have implemented are based on the following pattern. Each operation e_k has a *feasible scheduling interval*, defined by the earliest and latest times when e_k can be scheduled to allow enough time for the prior and subsequent operations:

$$\sum_{i=1}^{k-1} d_i \leq t_k \leq D - \sum_{i=k}^n d_i \quad (3)$$

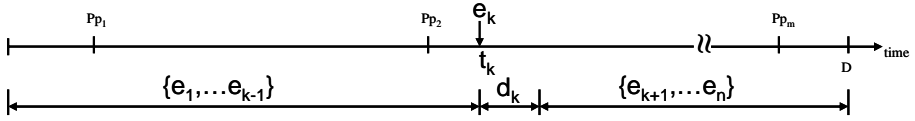


Fig. 4. Our Greedy algorithm for scheduling change operations first chooses the change operation e_k and the time t_k that yield the best business value. This placement splits the timeline indicating the future variation of the KPIs) that fall within this feasible interval. and the change-operation group in two, and we apply the same algorithm to the two halves of the problem.

Using these bounds, we try to schedule each change operation at the earliest possible time, the latest possible time and at all the m prediction points (time instants indicating the future variation of the KPIs) that fall within this feasible interval.

The baseline scheduler is a *backtracking algorithm* that generates and evaluates all of the possible placements for the change operations in a group. We start with the first event e_1 and we place it at all the prediction points from its feasibility interval ($t_1=0$, $t_1=Pp_1$, $t_1=Pp_2$, etc.). For each of these values of t_1 , we repeat the algorithm for the remaining operations and the new boundaries of the timeline (since we have started with the first operation, the deadline stays the same and the start time becomes t_1+d_1 , the time when e_1 will complete). When we have successfully scheduled all the operations from the change group, we compute the corresponding business value by invoking `GetImpactKPIs()` on the relevant advisors. We then backtrack to try other possible placements of e_n , then of e_{n-1} etc., and we save the schedule that generates the highest business value.

If the KPIs are expressed as step functions, as shown in Fig. 3, and the business values are linear functions of the KPI values (which would make them step functions as well), this algorithm generates the optimal schedule. For each operation e_k , there may be m assignments of t_k . An assignment of t_{n-1} will be tested in combination with m assignments of t_n . An assignment of t_{n-2} will be tested with m assignments of t_{n-1} , each of which will be tested with m assignments of t_n ; therefore, an assignment of t_{n-2} requires m^2 more operations for determining the best corresponding business value. By induction, this algorithm, henceforth called *Backtracking*, has the worst-case complexity $O(m^n)$.

A more realistic scheduler uses a polynomial best-effort algorithm that is not guaranteed to provide an optimal solution. We achieve this with a *greedy algorithm*: we place each operation e_k at each prediction point from its feasibility interval and we compute the business value that corresponds to this placement (during this step, we are only interested in the impact of e_k , so we invoke `GetImpactKPIs()` on the relevant advisors for a schedule that contains only e_k). We select the operation and the placement that yield the best possible business value. This placement splits the timeline and the change-operation group in two, and the same algorithm is applied recursively to the two segments of the problem, as shown in Fig. 4. Operations $e_1 \dots e_{k-1}$ will be scheduled between $[0, t_k]$, and operations $e_{k+1} \dots e_n$ will be scheduled between $[t_k+d_k, D]$.

The first iteration of this algorithm performs nm BV comparisons. In the worst case, the timeline partitioning will be skewed such that e_1 will be chosen and all the prediction points will fall after t_1+d_1 . the second iteration will then require $m(n-1)$ BV

comparisons. Since there are n iterations, this algorithm (Greedy1) has the complexity $O(n^2m)$.

This algorithm has the disadvantage that it tends to give priority to the short operations that have a small negative impact. These operations get the best placements, sometimes leaving the large operations to be scheduled during busier periods, thus affecting the overall business value. To avoid this situation, we can modify the selection condition in the following manner: at each iteration, we choose the operation e_k that displays the largest business value variation depending on the scheduling time. This strategy leads to selecting the operation most sensitive to placement first. This algorithm, called Greedy2, has the same complexity as the previous one: $O(n^2m)$.

Schedule Stability. The schedules generated by the orchestrator remain constant in the absence of any additional change requests, SLA updates or system management events such as faults or workload changes. Fig. 5 shows that all the changes that might affect the final schedules are always initiated outside the scheduling loop involving the orchestrator and the advisors, which ensures the stability of our protocol. The advisors generate deterministic KPI predictions for a given change group (*i.e.*, the same tentative schedule will yield the same predictions).³ The predictions returned by `GetCurrentKPIs()` will be adjusted in between change groups because the effects of the change that has just been scheduled are factored into the KPI predictions; however, no such adjustment is performed inside the scheduling loop. The algorithms presented above are guaranteed to converge if the KPI predictions are deterministic for a given change group. Other autonomic management systems based on iterative optimization loops [9, 21] may oscillate between borderline decisions because a resource reconfiguration will affect the performance metrics which may subsequently trigger another reconfiguration. Ecotopia, where all of the changes are initiated outside the scheduling loop and the “what-if” analysis considers a long time-horizon, guarantees that such infinite cyclic dependencies are broken and that thrashing cannot occur.

Canceling and Undoing Scheduled Changes. One corner case when the KPI predictions are not deterministic is when a fault or a load-surge prediction occurs while the scheduling loop is executing. Rather than updating the KPI predictions, in this case, we cancel the scheduling of the change group in order to avoid confusing the scheduler. Moreover, a fault or a load surge will typically be associated with a change request that has the highest urgency, so it is important to start scheduling this change as soon as possible. In general, whenever the orchestrator receives an urgent change request, it will preempt the currently executing scheduling process, and will start working on the new request immediately.

In some cases, it becomes obvious that a scheduled change does not have the desired effect and must be abandoned. If the change group has been scheduled but not yet implemented, it can be canceled easily. More often, however, this decision is taken only after the change has been finalized. In this case, another change has to be

³ The interaction protocol described in Section 3.2 also relies on this property because the orchestrator and the advisors cache the KPI predictions corresponding to partial schedules.

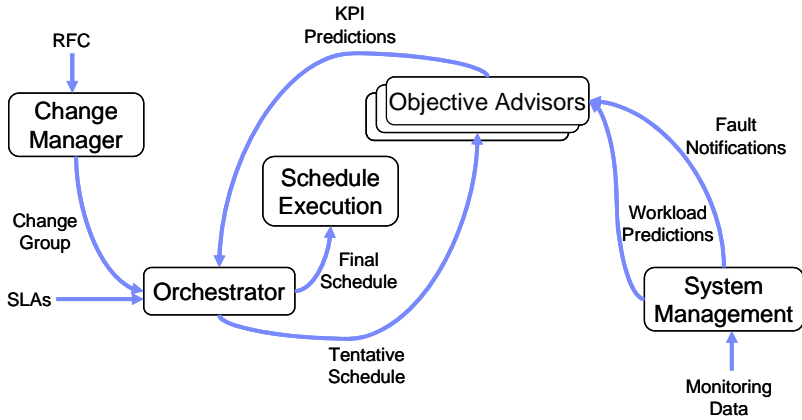


Fig. 5. The scheduling loop of Ecotopia is designed such that all the change requests originate from outside of the iterative interaction between the orchestrator and the objective advisors. This ensures that the scheduling process does not oscillate between borderline decisions.

scheduled to undo the effects of the previous one. The logs of the orchestrator can assist this operation by defining the reverse operations needed to undo the undesirable change, but the process must be guided by an administrator since the autonomic infrastructure has failed to take into account the negative effects of the change. In many cases, these errors are due to bad SLAs, which then have to be reworked by the system administrator. If the KPI predictions are accurate enough, we are confident that human interventions for correcting the orchestrator's decisions will be uncommon. Note that, since the decision to undo is not made by the orchestrator, the stability guarantees described above are not affected.

5 Case Study: Two-Tiered Enterprise Infrastructure

We consider a two-tiered system, where the physical hosts are organized in independently-managed node-groups. The first tier is a node group of application servers managed by application server middleware (*e.g.*, IBM WebSphere Extended Deployment [6]) and the second tier is a node group of database servers, managed by database cluster infrastructure (*e.g.*, Oracle Clusterware [27]). The two node-group managers perform various management tasks (*e.g.*, load balancing, request routing, fault recovery).

This infrastructure, illustrated in Fig. 6, provides two services, each mapped onto corresponding application-server and database services. The two services processing Web transactions are load-balanced across three application-servers, Srv_1 to Srv_3 . These front-end services query two database services that connect to separate database partitions. The database group comprises three nodes:

- DB_1 acts as primary server for Service1 and as backup for Service2;
- DB_2 is part of the logical primary server for Service2, which is distributed on two database nodes;

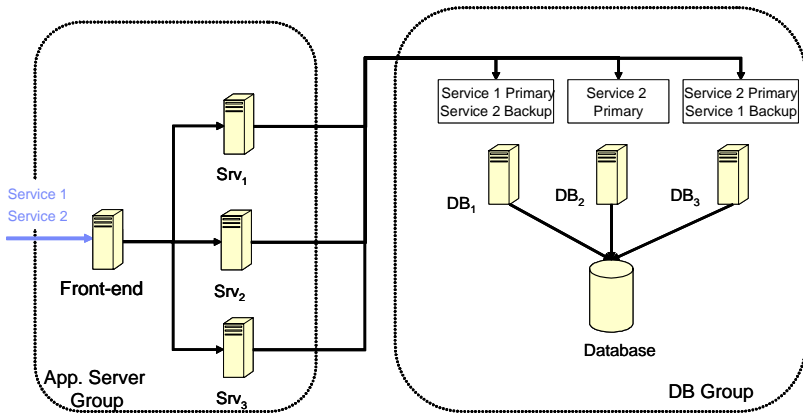


Fig. 6. Example: two-tier system

- DB_3 is also part of the logical primary for *Service2* and it is a backup for *Service1* as well.

Each of the two enterprise services has response time, recovery time and availability objectives. The business value associated with these SLOs depends on the related KPIs, such as ‘total number of transactions’, ‘number of transactions with response time below target’, etc.

A performance advisor evaluates the impact of change operations on the end-to-end response time for each service by exploiting the knowledge provided by the node-group managers (*e.g.*, expected workload variations, service overheads). Similarly, a dependability advisor evaluates the impact on the recovery time and the availability SLOs.

5.1 Qualitative Evaluation

For evaluating the Ecotopia change-management framework in this context, we discuss two realistic change-management scenarios for this case study: a crash of node DB_1 and an upgrade of the database software. We complement this analysis with measurements illustrating the trade-off between the cost and the loss of optimality of different scheduling algorithms (Section 5.2).

Scenario 1: Hardware crash. When the dependability advisor detects the crash of DB_1 , the corresponding node-group manager takes immediate recovery measures. The database recovery manager handles the failover of *Service1* to its backup node, DB_3 . As a result, DB_3 handles queries for both services, while DB_2 continues to handle only queries for *Service2*. However, since the database group now has fewer nodes, and an accompanying higher risk of failing the availability objectives, the change-management system must decide whether removing one node from the application server group and adding it to the database group would improve the overall business value and when these operations should be scheduled.

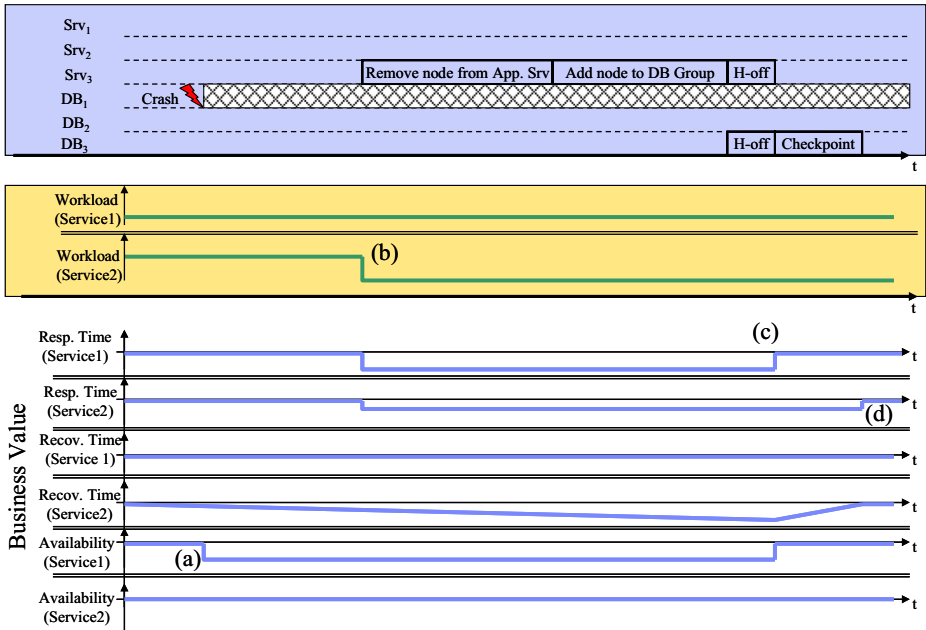


Fig. 7. Hardware crash and fault-management scenario

Fig. 7 shows the impact of these change operations. After the crash of DB_1 , the lack of a backup leads to a sharp decrease of the predicted availability of *Service1* and a drop in the corresponding business value – indicated by point (a) in the figure. However, since the load of *Service2* is high at this point, transferring a node from the application-server group to the database group would fail to meet the response time objective. Therefore, the orchestrator delays the change operations until the load of *Service2* decreases, at point (b). During the node transfer, the response time decreases for both services, but after the hand-off – point (c) – the response times, as well as the availability of *Service1*, may return to normal. However, since *Service2* has been continuously sending queries to the database, its log kept growing, leading to an increase of the recovery time. To solve this problem, the dependability advisor requests a proactive action in the form of a database checkpoint (synchronizing the modified data blocks in memory with the disk and shortening the log processed during recovery). After the checkpoint, indicated by point (d), the response time and the recovery time for *Service2* decrease to normal operating levels.

Scenario 2: Database upgrade. A similar impact analysis must be undertaken when upgrading the database software (Fig. 8). In this case, a request for change is decomposed into finer-grained change operations: each database node is upgraded separately and, for upgrading DB_1 , *Service1* is handed off to DB_3 (its backup) before the upgrade and restored at the end. The analysis must consider the impact of

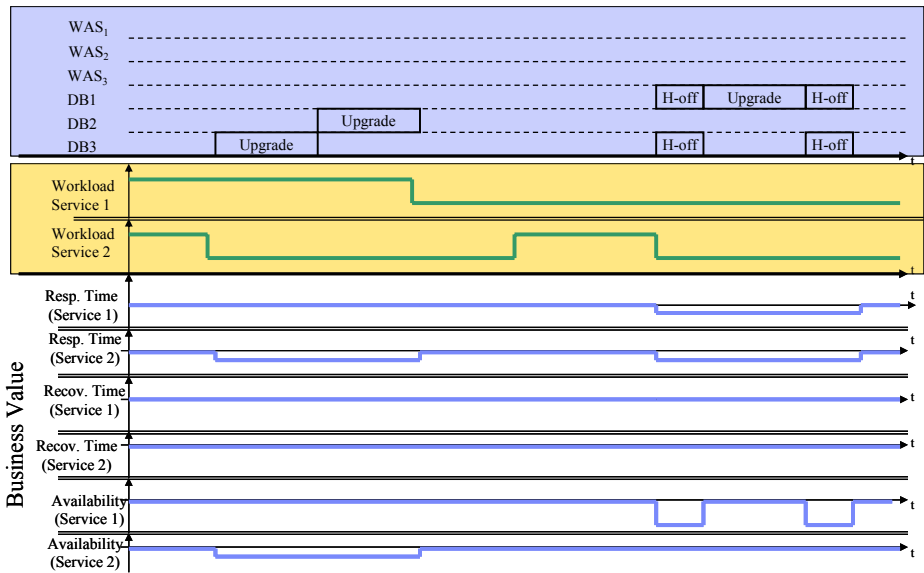


Fig. 8. Database-upgrade scenario

these operations on service objectives and their corresponding business values. For instance, if the load on *Service1* is high, we can reorder the change operations to perform the upgrades on nodes DB_2 and DB_3 , which are used by *Service2*. In fact, the upgrade of DB_1 must be delayed until both services register low incoming request rates because a high request rate during the upgrade may overload DB_3 , which also handles both *Service1* and *Service2*. By delaying the upgrade, the penalties incurred for violating the response time objectives are minimal, thus maximizing the aggregate business value for the duration of the changes. The reordering must take into account the dependencies between change operations; thus, the hand-offs of *Service1* should precede and follow the upgrade of DB_1 .

These scenarios show that delaying the change operations may sometimes improve the overall business value. Such situations are typical of change management in an enterprise infrastructure; similar operations occur at a much larger scale in many real-life deployments. This illustrates the complexity of predicting the impact of change due to the strong dependencies on the actual implementations of objective managers. Our framework addresses these issues by delegating the impact assessment to objective-specific advisors that encapsulate all the relevant domain knowledge.

5.2 Quantitative Evaluation

Using a traditional scheduler, which does not optimize for long-term impact [5, 7, 9, 21], would result in executing all of the change operations as soon as possible, instead of waiting for the most opportune time when the incoming load is low. The outcome of such impact-insensitive scheduling is a missed opportunity for optimizing the overall business value. Instead, the scheduling algorithms presented in Section 4.2

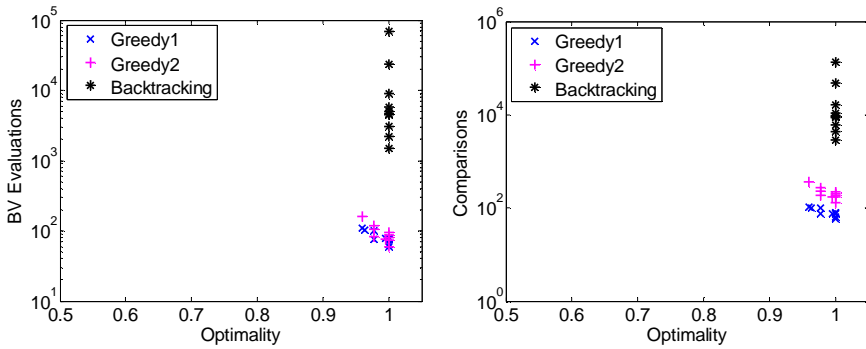


Fig. 9. Scheduling algorithms: trade-off between cost and loss of optimality. The Greedy algorithms are polynomial and yield schedules with a business value within 95% of the optimal achievable business value, which is computed using the exponential backtracking algorithm.

find the optimal schedule for these two scenarios and the run-times of all the algorithms – including the exponential backtracking scheduler – are comparable (less than 1s).

We also test our scheduler using several randomly-generated input sets, and we explore the trade-off between complexity and the loss of optimality. The most appropriate complexity measure is the number of times the business value needs to be evaluated, since these evaluations require communication between the orchestrator and the advisors; we do not report the run-times because they depend heavily on the hardware resources used for simulation. The loss of optimality shows how close the BV of the resulting schedule was to the BV of the optimal schedule, as generated by the backtracking algorithm. Fig. 9 shows that, for small problems (*e.g.*, 5 change operations and 10 KPI prediction points), the two (polynomial) greedy algorithms obtain near-optimal results and they need one or two orders of magnitude fewer BV evaluations than the exponential, optimal backtracking algorithm.

For larger problems, we cannot use the backtracking algorithm and, therefore, we cannot measure the loss of optimality of the greedy schedulers. For 100 change events and 100 prediction points, the greedy algorithms required up to 36673 business-value evaluations and 67342 comparisons, sometimes with significant differences between the two algorithms (between 3% and 68%). Greedy1 also exhibits a higher variance of the number of BV evaluations than Greedy2. While we could easily construct a scenario where Greedy2 performs better than Greedy1, the two algorithms produced identical schedules for all but one of the randomly generated scenarios.

6 Discussion

By focusing on the communication protocol for impact assessment rather than on building a monolithic change-management system, Ecotopia facilitates changes that might span multiple independent administrative domains and that might target heterogeneous software infrastructures. Our generic orchestrator can communicate

with third-party advisors, which are built with specific, proprietary domain knowledge about a service/system/vendor, and construct schedules using only the information available from such advisors. This approach mirrors the philosophy of Service-Oriented Architectures, which is to focus on interaction protocols rather than on implementation bindings.

The separation between scheduling and impact assessment makes Ecotopia applicable to realistic systems, although it may limit its optimization capabilities when the advisors cannot provide a comprehensive impact analysis (*e.g.*, some services may not provide latency estimations, which are required for end-to-end response-time management). Moreover, the KPI predictions will inevitably have a degree of inaccuracy, especially when the time frame of the predictions is far ahead in the future. The orchestrator will generate change schedules even with imperfect information about the system; however, the quality of the schedules will improve with accurate impact analysis. If the advisors provide incorrect information, the orchestrator might take the system to a state with unacceptable service levels; in this case, a downgrade or the rollback of the changes can be scheduled using the same process described above. This raises two questions that we plan to investigate in the future: how much prediction inaccuracy can the orchestrator tolerate while keeping its ability to offer meaningful recommendations, and what kind of predictions and impact analysis can the advisors perform to enable ecological change-management planning.

Another open question is how to determine the typical size of realistic change-operation groups, which is important for selecting a good scheduling algorithm. The optimal scheduling-algorithm works well for the case study presented in this paper; however, we cannot use it for change groups with more than 10 operations, because of its exponential complexity. For very large problem sets, we may need to use heuristics such as genetic algorithms or simulated annealing [30]. We also plan to investigate the possibility of defining an adaptive scheduler that selects the best algorithm depending on the properties of the change-operation group (*e.g.*, its size).

The best way to express the KPI variation in time also warrants further exploration. The step function representation used in this paper might be too constraining; for instance, it cannot describe a recovery time that increases linearly with the increase over time of the database log, as depicted in Fig. 7. However, this representation is easy to understand and to use (as opposed to a describing a generalized function), and it can approximate well an arbitrary KPI trajectory if enough change points are selected. Furthermore, using the change points as scheduling guidelines, allows us to use simple algorithms even for a scheduling problem with an unknown objective function.

7 Conclusions

This paper investigates the problem of performing dynamic change management while maximizing the aggregate business value across all SLOs of the enterprise. We propose Ecotopia, a novel ecological framework for change management that tackles the complexity and the distributed nature of SLO management in real-world systems by separating the impact assessment (performed by the objective advisors) from the scheduling and business-value computation and aggregation (performed by the

orchestrator). A novel “what-if” interaction protocol between advisors and orchestrator enables an efficient computation of SLO business values and change schedule refinement, Ecotopia performs ecological change management by taking into account the impact on the enterprise SLOs, the long-term KPI variations and the heterogeneous types and sources of change operations (both internal and external). We validate our framework using two realistic change scenarios that emphasize that impact assessment is essential for maximizing the business value. Our preliminary simulations compare the trade-offs between the cost and the loss of optimality of three scheduling strategies.

Acknowledgments. The authors would like to thank Biswaranjan Bhattacharjee and Joel Wolf of IBM Research, Florin Oprea of Carnegie Mellon University, as well as Jean-Charles Fabre of LAAS CNRS for their input during the early stages of this research.

References

1. Kirkley, J.: Aligning IT and Business as the Economy Rebounds. Enterprise Leadership, BMC Software 2 (2004)
2. Gartner Group: High Availability Q&A: Failures, Standards and Metrics. Networked Systems Management Research Note QA-05-2701 (1998)
3. Global Grid Forum: Web services agreement specification (WS-Agreement). Draft, version 11 (2004)
4. Whalley, I., et al.: Experience with collaborating managers: node group manager and provisioning manager. Cluster Computing 9, 401–416 (2006)
5. IBM Tivoli Intelligent Orchestrator, <http://www-306.ibm.com/software/tivoli/products/intell-orch>
6. IBM WebSphere Extended Deployment, <http://www-306.ibm.com/software/webservers/appserv/extend>
7. Keller, A., et al.: The CHAMPS system: Change management with planning and scheduling. In: Network Operations and Management Symposium, pp. 395–408. Seoul, Korea (2004)
8. Thereska, E., et al.: Informed Data Distribution Selection in a Self-predicting Storage System. In: International Conference on Autonomic Computing, Dublin, Ireland (2006)
9. Anderson, E., et al.: Hippodrome: Running Circles Around Storage Administration. In: USENIX Conference on File and Storage Technologies (FAST '02), Monterey, CA, 13(2002)
10. Segal, M., Frieder, O.: On-the-fly program modification: Systems for dynamic updating. IEEE Software 10, 53–65 (1993)
11. Kramer, J., et al.: Towards Unifying Fault and Change Management. In: Workshop on Future Trends of Distributed Computing Systems in the 1990s, Cairo, Egypt, pp. 57–63 (1990)
12. Moser, L.E., et al.: Eternal: fault tolerance and live upgrades for distributed object systems. In: DARPA Information Survivability Conference and Exposition (DISCEX 00), Hilton Head, SC, pp. 184–196 (2000)
13. Bloom, T., Day, M.: Reconfiguration in Argus. In: Workshop on Configurable Distributed Systems, London, England, pp. 176–187 (1992)

14. Dilley, J.: Web server workload characterization. Technical Report HPL-96-160, Hewlett-Packard Laboratories (1996)
15. Vallamsetty, U., et al.: Characterization of E-Commerce Traffic. *Electronic Commerce Research* 3, 167–192 (2003)
16. Arlitt, M., Jin, T.: A workload characterization study of the 1998 World Cup Web site. *IEEE Network* 14, 30–37 (2000)
17. www.alexa.com
18. Pertet, S., Narasimhan, P.: Proactive Recovery in Distributed CORBA Applications. In: *International Conference on Dependable Systems and Networks (DSN)*, Florence, Italy, pp. 357–366 (2004)
19. Zhang, Q., et al.: Workload-aware load balancing for clustered Web servers. *IEEE Transactions on Parallel and Distributed Systems* 16, 219–233 (2005)
20. Peltz, C.: Web services orchestration and choreography. *IEEE Computer* 36, 46–52 (2003)
21. Golding, R.A., Wong, T.M.: Walking toward moving goalposts: agile management for evolving systems. *Hot topics in autonomic computing, HotAC*, Dublin, Ireland (2006)
22. Beattie, S., et al.: Timing the Application of Security Patches for Optimal Uptime. In: *Large Installation System Administration Conference*, Philadelphia, PA, pp. 233–242 (2002)
23. Gorbenko, A., et al.: Dependable Composite Web Services with Components Upgraded Online. In: de Lemos, R., Gacek, C., Romanovsky, A. (eds.) *Architecting Dependable Systems III*. LNCS, vol. 3549, pp. 92–121. Springer, Heidelberg (2005)
24. Roşu, D., Dan, A.: Managing end-to-end lifecycle of global service policies. In: *International Conference on Service Oriented Computing*, Amsterdam, The Netherlands, pp. 570–575 (2005)
25. Keller, A.: Automating the Change Management Process with Electronic Contracts. In: *International Workshop on Service Oriented Solutions for Cooperative Organizations*, Yorktown Heights, NY, pp. 99–107 (2005)
26. WebSphere Extended Deployment Version 5.1 Information Center (2004)
27. Oracle Corporation: Oracle Real Application Cluster 10g. Oracle Technical White Paper (2005)
28. Lazowska, E., et al.: Quantitative System Performance: Computer System Analysis sing Queuing Network Models. Prentice-Hall, Englewood Cliffs (1984)
29. Urgaonkar, B., et al.: An analytical model for multi-tier internet services and its applications. In: *International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, Banff, Alberta, Canada, pp. 291–302 (2005)
30. Pinedo, M.: *Scheduling: Theory, Algorithms, and Systems*. Prentice-Hall, Englewood Cliffs (2002)

Generic-Events Architecture: Integrating Real-World Aspects in Event-Based Systems^{*}

António Casimiro¹, Jörg Kaiser², and Paulo Verissimo¹

¹ Univ. Lisboa

{casim,pjv}@di.fc.ul.pt

² Univ. Magdeburg

kaiser@ivs.cs.uni-magdeburg.de

Abstract. In a future networked physical world, a myriad of smart sensors and actuators assess and control aspects of their environments and autonomously act in response to it. To a large extent, such systems operate proactively and independently of direct human control. They include computer hardware and software parts mixed with mechanical devices. Besides the regular computer communication channels, they also establish interaction channels among them directly through the environment. These characteristics pose a number of fundamentally new consistency and correctness challenges which, if not met, may hinder the dependability of such systems, and ultimately lead to unexpected failures.

This paper describes an architectural framework and event model capable of solving these pressing problems. Firstly, we offer an innovative composable object model representing software/hardware entities capable of interacting with the environment. Secondly, we provide event-based communication seamlessly integrating real-world events and events generated in the system. The crucial parts of our work are the generic-events architecture GEAR, hosting the COSMIC middleware supporting the events model, with attributes to express spatial and temporal properties.

1 Introduction

In a future networked physical world, a myriad of smart sensors and actuators assess and control aspects of their environments and autonomously act in response to it. Examples range in telematics, traffic management, team robotics or home automation to name a few. In fact, this is also made possible by the continuous improvement of technologies that are relevant for the construction of these systems, including trustworthy visual, auditory, and location sensing [1], communication and processing. To a large extent, such systems operate proactively and independently of direct human control, instead driven by the perception of the environment and the ability to organize their own computations and actuations dynamically. The challenging characteristics of these applications include

^{*} This work was partially supported by the EC, through project IST-FP6-STREP-26979 (HIDENETS), and by the FCT, through the Large-Scale Informatic Systems Laboratory (LaSIGE).

sentience and autonomy of components, issues of responsiveness and safety criticality, geographical dispersion, mobility and evolution. In order to deal with these challenges, it is of fundamental importance to use adequate high-level models, abstractions and interaction paradigms. Unfortunately, when facing the specific characteristics of the target systems, the shortcomings of current architectures and middleware interaction paradigms become apparent.

As basic building blocks of such systems we often find computer hardware and software parts mixed with mechanical devices, and one or more network interfaces. In consequence, these components have different characteristics compared to pure software components. Moreover, these artifacts, besides the regular computer communication channels, also establish interaction channels among them directly through the environment. They are able to spontaneously disseminate information in response to events observed in the physical environment or to events received from other components. Larger autonomous components may be composed recursively from these building blocks.

However, classical event/object models are usually software oriented and, as such, when transported to a real-time, embedded systems setting, their harmony is cluttered by the conflict between, on the one side, send/receive of “software” events (message-based), and on the other side, input/output of “hardware” or “real-world” events, register-based. In terms of interaction paradigms, and although the use of event-based models appears to be a convenient solution [2,3], these often lack the appropriate support for non-functional requirements like reliability, timeliness or security.

The afore-mentioned characteristics pose a number of fundamentally new consistency and correctness challenges which, if not met, may hinder the dependability of such systems, and ultimately lead to unexpected failures. We believe that the first step in solving the former lies on an architecture allowing to: express and represent the software/hardware structure of these components and their composability; bridge the consistency gap between the events representing the physical environment generated by sensor readings and the events encapsulating state changes resulting from computations of the system.

This paper describes an architectural framework and event model capable of solving these pressing problems. Firstly, we offer an innovative composable object model which represents software/hardware entities capable of interacting with the environment. Secondly, we provide event-based communication seamlessly integrating real-world events and events generated in the system. After providing an overview of related work, the paper starts by clarifying several issues concerning our view of the system, about the interactions that may take place and about the information flows. This view is complemented by providing an outline of the component-based system construction and, in particular, by showing that it is possible to compose larger applications from basic components, following an hierarchical composition approach. The crucial parts of our work, the generic-events architecture GEAR, hosting the COSMIC middleware supporting the events model, are then introduced.

The Generic-Events Architecture (GEAR) describes the event-based interaction between the components via a generic event layer. This layer integrates different communication channels including the interactions through the environment. Because interaction with the physical world requires real-time properties, concepts of time describing the aging of information and temporal consistency have to be included, in order to allow correct representation of these interactions by algorithms. The paper devotes particular attention to this issue by discussing the temporal aspects of interactions and the needs for predictability.

Finally, an appropriate event model is presented, as well as the COoperating Smart devices (COSMIC) middleware, which reflects the properties of GEAR and allows specifying events with attributes to express spatial and temporal properties. This is complemented by the notion of *Event Channels (EC)*, which are abstractions of the underlying network and enforce the respective quality attributes of event dissemination. Event channels reserve the needed computational and network resources for highly predictable event systems.

The paper is organized as follows. Section 2 presents related work. Then, Section 3 introduces fundamental notions and abstractions related to the sentient object model adopted in this paper. GEAR is then described in Section 4, while Section 5 describes the event model and the COSMIC middleware, which may be used to specify the interaction between sentient objects. The temporal aspects of interactions, which are crucial in the context of this paper, are covered in Section 6. Then, Section 7 presents two examples of the applicability of GEAR concepts. Finally, Section 8 concludes the paper.

2 Related Work

Our work considers a wired physical world in which a very large number of autonomous components cooperate. They are interconnected by multiple heterogeneous networks. Islands of tight control may be realized by a control network and cooperate via wired or wireless networks covering a large number of these subnetworks. In an earlier work, we referred to such a network structure as a Wide-Area-Network of Controller-Area-Networks (WAN-of-CANs) [4]. Due to the dynamic nature of interactions, it will be difficult or impossible to know a priori the communication participants and secondly, because of heterogeneity, we have to support a variety of different addressing structures and mechanisms. Thirdly, components should be autonomous and no control transfer should be bound to communication. In general, event-based systems supporting autonomy of components and the spontaneous dissemination of information have been recognized as appropriate to support large scale distributed systems [5,6]. Autonomy is achieved by a data driven model in which components decisions and actions are based on sharing data rather than on explicit control transfer [7]. Dynamic interactions are supported by content and subject related addressing schemes that enable communication without a priori knowing addresses or names of communication participants. Intended for large general purpose distributed applications, systems like [5,8,3] require quite complex infrastructures and do

not consider stringent quality aspects like timeliness and dependability issues. A comprehensive overview and taxonomy is provided in [6]. Some publisher subscriber communication systems have been proposed to be used in control applications. The Information Bus [9] and its realization by the TIBCO Rendezvous software [10] integrates some soft real-time features. E.g., it is possible to create several prioritized queues and explicitly associate event types with event queues but real-time programming can be not handled at the same abstract level as the event programming. The NDDS protocol [11] and its successors from RTI [12] have some real-time properties. Its architecture explicitly assumes the Ethernet as the underlying interface. Therefore, real-time can only be assured on a probabilistic basis. It means that the communication load must be known in advance and that deviations from the load hypothesis are minimally tolerated. Deadlines are specified only for subscriptions. Therefore message scheduling is supported only in the queues maintained in the receiver side.

In the area of large control systems, Autonomous Decentralized Systems (ADS) have been proposed [13]. They provide a shared data field which decouples producers of information and consumers which autonomously may retrieve (control) information from the data field. ADS is based on a quite complex content-based addressing scheme similar to Linda [14] and thus is difficult to run on resource constraint components as smart sensors and actuators.

In [2] a real-time event system for CORBA has been introduced. The events are routed via a central event server which provides scheduling functions to support the real-time requirements. Such a central component is not available in an infrastructure envisaged in our system architecture and the developed middleware TAO (The Ace Orb) is quite complex and unsuitable to be directly integrated in smart devices. There are efforts to implement CORBA for control networks, tailored to connect sensor and actuator components [15,16]. They are targeted for the CAN-Bus [17], a popular network developed for the automotive industry. However, in these approaches the support for timeliness or dependability issues does not exist or is only very limited.

A new scheme to integrate smart devices in a CORBA environment is proposed in [18] and has lead to the proposal of a standard by the Object Management Group (OMG) [19]. Smart transducers are organized in clusters, connected to a CORBA system by a gateway. They form isolated subnetworks in which a special master node enforces the temporal properties. A CORBA gateway allows to access sensor data and to write actuator data by means of an interface file system (IFS). In contrast to the event channel model introduced in this paper, all communication inside a cluster relies on a single technical solution of a synchronous communication channel. Secondly, although the temporal behavior of a single cluster is rigorously defined, no model to specify temporal properties for cluster-to-CORBA or cluster-to-cluster interactions is provided.

It should be noted however that all the event-based systems discussed in this section lack the holistic view of an architecture which integrates the events of the environment with those of the system. This is a pre-condition to meet the fundamentally new consistency and correctness challenges brought by complex

systems of embedded systems. In particular, we address the kind of synchrony properties that allow the ordering and the scheduling of the events on the communication medium, and the conditions for their enforcement.

3 Sentient Object Model

3.1 Information Flow and Interaction Model

We consider a component-based system model that incorporates previous work developed in the context of the IST CORTEX project [20]. A fundamental idea underlying the approach is that applications can be composed of a large number of smart components, which can sense and interact with their surrounding environment. They are referred to as *sentient objects*, a metaphor elaborated in CORTEX and inspired on the generic concept of *sentient computing* introduced in [21]. Sentient objects accept input events from a variety of different sources (including sensors, but not constrained to that), process them, and produce output events, whereby they actuate on the environment and/or interact with other objects. The following kinds of interactions can take place in the system: (i) environment-to-object interactions, reporting about the state of the former, and/or notifying about events taking place therein; (ii) object-to-object interactions, complementing the assessment of each individual object about the surrounding space, or serving collaboration with other objects; (iii) object-to-environment interactions, with the purpose of forcing a change in the state of the latter.

The environment can be a producer or consumer of information while interacting with sentient objects, which it does through transducers: sensors and actuators. The latter perform the necessary transformations between the physical real-time entities and their computerized representations in the system [22]. In our architecture, there are two kinds of transducers: *dumb* sensors and actuators, which interact with the objects by disseminating or capturing raw transducer information; and *smart* sensors and actuators, with enhanced processing capabilities, capable of “speaking” the “dialect” of our event model. Transducers may, or may not be part of a sentient object’s body, as discussed in Section 3.4.

A distinguishing aspect of our work from many of the existing approaches, is that we consider that sentient objects may indirectly communicate with each other through the environment, through links established between actuations and sensing operations. Thus the environment constitutes an additional interaction and communication channel and is in the control and awareness loop of the objects. It has been shown that in systems ignoring these hidden channels (e.g., feedback loops) inconsistencies may arise that can lead to unexpected failures [22]. In order to deal with the global information flow through computer system and environment in a seamless way, handling “software” and “hardware” events uniformly, it is necessary to find adequate abstractions. As discussed in Section 4, the Generic-Events Architecture introduces the abstractions of *Generic Event* and *Event Layer* to deal with these issues.

3.2 Component-Based Object Model

The approach proposed in this paper is based on a component-based object model that incorporates some of the ideas developed in the context of the CORTEX project. Applications are composed of a (possibly large) number of smart components that are able to sense their surrounding environment and interact with it, which are referred to as *sentient objects* [23] (see Figure 1).

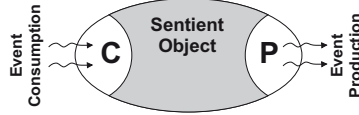


Fig. 1. The sentient object metaphor

Sentient objects can take several different forms: they can simply be software-based components, but they can also comprise mechanical and/or hardware parts, amongst which the very sensorial apparatus that substantiates “sentience”, mixed with software components to accomplish their task. We refine this notion by considering a sentient object as an encapsulating entity, a component with internal logic and active processing elements, able to receive, transform and produce new events. This interface hides the internal hardware/software structure of the object, which may be complex, and shields the system from the low-level functional and temporal details of controlling a specific sensor or actuator.

3.3 Sentient Object Composition

Given the inherent complexity of the envisaged applications, the number of simultaneous input events and the internal size of sentient objects may become too large and difficult to handle. Therefore, it should be possible to consider the hierarchical composition of sentient objects so that the application logic can be separated across as few or as many of these objects as necessary. On the other hand, composition of sentient objects should normally be constrained by the actual hardware component’s structure, preventing the possibility of arbitrarily composing sentient objects.

This is illustrated in Figure 2, where a sentient object is internally composed of a few other sentient objects, each of them consuming and producing events, some of which only internally propagated.

To give a more concrete example of such *component-aware* object composition we consider a fully-fledged sentient object, for example a car. The car is composed of several sentient objects, like: WLAN receiver and transmitter processor; velocity sensor processor; cruise speed control processor and actuator; doppler radar control; GPS CCD camera input treatment modules; control elements such as cruise speed, platoon, ambient, visual display, etc. The car (together with all of its embedded software) is in turn a sentient object, and the environment internal to its own structure becomes this larger object’s *body*.

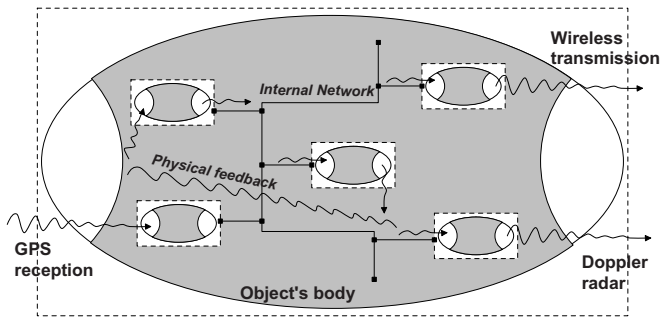


Fig. 2. Component-aware sentient object composition

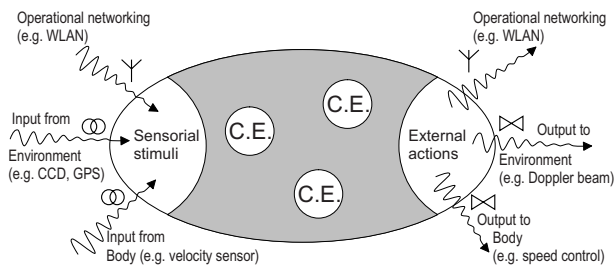


Fig. 3. Information flow through a sentient object (C.E- control element)

Figure 3 shows this perspective. The car as an object receives events from various different sources, namely operational networks (e.g., WLAN receiver), remote sources (e.g., GPS receiver) or local sources (e.g. velocity sensor). Likewise, it produces events to be consumed by different sinks, for instance events transmitted through networks (e.g., WLAN transmitter) to the environment or other objects, events to remote sinks (e.g., doppler radar actuator) or events to local sinks (e.g., speed control actuator). At this level of abstraction, it should be possible to define cooperation activities among several cars as sentient objects (say, platooning) without the need to know the internal structure of cars, or the events produced by body objects or by smart sensors within the body. Note that interactions within the local scope are referred to as interactions with the *body* of the object. This concept will be developed in the next section.

3.4 Encapsulation and Scoping

Now an important question is about how to represent and disseminate events in a large-scale networked world. As we have seen above, any event generated by a sentient object could, in principle, be visible anywhere in the system and thus received by any other sentient object. However, there are substantial obstacles to such universal interactions, originating from the components heterogeneity in such a large-scale setting.

Firstly, the components may have severe performance constraints, particularly because we want to integrate mobile units, smart sensors and actuators in such an architecture. Secondly, the bandwidth of the participating networks may vary largely. Such networks may be low power, low bandwidth fieldbuses, or more powerful wireless networks as well as high speed backbones. Thirdly, the networks may have widely different reliability and timeliness characteristics. Consider a platoon of cooperating vehicles. Inside a vehicle there may be a field-bus like CAN [24,17], TTP/A [18], LIN [25] or FlexRay [26], with a comparatively low bandwidth. On the other hand, the vehicles are communicating with others in the platoon via a direct wireless link. Finally, there may be multiple platoons of vehicles which are coordinated by an additional wireless network layer.

At the abstraction level of sentient objects, such heterogeneity is reflected by the notion of *body-vs-environment*. At the network level, we assume the *WAN-of-CANs* structure [4] to model the different networks. The notion of body and environment is derived from the recursively defined component-based object model. A body is similar to a cell membrane and represents a quality-of-service container for the sentient objects inside. On the network level, it may be associated with the components coupled by a certain CAN. A CAN defines the dissemination quality which can be expected by the cooperating objects.

In the above examples, a vehicle (robot or car) may be a sentient object, whose body is composed of the respective lower level objects (sensors and actuators) which are connected by the internal network (see Figure 2). Correspondingly, the platoon can be seen itself as an object composed of a collection of cooperating vehicles, its body being the environment encapsulated by the platoon zone. At the network level, the wireless network represents the respective CAN. However, several platoons united by their CANs may interact with each other and objects further away, through some wider-range, possible fixed networking substrate, hence the concept of *WAN-of-CANs*.

The notions of body-environment and *WAN-of-CANs* are very useful when defining interaction properties across such boundaries. Their introduction obeyed to our belief that a single mechanism to provide quality measures for interactions is not appropriate. Instead, a high level construct for interaction across boundaries is needed which allows to specify the quality of dissemination and exploits the knowledge about body and environment to assess the feasibility of quality constraints. As we will see in Section 4, the notion of an *event channel* represents this construct in our architecture.

4 Generic-Events Architecture

Although literature has classically studied the networking and sensing/actuating problems in isolation, we propose the innovative concept of *generic event*, be it derived from the boolean indication of a door opening sensor, from the electrical signal embodying a network packet (at the WLAN aerial) or from the arrival of a temperature event message.

Likewise, the programs running in sentient objects have very often consistency requirements that derive, even if remotely, from what are called *real-time entities*, in fact representations of state variables of the surrounding environment. Some of these, referred to as *time-value entities*, have consistency conditions based on the timeliness of the operations controlled by the computer, vis-a-vis their evolution in the environment (e.g., for the cooling system to consistently use the temperature of the engine it must obey some timeliness constraints) [22].

To fulfil this vision, we require an event model that satisfies these two sets of requirements. On the one hand, a model that treats the information flow through the whole computer system and environment in a seamless way, handling “software” and “hardware” events in a generic way. On the other hand, one that allows defining global, end-to-end, non-functional criteria in the time domain, such as temporal consistency, or QoS guarantees. We address these issues in this section and following ones.

We propose the **Generic-Events Architecture (GEAR)**, depicted in Figure 4, which we briefly describe in what follows (for a more detailed description please refer to [27]). The unusual L-shaped structure is crucial to ensure some of the properties described.

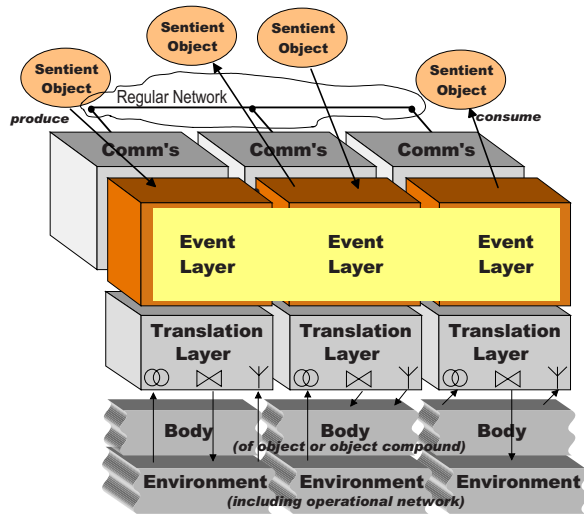


Fig. 4. Generic-Events architecture

Environment: The physical surroundings, remote and close, solid and etherial, of sentient objects.

Body: The physical embodiment of a sentient object (e.g., the hardware where a mechatronic controller resides, the physical structure of a car). Note that due to the compositional approach taken in our model, part of what is “environment” to a smaller object seen individually, becomes “body” for a larger,

containing object. In fact, the body is the “internal environment” of the object. This architecture layering allows composition to take place seamlessly, in what concerns information flow.

Translation Layer: The layer responsible for physical event transformation from/to their native form to event channel dialect (in Section 5 we further elaborate on the definition of the events flowing through event channels), between environment/body and an event channel. This layer performs observation and actuation operations on the lower side (as represented by the sensor and actuator symbols in the figure), and transactions of event descriptions on the other. On the lower side this layer may also interact with dumb sensors or actuators, therefore “talking” the language of the specific device. These interactions are done through *operational networks* (hence the antenna symbol in the figure).

Event Layer: The layer responsible for event propagation in the whole system, through several *Event Channels (EC)*:. In concrete terms, this layer is a kind of middleware that provides important event-processing services which are crucial for any realistic event-based system. For example, some of the services that imply the processing of events may include publishing, subscribing, discrimination (zoning, filtering, fusion, tracing), and queuing.

Communication Layer: The layer responsible for “wrapping” events (as a matter of fact, event descriptions in EC dialect) into “carrier” *event-messages*, to be transported to remote places. For example, a sensing event generated by a smart sensor is wrapped in an event-message and disseminated, to be caught by whoever is concerned. The same holds for an actuation event produced by a sentient object, to be delivered to a remote smart actuator. Likewise, this may apply to an event-message from one sentient object to another. Dumb sensors and actuators do not send event-messages, since they are unable to understand the EC dialect (they do not have an event layer neither a communication layer— they communicate, if needed, through operational networks).

Regular Network: This is represented in the horizontal axis of the block diagram by the communication layer, which encompasses the usual LAN, TCP/IP, and real-time protocols, desirably augmented with reliable and/or ordered broadcast and other protocols.

GEAR introduces some innovative ideas in distributed systems architecture. While serving an object model based on production and consumption of generic events, it treats events produced by several sources (environment, body, objects) in a homogeneous way. This is possible due to the use of a common basic dialect for talking about events and due to the existence of the translation layer, which performs the necessary translation between the physical representation of a real-time entity and the EC compliant format. Crucial to the architecture is the event layer, which uses event channels to propagate events through regular network infrastructures. The event layer is realized by the COSMIC middleware, as described in Section 5.

The GEAR architecture serves an object model based on production and consumption of generic events. Events are presented to objects through Event

Channels (EC), which are in charge of propagating them to the relevant objects (those having subscribed to that event class). However, not only objects produce or consume events, but this is transparent, since it is dealt with by ensuring all these entities (objects and other) speak the EC dialect.

Events are produced by several sources which are treated in a homogeneous way. Event sources include:

- the environment where physical events take place, such as the detection of the opening of a gate, or of the change of a semaphore light, the sampling of a temperature at a given time.
- the body of the object (or object compound), taken as that part of the environment which is aggregated to the object or object compound and would not make sense otherwise (e.g., the body of a robot, the hardware of a car, the embodiment of a mechatronic device), and where similar kinds of physical events take place, for example, the sampling of a car's velocity. Note that part of what is 'environment' for an isolated object, may become 'body' of a compound object of which that object is part.
- the objects themselves may generate an event, when they invoke *produce*, which manifests itself as: a piece of information or a command they wish to make available to other objects; a notification they produce "to whom it may concern"; an actuation command on the body or on the environment, for example, controlling the speed of a car, or telling a gate to close.

The flow of information (external environment and computational part) is seamlessly supported by the L-shaped architecture. It occurs in a number of different ways, as illustrated in Figure 5, which demonstrates the expressiveness of the model with regard to the necessary forms of information encountered in real-time cooperative and embedded systems.

Smart sensors produce events which report on the environment (they deserve the 'smart' adjective because they speak the EC dialect). Body sensors produce events which report on the body. They are disseminated by the local event layer module, on an event channel (EC) propagated through the regular network, to any relevant remote event layer modules where entities showed an interest on them, normally, sentient objects attached to the respective local event layer modules.

Sentient objects consume events they are interested in, process them, and produce other events. Some of these events are destined to other sentient objects. They are published on an EC using the same EC dialect that serves, e.g., sensor originated events. However, these events are semantically of a kind such that they are to be subscribed by the relevant sentient objects, for example, the sentient objects composing a robot controller system, or, at a higher level, the sentient objects composing the actual robots in a cooperative application. Smart actuators, on the other hand, merely consume events produced by sentient objects, whereby they accept and execute actuation commands. Alternatively to "talking" to other sentient objects, sentient objects can produce events of a lower level, for example, actuation commands on the body or environment. They

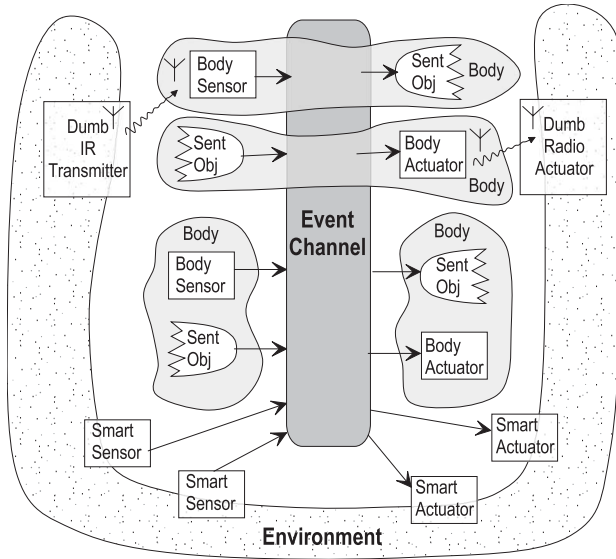


Fig. 5. Information flow in the whole system

publish these exactly the same way: on an event channel through the local event layer representative. Now, if these commands are of concern to local actuator units (e.g., body, including internal operational networks), they are passed on to the local translation layer. If they are of concern to a remote smart actuator, they are disseminated through the distributed event layer, to reach the former. In any case, if they are also of interest to other entities, such as other sentient objects that wish to be informed of the actuation command, then they are also disseminated through the EC to these sentient objects.

A key advantage of this architecture is that event-messages and physical events can be globally ordered, if necessary, since they all pass through the event layer. The model also offers opportunities to solve a long-lasting problem in real-time computer control and embedded systems: the inconsistency between computer message passing and physical feedback loop information flows. We address this issue in Section 6.3.

5 Event Model and Middleware

An event model and a middleware suitable for smart components must support timely and reliable communication and also must be resource efficient. The **COSMIC (COoperating Smart devices)** middleware, which is described ahead in this section, is aimed at supporting the interaction between those components according to the concepts introduced so far. However, we first introduce the event model.

5.1 Event Model

Based on the WAN-of-CANs model, we assume that components, smart sensors/actuators or sentient objects, are connected to some form of CAN as a fieldbus or a wireless sensor network which provides specific network properties. E.g. a fieldbus developed for control applications usually includes mechanisms for predictable communication while other networks only support a best effort dissemination. A gateway connects these CANs to the next level in the network hierarchy. The event system should allow the dynamic interaction over a hierarchy of such networks and comply with the overall generic event model.

We introduced the notion of an event channel to cope with differing properties and requirements, and to have an object to which we can assign resources and reservations, as needed when striving for predictability. Although the concept of an event channel is not new [2,8], it has not yet been used to reflect the properties of the underlying heterogeneous communication networks and mechanisms as described by the GEAR architecture. Rather, existing event middleware allows to specify the priorities or deadlines of events handled in an event server. Event channels allow to specify the communication properties on the level of the event system in a fine grained way. An event channel is defined by:

$$event_channel := \langle subject, quality_attributeList, handlers \rangle$$

The subject determines the types of events which may be issued to the channel. Every event subject is therefore associated to its own event channel. The quality attributes model the properties of the underlying communication network and dissemination scheme. These attributes include latency specifications, dissemination constraints and reliability parameters. The notion of zones supports this approach. Our goal is to handle the temporal specifications as $\langle bound, coverage \rangle$ pairs [28], which is orthogonal to the more technical questions of how to achieve a certain synchrony property of the dissemination infrastructure. Exception handlers can be provided to deal with situations such as the violation of specified timeliness bounds for the channel.

Events are typed information carriers and are disseminated in a publisher/subscriber style [9,29], which is particularly suitable because it supports generative, anonymous communication [14] and does not create any artificial control dependencies between producers of information and the consumers. This decoupling in space (no references or names of senders or receivers are needed for communication) and the flow decoupling (no control transfer occurs with a data transfer) are well known [9,29,7] and crucial properties to maintain autonomy of components and dynamic interactions.

Events are exchanged between sentient objects through event channels. To cope with the requirements of an ad-hoc environment, an event includes the description of the context in which it has been generated and quality attributes defining requirements for dissemination. This is particularly important in an open, dynamic environment where an event may travel over multiple networks. An event instance is specified as:

$event := \langle subject, context_attributeList, quality_attributeList, contents \rangle$

A subject defines the type of the event and is related to the event contents. It supports anonymous communication and is used to route an event. The subject has to match to the subject of the event channel through which the event is disseminated. Attributes are complementary to the event contents. They describe individual functional and non-functional properties of the event. The context attributes describe the environment in which the event has been generated, e.g. a location, an operational mode or a time of occurrence. The quality attributes specify timeliness and dependability aspects in terms of $\langle validity_interval, omission_degree \rangle$ pairs. These timeliness, and other temporal aspects of the interactions will be further addressed in Section 6.

5.2 COoperating Smart Devices Middleware

The COSMIC (COoperating Smart devices) middleware, maps the channel properties to lower level protocols of the regular network. Based on our previous work on predictable protocols for the CAN-Bus, COSMIC defines an abstract network which provides hard, soft and non real-time message classes [30].

Correspondingly, this allows us to distinguish three event channel classes with different synchrony properties: hard real-time channels, soft real-time channels and non-real-time channels.

Hard real-time channels (HRTC) guarantee event propagation within the defined time constraints in the presence of a specified number of omission faults. HRTECs are supported by a reservation scheme which is similar to the scheme used in time-triggered protocols like TTP [31], TTP/A [18], and TTCAN [24]. However, a substantial advantage over a TDMA scheme is that due to CAN-Bus properties, bandwidth which was reserved but is not needed by a HRTEC can be used by less critical traffic [30].

Soft real-time channels (SRTC) exploit the temporal validity interval of events to derive deadlines for scheduling. The validity interval defines the point in time after which an event becomes temporally inconsistent. Therefore, in a real-time system an event is useless after this point and may be discarded. The transmission deadline (DL) is defined as the latest point in time when a message has to be transmitted and is specified in a time interval which is derived from the expiration time: $t_{event_ready} < DL < t_{expiration} - \Delta_{notification}$

$t_{expiration}$ defines the point in time when the temporal validity expires. $\Delta_{notification}$ is the expected end-to-end latency which includes the transfer time over the network and the time the event may be delayed by the local event handling in the nodes. As said before, event deadlines are used to schedule the dissemination by SRTECs. However, deadlines may be missed in transient overload situations or due to arbitrary arrival times of events. On the publisher side the application's exception handler is called whenever the event deadline expires before event transmission. At this point in time the event is also not expected to arrive at the subscriber side before the validity expires. Therefore, the event is removed from the sending queue. On the subscriber side the expiration time

is used to schedule the delivery of the event. If the event cannot be delivered until its expiration time it is removed from the respective queues to prevent the communication system to be loaded by outdated messages.

Non-real-time channels do not assume any temporal specification and disseminate events in a best effort manner. An instance of an event channel is created locally, whenever a publisher makes an announcement for publication or a subscriber subscribes for an event notification. When a publisher announces publication, the respective data structures of an event channel are created by the middleware. When a subscriber subscribes to an event channel, it may specify context attributes of an event which are used to filter events locally. E.g. a subscriber may only be interested in events generated at a certain location. Additionally the subscriber specifies quality properties of the event channel. A more detailed description of the event channels can be found in [32].

On the architectural level, COSMIC distinguishes three layers roughly depicted in Figure 6. Two of them, the event layer and the abstract network layer are implemented by the COSMIC middleware. The event layer provides the API for the application and realizes the abstraction of event and event channels.

The abstract network implements real-time message classes and adapts the quality requirements to the underlying real network. An event channel handler resides in every node. It supports the programming interface and provides the necessary data structures for event-based communication. Whenever an object subscribes to a channel or a publisher announces a channel, the event channel handler is involved. It initiates the binding of the channel's subject, which is represented by a network independent unique identifier to an address of the underlying abstract network to enable communication [7]. The event channel handler then tightly cooperates with the respective handlers of the abstract network layer to disseminate events or receive event notifications. It should be noted that the QoS properties of the event layer in general depend on what the abstract network layer can provide. Thus, it may not always be possible to e.g. support hard real-time event channels because the abstract network layer cannot provide the respective guarantees. In [32], we describe the protocols and services of the abstract network layer particularly for the CAN-Bus.

As can be seen in Figure 6, the hard real-time (HRT) message class is supported by a dedicated handler which is able to provide the time triggered message dissemination. The HRT handler maintains the HRT message list, which contains an entry for each local HRT message to be sent. The entry holds the parameters for the message, the activation status and the binding information. Messages are scheduled on the bus according to the HRT message calendar which comprises the precise start time for each time slot allocated for a message. Soft real-time message queues order outgoing messages according to their transmission deadlines derived from the temporal validity interval. If the transmission deadline is exceeded, the event message is purged out of the queue. The respective application is notified via the exception notification interface and can take actions like trying to publish the event again or publish it to a channel of another class. Incoming event messages are ordered according to their temporal validity. If an

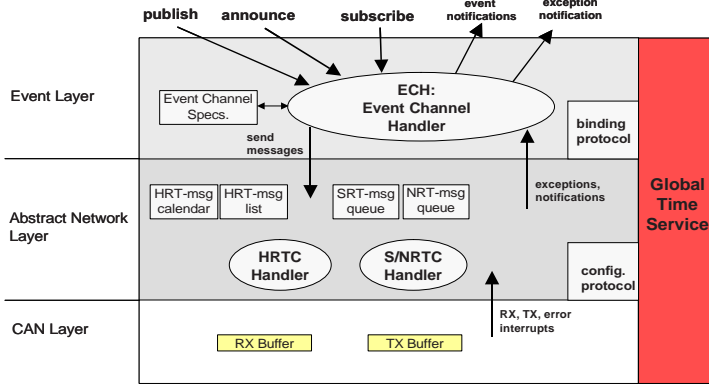


Fig. 6. Internal structure of COSMIC

event message arrive, the respective applications are notified. At the moment, an outdated message is deleted from the queue and if the queue runs out of space, the oldest message is discarded. However, there are other policies possible depending on event attributes and available memory space. Non real-time messages are FIFO ordered in a fixed size circular buffer.

6 Temporal Aspects of Interactions

Any interaction needs some form of predictability. If safety critical scenarios are considered, temporal aspects become crucial and have to be made explicit. The problem is how to define temporal constraints and how to enforce them by appropriate resource usage in a dynamic ad-hoc environment. In a system where interactions are spontaneous, it may be also necessary to determine temporal properties dynamically. To do this, the respective temporal information must be stated explicitly and available during run-time. Secondly, it is not always ensured that temporal properties can be fulfilled. In these cases, adaptations and timing failure notification must be provided [33,28].

In most real-time systems, the notion of a deadline is the prevailing scheme to express and enforce timeliness. However, firstly, a deadline only weakly reflects the temporal characteristics of the information which is handled. In a dynamic cooperative control system suffering from weak links it is important to consider issues like ageing of information or the urgency and importance of control actions, which is not possible within the notion of a simple deadline.

Secondly, a deadline often includes implicit knowledge about the system and the relations between activities. In a rather well defined, closed environment, it is possible to make such implicit assumptions and map these to execution times and deadlines. E.g. the engineer knows how long a vehicle position can be used before the vehicle movement outdates this information. Thus he maps this dependency between speed and position on a deadline which then assures that the position error can be assumed to be bounded. In an open environment, this

implicit mapping is not possible any more because, as an obvious reason, the relation between speed and position, and thus the error bound, cannot necessarily be reverse engineered from a deadline.

Thirdly, as is emphasized in the work of Mock [34], the notion of a deadline assumes a passive environment and an active control system. In this classical view, the physical properties of the environment define the real-time constraints which are imposed as deadlines on a control system. This simple distinction in a passive environment and an active control system does not reflect the situation in a cooperative scenario where multiple active and mobile entities interact. The cooperating entities form part of the environment perceived by an individual component and they are active. Consider a team of robots which cooperatively move or lift an object too heavy for a single robot. In such a coordination task where mutual dependencies exist, the notion of a deadline is hardly useful when timing the interaction.

Therefore, we need a more elaborated model of representing and exploiting temporal information in such an environment. To derive such a model, we briefly review the notion of events in the physical environment and their representation in a machine.

6.1 Events

In the physical world, an **event** is defined as a singular occurrence in space and time. It can be defined as a cut of the time line and point in space and thus has no temporal or spatial extension. An event may lead to a state change in the physical world which may be called an effect. The effect is the subject of an observation [35,36,22]. The observation has a temporal and spatial distance to the related event. In a dynamic system, it is beneficial to store and transfer information about the temporal and spatial occurrence of an event together with the event itself because implicit assumptions about where and when the event occurred may be difficult. Information related to these issues is called the event context. In GEAR, a **generic event** is a happening that takes place in the event layer at a given instant of the time line, $\langle E, T^{GE} \rangle$. The happening is internal to the system, has an event channel compliant representation, and may be related with physical events taking place in the environment. That is, the 'event' is the happening as seen by the event layer, at a given instant in the time line.

It is useful to consider the event model in the terms defined in [37,35,36,22], i.e. a real-time entity (RTE), which is the element of the physical world, a real-time image, which is a representation of the environment and can be a single image of a RTE or a complex representation derived from multiple sensor readings, and a real-time object, which refers to the data structure in which a real-time image or a real-time entity are stored and communicated. A real-time image derived from one or more observations may be encapsulated in a $\langle context, value \rangle$ pair, where context describes relevant aspects of the environment/system state in which the value has been derived. The classical $\langle time, value \rangle$ model of an event therefore refers to the temporal aspect of the context only. It is useful in

a dynamic system as envisaged in this paper to include more information, e.g. location information or a network zone.

6.2 Temporal Properties of Events

In order to deal with real-time sentient objects we need to understand the implications of timeliness requirements in the context of the proposed generic-events architecture. This will be done by establishing fundamental correctness criteria for the operation of the system. The system architecture, including the protocols and mechanisms materialising the event layer middleware, must be built so that the strict observation of the established criteria is ensured. On the other hand, given the distributed nature of the problem, the correctness of operation does not depend solely on the observation of timeliness constraints, but also on the consistency and coordination among the distributed actors in the system. In this respect, note that the information flow is defined in terms of events, and it is controlled at the event layer, where everything passes. As such, and very importantly, all consistency criteria that must be secured apply as well to regular messages, messages through operational network channels, and input/output feedback paths through the environment. No hidden channel problems need affect the operation of the system [22].

We illustrate the nature and the temporal properties of generic events with a few examples.

Examples of generic events

1	door opened;
2	door opened as observed at T ;
3	door is open;
4	door is open as observed at T ;
5	temperature is X;
6	temperature is X as observed at T ;
7	position of crankshaft is Y;
8	position of crankshaft is Y as observed at T ;
9	crankshaft reached ignition point I;
10	crankshaft reached ignition point I as observed at T ;
11	value of variable Z is 'entering-zone mode';
12	set variable 'cruise speed' to S;
13	set variable 'cruise speed' to S within T ;

The difference between (1) and (2) is that in (1) we know at T^{GE} that the door has opened in some (near) past instant, whereas in (2) we know at T^{GE} that it opened at T . The difference of the former to (3) and (4) is that here we know the state of the door, without necessarily knowing when it opened. In fact, note that generic events report state (3) as well as change of state (1), what in other more classical models used to denote “state” and “events”, with regard to the physical environment. This said, in GEAR nothing prevents the periphery of the system (e.g., smart sensors) from being organized in the best suited way (e.g., state sampling, event latching, etc.), for each generic-event flow to be produced.

Given that T^{GE} establishes the event production instant, there is apparently redundant timing information in some lines, e.g. (2) or (4). However, this is an important characteristic of GEAR: T^{GE} denotes the time at which E was produced on the event channel and serves any generic type of event; T is part of E, thus invisible to the EC, but it denotes the time at which a given real-time entity was observed to have a given state or to have changed its state. The separation of concerns enforced by T and T^{GE} is very important, as we detail ahead.

When we say in (11) that we know at T^{GE} that “ $Z = \text{'entering-zone mode'}$ ”, this marks the point at which this internal state change is relevant for the system, e.g., as alerted by the platoon leader sentient object, in a cooperating cars scenario. Likewise, in (12), when (e.g., the leader again) publishes the command to change the state of the ‘cruise speed’ variable to S, the reference point is T^{GE} .

However, when we say in (3) that we know at T^{GE} that “the door is open”, or in (5) that “the temperature is X”, we might as well try to know how trustworthy this information is, since the temperature and door are time-value entities. For example, by the time T^{GE} when we learn that “the temperature is X”, it might already be way higher than X! Even if, as we say in (4), we know at T^{GE} that “the door is open at T ”, or as in (6), that “the temperature is X at T ”, this still may not solve our problem. There are important implications of the way we handle time-value entities that we discuss below, using definitions in [22].

Firstly, saying, as in (6), that we know at T^{GE} that “the temperature is X at T ”, might seem to provide a precise indication. However, what T portrays is the time at which the periphery of the system observed the temperature. When observing the value of a continuous variable, it is relevant to define the error. For an observation $\langle r(E_i)(t_i), T_i \rangle$ of the value of an RTe E_i at t_i receiving timestamp T_i , the **observation error** in the *value domain* is given by $\nu_i = |E_i(T_i) - r(E_i)(t_i)|$: we expect the value of E_i at T_i , but we get an approximation of the value ($r(E_i)$), measured approximately (t_i) at T_i . That is, the error has two components, the inaccuracy of the sensing apparatus, and the observation positioning error. On the other hand saying, as in (4), that we know at T^{GE} that “the door is open at T ”, has analogous constraints. Here, when observing the time at which a given discrete value E_i occurs (e.g., opening of the door), the observation error (jitter) is defined in the *time domain*, $\zeta_i = |T_i - t_i|$: E_i assumed a given value at t_i , but the system logs it as having happened at T_i .

In order for our measurements to be useful, we establish bounds on the observation errors, for classes of events produced in certain conditions (e.g., that maxima of the sensor errors and of the clocks precision are known for that class). The fact that observations of a class of events meet an error bound allows us to abstract from measurement details and henceforth trust observations of that class. We say they are *consistent*:

- Given a known \mathcal{V}_o , we say that an observation is **consistent** in the **value domain**, if and only if $\nu_i \leq \mathcal{V}_o$
- Given a known \mathcal{T}_o , we say that an observation is **consistent** in the **time domain**, if and only if $\zeta_i \leq \mathcal{T}_o$

This deals with the consistency at the observation instant. Secondly, we must deal with the consistency at the use instant. We must ensure that the information is sufficiently fresh when it is about to be used. For example, when we say in (5) that we know at T^{GE} that “the temperature is X”, it is important that the interval between the time when it was measured and T^{GE} , is known and short enough to be useful, so that the temperature hasn’t drifted too much in the meantime. I.e. for the information provided by this generic event to be meaningful for whatever the system intends to do with it. This must be ensured by the infrastructure, and a practical way is to define a fixed parameter, known at design time, based on estimates of the variable’s dynamics. This interval has been called *absolute validity interval* for databases[38], or *temporal accuracy interval* for control [36].

In fact, in GEAR we generalise this concept and we use the following two important notions to relate the value domain and the temporal domain of information: temporal consistency and temporal validity.

Temporal consistency relates a bound ν in the value domain to a point in time t . In general, since we consider real-time entities, the value v of the RTe is dependent on the time and can be described as a function over time (value over time [22]): $f_c(t) = v$. For a point in time $t_a > t$, the equation becomes $f_c(t_a) = v + \delta_a$ where δ_a is the change of the value in the interval $[t, t_a]$. Temporal consistency defines a bound ν on the value v at time t_a for which v is an acceptable representation of the RTe. The time-value entity is temporally consistent at t_a if and only if $0 \leq |\delta_a| \leq \nu$.

Temporal validity considers f_c and defines a time interval during which all values v of the time-value entity can be considered to be temporally consistent. Thus, given the temporal validity interval $[t_a, t_b]$ and a function $f_c(t)$, for any t_c , $t_a \leq t_c \leq t_b$: $f_c(t_c) = v + \delta_c$ ($0 \leq |\delta_c| \leq \nu$).

Temporal consistency specifies how close an observed RTe captured in a time-value entity represents the value in the real world. Temporal validity intervals are derived from the knowledge of both the temporal consistency requirements and the function f_c . Both, temporal consistency and temporal validity can be used to reason about properties of an event independently of any implementation issues, thus they should be part of a specification. Figure 7a) shows the simple case when $f(t)$ is a linear function. Temporal consistency of a value v is defined by a unique interval which matches the temporal validity interval for t_0 as observation point.

Figure 7b) highlights the fact that temporal consistency is independent from a temporal validity interval. There is an implication in only one direction: A value in the temporal validity interval is always temporally consistent. However, the reverse is not true: A temporally consistent value may exist outside of a temporal validity interval. Note that if $f(t)$ is known, it may be possible to make predictions about the error and choose the right points in time for values bounded by ν just by a clock.

Figure 7c) shows a difficult case of a discrete function. It shows a discrete binary function which represents e.g. the state of a switch, a door, a traffic light, etc. It is very difficult to define temporal validity intervals because the function

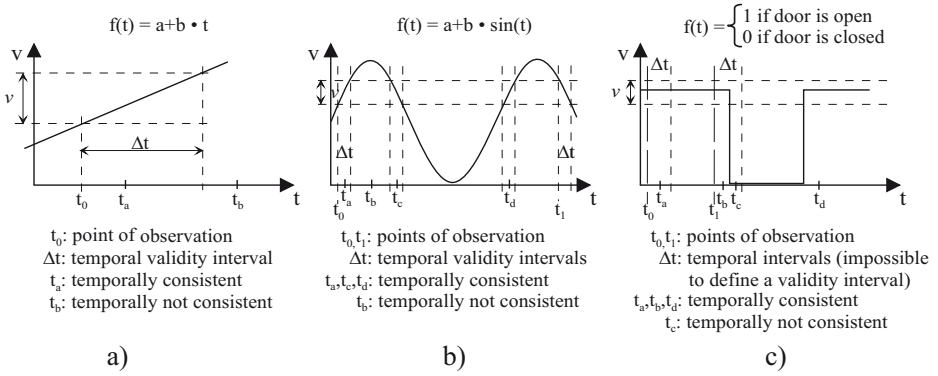


Fig. 7. RTe behaviours: a) Linear monotonic function; b) Continuous cyclic function; c) Discrete function

is not steady. How small the interval Δt ever is, it may include a rising or falling edge of the function and therefore a temporal inconsistency. Although temporal consistency can be easily specified, no temporal validity interval can be defined.

Finally, coming back to our examples of generic events, the difference between (1,2) and (5,6) concerns the nature of the variable: discrete in the former, and continuous in the latter. Lines (7-10) illustrate how this distinction, so much used in computer control, may turn out to be pretty much artificial. The position of an engine's crankshaft is a continuous variable: an angle that goes from 0 to 360 degrees (0 again) and so forth. So, there is apparently no difference between (5,6) and (7,8). However, the crankshaft evolves so quickly that addressing it as a continuous variable may imply a very high error. Hence, if we "fabricate" a discrete variable which is the arrival of the crankshaft to the ignition point I, as in (9), then this is equivalent to the kind of event in (1). This ambiguity was addressed in [22] as the duality between *value over time* and *time of a value*.

In conclusion, we have shown the fundamental consistency guarantees to be ensured by this kind of architectures: *value* and *time* domain observation *consistency*; and *temporal consistency*.

6.3 Event Ordering and the Hidden Channel Problem

As mentioned before, one important problem in real-time computer control and embedded systems is the inconsistency between computer message passing and physical feedback loop information flows. This stems from the difficulty in ensuring a proper ordering of event-messages (exchanged among computers in the system) and physical events (which may propagate through the environment, establishing precedence relations not perceived by the system). This is also referred as the problem of *hidden channels*, which we briefly introduce and discuss in this section. Let us consider a classical example that was given by Kopetz and Verissimo [35] long time ago (see Figure 8).

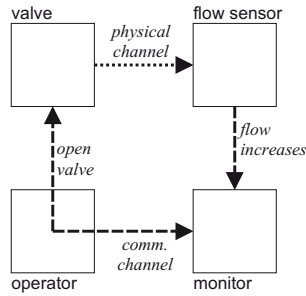


Fig. 8. Example of interaction through the environment

The operator sends a message to open the valve over the network. At the same time she informs the monitor to be prepared that the flow in the pipe will change. This message is delayed and thus, when the flow sensor reports an increased throughput, the monitor infers a broken valve and erroneously shuts down the system. The communication via the physical (hidden) channel between the valve and flow sensor is faster than the worst case delays in the network.

Now, what is the interpretation of this example in terms of the event architecture? The event layer supports multiple channels and channel classes to disseminate event messages. Event messages have a temporal validity, from which transmission and delivery deadlines are derived. A possible application design is depicted in Figure 9a). The command from the operator is not a safety critical event and thus, it is pushed to a soft real-time channel. The valve and the monitor have subscribed to this channel. The flow sensor may publish throughput periodically to a hard real-time channel because the monitor has to react on flow changes with a guaranteed latency. The problem with the scenario is that because of the interaction through the physical channel, it is not possible to apply one of the usual ordering schemes based on some history mechanism because there is no gap detection property. The physical channel simply cannot carry any ordering information. The problem is how to detect the causal relationship between the event which caused the valve to open and the subsequent message of the flow sensor. Any attempt to solve this problem in an asynchronous system will fail. We need some mechanism based on time to detect the (potential) causal dependency between the event generated by the flow sensor and the previous command of the operator.

The dissemination of events through the event layer alone does not help much if we only consider the communication over the regular network. The question is how to include the physical channel? Figure 9b) sketches the situation.

We recognize that we have a physical (hidden) channel between the valve and the flow sensor. This channel has a known latency. Within the GEAR architecture, it can be interpreted to be propagated via a kind of operational network coupling the valve and the flow sensor. The physical property of a flow speed is transferred through the translation layer of GEAR by the smart flow sensor.

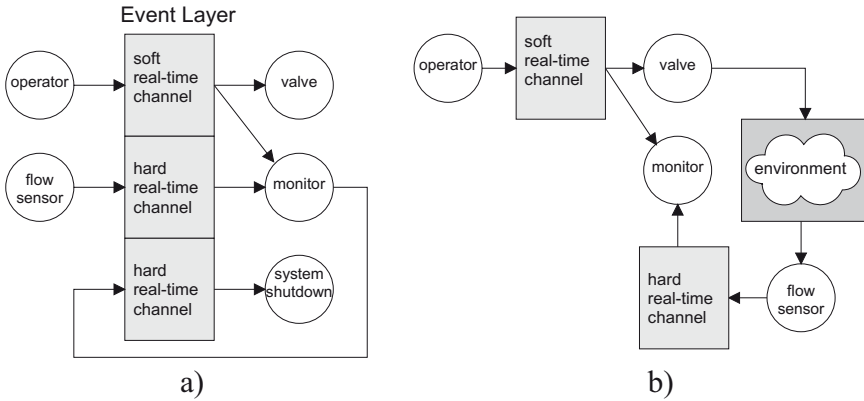


Fig. 9. The example modelled in an event system: a) without physical channel; b) including the physical channel

At the event layer it is provided as a periodic event propagated through a hard real-time channel. So far, no ordering property is available in the system. The monitor can not decide whether the event from the flow sensor is dependent on a soft real-time command event previously published by the operator. There is no gap detection property. We need a synchrony property to solve this problem.

We address the problem using the theory developed in [39]. Hidden channels are not directly controlled, instead their effect is incorporated in the design of the regular computer communication system. The latter is endowed with the adequate synchronism so as to guarantee that hidden channels do not cause ordering inversions. This is akin to designing a real-time causal delivery messaging system sensitive to both computer (send, receive) and real-world (sense, act) events. The communication system is tuned with the propagation time parameters of the environment.

Given the above, the problem can be addressed by resorting to a solution based on a model of partial synchrony, such as the TCB [28]. We make the assumption that we have a network where we can specify $\langle \text{bound}, \text{coverage} \rangle$ pairs and have best effort mechanisms to enforce deadline delivery for soft real-time events. The TCB provides awareness in the case an event misses the deadline or is delivered in a different order at different nodes. It doesn't make sense to defer a hard real-time event to provide a global order between hard real-time and soft real-time events. In other words, the event from the flow sensor is not deferred until the (soft real-time) command event has been delivered to the monitor. Thus, the HRT event from the flow sensor may be delivered before the soft real-time command event. However, this would be known to the monitor, which, with the help of the TCB, would be able to detect when an event has not been delivered within a certain time bound. The latency of the entire chain of event dissemination as depicted in Figure 10 would be used to set the TCB to an appropriate value (the minimum propagation time of the event chain, including the propagation time through the physical channel).

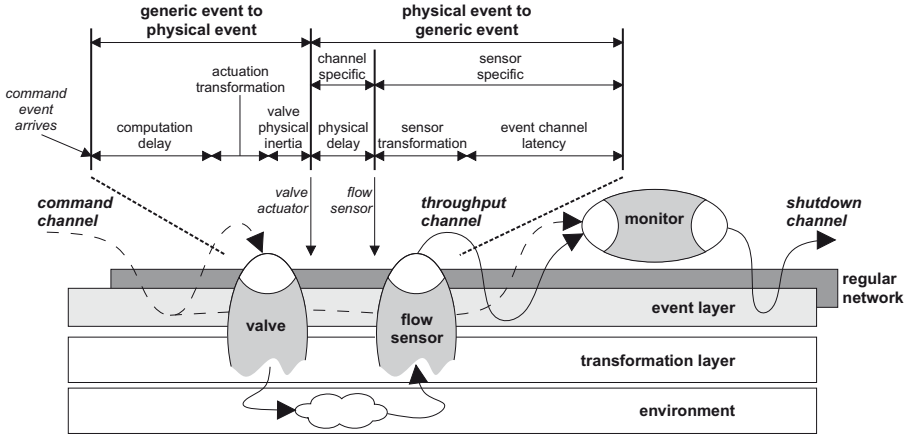


Fig. 10. Temporal analysis of event flow

Considering the flow sensor as a sentient object, it consumes the physical event from the environment generated by the actuation of the valve. In turn, it generates a generic event pushed to the respective channel. The latency can be calculated easily from the latency of the physical channel and the HRT channel of the event layer. At the abstraction level of sentient objects, we have three sentient objects and two event channels, one disseminating the commands and one disseminating the flow in the pipe (throughput) measured by the flow sensor. Figure 10 show this in different detail.

The main benefit including the physical channel would be the fact that now the bounds for the latencies as needed for setting the TCB is explicitly stated in the design phase.

7 Practical Examples

This section addressed two short examples of proof-of-concept prototypes that have been developed in the course of the CORTEX project, in which we used some of the concepts elaborated in this paper. A more detailed description of the cooperating cars example is provided in [40], while the coordinated robots example and the COSMIC middleware are further detailed in [41].

7.1 Cooperating Cars

In the first scenario we consider several cars (sentient objects) with the ability to 'consume' events from the environment and 'produce' events to it. These events are disseminated and received through Event Channels, using the GEAR architecture. We assume that the cars can freely move around and, for simplicity, we do not consider obstacles. Because of that, a fundamental safety rule consists in ensuring that no car crashes occur, which requires every car to know the

position of other cars. Therefore, each car periodically publishes its position as a GEAR event, which will be consumed by the cars that subscribed it and that fulfil the conditions to receive it (e.g. they are in the same 'zone').

One solution to satisfy the safety rule is to provide each car with a temporally consistent image of the external and internal (body) environment. Smart sensors publish events about the relevant real-time entities of the environment. Body sensors do the same with respect to the body. These events define the state of a 'zone' surrounding the car. Now, suppose all sentient objects feature a module, call it constructor, in charge of building a *real-time image* (RT-image) of the zone. The constructor subscribes to the environment and body events, and maybe other relevant events. The external environment events contribute to form a public RT-image of the zone. That image is enriched with inputs from other objects, and from the object's own body, part of which may not be made public. What is important is that *all* these events obey global consistency rules.

The safety rule is illustrated on the left of Figure 11. The grey car must keep outside the dashed circles around the black cars, and so he must always know the position of the black cars with a bounded error (ϵ). This error depends on how much time has passed since the last position of the car was published and on the maximum speed of that car. Therefore, both of them must be bounded.

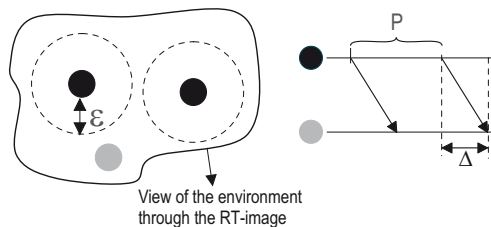


Fig. 11. Safety rules in the cooperating cars scenario

The problem would be easily solvable if we could assume a reliable and timely propagation of events through EC modules. The system would be configured so that at every P time units the grey car would receive information from the black car (see Figure 11 on the right). Every car would be able to construct a RT-image of the surrounding cars with a bounded accuracy (related with the propagation latency, Δ , and the period, P), and all the images would be consistent among them. However, since we have to assume asynchronous Event Channels (the abstract network layer is not able to provide real-time channels in the operational environment, typically wireless, considered here), it is not possible to ensure the temporal consistency of the RT-image. Events can take more than Δ to be propagated, incurring in timing failures.

In this example we assume that the architecture, despite uncertainty of communication, is endowed with perfect timing failure detection (TFD), such as supported by a timely computing base (TCB) [28]. This allows the construction

of event-channel handlers that, despite not being able to provide real-time properties, are able to provide indications when some specified timeliness bound is violated. This kind of EC modules can be exploited by fail-safe applications, such as the cars we consider here, which can stop as soon as a timing failure occurs. Considering our cooperating cars scenario, the required timeliness bounds would have to be specified as quality parameters of the EC modules. The EC module would then observe the timeliness of the events, with the help of the TFD service, and execute a fail-safety procedure to stop the car, in case of a timing failure.

7.2 Coordinated Robots

The coordinated robots example focuses on a demo of two cooperating robots, depicted in Figure 12. Each robot is equipped with smart distance sensors, speed sensors, acceleration sensors and one of the robots (the “guide” (KURT2) in front (Figure 12)) has a tracking camera allowing to follow a white line.

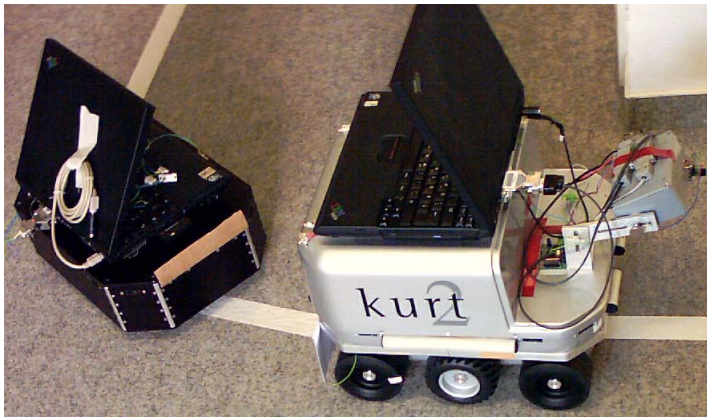


Fig. 12. Cooperating robots

The robots form a WAN-of-CANs system in which their local CANs are interconnected via a wireless 802.11 network. COSMIC provides the event layer for seamless interaction. The “blind” robot (N.N.) is searching the guide randomly. Whenever the blind robot detects (by its front distance sensors) an obstacle, it checks whether this may be the guide. For this purpose, it dynamically subscribes to the event channel disseminating distance events from rear distance sensors of the guide(s) and compares these with the distance events from its local front sensors. If the distance is approximately the same it infers that it is really behind a guide. Now N.N. also subscribes to the event channels of the tracking camera and the speed sensors to follow the guide. The demo application highlights the following properties of the system:

Dynamic interaction of robots which is not known in advance: In principle, any two a priori unknown robots can cooperate. All what publishers and subscribers have to know to dynamically interact in this environment is the subject of the respective event class.

Interaction through the environment: The cooperation between the robots is controlled by sensing the distance between the robots. If the guide detects that the distance grows, it slows down. Respectively, if the blind robot comes too close it reduces its speed. The local distance sensors produce events which are disseminated through a low latency, highly predictable event channel. The respective reaction time can be calculated as function of the speed and the distance of the robots and define a dynamic dissemination deadline for events. Thus, the interaction through the environment will secure the safety properties of the application, i.e. the follower may not crash into the guide and the guide may not loose the follower.

Cooperative sensing: The blind robot subscribes to the events of the line tracking camera. Thus it can “see” through the eye of the guide. Because it knows the distance to the guide and the speed as well, it can foresee the necessary movements.

8 Conclusion and Future Work

The paper addresses problems of building large distributed systems interacting with the physical environment and being composed from a huge number of smart components, in essence, systems-of-embedded-systems. We cannot assume that the network architecture in such a system is homogeneous. This work is a contribution towards achieving seamless integration of different components in such an environment, controlling the flow of information by explicitly specifying functional and temporal dissemination constraints.

The paper presented the general model of a sentient object to describe composition, encapsulation and interaction in such an environment and developed the Generic Event Architecture GEAR which integrates interactions through the environment and the network. To our knowledge, it is the first architecture to provide the possibility for hidden channel avoidance in the model, that is, a seamless integration of physical and computer information flows. This may have important implications on the way to architect systems-of-embedded-systems.

The notion of event channel has been introduced which allows to specify quality aspects explicitly, namely temporal properties. The COSMIC middleware is a first attempt to put these concepts into operation. COSMIC allows the interoperability of tiny components over multiple network boundaries and supports the definition of different real-time event channel classes.

We believe the concepts elaborated here to be of interest to build innovative event-based systems that have to portray some kind of real-time behaviour, because they deal with the environment. This serves application areas such as: small-scale embedded systems; federated large-scale embedded systems; ambient intelligence settings; ad-hoc and mobile dynamically configurable systems.

References

1. Hightower, J., Borriello, G.: Location systems for ubiquitous computing. *IEEE Computer* 34(8), 57–66 (2001)
2. Harrison, T., Levine, D., Schmidt, D.: The design and performance of a real-time corba event service. In: *Proceedings of the 1997 Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA)*, Atlanta, Georgia, pp. 184–200. ACM Press, New York (1997)
3. Meier, R., Cahill, V.: Steam: Event-based middleware for wireless ad hoc networks. In: *Proceedings of the International Workshop on Distributed Event-Based Systems (ICDCS/DEBS'02)*, Vienna, Austria, Vienna, Austria, pp. 639–644 (2002)
4. Casimiro, A.,(ed.): Preliminary definition of cortex system architecture. CORTEX project, Deliverable D4 (April 2002), IST-2000-26031
5. Bacon, J., Moody, K., Bates, J., Hayton, R., Ma, C., McNeil, A., Seidel, O., Spiteri, M.: Generic support for distributed applications. *IEEE Computer* 33(3), 68–76 (2000)
6. Meier, R., Cahill, V.: Taxonomy of distributed event-based programming systems. *The Computer Journal* 48(5), 602–626 (2005)
7. Kaiser, J., Mock, M.: Implementing the real-time publisher/subscriber model on the controller area network (CAN). In: *Proc. 2nd International Symposium on Object-oriented Real-time distributed Computing*, Saint-Malo, France (May 1999)
8. (OMG), O.M.G.: CORBAservices: Common Object Services Specification - Notification Service Specification, Version 1.0 (2000)
9. Oki, B., Pfluegl, M., Seigel, A., Skeen, D.: The information bus - an architecture for extensible distributed systems. *Operating Systems Review* 27(5), 58–68 (1993)
10. TIBCO: Tibco rendezvous concepts, release 7.0 (April 2002)
11. Parado-Castellote, G., Schneider, S., Hamilton, M.: NDDS: The realtime publish subscribe network. In: *IEEE Workshop on Middleware for Distributed Real-Time Systems and Services*, pp. 222–232 (1997)
12. RTI: Real-time inovations. network data delivery service, <http://www.rti.com>
13. Mori, K.: Autonomous decentralized systems: Concepts, data field architectures, and future trends. In: *Int. Conference on Autonomous Decentralized Systems (ISADS93)*. (1993)
14. Carriero, N., Gelernter, D.: Linda in context. *Communications of the ACM* 32(4), 444–458 (1989)
15. Kim, K., Jeon, G., Hong, S., Kim, T., Kim, S.: Integrating subscription-based and connection-oriented communications into the embedded CORBA for the CAN Bus. In: *Proc. IEEE Real-time Technology and Application Symposium* (May 2000)
16. Lankes, S., Jabs, A., Bemmerl, T.: Integration of a CAN-based connection-oriented communication model into Real-Time CORBA. In: *Workshop on Parallel and Distributed Real-Time Systems*, Nice, France (April 2003)
17. Robert Bosch GmbH: CAN Specification V2.0. Technical report (September 1991)
18. Kopetz, H., Holzmann, M., Elmenreich, W.: A Universal Smart Transducer Interface: TTP/A. *International Journal of Computer System, Science Engineering* 16(2) (2001)
19. (OMG), O.M.G.: Smart transducer interface, initial submission (June 2001)
20. CORTEX Consortium: CORTEX project Annex 1, Description of Work (October 2000) <http://cortex.di.fc.ul.pt>
21. Hopper, A.: The Clifford Paterson Lecture, 1999 Sentient Computing. *Philosophical Transactions of the Royal Society London* 358(1773), 2349–2358 (2000)

22. Veríssimo, P., Rodrigues, L.: Distributed Systems for System Architects. Kluwer Academic Publishers, Dordrecht (2001)
23. Meier, R.(ed.): Preliminary definition of cortex programming model. CORTEX project, IST-2000-26031, Deliverable D2 (March 2002)
24. Führer, T., Müller, B., Dieterle, W., Hartwich, F., Hugel, R.: Walther, M.: Time triggered communication on CAN (2000), <http://www.can-cia.org/can/ttcan/fuehrer.pdf>.
25. LIN Consortium: Local Interconnect Network: LIN Specification Package Revision 1.2. Technical report (November 2000)
26. FlexRay Consortium: FlexRay Communications System Protocol Specification, Version 2.0. Technical report (June 2004)
27. Veríssimo, P., Casimiro, A.: Event-driven support of real-time sentient objects. In: Proceedings of the 8th IEEE International Workshop on Object-oriented Real-time Dependable Systems, Guadalajara, Mexico (January 2003)
28. Veríssimo, P., Casimiro, A.: The Timely Computing Base model and architecture. Transactions on Computers - Special Issue on Asynchronous Real-Time Systems 51(8), 916–930 (2002)
29. Eugster, P.T., Felber, P., Guerraoui, R., Kermarrec, A.M.: The many faces of publish/subscribe. Technical Report DSC ID:200104, EPFL, Switzerland (2001)
30. Livani, M., Kaiser, J., Jia, W.: Scheduling hard and soft real-time communication in the controller area network. Control Engineering 7(12), 1515–1523 (1999)
31. Kopetz, H., Grünsteidl, G.: TTP - A Time-Triggered Protocol for Fault-Tolerant Real-Time Systems. Technical Report rr-12-92, Institut für Technische Informatik, Technische Universität Wien, Treilstr. 3/182/1, A-1040 Vienna, Austria (1992)
32. Kaiser, J., Mitidieri, C., Brudna, C., Pereira, C.: COSMIC: A Middleware for Event-Based Interaction on CAN. In: Proc. 2003 IEEE Conference on Emerging Technologies and Factory Automation, Lisbon, Portugal (September 2003)
33. Becker, L.B., Gergeleit, M., Schemmer, S., Nett, E.: Using a flexible real-time scheduling strategy in a distributed embedded application. In: Proc. of the 9th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), Lisbon, Portugal (September 2003)
34. Mock, M.: On the real-time cooperation of autonomous systems. Fraunhofer Series in Information and Communication Technology 6(2204) (2204)
35. Kopetz, H., Veríssimo, P.: Real-time and Dependability Concepts. In: Mullender, S.J. (ed.) Distributed Systems, 2nd edn. pp. 411–446. ACM-Press, Addison-Wesley, New York (1993)
36. Kopetz, H.: Real-Time Systems. Kluwer Academic Publishers, Dordrecht (1997)
37. Kopetz, H., Kim, K.H.: Temporal uncertainties in interactions among real-time objects. In: 9th Symp. on Reliable Distributed Systems (SRDS-9), Huntsville, AL, USA, pp. 165–174. IEEE Computer Society Press, Los Alamitos (1990)
38. Ramamritham, K.: The origin of TCs. In: Proceedings of the First ACM International Workshop on Active and Real-Time Database Systems, Skovde, Sweden, pp. 50–62. Springer, Heidelberg (1995)
39. Verissimo, P.: Causal delivery protocols in real-time systems: A generic model. Journal of Real-Time Systems 10(1), 45–73 (1996)
40. Martins, P., Sousa, P., Casimiro, A., Veríssimo, P.: Dependable adaptive real-time applications in wormhole-based systems. In: Proceedings of the 2004 International Conference on Dependable Systems and Networks, Florence, Italy, pp. 567–572. IEEE Computer Society Press, Los Alamitos (2004)
41. Kaiser, J., Brudna, C., Mitidieri, C.: Cosmic: a real-time event-based middleware for the can-bus. J. Syst. Softw. 77(1), 27–36 (2005)

Flexible Communication Architecture for Dependable Time-Triggered Systems

Christoph Heller, Josef Schalk, Stefan Schneelee,
Maria Sorea, and Sebastian Voss

EADS Deutschland GmbH,
Corporate Research Centre,
81663 Munich, Germany
`stefan.schneelee@eads.net`
`http://www.eads.net`

Abstract. The trend for more flexible communication architectures, in particular for safety critical aeronautic applications, reflects the growing need for an optimized design approach. Customer requirements for additional functionality and changes caused by unpredictable obsolescence policies may necessitate requests for renewal of technologies during the product life cycle in a maintainable approach. The challenge is to develop an architecture approach, allowing reusability of existing application code, scalability and providing independence of the underlying system communication.

Over recent years, shared networks have become a rapidly emerging technology in aeronautics and space, since they offer — in contrast to point-to-point connections — more flexibility in terms of architecture and reduced wiring. Thus, they present the prospect of potential savings in cost and weight. In particular, due to their reliability and strongly deterministic behavior, time-triggered shared networks have evolved as eligible communication protocols for safety-critical applications in aerospace.

We propose an approach in terms of dependable and flexible communication architecture that permits more flexibility in the use of time-triggered technologies and delivers a more effective, reliable and dependable system design.

1 Introduction

The present trend in aeronautic technology is to replace mechanical controls with electronic x-by-wire solutions. Fly-by-wire technology was already introduced in the 1980s [1]. Apart from the aeronautic industry, this trend also took place in the space and for the automotive industries, as well as many other sectors. These x-by-wire controls require a system to respond to stimuli within some small upper limit of time. Hence, they are classified as (*hard*) *real-time (RT) systems*. As outlined in section 2 there are applications where it is favorable to build up a control of this nature as a *distributed RT system*, instead of a

centrally controlled system. In many cases the subsystems, a distributed system is composed of, differ in terms of computing power and architecture, or they are dynamic in their temporal behavior. The distributed system can therefore be classified as *heterogeneous*.

Proper intra-system communication is crucial for a distributed system to be able to operate reliably and dependably [2] [3].¹ Commercial reasons entail that communication architecture is increasingly realized as shared communication buses instead of several discrete point-to-point connections (for instance ARINC-429) [4]. This replacement exerts an additional burden of complexity on system designers. Bus arbitration, mapping signals to frames, scheduling, and the electrical configuration of a bus with many subscribers are not trivial issues. Selection of a specific bus may introduce additional design constraints. Furthermore, it is an acknowledged fact that the impact of the underlying protocol on architectural decisions and on software development is highly significant. Taking the many different aspects into consideration, this situation may be regarded as leading to suboptimal system configurations, which are difficult to maintain and to scale.

In this paper we propose a novel architecture approach that supports a high degree of static and dynamic heterogeneity for the integrated components. Furthermore, we further argue that optimal configuration of the underlying communication protocol leads to a higher level for flexibility of the entire architecture.

The remainder of this paper is structured as follows. Section 2 provides background information on the characteristics of shared-bus architectures and the usage of time-triggered protocols.

Section 3 analyzes the requirements and challenges confronting a flexible communication architecture for dependable distributed systems in the aeronautic field of application.

Section 4 describes the implementation of methods designed to realize the approaches presented in section 3. The dependable and flexible architectural approach is introduced. The architecture consists of two key elements, namely the *Abstraction Layer* and the *Communication Layer*. The overall approach comprising both layers is referred to as *Communication Abstraction Layer*. The Abstraction Layer providing the physical implementation of these services is hidden to the application. This releases the system designer from bus specific considerations. The quality of services guarantees system requirements, in particular those related to timing and fault-tolerance. The Communication Layer comprises capabilities which enable bus designers to implement a protocol-specific and optimized bus configuration. There is the additional possibility of migration between the various time-triggered protocols. These capabilities consist of a set of design rules for supporting the migration, an application-specific set of physical parameters, and a feasible scheduling mechanism with the focus on an optimized bus load. The paper concludes with a discussion of the architecture presented and forecasts future developments in this research area.

¹ Dependability of a computing system is the ability to deliver service that can justifiably be trusted. Reliability is the continuity of a correct service.

2 Time-Triggered Protocols for Dependable Distributed Systems

The communication architecture presented in this paper has been developed for dependable distributed systems on the basis of a time-triggered bus. This section provides background information on shared-bus architectures and the properties of time-triggered communication.

2.1 Distributed Systems

An electronic control system can be built up either as a centrally controlled system or a distributed system. A centrally controlled system consists of a central processing system with all other functional units like sensors and actuators directly connected with this central unit via point-to-point connections. The peripheral units are more or less "stupid" devices, whereas all system intelligence is located inside the central unit.

The following definition of the term *distributed system* can be found in the literature [5].

"A distributed system consists of a collection of distinct processes, which are spatially separated, and which communicate with one another by exchanging messages. [...] A system is distributed if the message transmission delay is not negligible compared to the time between events in a single process."

If we elaborate on this citation, a distributed system consists of several subsystems, interacting by exchanging messages to render the functionality of the global system. Other forms of interaction like shared memory or repository systems are not feasible for aeronautic systems due to the spatial separation and encapsulation required.

The advantages of a distributed system architecture are increased composability, scalability and manageability as a result of the functional encapsulation of the subsystems. A distributed architecture is also preferable in terms of reliability, since if one subsystem failed, the remainder of the system might be able to continue operating. It is even possible that other subsystems take over the functionality of the failed one. Finally, a distributed architecture is the prerequisite to reduce the wiring complexity of a system, as for example several sensors and actuators can be integrated into one smart subsystem with a reduced number of interfaces to the other subsystems.

2.2 Shared-Buses Versus Point-to-Point Connections

A proper intra-system communication is crucial for the reliable operation of a distributed system.

From the communication point of view, the subsystems are called *nodes*. The interconnections between the nodes can be realized as several discrete point-to-point connections or as a shared bus. A point-to-point connection is a communication channel with one transmitting node and one or more receiving nodes. This

allows each transmission to utilize the full available bandwidth of the communication channel. If the system is based on a shared bus, only one communication channel is used, to which all nodes are connected. Since more than one node is generally able to transmit on the shared bus, the available bandwidth of the communication channel has to be shared among the nodes. Furthermore, it is conceivable that multiple nodes might attempt to transmit simultaneously. It is therefore necessary to use a dedicated protocol to organize bandwidth allocation and arbitration of the shared communication medium. The benefits of a shared bus architecture, which have to be assessed in tandem with these additional expenses entailed, are greater flexibility, scalability and composability, as well as a reduced and simplified wiring compared with an architecture based on point-to-point connections. Because reduced harnesses also lead to reduced weight, shared buses are the technology of choice in aeronautic applications. This paper only addresses shared-bus architectures.

2.3 Time-Triggered Versus Event-Triggered Communication

As previously mentioned, a suitable protocol is required to organize the bandwidth allocation and arbitration of a shared communication bus. The communication can either be organized according to an event-triggered or a time-triggered principle — both having their specific benefits and drawbacks.

In the case of event-triggered communication, the message sending is triggered by the occurrence of external events. Such an event might be the sudden change of state in a monitored physical variable or a user input. Event-triggered communication protocols have to implement mechanisms for ensuring that simultaneous events do not lead to message collisions. This can be achieved by assigning each type of message a certain priority level. The sending of a message is delayed until all transmissions with greater priority levels are completed. It is not possible to specify worst-case transmission delays or jitters for event-triggered messages, as events in general occur arbitrarily. In theory, a message could be infinitely delayed by other messages with greater levels of priority. If an event-triggered protocol (like CAN [6]) were be used for a dependable distributed system, additional complexity is required at application or even at system level to solve the problem of an unpredictable intra-system communication.

If a time-triggered protocol is used to organize communication on the shared bus, the message transmission is triggered by the progression of time. A periodically repeated communication schedule specifies which message is sent by which node at which point in time. A prerequisite for the correct operation of a time-triggered system is a consistent knowledge of the global time among all nodes. This requires either a clock master, which informs all nodes about the global time, or a distributed mechanism that synchronizes the local clocks of the nodes and generates a global timebase. The latter is used by most implementations of time-triggered protocols, FlexRay or TTP/C among them. Correct message scheduling and clock synchronization ensure collision-free accesses to the shared bus. The major advantage of time-triggered communication is its determinism. Since the message transmission times are specified a-priori, the transmission

delay of each message is known and the transmission jitter is reduced to the imprecision of the clock synchronization. Using time-triggered operating systems, it is possible to synchronize application and communication, resulting in a deterministic system with exact knowledge at which point in time a message is produced, transmitted and consumed. A side-effect of time-triggered communication is greater transparency of the system health state, derivable from the known sending instants of each node. Because of these characteristics, time-triggered protocols seem to be appropriate for the design of dependable distributed real-time applications.

Another advantage is their greater efficiency compared to event-triggered protocols. In order to allow for robustness against variations in the occurrence of events, an event-triggered system should be designed in a way that its expected busload is less than 60-70% [7]. Besides the gaps required for the clock synchronization, time-triggered protocols can utilize the full bandwidth of the shared bus. Since the messages for time-triggered communication are identified by their temporal occurrence, no identifiers for the type or sender of the message have to be transmitted. This also increases the efficiency of the time-triggered communication principle.

The main drawback of time-triggered communication is the lack of flexibility compared to event-triggered communication. In an event-triggered system, messages or nodes can simply be removed or added, provided that this is compliant with the expected busload. In a time-triggered system, the communication schedule defines the participating nodes and the messages they exchange. This schedule has to be specified at the design phase of the system. Adding or removing messages or nodes involves setting up a new communication schedule. Nevertheless, the advantages of time-triggered protocols make them important for dependable distributed systems, especially in the aeronautic domain.

Table 1 gives a comparison of the event-triggered and time-triggered communication principle.

The analyses and the developed communication architecture presented in this paper are based on TTP/C [8] and FlexRay [9], as at the date of origin of this paper, they are the most common implementations of a time-triggered protocol.

3 Requirements in a Communication Architecture for Time-Triggered Systems

In designing a distributed system architecture with a time-triggered bus, there are a lot of requirements which have to be taken into account. In this chapter we address the requirements of aeronautic communication systems in general, and especially under the consideration of a time-triggered protocol. Due to their reliability, strongly deterministic behavior and heterogeneous character the configuration get rather complex. The complexity involved in defining a consistent parameterization of a time-triggered protocol means that we need to describe the requirement for a scheduling strategy, in order to meet system requirements and handle complexity in configuration. Furthermore, on the basis of all these

Table 1. Comparison of event-triggered and time-triggered systems

	EVENT-TRIGGERED
Flexibility	nodes/messages can be added/removed even at runtime
Reliability	communication scenario unpredictable
System Health	opaque: no message = no event or faulty node?
Bus Load	not deterministic, depends on occurrence of events
Efficiency	larger frame overhead bus load has to be less than 60-70%
Message Content	event observation ideal for rare events
	TIME-TRIGGERED
Flexibility	communication schedule has to be specified at design time
Reliability	deterministic communication
System Health	transparent: each node sends periodically
Bus Load	as defined in the communication schedule
Efficiency	less frame overhead (no message/node identifiers) full bandwidth can be used
Message Content	state observation ideal for frequently changing variables

requirements we evaluate the need for a middleware approach and a suitable Communication Abstraction Layer.

3.1 Requirements of Distributed Aeronautic Systems

The functional and architectural requirements in communication are manifold in distributed aeronautical systems. The functional hard real-time demands placed on aeronautic systems are obvious. They necessitate bounded and acceptable message transmission delays and jitters from the underlying communication.

In terms of architecture, distributed aeronautic systems often have to cope with heterogeneity, demanding flexibility from the intra-system communication. This paper distinguishes between two forms of heterogeneity: static heterogeneity on the one hand, and dynamic heterogeneity on the other one.

A system is heterogeneous in a static manner, if, for example, it consists of a heterogeneous set of subsystems, differing in their hard- or software architecture. As a consequence of this, the individual subsystems might differ in their computing power and the way in which the communication medium is interfaced. In modern civil aircraft ² Integrated Modular Avionics (IMA) is the reference architecture for control systems. An IMA module is a powerful computation unit, running an ARINC 653 [10] compatible operating system to execute complex control algorithms. In a distributed system, such an IMA module might be connected with rather simple sensor/actuator modules, equipped with a low-cost microcontroller and without a dedicated operating system. If the time-triggered

² e.g. Boeing 787, Airbus A380.

bus is the only communication interface of a subsystem, it might be beneficial to synchronize its application with the timebase of the time-triggered bus. This could either be done by using a time-triggered operating system or by using a simple sequencing routine. Both would call the application tasks in a predefined sequence, synchronized with communication on the bus. As the name IMA implies, several functions are integrated in one module. Thus, the interface to the time-triggered bus is probably not the only communication interface of the IMA module under consideration. The AFDX [11] network that acts as a backbone, interconnecting the individual modules of the IMA architecture, might — from a hierarchical perspective — be more important than the time-triggered bus interface. It might therefore be detrimental to synchronize the operating system with the less important time-triggered bus. Since the IMA software architecture restricts the usage of interrupts — their usage could either be completely or temporarily prohibited — an IMA module would have to interface a time-triggered bus in an asynchronous manner. Hence, the communication architecture developed has to support software architectures with and without operating system, and synchronous and asynchronous bus accesses.

Apart from its heterogeneous hardware and software architecture, another aspect of a statically heterogeneous communication system might be unequal assignment of the (static) bus bandwidth among the nodes. The system might consist of nodes sending a lot of messages requiring significant bandwidth, as well as nodes, requiring only a small amount of bandwidth.

Finally, the topology of the bus itself could be heterogeneous. A set of nodes could be located close to each other, resulting in short interconnections, whereas other nodes could be located in a remote distribution from the others. The structure of the bus could be a linear bus, a star or even a combination of these two topologies.

Dynamic heterogeneity is the other form of heterogeneity a communication system might have to deal with. This is characterized by a change in the communication scenario at runtime. If a system has to switch between different operational modes, it might be expected that such a change also implies a change in the intra-system communication. Hence, it might be necessary to change the bandwidth allocation among the nodes dynamically at runtime. In an aeronautic application, the communication might be different depending on whether the aircraft is parked on the ground or flying at cruising altitude. The communication architecture should be able to adapt to changing communication requirements at run-time. In some situations, e.g. related to maintenance, an important communication requirement relates to integrating a new device on-the-fly without having complete knowledge of the communication architecture.

Another form of dynamic heterogeneity lies in the type of messages which the subsystems exchange. Generally, there are two types of message: *event messages* and *state messages*. A state message is sent periodically, corresponding to the sampling of the message value. State messages are therefore ideal for the cyclic communication of a time-triggered system. Nevertheless, scenarios are conceivable, where the transmission of a state message is inappropriate, as it would

cause a lot of transmissions without new information content. This applies to rarely changing values, for example observation of whether the user has pressed a button. Instead of periodically sending a message with the content "button not pressed", it is more efficient just to send a message for the event when the button is pressed. Since such a message is coupled to the occurrence of an event, it is called *event message*. Both types of messages should be supported by the communication architecture.

3.2 The Complexity of Time-Triggered Protocols

Using a time-triggered protocol for the communication of a distributed system significantly increases the necessary effort for the system design process compared to using an event-triggered protocol. If an event-triggered protocol is used, the application can be implemented regardless of the system communication, provided that the expected busload lies in a valid range. When using a time-triggered protocol, the communication also has to be precisely specified. If the application is synchronized with the bus, communication and application have to be well inter-coordinated.

System designers who are highly qualified in their system domain have to address complex communication aspects which may lead to a suboptimal configuration. Such issues are very often solved by using a higher bandwidth but this adds additional and avoidable electrical complexity to the system. Taking all these aspects into consideration, the impact of introducing a time-triggered bus into existing design processes is immense. There are also many open issues pending related to integration within chance and configuration management.

Even the generation of a proper parameter set to realize the bus configuration obtained from the bus designer, is a complex problem for time-triggered protocols subject to analysis. A TTP/C node is configured using a data structure called message description list (MEDL) [12], which has to be loaded in the communication controller³ before communication can be established. A MEDL is organized in the form of interlinked tables consisting of binary values. For example, more than 750 parameters have to be set for the most simple TTP/C configuration.⁴

The complexity of FlexRay parameterization lies in the same order of magnitude. The analyzed FlexRay communication controllers⁵ are parameterized using register settings. The existence of different communication controller implementations for FlexRay differing in their parameter sets again increases the complexity of the problem. The most common data interchange formats for FlexRay parameter sets are the Controller-Host Interface (CHI) file and the Field Bus Exchange Format (FIBEX) [16]. The CHI file is based on a proprietary format and contains a mixture of functions and configuration parameters,

³ The analyses are based on the AS8202NF TTP/C controller. [13].

⁴ A network consisting of four nodes. Only one cluster mode implemented, consisting of four round slots (cf. [8]).

⁵ MB88121B [14], MFR4300 [15].

as well as write instructions that directly set the registers of the communication controller to the desired values. Similar to the MEDL files, the parameter settings are not directly human-readable. The FIBEX file, in contrast, is a standardized XML⁶ file format for describing message-based communication systems. Nevertheless, the large number of parameters and numerous interdependencies means that setting the parameters by hand is not advisable. A dedicated tool is inevitably required to generate the parameter defined in the bus design.

3.3 Scheduling in Time-Triggered Architectures

A scheduling strategy is a key component for obtaining high level of performance with time-triggered systems. In order to fulfill system requirements, time-triggered systems must react within precise timing constraints. As a consequence, the correct behavior of these systems depends on the value of the computation and on the time at which the results are produced and transmitted.

Schedulers coordinate the execution of system tasks in order to meet the requirements for their temporal behavior. Since a shared system is a multiprocessor system, these tasks can overlap in time and may be characterized by different priority and computation duration. The CPUs are assigned to various task according to a scheduling policy with the help of a scheduling algorithm. A feasible schedule is required to take into account the different system requirements, e.g. fault tolerance, determinism, scalability, reliability, validation and verification, etc. . This schedule must be compliant with the system requirements. In addition to task scheduling at system level, a time-triggered system must also involve communication of messages at the protocol level. This means that the traditional scheduling approach has to be extended by considering message scheduling to address the whole complexity in designing heterogeneous systems based on a time-triggered bus.

3.4 Middleware Approaches and Abstraction Layer

Common bus systems only specify the Physical Layer and the Data Communication Layer with respect to the ISO/OSI model [17]. For instance, the transportation of messages with more than 8 Byte is not addressed in a CAN [6] based system. There are basically two methods to access the Data Communication Layer in embedded software development. On one hand, software engineers can interface the bus hardware directly. On the other hand, a common *Communication Abstraction Layer* could be introduced to provide a standard interface even for applications which are addressing different functionalities. The first approach is proprietary and may lead to problems with integrating subsystems from different vendors on the same shared bus. Different strategies related, e.g. to start-up and fault scenarios, means that the second approach clearly demonstrates significant benefits in terms of dependability and scalability. An adoption of the generic Communication Abstraction Layer to a specific application has to be

⁶ <http://www.w3.org/XML/>

carried out by configuration files. However, the use of common Communication Abstraction Layers is not yet state-of-the art in the aeronautical sector for subsystems. This is because there is no open industry standard for systems without an operating system in conformity with ARINC 653 [10]. There are a number of open standards in other industry sectors, such as the OSEK [18] initiative within the automotive industry, or the Smart Distributed System (SDS) from Honeywell [19] used for production equipment. The differences between these layers are mainly based on optimization for a specific field of application (e.g. real time constraints) and hardware environments (e.g. size of ROM, RAM, ...). An overview of all the available and ongoing work related to Middleware and service oriented computing would be beyond the scope of this paper. [20] gives an interesting overview of the state-of-the art in Middleware technology. A promising domain-independent approach addressing the whole design methodology and validation and the verification of integrated embedded systems is being developed in the context of the EU sponsored project "Dependable Embedded Components and Systems" (DECOS) [21]. The goal of DECOS is to develop homogeneous and operation system based platforms. Services such as virtual communication links for running CAN based legacy code on top of time-triggered networks are undoubtedly of interest, but they are outside the scope of the systems and hardware addressed in this paper, due to the overhead they introduce [22].

The next section introduces a Communication Abstraction Layer, optimized for heterogeneous embedded aeronautical applications, and strategies for an optimal configuration of the underlying protocols are presented.

4 Achieving Flexibility with Time-Triggered Systems

In order to meet the demands of the heterogeneous time-triggered communication system presented in section 3.1, the system architecture and its design process have to be flexible. Some flexibility aspects have to be implemented in the configuration of the protocols, while others are implemented in the software architecture and the system design. The objective is to have a design process that supports different types of nodes and different implementations of time-triggered protocols to permit code reusability ease of possible migration and facilitate certification issues.

4.1 Communication Abstraction Layer Concept

As referred to above, nodes with different hardware configurations might require different routines for accessing functionalities and data provided by the time-triggered bus. In order to alleviate the dependence between hardware and software, the underlying hardware has to be abstracted. This could either be done by implementing a dedicated driver for each combination of host and bus controller implementation, or by introducing a Hardware Abstraction Layer (HAL) and a generic driver covering all time-triggered protocols. The set of dedicated drivers has to provide the same function calls and data structures for the higher levels

of the software architecture. A drawback of this approach is the complexity involved. Up to $n \cdot m$ different drivers would have to be implemented for n different host controllers and m different bus controllers to be covered by the approach. A reasonable mitigation could be usage of a HAL in combination with a generic driver. This would require the implementation of n HALs for each different host controller and m instances of the generic FlexRay interface. This results in only $n + m$ different implementations and is therefore the approach of choice.

The driver itself should support both operation modes, the asynchronous and synchronous mode — depending on the requirements of the application. On top of these drivers several layers are settled. Figure 1 gives a comprehensive overview of the proposed architecture. A Communication Abstraction Layer itself consists of a two-layered approach, namely an Abstraction Layer and a Communication Layer.

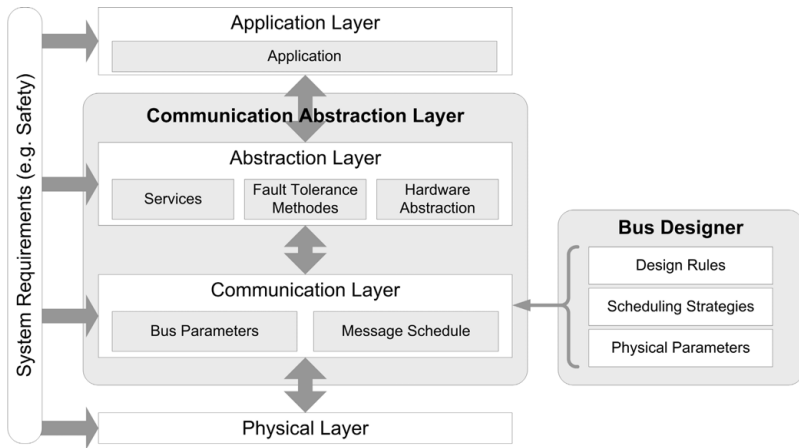


Fig. 1. Architectural Approach

All layers are implemented in ANSI C. The standard interface of the driver is the *Communication Layer*. This layer provides basic services to the operation system and to a sequencer:

1. Start Communication
2. Stop Communication
3. Sending of Raw Data
4. Receiving of Raw Data
5. Mode Change
6. Local Clock Synchronization
7. Error Handling

The adaptation to specific bus configuration is performed via configuration files to maintain the integrity of the driver code. These configuration files can

be mainly interpreted as a C structure containing the register settings of the respective bus controller. There is currently no standard available incorporating different bus systems. Typical proprietary options would have been the tabular MEDL format for TTP/C or the CHI format for FlexRay. A new structure was therefore introduced. As referred to above, such structures consist mainly of a large number of parameters and should be automatically checked for consistency and errors. These configuration files are processed during the initialization phase of the communication controller. The example in figure (2) shows the control flow of a TTP/C initialization sequence.

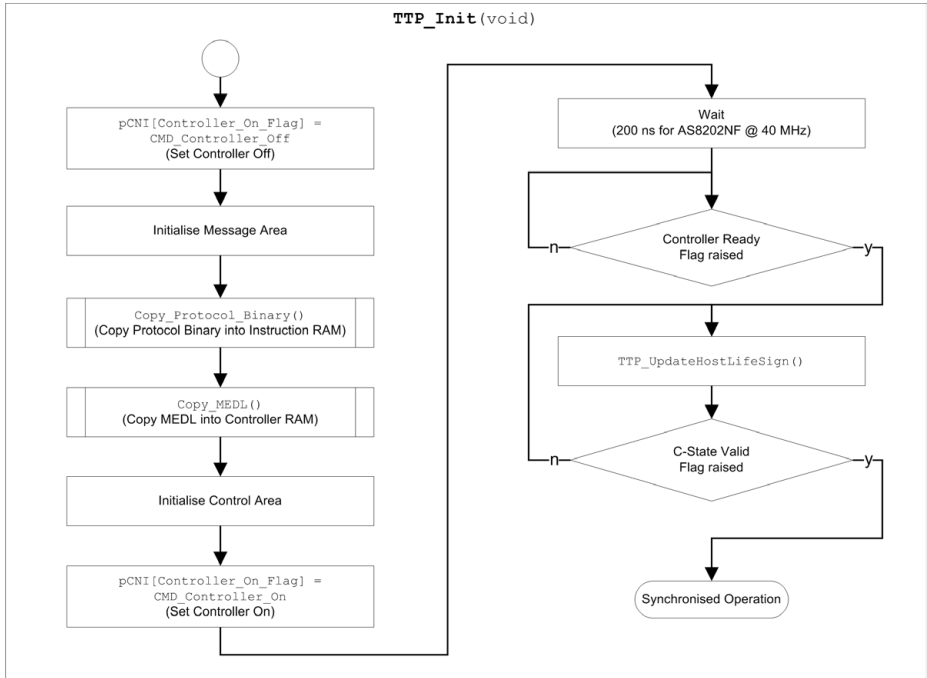


Fig. 2. Initialization Sequence

The function calls of the Communication Layer are intended to access the buffers on the frame level of the communication controller directly. They are therefore more suitable for integration within system calls. Performance tests showed that addressing a bus controller at signal level overloads common CPUs. Cyclic redundancy checks (CRC) and specific status are usually attached to the application's signals. Moreover, several instances of a signal are triggered to perform redundant data transmission. A higher level of abstraction is therefore desirable for working with signals at application level. This is achieved by introducing the *Abstraction Layer*. The core services of the Abstraction Layer are:

1. Set Message Value
2. Get Message Value

3. Sender Status
4. Receiver Status
5. Message Refresh Rate
6. Host Specification
7. Controller Status
8. ARINC 653 Calls

The service *Host specification* is necessary because referring signals by name is not always adequate. For example, redundant systems might entail specifying the dedicated sending or receiving host of a signal in order to perform voting algorithms at application level. These functions offer an appropriate level of abstraction to the application developer by using additional information from a bus independent communication matrix. The communication matrix specifies the system level requirements related to communication, such as bit size, sign, redundancy level, value range and maximal latency. This allows for syntactic and semantic run-time checks. As yet, there are also no open standards for defining the communication matrix.

4.2 Integration into Application Layer

A standard approach in aeronautics is to generate the application code with SCADE [23]. As this C code does not contain any system calls, the Abstraction Layer does not need to be integrated into the SCADE design process. In order to satisfy the inputs and outputs of SCADE Tasks, communication tasks have to be appropriately scheduled to access the time-triggered bus. There are basically two ways of integrating these communication tasks into the Abstraction Layer: deployment with or without an operating system in conformity with ARINC 653. In an environment without an operating system, the standard interface of the Abstraction Layer would be utilized. The benefit is that this interface software relies on an approved framework, and is not affected by changing the communication technology.

The ARINC 653 standard specifies two possibilities for establishing communication. The standard provides for a *Queuing Port* and a *Sampling Port*. In queuing mode, each new occurrence of a message overwrites the previous one. This represents the typical time-triggered communication behavior. In the sampling mode, the messages are stored in FIFO order. This mode of communication is specified for each signal within the communication matrix. Depending on the underlying time-triggered protocol the sampling mode places heavy constraints on the task schedule. The necessary API calls for achieving an ARINC 653 compliance are limited to *CREATE PORT*, *WRITE MESSAGE*, *READ MESSAGE*, *GET PORT STATUS* and *GET PORT ID*. Figure 3 illustrates the interdependence between the different software layers.

4.3 Introduction of the Bus Designer

As referred to section 3.2, the complexity of configuring a system based on a time-triggered bus is immense. In order to address this complexity and keep the

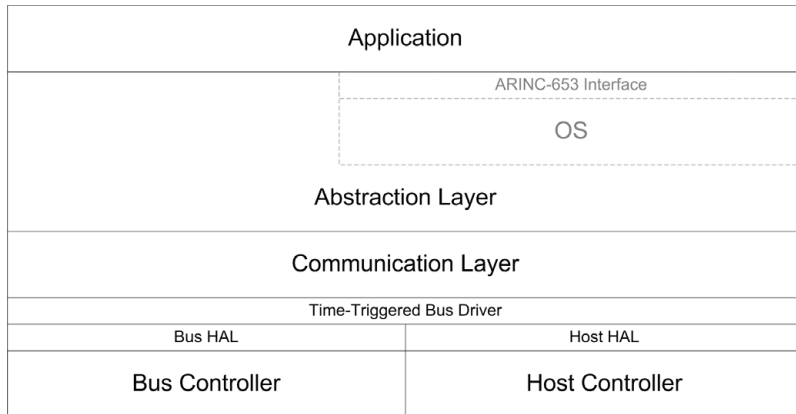


Fig. 3. Application Integration

system design approach feasible, the role of a bus designer should be integrated into the design process as shown in figure 4. The bus designer will support the whole process right from the start by providing design constraints for the system designer and will deliver an optimized protocol specific bus configuration to the final system integration. The bus configuration has to satisfy all constraints expressed by the hardware and protocol independent communication matrix and the application task schedule, delivered by the system designer.

4.4 Implementation of Scheduling Strategy

As stated in section 4.3, the role of the bus designer is to support the entire design process by providing design constraints and suitable scheduling strategies. As part of the scheduling strategy, task scheduling and message scheduling have to be considered in the new design process that incorporates the bus designer. Although, task and message scheduling are two separate activities under the responsibility of the system designer and bus designer respectively, there is a strong interrelation between those two activities. It is therefore necessary to combine the detailed knowledge of system designer, for example protocol-independent task schedule with all its parameters (e.g. response times, deadlines, computation, etc.,) with the protocol-specific knowledge of the bus designer, in order to obtain a possible optimized task and message scheduler for the parameters under consideration. A scheduling strategy therefore has to support the system development by defining the dependencies between task and message schedule. The scheduling strategy should also allow for finding feasible scheduling policies to provide improved system utilization with enhanced predictability on the basis of different system requirements as well as requirements imposed by the time-triggered communication.

When designing a communication scheduler the system designer usually will take system specific and protocol specific constraints into account. When

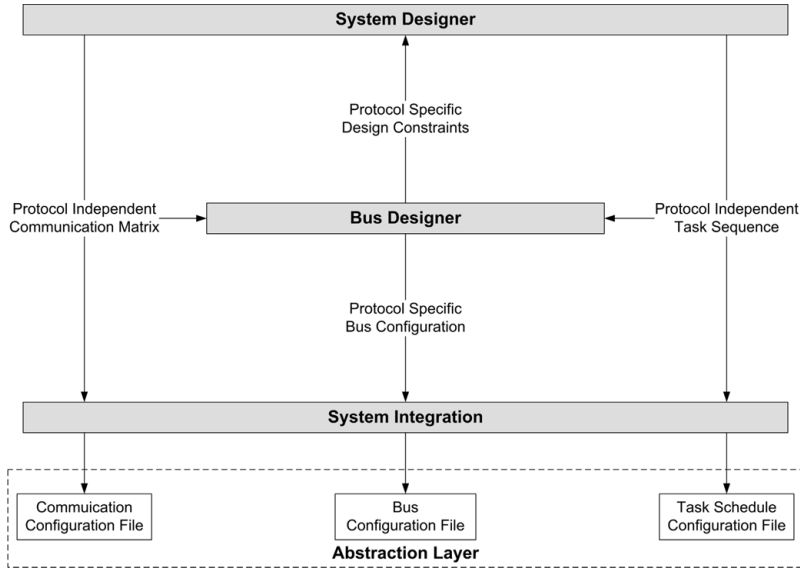


Fig. 4. Role of Bus Designer

considering different time-triggered protocols, such as TTP/C and FlexRay, this approach would lead to sub-optimal bus configurations, given the higher level of complexity by comparison with traditional communication protocols, like CAN. The amount of parameters in the mentioned time-triggered protocol above attains a complexity requiring detailed knowledge of the underlying protocol. An additional scheduling strategy is needed for optimizing parameters and achieving a feasible schedule to meet all system requirements:

We therefore address this problem by proposing a scheduling approach based on separation of scheduling related parameters using the knowledge of the system and bus designer.

The application-specific parameters can be assigned to the role of the system designer who provides specific information about the system requirements. These parameters can be divided in mainly architectural, physical, and function related parameters. Architectural Parameters describe the topology, such as number of nodes, star, bus, or hybrid architecture, etc. Function related parameters can be described by the attributes of signals or messages which realize system functionality, such as refresh rate, size, priority, etc. The definition of physical parameters and bus specific parameters is assigned to the role of the bus designer. These parameters include cycle length, slot length, number of slots, etc., and are rather protocol specific.

The role of the bus designer is indisputable to obtain optimal bus configuration on the basis of the given system requirements.

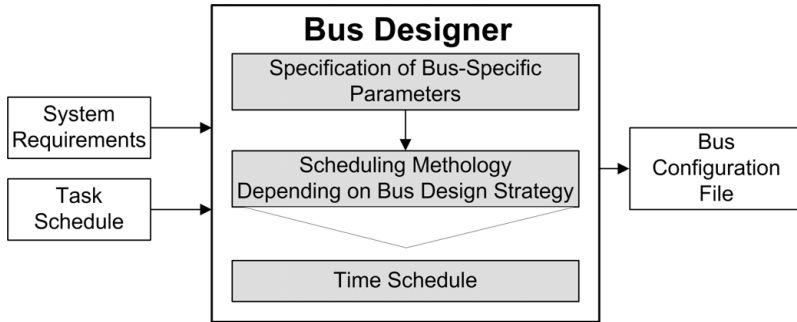


Fig. 5. Scheduling strategy

A two-step approach is proposed for achieving this goal, as illustrated in figure 5:

- Firstly, the bus-specific parameters are set in compliance with the time-triggered protocol under consideration. This is done with reference to the given system requirements as an input provided by the system designer.
- The allocation of payload to time-triggered slots has to be performed as a second step. This entails developing a feasible scheduling policy which is characterized by a set of rules, also known as scheduling algorithms. These algorithms have to be developed in accordance with system requirements. The task schedule provided by the system designer has a major impact on designing a feasible communication schedule. In this context, the relation between task and message schedule is significant.

4.5 Necessary Tools

It is obvious the approach described above requires consistent tool support. Stand-alone tools for all common bus systems are available on the market and usually have adequate maturity. The challenge is to integrate them into the existing design processes, e.g. code generation tools have to work in accordance with high aeronautic safety requirements.

There is therefore a need to develop a tool which integrates the configuration of the different layers and the task and message scheduling with the aim of supporting optimized bus parameterization. Research is currently being carried out into arranging such tools around a common repository. The aim is not to generate certified code, but to create configuration files for certifiable layers that can be verified automatically.

4.6 Configuration of Time-Triggered Protocols

Aside from the system design process and software architecture, many flexibility aspects of a time-triggered system have to be implemented with configuration

and parameterization of the underlying protocol. Further information about the methods presented in this subsection can be found in [24].

One of these aspects is the bandwidth allocation which is specified in the scheduling-related parameters of the protocols. FlexRay and TTP/C use a time division multiple access (TDMA) scheme to access the shared bus. The available transmission time on the bus is split into slots which are exclusively assigned to a specific node. The nodes are allowed to utilize the full communication bandwidth of the bus within their assigned slots. FlexRay additionally provides a dynamic segment, which is executed alternately with the static TDMA scheme. The dynamic segment can be used by each node. An arbitrating mechanism based on the minislotting scheme with the use of message priorities regulates bus accesses in this segment.

Organization of the TDMA scheme is different for FlexRay and TTP/C. In the case of FlexRay, all the slots must have the same length but a node can be assigned to more than one slot per round. In the case of TTP/C, each node can only utilize one slot per TDMA round, whereas the length of the slots may be different. The configuration therefore has to assign the individual nodes with the correct number of slots or a correct slot length on the basis of the node's communication requirements. Changing the bandwidth allocation at runtime might be necessary for a dynamically heterogeneous communication system. This is not trivial for both protocols, since they are intended for use with rather static communication schedules. The FlexRay implementations analyzed do not allow changes in slot size or slot allocation within the static TDMA segment. The dynamic segment with its priority management therefore has to be used for this purpose. TTP/C offers the possibility of implementing different communication schedules (*cluster modes*). The protocol is then able to switch between the schedules at runtime. This can be achieved using a special service provided by the protocol, called *cluster mode change*. The service has restricted usability, because the length of the individual slots has to be the same for all cluster modes and it is not possible to assign nodes for different slots in different cluster modes.

A heterogeneous architecture for the bus, interconnecting the individual nodes of a time-triggered network also has to be considered for the configuration of the protocol deployed. Above the physical layer (cf. ISO/OSI model [17]), the impact of the bus topology is reduced to different transmission delays resulting from the physical layer drivers, star couplers and the cables themselves. Since the clock synchronization mechanisms used by the analyzed protocols are based on comparison of the expected arrival time and the actual arrival time of received frames, knowledge of the delay caused by the bus topology is required. In order to achieve this, each transmission on the shared bus has to be regarded as a *virtual point-to-point connection* between the sending and receiving node. The delay caused by the hardware involved in this virtual connection is summed up and used as a correction term for the expected arrival time of the frame. These terms are the *action point offsets* for FlexRay and the *delay correction* parameters for TTP/C.

Time-triggered communication systems are intended for periodic transmission of state messages (cf. 2.3). Special mechanisms are required, if event messages should be transmitted in addition. For this purpose, FlexRay implements the dynamic segment that is not assigned to a specific node but can be used by any node in the network. The decision on which messages are transmitted in the dynamic segment is based on the message priorities and is performed according to a minislotting scheme [9]. TTP/C does not provide a mechanism for the transmission of event messages at protocol level. Such a mechanism has to be implemented on top of the protocol at application level. A certain amount of bandwidth has to be reserved and — if required — used for the transmission of an event message.

5 Discussion

The approach presented above provides the fundamental benefit of a flexible communication architecture for time-triggered distributed systems. Important aspects of the proposed approach are summarized in this section.

As already anticipated in section 2.3, it is not possible to achieve the same level of flexibility with a time-triggered protocol as with an event-triggered one. Parameterization of the analyzed protocols FlexRay and TTP/C permits incorporation of a heterogeneous bus topology or statically heterogeneous assignment of the available bandwidth among the nodes. The weak point in time-triggered communication is dynamic flexibility. In particular, dynamic redistribution of the bandwidth at system runtime is cumbersome. Flexibility aspects aimed at supporting different hardware and software architectures are addressed with the presented Abstraction Layer and introduce the bus designer into the system design process.

One of the central assumptions made for the Abstraction Layer and its related configuration files is that the description of the overall communication matrix of a system can be realized in practical terms independently of hardware and protocol to obtain a clear separation between communication and bus configuration. If this were not the case, a standard aeronautical utility bus system would be selected. This might anyhow be the most promising way forward. All configurations could then be optimized for this unique solution. The Abstraction Layer is eligible because it acts as a standard interface. However, all migration aspects would become obsolete. Nevertheless, the role of a bus designer is essential for achieving optimal bus configuration. The interfaces between system and bus designer should be carefully defined to avoid interdisciplinary misconception. An additional risk for the introduced concept arises from the complex industrial relations between airframers and multiple suppliers. A solution can only bring benefit if it is supported by both sides. Many years of experience demonstrates that international efforts for open standard specifications such as ARINC 664 (AFDX) [11] are the most promising way forward. The usability of this approach for an open specification should be determined after successful integration within first prototypes.

The resources required to develop the required tools referred to in the previous chapters are immense. Each single problem that is targeted by this approach, in terms of scheduling or optimal parameterization, relates to ongoing fields of research. These problems are not covered by standard tools.

In many sectors, a trend is emerging with standardized interfaces and hardware abstraction for embedded systems. This trend should also be assimilated by the aeronautical industry at the utility system level. The approach presented in this paper is an initial step towards a standardized and industry-wide interface description for dependable distributed systems.

6 Conclusion and Future Prospects

The objective of this paper was to present an approach for a dependable and flexible communication architecture in shared systems, based on a time-triggered protocol. The approach allows more flexibility and scalability in the use of the protocol and provide an efficient and reliable system design because it permits reusability of existing code.

The paper began with a comparison of shared buses and point-to-point connections for the interconnection of a distributed system. It highlighted the advantages of greater flexibility provided by a shared bus architecture. The advantages offered by a deterministic time-triggered intra-system communication for a distributed system have been identified, in particular those related to reliability.

Section 3 of the paper presented the requirements defined for a distributed system by the aeronautic environment. The challenges posed by the complexity of time-triggered protocols were identified and the need to adapt the system design process and introduce a Communication Abstraction Layer was explained. Explanations were provided as to why current middleware approaches are not appropriate to meet these challenges.

Section 4 described the implementation of an Abstraction Layer to provide flexible support for different hardware and software architectures by alleviating dependency between these two. An outline was provided for the manner in which the Abstraction Layer is integrated into an application and the kind of tools required for the system design process. In addition, an initial scheduling strategy approach was proposed to meet the higher requirements of dependable time-triggered system design. Flexibility aspects that have to be considered in the protocol configuration were also addressed in this section.

Possible weak points and open issues of the presented approach have been discussed in section 5.

The stand-alone tools already developed for configuration and scheduling of the time-triggered bus will be integrated in a single coherent framework. The whole approach will be implemented in a realistic use case system within the coming months. More research has to be carried out into developing suitable hardware platforms. The level of support of the event driven protocol CAN will be a big challenge in the future. Since it is based assumptions other than time-triggered protocols, harmonization of the Abstraction and Communication Layer

will therefore be more difficult but still possible. As far as the FlexRay driver developed in the course of this project is concerned, improving integration of different FlexRay controller implementations, namely the Bosch and Freescale chips, remains an open issue. At the current stage of development, there is no unique FlexRay driver, although there are platform-specific drives available for both implementations. The configuration of the FlexRay protocol and the implementation of a specific configuration should be separated and highlighted. Providing practical proof of hardware independent communication design will be one of the most important outputs of the exercise with use-case system being deployed.

References

1. Traverse, P., Lacaze, I., Souyris, J.: Airbus fly-by-wire: A total approach to dependability. IFIP (2004)
2. Laprie, J.: Dependable computing and fault tolerance: concepts and terminology. Digest of FTCS-15, 2–11 (1985)
3. Echte, F.: Fehlertoleranzverfahren, 3rd edn. Springer-Verlag, Berlin (1990)
4. Aeronautical Radio Inc.: ARINC Specification 429-17 Part 1. Prepared by the Airlines Electronical Engineering Committee, Annapolis (2006)
5. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM 21(7), 558–565 (1978)
6. Lawrenz, W.: Controller Area Network. Huethig (2000)
7. Claesson, V., Ekelin, C., Suri, N.: The event-triggered and time-triggered medium-access methods. In: 6th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing ISORC, Hakodate, Hokkaido, Japan, pp. 131–134. IEEE Computer Society Press, Los Alamitos (2003)
8. TTTech Computertechnik AG: Time-triggered protocol TTP/C high level specification document, protocol version 1.1. Technical report, TTAgroup (2003)
9. FlexRay Consortium: FlexRay communications system protocol specification, version 2.1, revision A (URL) last visited (September 2006), <http://www.flexray.com>
10. Aeronautical Radio Inc.: ARINC Specification 653: Avionics Application Software Standard Interface. Prepared by the Airlines Electronical Engineering Committee, Annapolis (1991)
11. Aeronautical Radio Inc.: ARINC Specification 664 3. ed.: Aircraft Data Network, Part 5 - Network Domain Characteristics and Interconnection. Prepared by the Airlines Electronical Engineering Committee, Annapolis (1991)
12. TTTChip Entwicklungsgesellschaft mbH: TTP/C controller C2NF, controller schedule (MEDL) structure document, firmware version 2.0.3. Technical report, TTAgroup (2004)
13. Austriamicrosystems AG: TTP-C2NF communication controller –data sheet. Data Sheet Rev.1.6 (2006)
14. Robert Bosch GmbH: FlexRay IP-Module. User's Manual Revision 1.2.3, Automotive Electronics (2006)
15. Freescale Semiconductor: MFR4300 data sheet, rev. 1. Data sheet (2006)
16. Association for Standardisation of Automation and Measuring Systems: FIBEX - Field Bus Exchange Format Version 2.0. (2006)
17. ISO/IEC: Information technology - open systems interconnection - basic reference model - the basic model. Technical Report 7498-1 (1994)

18. OSEK. Technical report, TTAgroup (2005)
19. Woolever, B.: Application Layer Protocol Specification Version 2.0. Technical report, Honeywell Inc. MICRO SWITCH Division (1999)
20. RWTH Aachen: ULLA integration to middleware and existing technologies. Project report, RWTH Aachen University (2005)
21. Kopetz, H., Obermaisser, R., Peti, P., Suri, N.: From a federated to an integrated architecture for dependable real-time embedded systems. DECOS (2005)
22. Fuhrmann, H., Geilsdorf, H., Klein, L., Schnee, S.: System entwicklung basierend auf der decos architektur. Informationstagung Miktroelektronik 2006 pp. 139–150 Wien, Austria (2006)
23. Berry, G., Michael Kishinevsky, S.S.: System Level Design and Verification Using a Synchronous Language. Technical report, Esterel Technologies (2001)
24. Heller, C.: Time-triggered protocols for heterogeneous communication systems. Diploma Thesis, Department of Wireless Networks, RWTH Aachen University, Kackertstr. 9, 52072 Aachen, Germany (2006)

Business Process Monitoring for Dependability

Luciano Baresi, Sam Guinea, and Marco Plebani

Dipartimento di Elettronica e Informazione - Politecnico di Milano
Piazza L. da Vinci 32, I-20133 Milano, Italy
{baresl,guinea,mplebani}@elet.polimi.it

Abstract. This paper studies dependability in the context of service-based business processes, and proposes a dynamic technique for ensuring dependability requirements are met. On one hand, business processes are modeled using BPMN, which provides stakeholders with a suitable level of abstraction. On the other, we provide Dynamo, a run-time business process supervision framework that guarantees the dependability requirements are satisfied. Supervision rules let the user customize how the system deals with business-related situations that might hamper the dependability of the application. The main features of the proposed infrastructure are demonstrated on a simple case study in the domain of banking services.

1 Introduction

Dependability is a key concept in the actual use of software systems, however it may be fuzzy to some. Therefore, we will start by giving a classical definition of what is intended by dependability, which also serve as a way to frame our research.

As placed by Laprie et al. in [1], dependability is “the ability to deliver a service that can justifiably be trusted”. They also go on to say that a service is its behavior as it is perceived by its users, be them other systems or people. In their definition, they stress that dependability is a three-way problem. In fact, to be able to understand and ensure dependability, we must study: (a) the threats that can lead us to a situation in which the system is not dependable, (b) the attributes we want to maintain in the system, and (c) the means we can adopt to ensure overall dependability. It is not our interest to give a thorough presentation of what dependability means; however, in order to position our work we shall very briefly go over the three main concepts.

Typical threats can be either faults, failures or errors. The principal attributes encompassed by dependability, and that we want to maintain, are: availability (i.e., the readiness for correct service), reliability (i.e., continuity of correct service), safety (i.e., absence of catastrophic events on the user or the environment), confidentiality (i.e., no unauthorized disclosure of information), integrity (i.e., no improper state alterations), and maintainability (i.e., ability to undergo repairs).

Finally, the means we can adopt to ensure these properties can be subdivide into: fault prevention, fault tolerance, fault removal, and fault forecasting.

In our work, we tackle the dependability of composite Web services defined using BPEL (Business Process Execution Language). With respect to the very simple presentation we have given of threats, properties, and means involved in dependability, we can position our research as follows.

First of all, in our business processes we use external services that are not under our jurisdiction. This means that we cannot ensure that, during the system's life-cycle, they will not change in quality and/or functionality, for better or for worse. As we shall see, this has direct consequences on the nature of the threats we are exposed to. Failures, in fact, can be related to functional and/or non-functional properties that were not ensured by the parties involved in the distributed system.

Second, in our research we concentrate on providing a general and flexible means to ensure safety properties. More properly, since our context is that of business processes, we will go one step further and say that we ensure "business-related safety" properties. Ultimately, what we provide is a means that designers can use to prevent catastrophic events (from a business point of view) for their users.

However, it is very common for such systems to have an overwhelming number of stakeholders, which leads to the existence of many different perceptions of the offered service. For example, if we think of a web-based bookstore, an experienced user might be confident with using the system to buy his goods. In contrast, an unexperienced user might be much more cautious and feel the need to check every detail several times before placing an order. Even though the two users interact with the same business process, the perception they have of the service is different. In the first case, the user trusts the technology (and the brand) he utilizes, but in the second case, the user might have preferred more controls and more user-oriented warnings. This leads us directly to our next consideration.

We believe that, in very much the same way, the extremely high number of stakeholders demands that we consider the existence of diverse dependability requirements. Our working hypothesis, therefore, is that if a given business process is released for different (classes of) users, which will use the system at different times and within different contexts, we cannot conceive a single set of safety properties. This imposes that the application that is run must come from the intertwining of the actual business process and diverse sets of dependability constraints.

These concepts integrate well with Dynamo [23,25,24], our prototype framework for the run-time monitoring and recovery of BPEL processes [3]. Dynamo is conceived to:

- support for *separation of concerns*. We want process designers to be able to concentrate on one aspect of their system at a time. They must be able to focus on defining the business logic without having to necessarily think about supervision and dependability. These aspects should be designed later on, and automatically intertwined with the process to create a supervised version of the application.

- provide a *timely reaction* to anomalous situations. Our approach wants to be a solution capable of discovering anomalies as soon as they occur, and trying to fix them immediately. Dynamo supports an ECA (Event - Condition - Action) approach. Given a supervision rule, the triggering event is the execution of the *invoke* activity the rule is associated with. The condition is the monitoring constraint expressed in the form of pre- and post-condition on the external invocation. The action is the strategy that must be applied when the constraint is violated.
- offer a *high degree of customizability*. The degree of invasiveness of the supervision activities must be modifiable. The use of pre- and post-conditions in a process requires a certain degree of invasiveness. The process provider might want to be able to switch on and off certain checks in order to favor performance over everything else, thus we need dynamic monitoring and the degree of monitoring must be a direct consequence of how monitoring has been “doing” up until then. If things are going well, we might want to switch some of the upcoming checks off, or, on the contrary, if things are going wrong we might feel the need to notch the supervision up a bit.
- embed a *high degree of adoptability*. To foster the adoptability of our approach a requirement was to use as much pre-existing technologies as possible. We did not want to try to re-invent what is already standard or widely accepted. This is one of the main reasons for using BPEL as our service composition technology.

The paper introduces the concept of supervision rule as the union of user-defined constraints, to monitor how the BPEL process evolves, and reaction strategies, to specify corrective actions that must be executed when set constraints are violated. The former defines the triggering conditions that make the system apply the latter. As soon as a constraint is violated, the associated reaction warns the system manager (and the user if needed) and tries to keep the system on track by adopting corrective actions.

Our main requirement is to provide a framework in which dependability is ensured, in accordance with the user’s “perception” of the system.

Since stakeholders are our key players, we also want to provide them with suitable languages and tools to abstract from the underlying technologies and hide how the system guarantees the dependability. To this end, we adopt BPMN (Business Process Modeling Notation, [5]) as high-level notation to let users design business processes. Dynamo usually works by annotating standard BPEL processes with supervision directives. In this paper, however we propose to extend our business models with a high-level means of modeling dependability aspects. Therefore, we introduce high-level versions of the languages supplied by Dynamo: WSCoL (Web Service Constraint Language,[25]) for the monitoring directives and WSRS (Web Service Reaction Strategies, [38]) for the reaction strategies. A semi-automatic process is then in charge of transforming the high-level artifacts into the annotated BPEL processes required by Dynamo.

The rest of the paper is organized as follows. Section 2 introduces the case studies used in the paper and arguments dependability. Section 3 presents

Dynamo, while Section 4 introduces our prototype tools. Section 5 surveys some related approaches and Section 6 concludes the paper.

2 Dependability: An Example

In order to better clarify the motivations behind our proposal, we will introduce a simple case study. It will also come in handy, as a running example, when we will analyze our approach in greater depth in the upcoming sections.

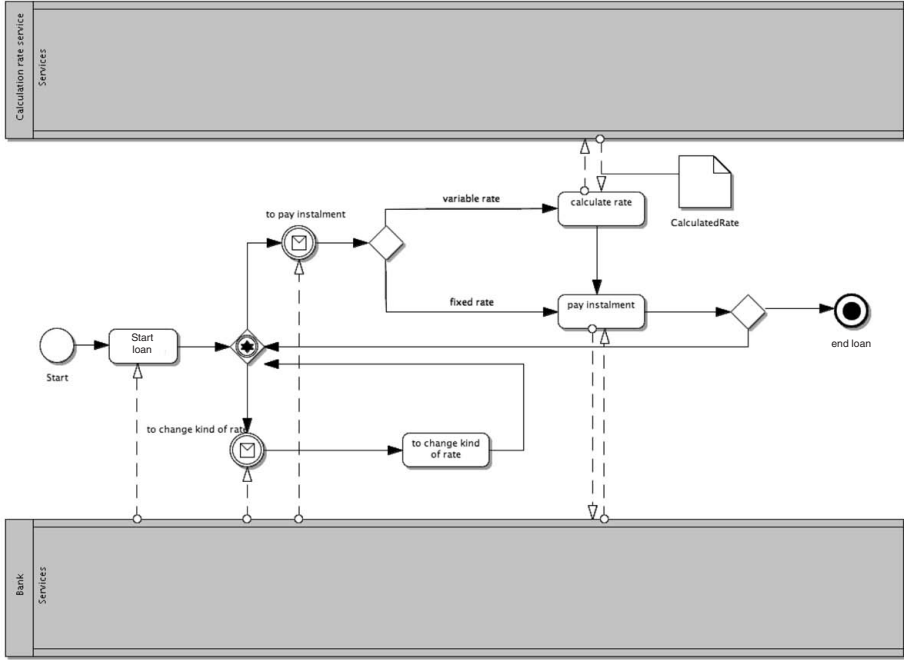


Fig. 1. The loan management example

The example takes place in the field of banking services. In particular, *Flex-iBank* is interested in providing flexible loans to their customers. These loans are managed by the bank so that each customer must pay back a certain quota once a month. The amount paid by the customer, and the date in which the payment is due, are defined in accordance with specific customer data. The bank allows their clients to choose between two different kinds of interest rates, which will be used to determine their monthly pays. The first is a static interest rate, which is defined once and for all during the contractual phase when the loan is discussed between the client and a bank officer. The second is a variable rate. This interest rate is determined as a function of the ECB's (European Central Bank) interest rate. This can be advantageous for the client, but could also prove to be a bad

choice if the ECB rate rises too much. Therefore, the bank also allows the client to change the kind of interest rate being used once every five years. If the loan is shorter than five years, then the kind of interest can only be changed once. If the clients should ever want to change it again after it has already been changed during such a period of time, then the loan payment will need to be stopped and re-negotiated.

FlexiBank has decided to manage these loans through the use of long-running and stateful BPEL processes. Therefore, their starting point was to model their process using BPMN. This modeling notation was chosen since it provides the level of abstraction needed to allow the business expert to collaborate in its definition. An example of BPMN is given in Figure 1, in which it is used to define our example. A new instance of the process is created each time a client signs a contract with the bank. The internal state of the process contains the total amount of the loan, the number of scheduled payments, the kind of interest rate chosen, and the value of the interest rate currently being used. The process continuously cycles between client payments. It distinguishes between two possible sequences of activities for managing each payment. The first consists in simply paying, taking into account the value of the static interest rate stored within the process. The second, on the other hand, interacts with an external service to obtain the ECB's interest rate, prior to issuing the payment. The process also allows to change the kind of interest rate being used between payments. More subtle business rules, such as the frequency in which a client can issue a change in interest rate type, are not shown in the figure as to keep it as simple as possible for the business expert.

At this point, the model needs to be completed with notions of dependability. We propose to use special purpose supervision rules that state the safety properties we want to ensure, as well as appropriate reaction strategies (or corrective measures) we want to attempt if anomalies are discovered at run-time. In the following we will refer to these rules as supervision rules. Together with our supervision-aware run-time framework called Dynamo, they allow us to guarantee dependability. We will now look at an example of such a rule by stating it informally. How to express the same rule in a more rigorous fashion will be treated later.

An example could be: “If the process is using a variable interest rate and its value approaches 3.0% (let us say more than 2.5%) then we should let the end-user know via e-mail. If the value surpasses 3.0% and the rate type has not already been changed once in the last five years then we should change the rate type to *fixed* to favor our clients. If the value surpasses 3.0% and the rate type has already been changed, then we should notify the end-user immediately as well as the process management team at the bank, so they can start re-negotiating the loan payment”.

While remaining at the same level of abstraction as BPMN, these rules can be stated in a high-level version of our WSCoL and WSRS languages, which represent the core matter of upcoming sections. The abstraction we provide is interested in stating two things: (1) properties that must hold for the data objects

introduced in our business model, and (2) what actions should be attempted when these properties are unsatisfied at run-time. The following models the dependability aspects we want to introduce.

Monitoring expression:

```
CalculatedRate.get(rate) <= 2.5
```

Recovery strategies:

```
if CalculatedRate.get(rate) >= 3.0 &&
    ConfigParams.get("lastChange") - now >= 5Y
then notify(client_email, mess) &&
    Call changeRateType with fixedRate at process
else if CalculatedRate.get(rate) >= 3.0 &&
    ConfigParams.get("lastChange") - now < 5Y
then notify(client_email, mess1) &&
    notify(bank_email, mess2)
else
    notify(client_email, mess)
```

It can be disputed, however, that the above rules, although adopting the correct level of abstraction, are still too hard to write for a business expert. We make this dispute our own, and advocate that the same is also true for business models in general. In fact, the correctness and soundness of a BPMN business model is the direct consequence of a collaboration between the business expert and a process designer. The same is true for the modeling of the dependability requirements. This is easy to see if we look at the impact a certain recovery strategy might have on the business; the extent of the impact can only be known by a business expert and therefore his/her input is necessary.

The high level of abstraction used in defining these supervision rules represents a valid middle ground between a natural language description (given by a business expert), and a thorough specification given in WSCoL (the language our run-time framework understands). The gaps between BPMN and BPEL, and between the so-designed supervision rules and their run-time counterparts, are, as we shall see, managed by means of proper semi-automatic translations.

Dynamo was born as a tool for fostering the joint use of run-time monitoring, to identify the triggering conditions, and reaction strategies to make a BPEL system behave accordingly. The approach is quite invasive: extra time and resources are spent at run-time to verify, step-by-step, if defined pre- and post-conditions hold. The payoff is that any situation we might perceive as problematic is discovered immediately, giving us a chance to do something about it.

3 Our Approach

Our approach ensures dependability capabilities to BPEL processes through the execution of three main activities: *data collection*, *data analysis*, and *recovery*. All three activities contribute significantly to the effectiveness of the framework.

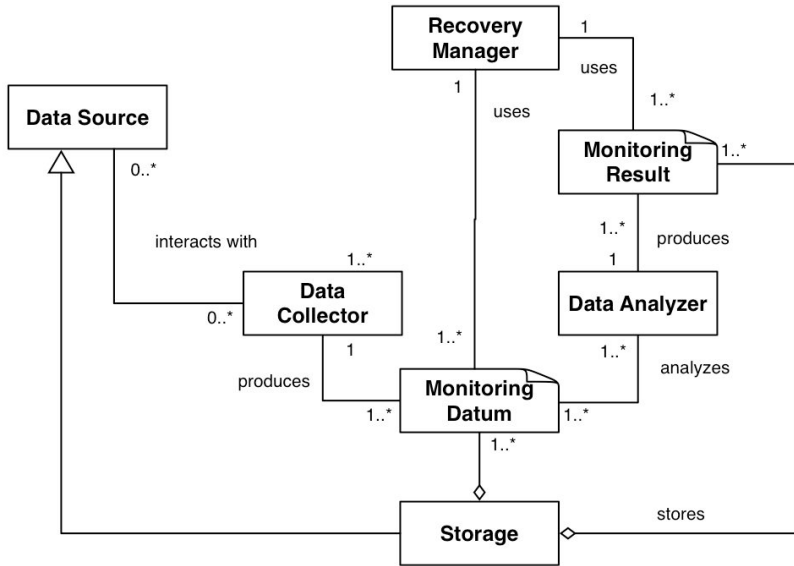


Fig. 2. Conceptual model for the supervision framework

Figure 2 illustrates how the three main activities are tight together to provide a single coherent framework. Conceptually there are five main component types:

- **Data Collectors** are responsible for the monitoring data required by the framework to monitor the functional and/or non-functional properties of the system. This can consist in simply obtaining the data from appropriate probes, or in manipulating them for analysis.
- **Data Sources** are probes that can be called by the data collectors when supervision is being achieved. The most typical data source in Dynamo is the business process itself. Most of the time, in fact, the end-user will be interested in asserting properties for the data the process sends to, and receives from, a remote service. These data represent a subset of the process' internal state.

However, at times, the data we can obtain from the process is simply not enough. Conceptually, in fact, although the data pertaining to the business logic and the monitoring data may overlap, they are not necessarily the same. Let us imagine, for example, we need to check the amount of money involved in a bank transaction and that the process only knows the amount in US dollars. If the property is defined in euros, and the correct amount does not only depend on the exchange rate between dollars and euros at the exact moment of the analysis (a factor the process does not internally know), but also on the price of a certain stock at the NASDAQ stock exchange. The exchange rate and the stock price need to be collected from data sources that are *external* to the process. In our approach we allow

data to be collected from any source. The only limitations are imposed by the technological aspects of how this is achieved.

- The **Storage** component is a component that can be used to persistently store monitoring data (and/or monitoring results —as we can see in Figure 2). It is, to all means, a special case of data source. Its particularity is that it does not give you data pertaining to the current activation of the framework, but data that pertained to a previous activation. This means that it can provide data from a previous process execution, or from a previous step in the same process. This allows us to confront how well the process is executing with respect to previous instances, and to define, for example, the margin of fluctuation of a given response time we are willing to accept, and to react if a service seems to be behaving “differently” to how it has in the past.
- **Data Analyzers** verify whether properties hold by analyzing the monitoring data received from data collectors. In the conceptual model we allow the framework to possess more than one data analyzer. It might be convenient, in fact, to have special purpose analyzers for certain monitoring activities, and for certain kinds of monitoring data. When the actual analysis has been completed, the analyzer produces a monitoring result which can cause recovery to be activated, and can be stored for future analysis should it be needed.
- The **Recovery Manager** is the component responsible for activating the strategies that will try to “keep the process on track” when an anomaly is detected. Access to the monitoring results and monitoring data allows the component to determine the degree to which a certain property has been infringed, and to choose the most appropriate recovery. As we shall see in the upcoming sections, there is a wide range of activities the stakeholder can choose from when defining his/her recovery strategies. In the conceptual model, recovery cannot require any extra data. This was chosen to preserve a clear separation between the recovery itself and the “reasons” for which recovery is being performed.

3.1 Supervision Rules

Figure 3 illustrates the conceptual model behind the definition of *Supervision Rules*. A rule is made up of a **Monitoring Expression** and a set of **Reaction Strategies**. There are also two further elements that help maintain and apply rules: a **Location**, which is an XPath expression to indicate the point of the process for which we a supervision rule, and a set of optional **Parameters**, which are meta-level information used at run-time to decide whether the supervision rule should be taken into account or not. In the case no parameters have been defined the monitoring property is evaluated by default.

Currently, supported parameters are *priority*, *validity*, and *trusted providers*, but other attributes can be added easily. When the designer decides to attach a *priority* to a rule, he uses a simple notion of “importance”. When the process in execution has reached a point for which a supervision rule is defined, we compare the priority associated with the supervision rule with a process threshold value.

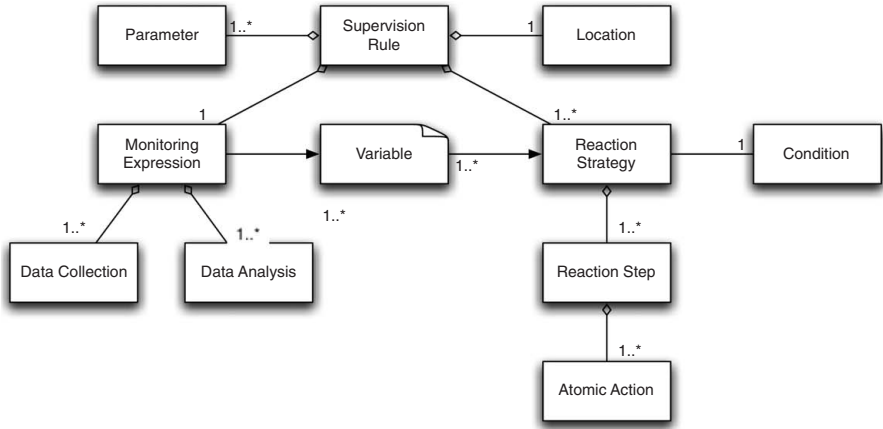


Fig. 3. Conceptual model of a supervision rule

The rule is taken into account only if its priority is less than or equal to the threshold value. The *validity* parameter defines a time window. If the process is run within this window, supervision is performed. On the other hand, every time the process is run outside of this window the specific rule is ignored. A validity parameter is defined using three optional values: *from*, *to*, and *every*. This means that the time window may have both a lower and an upper time bound and it can be repeated with a user-defined frequency. *Trusted providers* are service providers for which supervision is not necessary. This is useful because, in abstract process definitions, the actual service to which the process binds to could be chosen at deployment-time or at run-time. If no trusted providers are defined the supervision rule is always checked.

3.2 Monitoring Expressions

WScOL is the language we propose for defining monitoring expressions and thus it allows the designer to define **Data Collection** and **Data Analysis**. The language was inspired by the lightweight version of JML [27] [28] (Java Modeling Language), a language designed for specifying behavioral interfaces in Java programs, which combines design by contract [26] and interface definition languages such as Larch [18]. At its core WScOL is a first-order logic that allows the designer to:

- define and predicate on variables containing data originating both within the process and outside of the process, and to retrieve data previously stored in a storage component. It is also allows designers to define variable aliases to simplify their WScOL expressions and recovery strategies.
- use pre-defined variable functions for data manipulation, depending on the variable's *dataType*, e.g. string concatenation.

- use the typical boolean operators such as `&&` (and), `||` (or), `!` (not), `=>` (implies), and `<=>` (if and only if), the typical relational operators, such as `<`, `>`, `==`, `<=`, and `>=`, and the typical mathematical operators such as `+`, `-`, `*`, `/`, and `%`.
- predicate on sets of variables through the use of the universal and the existential quantifiers, and of special language constructs such as *min*, *max*, *sum*, *product*, and *avg*.

WSCoL supports data collection through the concept of **Variables**, which play a pivotal role in the definition of monitoring properties. In fact, a WSCoL monitoring property simply states relationships that must hold between variables. While defining these variables, the designer implicitly defines the data collection needed to obtain their values. Depending on the nature of the data collection being performed, WSCoL supports three possible kinds of variables:

- A WSCoL *internal* variable corresponds to a monitoring datum that originates within the process in execution. In WSCoL internal variables can only contain simple XSD typed values. These variables should not be confused with BPEL variables, which on the contrary are meant to match and hold the complex XSD types defined in the WSDLs of the services with which the process interacts. Since we only accept simple XSD types as valid WSCoL variable contents, when we define a WSCoL internal variable, what we are actually doing is to define some *data extraction* from a complex BPEL variable.
- A WSCoL *external* variable indicates a monitoring datum that originates outside of the process in execution. This is useful when the correctness of a service invocation can only be established by referring, for example, to contextual data such as the time and/or place of execution, or the ID of the end-user. External variables can also represent a way to achieve data transformation on WSCoL internal, external, and historical variables.

WSCoL provides a special purpose mechanism for retrieving external variables if the data collector we want to use supplies a WSDL interface. `return<X>()`, where `X` indicates the XSD type of the return value, allows us to query any external data source.

- a WSCoL *historical* variable consists of a monitoring datum that is related to previous process executions, or to previous activations of the Dynamo framework on the same process in execution. WSCoL allows designers to store variables —into a persistent storage component— contextually to the analysis of a pre- or post-condition. A historical variable is therefore implicitly tied to the process definition and to the process instance it belonged to, as well as to the pre- or post-condition in which the storage operation was performed. The syntax for the storage operation (**store**) also creates an alias as a side effect.

WSCoL also supports aliasing (**let** construct). Aliasing during data collection allows us to refer to a certain WSCoL variable with another name. This has two main advantages. The first is that it greatly simplifies the specification of data analysis and of recovery strategies, allowing for less verbose expressions. The

second is that it allows the specific variable to be collected only once, but to be referenced any number of times in the analysis or recovery specifications. This is particularly important when we have to deal with external variables that may return different values depending on the exact moment in which they were collected. Moreover, it is possible to append data extraction (under the form of XPATH expressions) to aliases to further refine their contained values.

Returning to the example illustrated in Figure 1, we are now capable of specifying the monitoring expression Dynamo will need to check at run-time in order to guarantee dependability. The monitoring expression is defined as a post-condition to the call to the external service responsible for calculating the variable interest rate. Figure 4 uses a tree-like representation to illustrate the XML structure of the BPEL variable that contains the remote service's response. This BPEL variable is the concretization of the abstract data object shown in Figure 1 with the name "CalculatedRate". Therefore, the WSCoL monitoring expression can be defined as:

```
($rateOut/parameters/rate) <=2.5;
```

The expression states "when" the returned message can be considered correct. In practice, it states that the BPEL internal variable called `rateOut` must contain a rate value which is less than or equal to 2.5%.

3.3 Recovery Strategies

WSRS (Web Service Reaction Strategies, [38]) allows the designer to associate a set of Reaction Strategies to the same Monitoring Expression. The idea of having more than one strategy allows the designer to differentiate the actual strategy depending on the extent to which a monitoring rule was not honored.

Recovery strategies are made up of **Strategy Steps**, which in turn are made up of **Atomic Steps**. In presenting our solution, we will start by explaining the

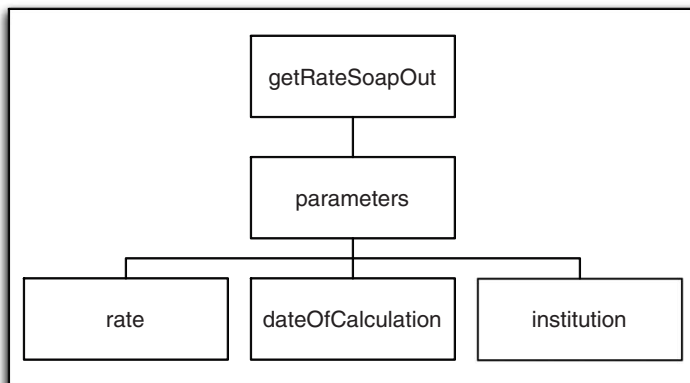


Fig. 4. Structure of the message containing the calculated rate

atomic recovery actions, and then proceed to present how they can be mixed and matched to create complex strategies, and how the framework can choose between one or the other.

We have defined ten different possible “atomic” recovery actions that can be performed when anomalous situations arise. Keep in mind that these actions are always performed immediately after a pre- or a post-condition has been evaluated, and therefore while the process is momentarily blocked. This is due to the fact that our approach is intrinsically synchronous, and that once the supervision framework is activated the process must wait before proceeding.

Moreover, all of these actions have instance validity, and do not—in any way—alter the process definition itself. The syntax and the semantics of these actions are:

- *ignore* simply ignores the fact that an anomaly has arisen.
- *notify(message, email)* takes two parameters: the former is a notification message and the second being the email address to which it should be sent.
- *halt* takes no parameter and stops the process in execution.
- *retry(times)* declares that, in the wake of an anomalous situation, the system should continue to retry to invoke the Web service up to a certain number of *times*. This recovery action should only be used in conjunction with post-conditions. Once a service has been reinvoked the post-condition needs be re-evaluated to see if the recovery was successful.
- *rebind(new_service_URI)* indicates that the service being invoked is to be substituted with another service that implements the same WSDL interface, and whose URI is passed as a parameter. Once the dynamic rebinding has been achieved, the new service is invoked and the monitoring step is reiterated. In our approach, the effects of the rebind action remain valid for every other call to the same service in the process.
- *change_monitoring_rules(analysis, recovery)* allows the designer to modify how monitoring is being achieved and therefore to relax (or reinforce) the constraints. The only parameter that is compulsory is **analysis**. It replaces the old monitoring analysis specifications with new ones. If necessary, the designer can also modify data collection accordingly by adding new variables (internal, external, or historical) or by creating new aliases. The definition of new recovery strategies is also optional. Once the modifications have been taken out, the monitoring activities are re-performed.
- *change_monitoring_params(params)* modifies the monitoring parameters [25] associated with the monitoring activities being considered. This allows the designer to modify the monitoring activity’s priority level, redefine the list of trusted providers by adding (or removing) providers, or change the time-frames within which the monitoring activity is to be performed. All these actions may result in the monitoring activity being “switched-off”. Once these modifications have been taken out, the parameters are reconsidered to decide whether the monitoring activities should be re-performed.
- *change_process_params(params)* modifies the global parameters associated with the process instance in execution. This allows the designer to dynamically activate or de-activate sets of monitoring rules, based on the actual

way services are performing during the execution. For example, as we have already mentioned, monitoring rules can be grouped together by giving them the same *priority* level. Therefore, using different levels a designer can create up to five monitoring rule sets. So if he/she decides to modify the overall process priority in the wake of a certain anomaly, this can have the effect of activating monitoring rules for certain BPEL activities further down the line.

- *call(wsdl, operation, ins, sendback, xslt)* consists of a call to an external Web service. The first two parameters identify the service (through a WSDL URI) and the name of the operation that should be called. The third parameter (*ins*) represents the data that are to be sent to the service. *sendback* is a boolean value (to be used exclusively in post-conditions) which states whether the service being called returns a message type which is compatible with that of the service for which we are verifying the post-condition, and, therefore, whether it should substitute it. If the *sendback* value is false, the designer can use the optional *xslt* parameter to define a transformation capable of mapping the values returned from the called service onto the soap body expected by the process.

The external service being called obviously does not share the same data space as the process. However, this recovery action is still useful since it gives designers great flexibility. A designer can statically devise a new external BPEL process containing quite complex logic. For example, he/she might define a new process that uses run-time information (coming both from the process and from the monitoring activities) to provide an augmented rebinding where UDDI registries are queried for appropriate candidates.

- *process_callback(event_name, params)* is, potentially, the most disruptive of the nine recovery actions, since it allows direct access to a process' internal state. This action allows complex recovery logic to be directly embedded into the executing process by means of a specific event handler (*event_name*), which is called by the recovery subsystem when the anomaly arises.

A BPEL event handler is an inner-process that is made public through the business process' WSDL interface. When we send it an event, the handler executes in an independent thread from the main business logic, which in the meanwhile continues to remain synchronously blocked, since it is waiting for an answer from the monitoring subsystem. The advantage is that the event handler and the main process share the same state. Once the event handler thread completes, the monitoring subsystem is warned to unblock the main business process.

The disadvantage is that event handlers will have to be statically embedded into the process prior to deployment. This means that the recovery logic is defined once and for all, and that it can be personalized—in order to take into account contextual information such as who is executing the process, and how and from where—only through the parameterization of the event handler itself. *Params* contains such data (under the form of internal, external or historical variables). Moreover, when the event handler is created, the

designer also has to provide a proper correlation set that can be shared with the monitoring subsystem to ensure correct interactions.

Different strategies are associated with **Conditions**, which are heavily based on aliases. This allows us to be less verbose, and conditions to be evaluated only once. **Reaction Strategies** define what the recovery subsystem should do to attempt to keep things on track. We define complex strategies by combining **Strategy Steps**, which are in turn conjunctions of atomic recovery actions. Strategy steps are separated by the traditional “or” symbol (`||`), and the order in which they are specified once again is important. If we write *strategy-step-1 || strategy-step-2 || ...* we intend the recovery subsystem should first apply step 1 and, if that is not effective, then try step 2, and so on. If no step is successful, the process provider is notified and the process is halted. On the other hand, a single strategy step is a conjunction (`&&`) of atomic recovery actions. Once again, the order in which the atomic actions are specified is relevant, since it also defines their order of execution.

How to know if a single strategy step is successful or not depends on the actions it contains. We can distinguish between two groups of recovery actions: those that require monitoring to be re-evaluated at the completion of the strategy step, and those that do not. In particular, *retry*, *rebind*, *change_monitoring_rules*, *change_monitoring_params*, *change_process_params*, and *call* all need monitoring to be re-evaluated. This re-evaluation tells us whether the strategy step was successful. Notice that every time monitoring is re-evaluated, all pertinent aliases are updated, since it means re-performing both data collection and data analysis). If the recovery step is unsuccessful, then the overall state is restored, as if the strategy step had not been performed. If a strategy step does not contain activities that require monitoring to be re-evaluated, the step is automatically considered successful, unless the strategy step could not be performed for some reason (e.g. a necessary external service is unreachable).

When defining complex recovery strategies, a designer must keep in mind that it is not always possible to aggregate steps arbitrarily. In general, strategy steps that do not contain actions that require monitoring to be re-considered should only be used as last steps. For example, steps containing only *halts*, *ignores*, or *notifies* are always considered successful, and therefore always invalidate any following steps. This is exactly the case of the above example, in which step number four can always be considered successful. *change_monitoring_rules* can also invalidate following steps by replacing the recovery strategies that are going to be used thereon. Once again, such issues are left up to the designer. Figure 5 illustrates the rules for the coexistence of atomic actions within a same strategy step.

Returning to our running example, the client stated three possible recovery strategies. The first asked the system to notify him/her when the variable rate reached 2.5%. The second was to change the rate type to “static” should the variable rate grow beyond 3.0%. But this strategy is only feasible when the rate has not been changed in the last five years. Therefore, another strategy is needed. In that case the system should just notify the client, and tell the bank that they are going to have to re-negotiate the payment.

	ignore	notify	halt	retry	rebind	ch_mon_r	ch_mon_p	ch_proc_p	call	p_callback
ignore		Y	N	N	N	N	N	N	Y	Y
notify			Y	Y	Y	Y	Y	Y	Y	Y
halt				N	N	N	N	N	Y	N
retry					N	Y	Y	Y	Y	Y
rebind						Y	Y	Y	Y	Y
ch_mon_r							Y	Y	Y	Y
ch_mon_p								Y	Y	Y
ch_proc_p									Y	Y
call										Y
p_callback										

Fig. 5. Rules for combining atomic actions within single strategy steps, where **ch_mon_r** stands for `change_monitoring_rules`, **ch_mon_p** for `change_monitoring_params`, **ch_process_p** for `change_process_params`, and **p_callback** for `process_callback`

Such recovery strategies can be defined in WSRS in the following manner:

```

let $vRate = ($rateOut/parameters/rate);
let $lastChange = ($configParams/lastChange);
let $daysPassed = (returnInt(WSDL, timePassed, '<input><date>' +
    $lastChange + '</date></input>', /timePassed/days));

if ($vRate>2.5 && $vRate<3)
    notify(client_email, mess);
else if ($vRate>=3 && $daysPassed >= 1825)
    notify(client_email, mess) && call(processWSDL,
        changeRateType, inputs, false, null);
else if ($vRate>=3 && $daysPassed < 1825)
    notify(client_email, mess1) && notify(bank_email, mess2);

```

Three aliases are defined to simplify the definition of the strategies. The first contains the value of the variable interest rate. The second contains the date of the last time the rate was changed. In order to simplify both the process and the definition of the recovery, this value defaults to exactly five years prior to the instantiation of the process. The third, on the other hand, uses a WSCoL external variable to discover how long it has been since the last time the rate was changed. It calls a special utility service which provides functions for treating dates and times. For the sake of clarity, the actual atomic recovery strategies are defined using simplified versions of their real counterparts.

4 Supporting Tools

The solution to dependability introduced in the previous sections, and shown in Figure 6, requires two different classes of supporting tools. First of all, we need to conceive a viable solution to the problem of mapping BPMN models

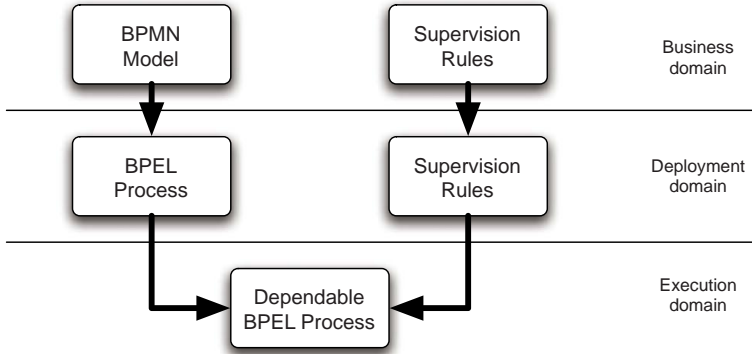


Fig. 6. Elements and transformations for *dependability*

onto executable BPEL processes. Then, we can think of exploiting the last version of *Dynamo* to supply the run-time checks required to enforce dependability. The former problem can easily be solved by exploiting already available tools and integrating them into the proposed framework. The latter problem requires that the terms used in the *supervision rules* be suitably translated into their WSCoL and WSRS equivalents. Moreover, the solution we propose treats business and supervision as separated concerns and thus we need to emphasize how *Dynamo* exploits aspect-oriented programming [22] to support the independent deployment of a business process and its supervision logic.

4.1 BPMN vs. BPEL

One of the pivotal moments in allowing stakeholders to design their models with BPMN and to obtain *dependability* by means of *Dynamo* is the translation that must occur between BPMN models and BPEL processes. This is a well-known problem in the area of business process modeling and BPMN has demonstrated to be a valid abstraction of business processes [35]. The BPMN specification includes BPD (Business Process Diagram [40]), which is a graphical and human-readable description of a business process. The execution of these processes are then guaranteed by a more technical standard, such as BPEL.

In the last few years, there have been several initiatives aimed at mapping BPMN onto BPEL [6]. This means that we do not need to develop a completely new solution, but we can exploit existing ones, and complement them with our tools. The selection of the right tool was guided by two factors. First, it had to offer the capability of augmenting BPMN models with information regarding the external services chosen for interaction. Second, it had to support us in our definition of *supervision rules*. Moreover, the results of the BPMN to BPEL transformation had to be “standard” and “visible” to ease the integration with *Dynamo*.

In the end, we selected *Borland Together* [2]. We envision the tool being used in a two-step process. First, the business process is designed through the collaboration between the business expert and the process designer. Second, the process is augmented by the process designer, in order to make it runnable.

The business expert starts by defining who participates in the process, its main tasks, and the message flow between the BPMN model and its participants. During the definition of the business process, the business and the designer experts can identify three different kinds of participants: the process itself, its clients, and the external services with which to collaborate (which must be identified through their WSDL interfaces). Furthermore, there are three kinds of tasks: receive tasks, which are only able to receive messages from participants, reply tasks, which send messages to participants, and service tasks, which create a conversation between the process and an external Web service.

At this point of the design process, the inherent structures of the exchanged messages are not important. A high-level abstraction in which only the names of the messages, and the names of the messages' sub-parts are enough. As soon as these are defined, the tool can check whether the process is well-formed with respect to BPMN's constructs. These information are what the end-user then uses as the starting point for defining high-level supervision rules.

Such a specification, however, is not enough to guarantee dependability, since messages are only names, while our approach requires that the supervision rules relate to real BPEL messages which are mapped to complex XML schema definitions. This is only possible after the process designer complements the BPMN process with the real service endpoints, and the real structure of the *Data Objects*. All the required information can be retrieved automatically from the WSDL interfaces of the external Web services.

Moreover, the BPMN-BPEL translation also imposes the definition of: (a) the conditions associated with automatically generated BPEL branch statements (like **switch** and **while** elements) and (b) the role played by each participant.

The filling of these gaps is the responsibility of the process designer. Using Borland's tool, the designer translates the high-level conditions into BPEL specific language conditions; e.g. the switch condition simply referred to as variable rate in Figure 1 must be translated into

```
bpws:getVariableData(loan,currentRateType)=variable.
```

In other words, these last definitions narrow the gap between the source BPMN model and the target BPEL process. A second validation step then checks whether all the relevant information has been defined and if the tool can proceed to generate the actual BPEL code.

This activity creates a set of files: the set of wrapper WSDL interfaces used locally to refer to the external Web services, the BPEL process, the WSDL interface of the process itself, and all the files needed to deploy the process into the *ActiveBPEL* [16] execution engine, which is the standard BPEL engine supported by Together.

At the end of the transformation the designer is left with a BPD diagram augmented with the structure of the data exchanged between the process and

its partners, and a BPEL process that is agnostic with respect to all possible supervision rules, and that can be deployed once and for all to the our Dynamo framework.

The BPD diagram represents the starting point for the definition of the high-level supervision rules. Before going to the end-client the process designer and the business expert must once again collaborate to identify the subset of the service invocations for which the user can define rules, and the set of data objects he/she can refer to during the specification.

Referring to the example provided in Section 2 and illustrated in Figure 1, we provided the stakeholder with the BPD diagram, the *CalculatedRate* data object, and a data object containing a subset of the configuration parameters used by the business logic during its life-cycle. These two data objects contain the variable rate the end-client wants to check and the last time the rate type was changed, that is the data the end-user needed to define his/her rule.

The stakeholder first defines his/her dependability supervision rules using natural language, while the design expert annotates them in our high-level version of WSCoL. Once the rules have been agreed on, the designer can proceed to translate them into the real WS-CoL rules that will instruct our supervision framework.

4.2 Dynamo

Dynamo is based on AOP-like techniques. These techniques play a pivotal role in allowing us to keep the business logic separate from the definition of the supervision activities, and thus foster the concept of *dependability*. AOP allows us to envision the supervision activities as a cross-cutting concern that can be “weaved” directly into the process. It also guarantees that the business process is defined once and for all, regardless of the different supervision requirements imposed by different stakeholders.

We have decided to relegate all the supervision activities to special purpose components that exist outside of the execution engine. This gives us two main advantages. The first is that this allows us to distribute the supervision environment more freely, for example as a stand-alone service that any process developer can take advantage of. No knowledge of monitoring or recovery needs to be embedded into the execution environment, meaning that our solution can be used in conjunction with whatever BPEL execution engine the designer chooses to adopt. The second main advantage is that, thanks to this decision, we can manage the framework’s own configuration and setup more freely. For example, we can choose to change the data analyzer we are using on-the-fly, or we can use a different storage component, etc.

The main obstacle at this point becomes how to automatically instruct the business process to interact correctly with the external supervision framework. This is where the AOP-like techniques come in handy. They provide us with the “glue” we need to tie the process to Dynamo.

Figure 7 illustrates the solution we propose. Reading the figure from left to right it is possible to understand the transformation the process must undergo

in order to allow for run-time supervision. The automatically generated BPEL process only considers the business logic, without having to take into account any notion of supervision. The process, as we see it on the left-hand side of Figure 7, in fact, interacts with the partner services on the internet through BPEL `invoke` activities.

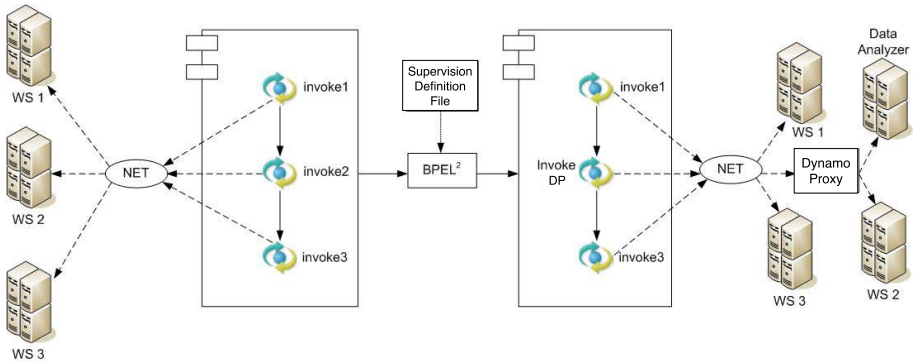


Fig. 7. An overview of our proxy based implementation

An external *supervision definition file* (once again see Figure 7) contains all the information needed to actually tie the process definition to the Dynamo framework. This file is defined conjunctively by the business expert and the design expert. In fact, it contains information regarding the subset of activities and data objects the end-clients can use in their supervision rules.

This file is used at deployment-time by a component called BPEL², responsible for producing the BPEL code that will “glue” the process to Dynamo. This code is responsible for “pushing” the run-time data needed to provide supervision towards the Dynamo framework. The code that is created is then automatically inserted into the process definition, creating a new version of the process capable of interacting correctly with Dynamo. This process can be seen on the right-hand side of Figure 7.

In this modified version of the process, every time there is a pre- or post-condition might need to be checked, the interaction the process would have had with the outside world is channeled through the *Dynamo Proxy*. This component is the main component of this prototype. It is responsible for managing external or historical data collection, data analysis and recovery. It is also responsible for invoking the original Web service (the one for which we are verifying pre- and/or post-conditions), since the original process no longer performs the invocation, and for returning its response to the process.

Performance. The loss in performance can be attributed to (a) the time it takes the execution to figure out if supervision is required, and (b) the amount of time it takes to actually perform monitoring and recovery. Evaluation has demonstrated that, in average, it takes the system less than 2ms to figure out if

supervision is required. These tests have been performed on an AMD Athlon(tm) XP 2600+ (1.93Ghz) with 512MB of RAM, running Windows XP. The loss due to the actual supervision, on the other hand, depends on the nature of the monitoring property and of the recovery definition. The reason for this is that it depends on how long it takes the external services we need invoke to complete, be them external variables in WSCoL properties, or various atomic action, such as retry, rebind, call, and callback.

5 Related Work

Many approaches concentrate on a more traditional definition of dependability, and often build upon results previously obtained for component-based systems. Candea et al. [17] propose a solution to enforce greater availability through the use of micro-reboots, in which single components are preemptively rebooted to assure a ready-state where upcoming failures should be less likely to occur. This requires a continuous monitoring of the different Web services, which might not be feasible when the various components are distributed or under diverse jurisdictions.

Hall et al [39] propose a solution to the reliability problem. They propose a container-based approach in which a transparent proxy places itself between the client and the Web service itself. The proxy can then implement a different number of fault-tolerance policies. One such policy could be to adopt replication, and then have the proxy use a certain voting scheme when it receives multiple answers.

Fault-tolerance is a highly perceived problem in the domain of grid computing as well. A common solution, present in systems such as Condor [4], is to use check-pointing. This involves periodically saving the state of an application, in order to be able to restart it promptly. This typically introduces high levels of performance overhead. Plank [21] proposes concurrent check-pointing as a way to avoid such overhead. A second problem arises in dynamic and heterogeneous systems, such as SOAs, where a restart can require the data to be sent to all the nodes in the system.

An approach which presents an abstraction-level more similar to that of our solution, and which also provides service execution monitoring, is Cremona (Creation and monitoring of WS-Agreements) [19]. WS-Agreement is a standardization effort of the Global Grid Forum [9] that defines an agreement protocol based on XML. This standard defines agreements for interfaces, security and quality of service properties. Cremona provides a framework that simplifies the definition, management, and run-time monitoring of the state of the agreements.

Spanoudakis and Mahbub [30] also propose a framework for monitoring requirements of WS-BPEL-based service compositions. Their approach uses event-calculus for specifying the requirements that must be monitored. Requirements can be behavioral properties of the coordination process or assumptions about the atomic or joint behavior of deployed services. The first can be extracted automatically from the BPEL specification of the process, while the latter must be

specified by the user. Events are then observed at run-time. They are stored in a database and the run-time checking is done by an algorithm based on integrity constraint checking in temporal deductive databases. Like in our approach, erroneous situations can be found only after they occur. However, it is much less responsive in discovering run-time erroneous situations and cannot be used to enforce dependability.

Exception handling in business workflows has also been considered outside the domain of service oriented computing; Casati et al. [12] provide an environment for designing reaction strategies in generic business workflows. Exceptions are specified using ECA rules that consider data, workflow, temporal and external events. Their approach is similar to ours since events are raised when a task is started or completed. However, their rules are less flexible; in fact, the definition of their rules is pattern-oriented; instead, in our work we do not force rigid schemas onto the designers. F. Daniel [15], on the other hand, proposes a portable approach to exception handling in workflow management systems. He proposes a means to enrich XPDL workflows with standard exception handling constructs, starting from a universal and high-level event-condition-action language. Through a rule compiler he then yields portable process and exception definitions in an automated way.

Rule engines are widely exploited in recovery management. DIOS++ [29] is a framework for rule-based autonomic adaptation and for controlling distributed sensor-monitored scientific applications. DIOS++ provides a distributed rule engine that adopts *if - then - else* client-defined rules like those described in this paper. RuleBAM [20] is an another framework that uses policies and rules; it uses Business Activity Management (BAM) policies to define the system requirements and to automatically produce executable business rules that implement recovery. These approaches however are not specifically tailored towards SOAs. They also do not provide specific mechanisms for complex strategies, but only atomic actions.

As far as recovery actions for SOAs are concerned, Pautasso et al. [34] provide an autonomic reconfiguration component for JOpera, a proprietary distributed service composition and deployment platform. It allows the system to automatically reconfigure its deployment strategy in the wake of QoS problems (e.g. excessive workload). Recovery strategies are chosen according to goals such as the minimization of resource allocation or response times. However, the use of proprietary languages and tools greatly reduces the possible diffusion of JOpera. Canfora et al. [11] propose a re-planning technique for QoS-based dynamic binding. Their goal is to trigger re-planning when the measured QoS violates an SLA or deviates greatly from estimations, and to obtain new service compositions that guarantee the needed functionality and the overall desired QoS. With respect to our approach, however, they do not cover functional properties.

Regarding the use of aspect oriented programming to weave cross-cutting concerns into BPEL processes, Courbis et al. [14] propose a proprietary solution based on their semantic analyzer toolkit called SmartTools. Using this tool they have produced a process engine that uses the visitor design pattern to traverse

abstract syntax trees (which represent their processes) in order to execute appropriate code for each language construct they encounter. Aspects are activated through redirection. With respect to our approach, this solution is closed and more research oriented. Charfi et al. [13], on the other hand, propose a slightly different approach in which AOP is used to produce a container-based middleware to be used in conjunction with a BPEL engine. This middleware intercepts the process calls to enforce policies such as security, persistence, and reliable messaging. With regards to our approach, such a solution is much more QoS oriented and less functional.

Much work has been accomplished in the field of dynamic composition of SOAs, introducing interesting techniques that can be used for process reorganization in the wake of anomalies. Mecella et al. compose services [10] starting from a finite transition system representation of the needed service. In a different way, Traverso et al. [36] [41] consider dynamic compositions a planning problem. Given a set of abstract BPEL descriptions, and a composition requirement, they generate an executable BPEL process by combining all the possible behaviors of each service into a parallel state transition system. This representation, together with the process requirement, is the input for a planner that extracts the executable BPEL process. However, both Mecella and Traverso limit client intervention to the definition of the set of services that can be used in the composition. Again our approach allows for greater flexibility and for personalized recovery.

Finally, we can also mention the work by Di Nitto et al. [31] and SH-BPEL (Self-Healing BPEL [33]). They both propose to extend BPEL by means of suitable external rules to increase its flexibility and its capability of dealing with anomalous situations. Both the approaches do not consider monitoring and dependability as first class citizens, but they are more interested in supporting the dynamic reconfiguration of BPEL processes. To this end, some of the solutions they propose could easily become new atomic actions for our reaction strategies, but the key motivations of our approach and these proposals are different.

A business process can also be made personal by using some well-known workflow patterns [8]: for example *exclusive choice*, *deferred choice* and *multiple instance*. These patterns help us customize a business process according to the dependability constraints of the different stakeholders. These patterns are used as starting point for enforcing dependability by means of pi-calculus. Puhlmann [37] proposes a formal method where each workflow activity is mapped onto a pi-calculus process with pre- and post-conditions (to constrain the evolution). Even if the solution is interesting for its degree of precision and rigor, it lacks flexibility, which hampers the idea of stakeholder-oriented dependability, and user-friendliness given the choice of pi-calculus as model notation.

6 Conclusions

The paper studies the concept of dependability in the context of business-oriented BPEL processes. It concentrates on the capability to tailor the amount

of checks that must be done while the process executes, as well as the capability to react as soon as a given dependability condition is violated. Dynamo oversees the execution of dependable processes by checking monitoring expressions and by reacting as soon as they are violated by means of the associated reaction strategies.

In this paper, the work is intentionally general. There are many other kinds of properties that reside under the dependability umbrella, such as confidentiality, integrity, etc. Not all of them can be easily guaranteed using our approach. However, since our work is intentionally general, in order to achieve versatility, sometimes it is simply a matter of finding ways to obtain the run-time information needed for verifying such properties. In fact, we have studied the use of our techniques, and in particular of WSCoL and Dynamo, in expressing and guaranteeing confidentiality requirements [42].

The main novel contributions of this paper is the adoption of BPMN as friendly notation for business process modeling, but this choice also leads to other novelties because BPMN models must be semi-automatically translated into executable BPEL process and supervision rules must be re-casted accordingly.

The paper describes our first experiments to ease the definition of supervision rules. Our base languages (WSCoL and WSRS) supply the necessary functionality, but they are not easy and straightforwardly usable by non experts. Besides proposing to raise the abstraction level as for modeled processes, we are also working on form-based languages and supervision patterns to support the definition of the rules themselves.

References

1. Avizienis, A., Laprie, J.-C., Randell, B., Landwehr, C.: Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing* 1(1), 11–33 (2004)
2. Borland Together technologies. <http://www.borland.com/us/products/together/index.html>
3. Business Process Execution Language for Web Services v1.1, <http://www-128.ibm.com/developerworks/library/specification/ws-bpel/>
4. Condor. <http://www.cs.wisc.edu/condor>
5. Business Process Modeling Notation Specification. <http://www.bpmn.org/>
6. Object Management Group (OMG) - Business Process Management Initiative. <http://www.bpmn.org/>
7. Wikipedia. <http://www.wikipedia.org/>
8. Workflow patterns. <http://is.tm.tue.nl/research/patterns/patterns.html>
9. WS-Agreement Structure. <http://www-unix.mcs.anl.gov/~keahey/Meetings/GRAAP/WSAgreement Structure.pdf>
10. Berardi, D., Calvanese, D., De Giacomo, G., Mecella, M.: Composition of services with nondeterministic observable behavior. In: Benatallah, B., Casati, F., Traverso, P. (eds.) *ICSOC 2005*. LNCS, vol. 3826, pp. 520–526. Springer, Heidelberg (2005)
11. Canfora, G., Di Penta, M., Esposito, R., Villani, N.L.: Qos-aware replanning of composite web services. In: *2005 IEEE International Conference on Web Services (ICWS 2005)*, pp. 121–129. IEEE Computer Society Press, Los Alamitos (2005)

12. Casati, F., Fugini, M.G., Mirbel, I.: An environment for designing exceptions in workflows. In: Pernici, B., Thanos, C. (eds.) CAiSE 1998. LNCS, vol. 1413, pp. 139–157. Springer, Heidelberg (1998)
13. Charfi, A., Mezini, M.: An aspect-based process container for bpm. In: AOMD '05: Proceedings of the 1st workshop on Aspect oriented middleware development, ACM Press, New York (2005)
14. Courbis, C., Finkelstein, A.: Towards aspect weaving applications. In: Roman, G.-C., Griswold, W.G., Nuseibeh, B. (eds.) ICSE, pp. 69–77. ACM Press, New York (2005)
15. Daniel, F.: A portable approach to exception handling in workflow management systems. Technical report, Politecnico di Milano - Dipartimento di Elettronica e Informazione (2006)
16. Active Endpoints. ActiveBPEL engine architecture. <http://www.activebpel.org/docs/architecture.html>
17. Candea, G., Kawamoto, S., Fujiki, Y., Friedman, G., Fox, A.: Microreboot - A Technique for Cheap Recovery. In: 6th Symposium on Operating System Design and Implementation (OSDI 2004), pp. 31–44 (2004)
18. Guttag, J.V., Horning, J.J., Garland, S.J., Jones, K.D., Modet, A., Wing, J.M.: Larch: languages and tools for formal specification. Springer, Heidelberg (1993)
19. Ludwig, H., Dan, A., Kearney, R.: Cremona: An Architecture and Library for Creation and Monitoring of WS-Agreements. In: Service-Oriented Computing - ICSOC 2004, Second International Conference, New York, USA, November 15-19, 2004, pp. 65–74 (2004)
20. Jeng, J.J., Flaxer, D., Kapoor, S.: RuleBAM: A rule-based framework for business activity management. In: IEEE SCC, pp. 262–270. IEEE Computer Society Press, Los Alamitos (2004)
21. Plank, J.S.: Efficient checkpointing on MIMD architectures. PhD thesis, Princeton, NJ, USA (1993)
22. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Videira Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In: Aksit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
23. Baresi, L., Ghezzi, C., Guinea, S.: Smart monitors for composed services. In: Service-Oriented Computing - ICSOC 2004, Second International Conference, New York, USA, November 15-19, 2004, pp. 193–202 (2004)
24. Baresi, L., Guinea, S.: Dynamo: Dynamic Monitoring of WS-BPEL Processes. In: Benatallah, B., Casati, F., Traverso, P. (eds.) ICSOC 2005. LNCS, vol. 3826, pp. 478–483. Springer, Heidelberg (2005)
25. Baresi, L., Guinea, S.: Towards Dynamic Monitoring of WS-BPEL Processes. In: Benatallah, B., Casati, F., Traverso, P. (eds.) ICSOC 2005. LNCS, vol. 3826, pp. 269–282. Springer, Heidelberg (2005)
26. Leavens, G., Cheon, Y.: Design by Contract with JML. Java Modeling Language Project (2003), <http://www.jmlspecs.org>
27. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: a behavioral interface specification language for java. ACM SIGSOFT Software Engineering Notes 31(3), 1–38 (2006)
28. Leavens, G.T., Cheon, Y., Clifton, C., Ruby, C., Cok, D.R.: How the design of JML accommodates both runtime assertion checking and formal verification. Science of Computer Programming 55(1-3), 185–208 (2005)

29. Liu, H., Parashar, M.: DIOS++: A framework for rule-based autonomic management of distributed scientific applications. In: Kosch, H., Böszörményi, L., Hellwagner, H. (eds.) Euro-Par 2003. LNCS, vol. 2790, pp. 66–73. Springer, Heidelberg (2003)
30. Mahbub, K., Spanoudakis, G.: A framework for requirements monitoring of service based systems. In: Service-Oriented Computing - ICSOC 2004, Second International Conference, New York, USA, November 15–19, 2004 (2004)
31. Colombo, M., Di Nitto, E., Mauri, M.: SCENE: A Service Composition Execution Environment Supporting Dynamic Changes Disciplined Through Rules. In: Dan, A., Lamersdorf, W. (eds.) ICSOC 2006. LNCS, vol. 4294, pp. 191–202. Springer, Heidelberg (2006)
32. Momotko, M., Nowicki, B.: Visualisation of (Distributed) Process Execution based on Extended BPMN. In: Mařík, V., Štěpánková, O., Retschitzegger, W. (eds.) DEXA 2003. LNCS, vol. 2736, pp. 280–284. Springer, Heidelberg (2003)
33. Modafferi, S., Mussi, E., Pernici, B.: Sh-bpel: a self-healing plug-in for ws-bpel engines. In: MW4SOC '06: Proceedings of the 1st workshop on Middleware for Service Oriented Computing (MW4SOC 2006), pp. 48–53. ACM Press, New York (2006)
34. Pautasso, C., Alonso, G.: Flexible binding for reusable composition of web services. In: Gschwind, T., Aßmann, U., Nierstrasz, O. (eds.) SC 2005. LNCS, vol. 3628, pp. 151–166. Springer, Heidelberg (2005)
35. Irassar, P., Kloppmann, M.: From Business Process Modeling with BPMN and BPDM to Business Process Execution with BPEL and SCA. In: Dan, A., Lamersdorf, W. (eds.) ICSOC 2006. LNCS, vol. 4294, Springer, Heidelberg (2006)
36. Pistore, M., Traverso, P., Bertoli, P., Marconi, A.: Automated synthesis of composite BPEL4WS web services. In: ICWS, pp. 293–301. IEEE Computer Society Press, Los Alamitos (2005)
37. Puhlmann, F., Weske, M.: Using the *pi*-calculus for formalizing workflow patterns. In: van der Aalst, W.M.P., Benatallah, B., Casati, F., Curbera, F. (eds.) BPM 2005. LNCS, vol. 3649, pp. 153–168. Springer, Heidelberg (2005)
38. Guinea, S.: Dynamo: a Framework for the Supervision of Web Service Compositions. PhD thesis, Politecnico di Milano (2006)
39. Hall, S., Dobson, G., Sommerville, I.: A Container-based Approach to Fault Tolerance in Service-Oriented Architectures (2004), <http://www.cs.wisc.edu/condor/>
40. White, S.: Introduction to BPMN (2003), <http://www.bpmn.org/>
41. Trainotti, M., Pistore, M., Calabrese, G., Zacco, G., Lucchese, G., Barbon, F., Bertoli, P., Traverso, P.: Astro: Supporting composition and execution of web services. In: Benatallah, B., Casati, F., Traverso, P. (eds.) ICSOC 2005. LNCS, vol. 3826, pp. 495–501. Springer, Heidelberg (2005)
42. Baresi, L., Guinea, S., Plebani, P.: WS-Policy for Service Monitoring. In: Bussler, C., Shan, M.-C. (eds.) TES 2005. LNCS, vol. 3811, pp. 72–83. Springer, Heidelberg (2006)

Achieving Dependable Systems by Synergistic Development of Architectures and Assurance Cases

Patrick J. Graydon¹, John C. Knight¹, and Elisabeth A. Strunk²

¹ Department of Computer Science
University of Virginia
151 Engineer's Way, P.O. Box 400740
Charlottesville, VA 22904-4740, USA
{graydon,knight}@cs.virginia.edu
² Software Systems Engineering Dept.
The Aerospace Corporation
15049 Conference Center Drive, CH3/320
Chantilly, VA 20151-3824
elisabeth.a.strunk@aero.org

Abstract. *Assurance Based Development* (ABD) is an approach to the construction of critical computing systems in which the system and an argument that it meets its assurance goals are developed simultaneously. ABD touches all aspects of the system lifecycle, but in this paper we focus on how the evolving assurance argument can guide architectural choices to increase system dependability. The goals with this approach to architectural choice are twofold. The first is to develop the architecture so that it provides the required evidence. The second is to refine the assurance case as architectural choices are made so that the evidence that will be provided supports the assurance claims. Combining development and assurance in this way facilitates detection—and thereby avoidance—of potential assurance difficulties as they arise, rather than after development is complete.

1 Introduction

It is essential that there be a high degree of assurance that a critical computing system will operate dependably in its expected environment, and system architecture plays a major role in achieving that dependability. Unless the architecture of the system is well-matched to both its dependability needs and the associated assurance of that dependability, developers may waste effort on activities that bring unnecessary gains in one part of a system while failing to provide the needed assurance of dependability in others. This can happen in any phase of the lifecycle, but it is especially important for a system's architecture because inappropriate architectural decisions can have a major impact on subsequent development activities. Nevertheless, current approaches to assuring dependability—in the system's architecture, and in other aspects of its development—are frequently ad hoc.

Assurance Based Development (ABD) is a novel approach to the development of critical computing systems in which development of the system and of its assurance argument are integrated. The assurance argument, presented in an *assurance case*, documents how evidence from the system and the process used to construct it supports the system's dependability claims. Integrating system development and system assurance means that dependability-related development objectives are clearly laid out, and can be addressed specifically when making architectural choices. In this paper, we describe the ABD process and how it can be used to develop a software architecture.

In ABD, the assurance case and architecture are developed in parallel. Each architectural choice a developer makes is assessed in terms of its impact on: (1) the system's functionality; (2) the development activities that will be needed; (3) the evidence that will be needed in the assurance case; and (4) the argument structure of the assurance case. The goals with this approach to architectural choices are twofold. The first is to develop the architecture so that it provides the evidence required in the assurance case. The second is to refine the assurance case as architectural choices are made so that the evidence which will be provided supports the claim that the system is adequately dependable.

Combining architecture development and dependability assurance in this way promotes detection—and thereby avoidance—of potential assurance difficulties as they arise. Inevitably, architectural choices are made without complete knowledge of their impact on subsequent development, and so it is always possible that a decision will have to be rethought. Nevertheless, by including explicit attention to dependability assurance goals while creating the system's architecture, the chances that a decision will not support the overall dependability goal are reduced. Where this is not done, there is the very real danger that inadequate dependability might only be revealed during evaluation carried out after development is complete.

Since dependability assurance is considered and addressed throughout development, ABD can increase the confidence that can be placed in an architecture. Furthermore, the increased efficiency of the development processes allows resource savings during development of typical critical systems when ABD is used. Nevertheless, it is not possible to show that architectures chosen with the help of the assurance case will always be superior to architectures chosen in an ad hoc manner because of the many variables involved in development. The architect will have been able to make better informed choices than that would have been possible otherwise, and so his or her goals are more likely to be achieved.

2 Assurance Cases

Assurance cases are the state of the art in rigorous but non-formal dependability argumentation and, as such, provide the foundation on which the ABD approach to architectural development rests. The most common use of assurance cases at present occurs in the documentation of *safety*, and safety cases have been built for a variety of production systems. In general, a safety case is “a documented body of evidence that provides a convincing and valid argument that a system is adequately safe for a given

application in a given environment” [2]. Special graphic notations have been designed to enable the documentation of assurance cases in a manner that is easy for humans to understand and that can be manipulated by machine. The most widely used of these notations is the Goal Structuring Notation (GSN) [17].

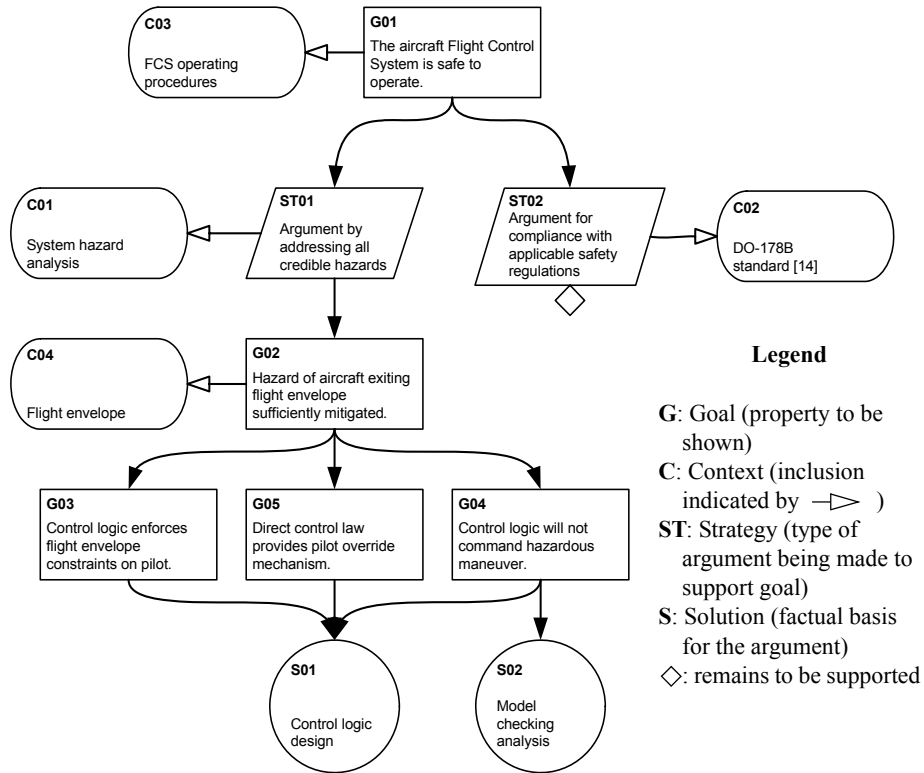


Fig. 1. Example assurance case

In its simplest form, an assurance case contains an instance of each of three essential elements: (1) an assurance goal or claim; (2) evidence that the goal has been satisfied; and (3) an argument linking the evidence to the goal in a way that leads one to believe that the goal is justified by the evidence. This basic structure is supplemented with a variety of other elements, including assumptions, justifications, and context, and applied recursively to produce, for real systems, a hierarchic structure with the overall goal for the system at the root. The hierarchic structure makes the overall argument manageable at each level. Figure 1 illustrates the use of GSN in a simple *completely hypothetical* safety case. In the figure, the assurance goal is stated in the box at the top and the remainder of the figure documents the argument for belief in that goal.

The overall argument in an assurance case is a set of logical inferences that show why the evidence implies that the system’s assurance goals have been met. The goals,

evidence, and hence the assurance argument are specific to a particular system, and so each assurance case is unique. However, patterns have been developed for common argument fragments [10].

3 Assurance Based Development

Assurance Based Development integrates the various separate activities that occur in the construction of a critical system, i.e., it integrates requirements and context analysis, system development, and assurance case creation. The primary goal is to ensure that the choice of development techniques will allow evidence generated during development to be sufficient for the system's assurance case. Creating a system architecture is one stage of ABD. We describe the overall ABD process in this section, elaborating its implications for system architecture below.

The major components of Assurance Based Development and their high-level interactions are shown in Figure 2. Shown on the left of the figure are components labeled *system context* and *system requirements*, with the former enclosing the latter; the context in which a system operates influences the system requirements in many ways. The system requirements are used by both the system assurance case and the system development artifacts. The system requirements include the dependability requirements such as availability and safety, and thus determine the primary goal of the assurance case. The system requirements include the functional requirements also and so are the starting point for the development lifecycle.

At the center of the technique are the *system assurance case* and the *system development artifacts*. These two components are developed in parallel, and their

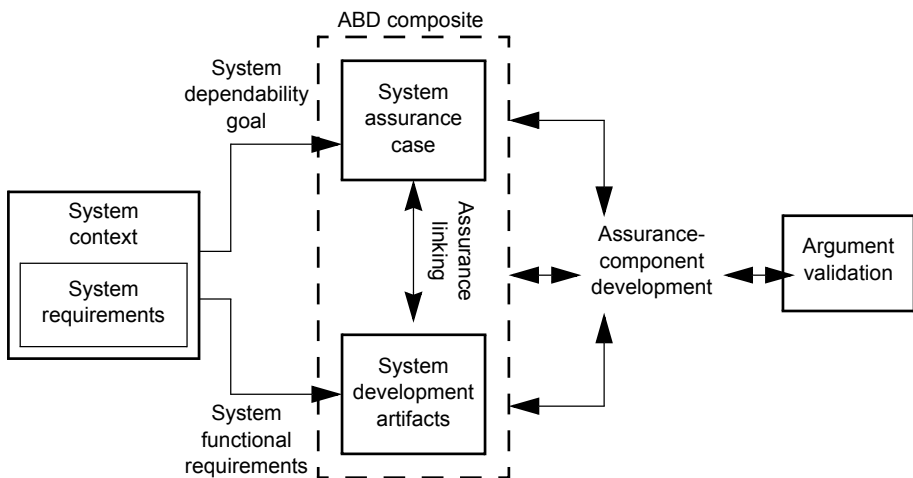


Fig. 2. Assurance Based Development

development is coordinated using a technique that we refer to as *assurance linking*. Assurance linking ensures that assurance goals and development artifacts are coupled explicitly and systematically so as to reveal the evidence needed by the assurance case from the development artifact. Assurance linking enables developers to check that the properties possessed by development artifacts are the properties necessary to support the goal in the assurance case for which the development artifact provides evidence.

As an example of assurance linking, consider the problem of developing a software component in a safety-critical system such that the component meets an assurance goal of having a failure rate per unit time below some threshold p , where p is perhaps 10^{-3} , i.e., not in the ultra-dependable range. Testing might be the basic strategy chosen to meet this goal. Such a goal requires several pieces of evidence if it is to be believed as part of an assurance case. These pieces of evidence are: (1) that the specification for the component is correct (a complex assurance subgoal); (2) that the component has been tested according to its test plan (a development subgoal); (3) that testing according to the test plan demonstrates in a statistically valid way that its failure rate is below the threshold (a developmental and documentation subgoal); (4) that the test cases were the result of a random process of selection from the expected operational environment (an analysis subgoal); and (5) that these items of evidence were recorded and reported accurately.

A technique called *ABD composite production* is the process of developing a component and its associated assurance case elements. The assurance case plays a predictive role in ABD since it is used to determine the necessary properties that each development artifact must have in order to support the system assurance argument.

Development of an architecture, therefore, consists of repeatedly making an architectural choice, assessing the value and suitability of the evidence that will result from it, and developing the associated assurance argument fragment. The assurance argument fragment is then analyzed to ensure that its derived premises are both satisfiable and practical, and that it is free of fallacious reasoning and other flaws.

If the synergy between the system and its assurance case is not present, then the basis for each development choice, including those made in architecture, will tend to be factors such as cost, experience and convenience although dependability will sometimes be considered. Thus there will be no guarantee that the choices made will be those that facilitate the system meeting its dependability goals nor that the evidence developed will be that needed in the assurance case.

4 Existing Techniques for Architecture Development

In any discussion of architecture, it is important to have a precise definition of what the term means. In this paper, by the term architecture, we mean the following:

The architecture of a system is the system's fundamental organization, embodied in its components, their relationships to each other and to the environment, and the principles guiding its design and evolution. [1]

There is no rigorous, prescribed process for developing a system architecture or a software architecture. There are several principles that are known to promote certain useful qualities in an architecture, e.g., modularization that facilitates information hiding, and there are architectures that are commonly used because they possess important properties, e.g., redundancy allows certain types of fault to be masked.

In practice, most architectural development tends to be driven primarily by desired system functionality. The dominant technique used, therefore, is functional decomposition using informal “box-and-arrow” models. Once a model is thought capable of supplying the necessary functionality, attention turns to issues such as encapsulation and performance. Dependability as an architectural issue is often either: (1) assumed to be addressed by some architectural pattern; (2) an afterthought that has to be argued with an architecture that is a *fait accompli*; (3) argued in a fragmented and uncoordinated manner as development proceeds; or (4) some combination of items (1)-(3). The first approach leads to a partial argument where the contribution to dependability or lack thereof by many of the architectural decisions is missing. The second approach can lead to a lot of rework as architectural decisions are discovered that do not meet the needs of the assurance argument. Finally, the third approach leads to an incomplete and unsatisfactory argument.

Developers of safety-critical systems pay attention explicitly to system dependability both during development and during evaluation, and the process they follow shares some characteristics with ABD. However, although dependability goals influence architectural choices, the goals considered are typically just overall system reliability or availability targets, and the process tends to have the characteristics of item (3) above.

In the experience of the authors, for systems in general, architects often use an iterative process such as the one shown in Figure 3. In each iteration, the architect makes a tentative architectural choice based on functional decomposition and then examines the software as it would be given that choice using a box-and-arrow style of model. He or she speculates as to whether, with respect to obvious alternative choices, the new choice combined with the previous choices would:

- allow the resulting system to meet its functional requirements;
- allow partitioning of the software into modules that can be divided evenly over the available team members;
- result in a system with adequate performance; and
- result in software with acceptable volume and complexity.

An architect typically starts the development of an architecture with a choice that he or she hypothesizes will allow the required overall system functionality to be provided. Understandably, an architect usually proposes architectural choices with which his or her team is already familiar: a team that has used the same architectural pattern in the last several projects will often start by proposing that pattern for the next project. Clearly, they could not propose an architectural choice with which they were totally unfamiliar. Also, architects are hesitant to use architectural choices with which they have little or no experience, as their unfamiliarity will make their

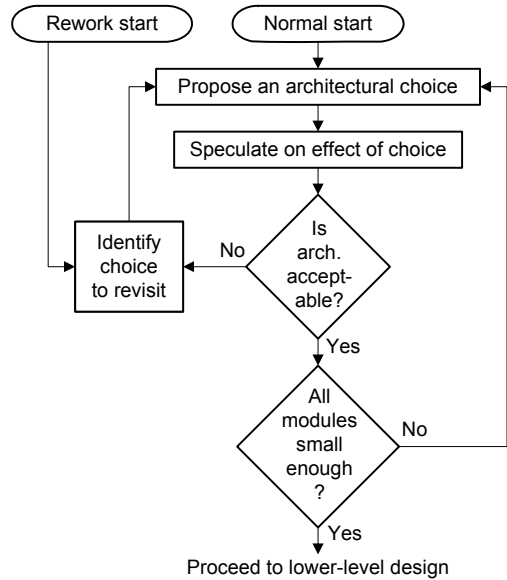


Fig. 3. Traditional process for architecture development

conclusions about the acceptability of the choice tenuous, thus adding to perceived project risk.

Selection of architectural choices is guided by the architect's sense of the next most "pressing" issue. Once convinced, albeit informally, that their choice could supply the necessary functionality, the architect turns to issues such as performance, flexibility, maintainability or similar. Following that the architect might turn to mundane but important issues such as whether the components of the architecture can be implemented effectively by the available team structure.

This development process does not guarantee an acceptable architecture. As a tentative architecture becomes more complex, answering questions such as whether proper functionality will be supplied or whether acceptable performance will be achieved requires techniques such as architectural modeling and prototyping. In some cases, the answers are not obtained until the system is built at which point the "wrong" answer can be disastrous because selecting a new choice is either impractical or very expensive.

When the perceived drawbacks are costly enough, however, architects will revisit choices. They will replace them with alternatives and re-assess the system as a whole. They will then consider replacing any further choices that examination reveals to be sub-optimal given the repaired choice.

In some cases, architectural choices are affected by non-technical considerations such as standards, imposed choices, and workload balancing. External standards often have to be followed to facilitate certification in regulated industries such as medical devices. Similarly, development decisions such as the choice of target hardware platforms or the choice of software development tools are sometimes imposed as an

enterprise-wide decision. Finally, limited development resources may force engineers to accept compromises that increase development risk, reduce functionality, or make maintenance or enhancement more difficult.

In the special case of safety critical systems, the general architectural approaches summarized above are sometimes enhanced by including consideration of the safety argument during development. For example, it has been advocated that developers create a preliminary safety case early in the system lifecycle and update it periodically as development progresses [9, 12]. ABD takes this idea to its limit by tightly coupling the development of the system and its assurance argument so as to make visible to the developer the assurance obligations incident upon each part of the system.

Attribute Driven Design (ADD) is an architectural development technique that is closely related to ABD [18]. In ADD, architectural choices are influenced by quality properties as well as more conventional software architecture considerations. The major difference between ABD and ADD is that ABD uses the rigorous argument of a safety case to fully document the rationale for believing that quality attributes have been achieved and guides development to ensure that this argument will be valid.

5 Architecture in Assurance Based Development

As discussed in section 3, the fundamental concept underlying Assurance Based Development is that system development decisions are based on being able to meet assurance goals and on supporting the associated assurance argument. The development of a system's architecture is a part of the overall development, and so the architectural decisions in ABD are based on meeting assurance goals. With this approach, the role of architecture in dependability is clear, and the architectural choices made can be optimized to achieve the desired dependability and the associated assurance without expending unnecessary cost. The ABD process is summarized in Figure 4.

The starting point, as with any development, is the *given architecture*. This is the high-level architecture within which the computing system will operate. The high-level architecture will define the functional requirements for the computing system and the associated dependability requirements. The top-level goal in the computing system's assurance argument is derived from these requirements. Thus, any system that demonstrably meets this goal solves the problem that it was created to solve, providing the desired functionality with the desired dependability when operated in the intended context.

The ABD process for developing a system architecture proceeds differently from the process discussed in the previous section. The architect begins by examining the top-level goal in the assurance case. This goal will include the dependability requirements as well as the functionality itself. The architect seeks an architectural choice that will allow the goal to be met. His or her first step is to make an architectural choice that results in a satisfactory argument that the goal has been met and sub-goals that can be addressed practically by subsequent development activities. Each choice is evaluated in terms of seven criteria, which are discussed in detail in section 5.2. These criteria have been selected so as to help developers avoid making

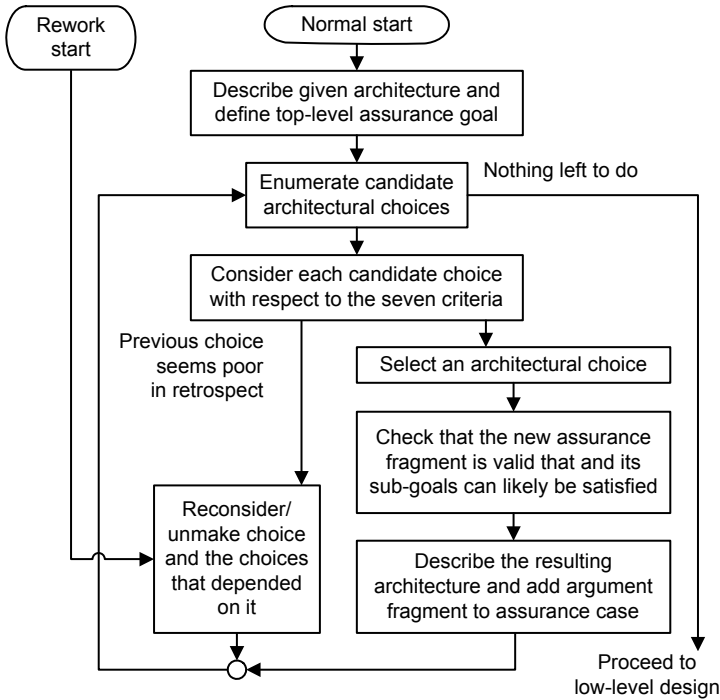


Fig. 4. Architecture development in ABD

choices that will need to be re-visited because they make successful completion of the system and its assurance case impractical or even impossible.

Once the choice is made, the evidence that results from the choice becomes an integral part of the argument that the top-level goal of the assurance case is met. Thus acceptance of the choice by the architect requires that he or she develop the associated argument fragment for the assurance case. This argument fragment must show that the top-level goal will be met, provided the premises generated by the architectural choice are met.

As an example of the concept, consider the development of a simple autopilot system that has to provide an altitude-hold capability for a general-aviation aircraft. Such a requirement could be implemented as a control loop that reads a set of sensors, computes an adjustment to the actuators, and sends commands to the actuator servos according to a simple, periodic real-time schedule. The dependability requirements would be quite extreme, however, because safe flight depends on the system operating correctly when in use [14]. In this case, a system reliability of 1×10^{-7} over three hours together with a significant safety requirement might be required.

The given architecture for such a system would include much of the rest of the aircraft's avionics system. In particular, the mechanisms by which the sensor signals are made available and the actuators are commanded would be defined.

The architect faced with the stated assurance goal would need to make an initial architectural choice that would ensure hardware and software failure rates which, when suitably composed, would meet the goal. The choice, therefore, might include an NMR hardware system¹ combined with software developed with a rigorous process and a comprehensive test plan. The detailed characteristics of the selected NMR hardware system and of the software development process would be the evidence for the argument fragment, and the architectural choice would not be accepted unless this argument was considered adequate.

In the remainder of this section, we describe the Assurance Based Development of architecture in detail. In section 5.1 we discuss the determination of architectural choices and in section 5.2 selection from them. In section 5.3 we present the use of architectural choices, and we review the overall process in section 5.4. A detailed example of ABD applied to architecture is presented in section 6.

5.1 Candidate Architectural Choices

With dependability as a central criterion in making architectural choices, it is important to ensure that all possible candidate choices are considered. Several general textbooks have been written on architecture [15, 3, 4, 5], and a great deal of research has been conducted on architectural support for dependability [e.g., 16], and it is not necessarily the case that the architect will be familiar with the field. This research has been motivated in most cases by seeking to achieve specific dependability metrics (e.g., reliability, availability and safety) within general system categories (e.g., servers, clients and embedded systems), and the relevant literature tends to be organized from these perspectives.

Determining the architectural choices in ABD is a two-phase process. In the first phase, the architectural choices that might be able to address the current subject assurance goal are enumerated. Care has to be taken to include: (1) all available general architectural patterns; (2) all architectural choices from experience with similar systems; and (3) architectural choices from beyond the architect's personal experience.

In the second phase, the functionality and assurance evidence that each choice can provide is elaborated. Because the candidate choices are necessarily generic, the architect must consider how the choice would apply to the situation at hand and determine the functionality and evidence that would result.

Patterns are a general and commonly used technique, and they have proven especially important in architecture and design. They can be used in Assurance Based Development where they consist of architectural choices coupled with assurance case argument fragments, the composite being used to capture experience thereby allowing future developers contemplating similar architectural decisions to benefit from that experience.

¹ N Modular Redundancy (NMR) executes N replicates of a system in parallel on separate hardware and selects an output by voting. Defects in a minority of the replicates can be masked with this technique. N is often three giving Triple Modular Redundancy (TMR).

5.2 Selection of an Architectural Choice

Selection of a suitable architectural choice from the enumerated candidate set is based on seven criteria (discussed below): functionality, subsequent restrictions, dependability, cost, feasibility, standards, and additional non-functional requirements. A candidate architectural choice can be rejected based on just one or several of the criteria, or it can be modified to suit the needs of the system under development if a change can deal with the problem.

Much of the pruning of the set will be based on the architect's experience. In many cases, an experienced architect might consider only a single candidate architectural choice in which he or she has considerable confidence. In such a case, these criteria are exit criteria from the selection process for that choice.

It might appear that, except for dependability, these criteria will have the same meanings and roles in ABD as they do in traditional architectural development. This is not the case, however, because several of the criteria can influence and can be influenced by the dependability argument. Note also that these criteria are not disjoint, and so evaluating a criterion cannot necessarily be done in isolation. We examine each criterion briefly with an emphasis on its overall role in dependability.

- **Functionality.** Once an architectural choice has been instantiated for the system under development, the architect needs to check by inspection, analysis, prototyping and/or modeling that there are no detectable deterrents to achieving the desired functionality. The functionality criterion is the same in ABD as in existing techniques.
- **Restrictions imposed on later choices.** To a greater or lesser extent, each architectural choice that is made restricts subsequent choices throughout the rest of software development. Making an architectural choice at one level of abstraction generates subgoals for subsequent refinement, and those subgoals can only be met in certain ways. Of particular importance in this context is the possibility of an architectural choice restricting the selection of techniques later in the process that either facilitate dependability or contribute to its assessment.
- **Evidence of dependability.** Each system development choice must give rise to evidence that, along with an assurance strategy, is sufficient to argue that the assurance goal will be met.
- **Cost.** Clearly, any architectural choice has to be cost effective, not only in terms of software construction effort but in a complete sense. If provision of adequate evidence for the assurance argument would require resources beyond those available, the candidate architectural choice has to be rejected. The issue is not whether the system can be built within budget but whether the system can be built and have a satisfactory assurance case within budget. While cost is a consideration in all systems, regulations sometimes demand justification that risk has been reduced as low as reasonably practical (ALARP). In such systems, cost may directly appear in the assurance case in addition to providing guidance on selection.
- **Feasibility.** The architectural choice must not be infeasible. Moreover, it must not preclude the completion of an architecture that can be instantiated, the

completion of a system that is fit for use in its intended context, or the creation of a convincing assurance case for that system.

- ***Applicable standards.*** Applicable standards have two effects on the selection of an architectural choice. First, a standard might preclude certain choices by definition. Second, standards might require certain development practices that restrict or preclude certain forms of evidence that would otherwise be required for the assurance case.
- ***Non-functional requirements.*** Non-functional requirements derive from stakeholder interests, and they have an effect that is similar to the effect of a standard. Non-functional requirements often prescribe certain aspects of development or certain characteristics of the desired system. Such prescriptions limit the available architectural choices and are likely to affect the assurance evidence in the same way that a standard can.

As an example of the application of these criteria, consider again the simple autopilot example mentioned earlier. Assume that the choices that the architect can make are: (1) a single processor running the entire application; (2) a pair of processors both running the entire application and comparing outputs; (3) three processors with each running the entire application and voting on their results (TMR); and (4) a distributed implementation in which several processors are connected together with a real-time bus and different parts of the application are run on different nodes.

The evidence for the assurance case that each choice provides would depend on the specific characteristics of the equipment chosen and the planned approach to the development of the software. The dependability requirements from the given architecture are such that options (1) and (2) might have to be rejected based on the dependability criterion. Option (4) might have to be rejected because of cost.

Applying these criteria can be quite involved since they are neither independent of each other nor independent of decisions at other points in development. Consider, for example, the applicable standards criterion. If such a standard prescribes use of a particular programming language, this might preclude the subsequent use of certain forms of static analysis that depend on certain language features (such as strong typing) or on the existence of a formal semantic definition of the language.

5.3 Using an Architectural Choice

Once an architectural choice has been made, precise descriptions (i.e., specifications) of the choice itself and the assurance evidence that implementing it will provide constitute an ABD composite. The ABD composite documents the link between the choice and the evidence.

Once the ABD composite has been formed, the choice is integrated into the evolving system architecture. The architecture can be documented in any manner that is deemed appropriate. In particular, an architectural description language can be employed thereby facilitating a variety of analyses.

The next step is to document the argument fragment to which the evidence applies and to integrate the fragment into the evolving assurance case. As with the

architecture itself, the assurance case can be documented in any suitable manner, but the use of GSN would be a likely choice.

The assurance case fragment to be added as the result of a choice identifies the affected development artifacts and describes the contribution that these artifacts will make to the argument. In some cases, the choice will introduce new goals, obligating the developers to supply specific evidence later in the process, while in others the choice will directly support a goal with evidence from a development artifact.

The role of the ABD composite is to document the link between the development stage where the evidence is created and the location in the assurance case where the evidence is part of the argument. As development proceeds, there is an obligation to ensure that the evidence is prepared as expected. Any changes in the anticipated development activity must be traced back to the assurance case so as to check that the effects of the change on assurance have been considered, and the mechanism that supports this traceback is the ABD composite.

It is possible that an architectural choice will prove unacceptable after it has been selected and subsequent choices have been made. To address this, a developer must isolate the problematic choice, select an alternative, and re-examine any choices made after the readdressed choice.

5.4 Termination of the Architectural Development Process

The ABD process continues as long as architectural choices are being made that produce subgoals which need to be refined using architectural techniques. Thus, as each architectural choice is made, new subgoals will be generated to support the assurance argument fragment that derives from the choice.

Each of these subgoals starts the process described in this section over again unless the architect is convinced that the subgoal is not best addressed by an architectural solution. The architect deems the architecture complete when, in his or her judgement, all of the modules in the system require no further decomposition, are sufficiently well defined, and are cohesive enough that it is likely that those responsible for designing and implementing them will be able to do so successfully.

An assurance case for a complete architecture may contain unsubstantiated goals that could be addressed through architecture; these will be addressed by design, implementation, and verification choices as the ABD process proceeds. Many architectural patterns are also design patterns, and so it is likely that some goals could be addressed through either architecture or design. Because architecture is more centralized than design, goals that can be addressed through design should be left to designers in order to keep the architecture phase from becoming a bottleneck.

6 An Illustrative Example

In order to illustrate the process of developing an architecture using ABD, we present an illustrative example of the use of the technique on a realistic application. The process is quite extensive, and so in the summary we examine only two architectural choices. In addition, although the application is real, we have made a number of

assumptions about aspects of the application that either have not been documented by the system developers or are necessary for ABD but not for the application in its present form.

The system we use for illustration is part of a software-based system for alerting pilots to *runway incursions* at airports. Lockheed Martin, in collaboration with the National Aeronautics and Space Administration (NASA), is developing a research prototype system known as the Runway Incursion Prevention System (RIPS) [7, 8] to address this realistic safety-related problem. The Federal Aviation Administration (FAA) defines a runway incursion as “any occurrence at an airport involving an aircraft, vehicle, person, or object on the ground, that creates a collision hazard or results in the loss of separation with an aircraft taking off, intending to take off, landing, or intending to land.” [7]. The RIPS system operates in the cockpit of an aircraft (referred to as *ownship*), collects information about the position of the aircraft and of other aircraft and traffic in the vicinity, examines that information for evidence of a runway incursion involving the ownship aircraft, and alerts the pilot to such incursions via an Integrated Display System (IDS) if a collision is possible.

Our illustrative example is based on a part of RIPS called the Runway Safety Monitor (RSM). The RSM was not developed using ABD, and so our example is strictly for purposes of illustration. Our work is not part of the RIPS development activity. In constructing the example, we have drawn upon the RSM documentation for descriptions of the problem to be solved, the sources of data available for the purpose of detecting incursions, and of the systems on board the aircraft and on the ground with which an incursion detection system might interact.

6.1 The Given Architecture

The RSM makes use of the existing systems on board the aircraft including a computer, the aircraft’s ground location system that provides the aircraft’s position, and broadcasts on the Automatic Dependent Surveillance - Broadcast (ADS-B) link that provides the positions of other traffic. These sources of data are known to be unreliable in that data might be unavailable for periods of up to several seconds because of limitations in the basic equipment. This lack of reliability in the data is not a serious problem provided the pilot knows that RIPS is not able to report incursions.

While the decision to implement RSM as a software entity that uses this equipment is an architectural decision, it is an architectural decision at the level of RIPS rather than that of the RSM system. In effect, the architects of RIPS decided to delegate the task of alerting the pilot to a software sub-component rather than a separate system running on its own processors. The outcome of these decisions constitute the given architecture.

The given architecture is shown in Figure 5. The IDS system polls the RSM at a frequency of 1 Hz to determine whether a runway incursion involving ownship is in progress. To perform its computation, the RSM will need to know where the ownship aircraft (the aircraft it is installed in) is located, and where other aircraft that might conflict are located. It will obtain the former from the aircraft’s ground location system and the latter from the contents of broadcasts on the ADS-B bus.

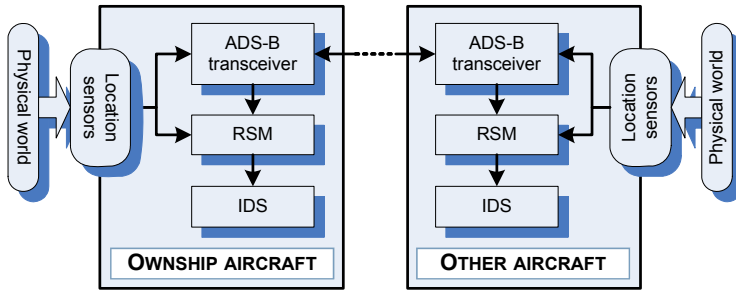


Fig. 5. The given RSM architecture

6.2 The Top Level Assurance Goal

The problem to be solved is to detect incursions involving ownship. The top-level goal of our assurance case states both the required functionality and dependability of the system as shown in Figure 6. For purposes of illustration, we have assumed dependability requirements for the RSM that place it in the ultra-dependable category and classify the system as safety critical.

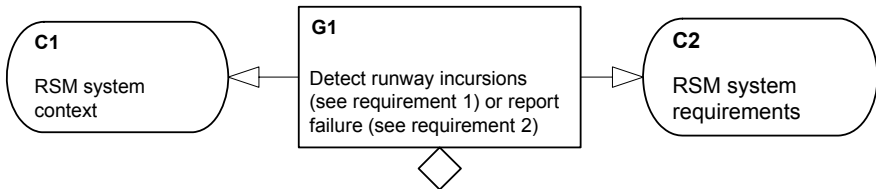


Fig. 6. Top-level assurance case goal

In this example, we assume that the RSM is required to meet the following two requirements (recall that the data sources are unreliable):

- **Requirement 1:** If the quality of the supplied data is adequate, to detect runway incursions involving ownship within t time units after they begin with probability greater than or equal to p_0 .
- **Requirement 2:** If the quality of the supplied data is inadequate, to report a failure of RSM with probability greater than or equal to p_1 within u time units.

Note the inclusion in Figure 6 of the system's context in GSN. The details of the system's context are crucial to the proper refinement of the goal and the analysis associated with both the functionality and the dependability of the system.

6.3 The First Architectural Choice

There are many candidate architectural choices that meet the two requirements in the top-level goal. For example, the overall approach to the real-time requirements could

be either sequential or concurrent, and if concurrent then either synchronous or asynchronous. The choice will be influenced, in part, by the services available from the target operating system, in part, by the anticipated verification approach, and by several other factors.

The requirement for the detection of missing or corrupt data can similarly be addressed using various architectural mechanisms. A number of different system modules could take action when data is missing, and data defects could be signaled by a data collection module by generating an event, by a time-out, or by using special coded data values. Feasibility is an important criterion in this aspect of selection because there has to be a high level of assurance that defective data will be detected and that the timing element of the requirements is met.

The experience of the authors leads us to select a sequential code implementation with each software module responsible for detecting and reporting errors in the data it handles. Choosing sequential code with distributed error detection allows us to divide the top-level goal into three concerns: 1) RSM primary functionality; 2) RSM timing, and 3) RSM detection of defective data. The resulting assurance case fragment is shown in Figure 7. An important item in this fragment is goal G2.4. This goal requires that evidence be supplied and that an argument developed which shows that the three modules do not interfere with each other. This is an important aspect of the verification that must be constructed if this architecture is used, yet this is not obviously so without the assurance case as a reference.

6.4 The Second Architectural Choice

The first architectural choice generated four subgoals, and in a complete application of ABD all four would be addressed. For purposes of illustration, we address only one, the RSM functionality (goal G2.1 in Figure 7).

There are many candidate architectures that might be used including several patterns, an object-oriented approach, and functional decomposition. We selected functional decomposition of the RSM functionality (see Figure 8) because it facilitates the use of some forms of static analysis including determination of worst-case execution time. That decision leads in this example to the following six modules:

- The *ownership runway locator*, which determines whether the aircraft in which the RSM is presently using a runway, and, if so, builds a model of that runway;
- The *runway database*, which stores the location and necessary geometric details of all of the runways for which RSM service will be available;
- The *runway model*, which stores the geometry of the runway including the bounds of the incursion zone;
- The *ownership position* component, which collects information about the position of the aircraft from the aircraft's ground location system;
- The *conflict detector*, invoked if the aircraft is found to be using a runway, determines whether ownership is in conflict with any other monitored traffic within that runway's incursion zone, and
- The *traffic positions* component, which collects information about the position of other traffic within a specified region from ADS-B broadcasts.

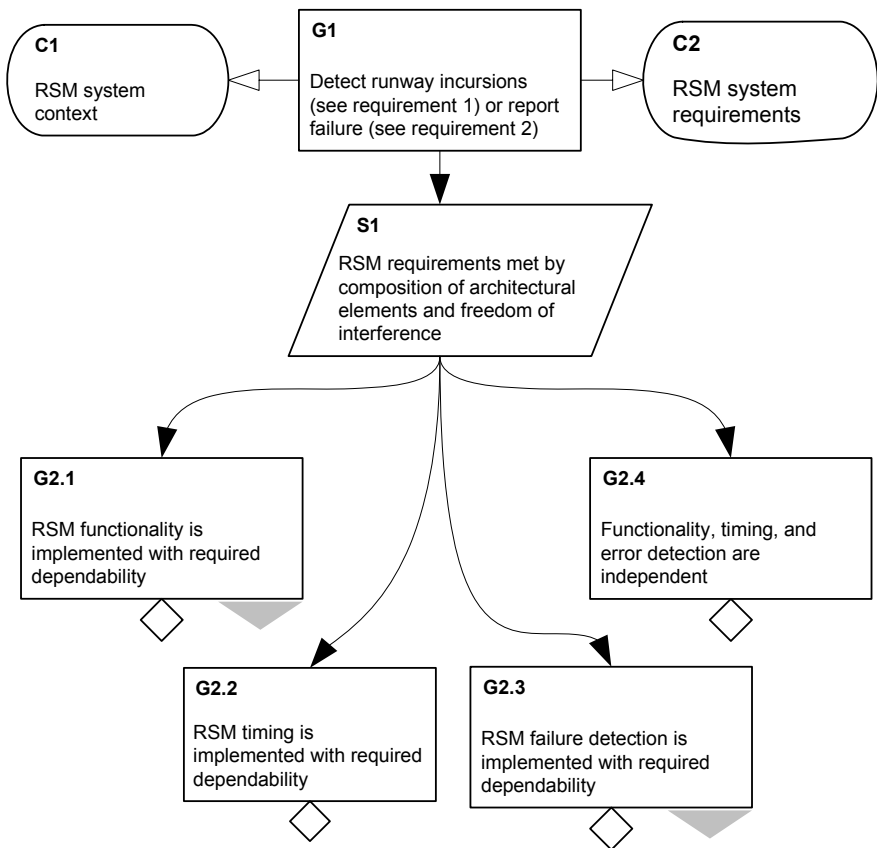


Fig. 7. Argument fragment from first architectural choice

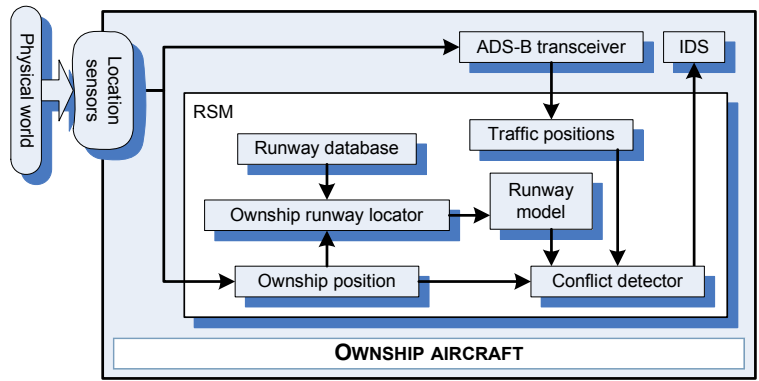


Fig. 8. Functional decomposition of RSM functionality

The assurance case fragment that accompanies this architectural choice is shown in Figure 9. It details the assurance responsibility allocated to each of the new components listed above and how these responsibilities, if satisfied, demonstrate the satisfaction of sub-goal G2.1.

Although not shown, the detailed arguments for goals G2.2 and G2.3 in Figure 7 are similar to the argument for goal G2.1. The argument for goal G2.2 is facilitated by the decision to use functional decomposition for goal G2.1. To show that goal G2.2 is met requires several forms of evidence, including assurance that various modules will execute within specified time bounds. One of those modules is that which is associated with goal G2.1. Functional decomposition as the architectural choice for goal G2.1 eases the task of determining worst-case execution time (WCET) for that module. WCET is not easy to establish with any architecture and can be essentially impossible with some modern processors. However, assurance over timing is essential, and that makes many other candidate architectural choices unacceptable.

The argument for goal G2.4 will be different from goals G2.1, G2.2 and G2.3. This goal is concerned with the composition of the evidence from the other three. It is not sufficient to know that each of the other three goals will be met in order to use that evidence to argue that goal G1 will be met. Goal G2.1 might be met, for example but there might be side effects that impact the composition of goals G2.1, G2.2, and G2.3.

Turning now to the other selection criteria, we ask ourselves whether, given this architectural choice, it is likely that the system can be built within the specified budget, schedule, technology constraints, etc. At this point the architecture is not yet complete, much less the low-level design and implementation, so our assessment will be speculative—as would any such assessment at this point in the development of a system. Given our experience and knowledge and the proposed architecture as it stands, how likely do we think it is that we will encounter a difficulty that will force revisions that would cause the project's schedule to slip or, worse, cause the effort to fail completely?

To perform this assessment, we consider the components in our candidate architecture and the responsibilities upon them described in our assurance case fragment. Will it be possible, for example, to construct the ownship runway locator so that, provided the components it depends upon perform as described, the ownship runway locator demonstrably satisfies the goals with which it is associated? Given what we know about the probability of data errors, reasonableness checks on the incoming data coupled with the use of formal techniques to implement and verify the algorithm seem like a plausible way to construct a component, later in the process, that can be shown to meet its goals using a software reliability modeling technique. Thus, we decide that the architectural choices we made are appropriate given the knowledge we have of the system at the time the architecture is created. Furthermore, although there is no way to know for sure that the architecture is the best one, we were able to assess it against the system's assurance goals, and that assessment gives us much higher confidence that the architecture is satisfactory than would experience and intuition alone.

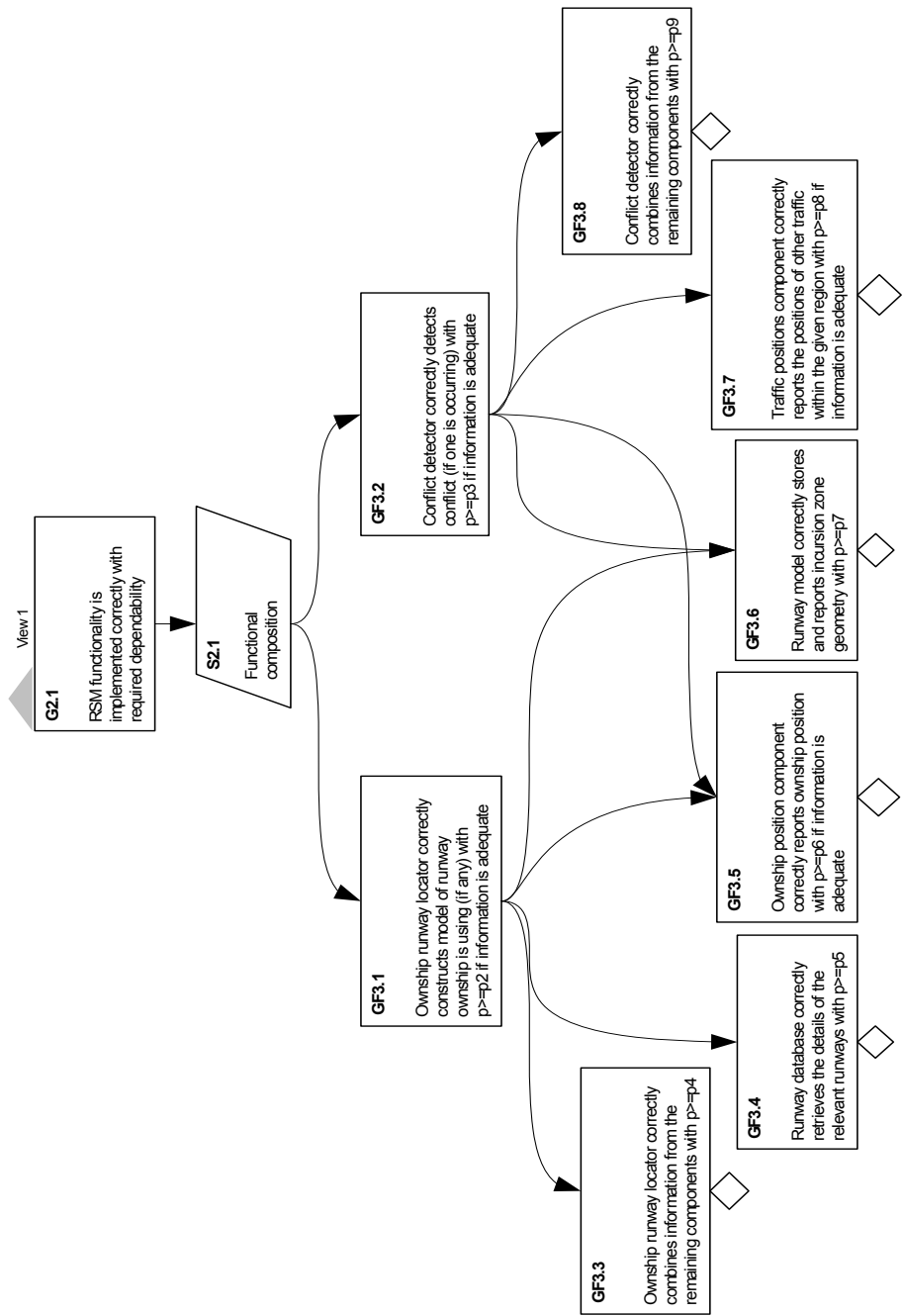


Fig. 9. Assurance of functionality split across functional decomposition

7 Conclusion

There are a number of choices that must be made when designing a system's architecture, and those choices can have a profound impact on the finished system's dependability. Currently, there is little guidance for making the right choices, given the level of dependability that must be met by the system. If system development is coupled with system assurance, however, the system's assurance case can guide architectural choices, providing concrete dependability criteria against which to gauge potential alternatives.

In this paper, we have explained the basic principles of Assurance Based Development, and shown how this development paradigm can be used to provide assurance case goals for architectural choices. We have presented an example system architecture and shown how evolving its assurance case in parallel with the architecture kept us continuously apprised of the specific dependability goals each part of our system was obliged to meet. Whereas with standard architecture development techniques we would have had to wait until system development was more complete to analyze the effect of our choices on the system's dependability, we were able to assess the choices against specific assurance case goals for the RSM.

Finally, software system architecture is currently very much an art, and the creativity in finding a good architecture is due in large part to the difficulty in creating general guidelines from a wide variety of systems. Because it is not clear whether the context in which one successful decision was made is similar to that in which a new choice must be made, whether the same choice should be made for the new system is likewise unclear. With ABD, the specific situation in which a choice is made is much more clearly defined because of the assurance goal that accompanies it. Thus, not only does ABD guide specific architectural choices, it helps lay a foundation for good architectural engineering.

Acknowledgements

We thank David Green of Lockheed Martin for giving us extensive help in understanding the RSM and all of the associated artifacts. We are very grateful to NASA Langley Research Center for suggesting the use of the system for study. We appreciate William Greenwell's assistance with the assurance case material presented here and his contribution of the hypothetical safety case in Figure 1. This work was sponsored in part by NSF grant CCR-0205447 and in part by NASA grant NAG1-02103.

References

- [1] ANSI/IEEE standard, 1471-2000, Recommended Practice for Architectural Description of Software-Intensive Systems -Description
- [2] Bishop, P., Bloomfield, R.: A Methodology for Safety Case Development. In: Proc. of the Sixth Safety-critical Systems Symposium, Birmingham (February 1998), <http://www.adelard.co.uk/resources/papers/index.htm>

- [3] de Lemos, R., Gacek, C., Romanovsky, A. (eds.): Architecting Dependable Systems. LNCS, vol. 2677. Springer, Heidelberg (2003)
- [4] de Lemos, R., Gacek, C., Romanovsky, A. (eds.): Architecting Dependable Systems II. LNCS, vol. 3069. Springer, Heidelberg (2004)
- [5] de Lemos, R., Gacek, C., Romanovsky, A. (eds.): Architecting Dependable Systems III. LNCS, vol. 3549. Springer, Heidelberg (2005)
- [6] EUROCONTROL. The EUR RVSM Pre-Implementation Safety Case, ver. 2.0. Document RVSM 691 (August 14, 2001)
- [7] Green, D.F.: Runway Safety Monitor Algorithm for Runway Incursion Detection and Alerting. Technical report NASA CR-2002-211416 (January 2002)
- [8] Green, D.F.: Runway Safety Monitor Algorithm for Single and Crossing Runway Incursion Detection and Alerting. Technical report NASA CR-2006-214275 (February 2006)
- [9] Kelly, T.P.: A Systematic Approach to Safety Case Management. In: Proc. of SAE 2004 World Congress, Detroit, MI (March 2004)
- [10] Kelly, T., McDermid, J.: Safety Case Patterns – Reusing Successful Arguments. In: Proc. of IEE Colloquium on Understanding Patterns and Their Application to System Engineering, London (1998)
- [11] Kinnersly, S.: Whole Airspace ATM Safety Case - Preliminary Study (November 2001)
- [12] MoD, 00-56 Safety Management Requirements for Defence Systems, U.K. Ministry of Defence, Defence Standard, Issue 3 (December 2004)
- [13] Nagra. Project Opalinus Clay: Safety Report. Technical report NTB 02-05. (December 2002)
- [14] RTCA. Software Considerations in Airborne Systems and Equipment Certification, document RTCA/DO-178B. Washington, DC: RTCA (December 1992)
- [15] Shaw, M., Garlan, D.: Software Architecture: Perspectives On An Emerging Discipline. Prentice-Hall, Englewood Cliffs (1996)
- [16] Strunk, E.A., Knight, J.C.: Dependability Through Assured Reconfiguration in Embedded System Software. IEEE Transactions on Dependable and Secure Computing 3(3), 172–187 (2006)
- [17] Weaver, R.A., Kelly, T.P.: The Goal Structuring Notation - A Safety Argument Notation. In: Proc. of Dependable Systems and Networks, Workshop on Assurance Cases (July 2004) <http://www-users.cs.york.ac.uk/tpk/dsn2004.pdf>
- [18] Wojcik, R., Bachmann, F., Bass, L., Clements, P., Merson, P., Nord, R., Wood, B.: Attribute-Driven Design (ADD), Version 2.0. Technical report CMU/SEI-2006-TR-023 (November 2006)

Towards Evidence-Based Architectural Design for Safety-Critical Software Applications

Weiham Wu and Tim Kelly

Department of Computer Science, The University of York, York YO10 5DD
{Weiham.Wu, Tim.Kelly}@cs.york.ac.uk

Abstract. Robust software and system architectures have been increasingly recognised as one of the keys to improving dependability. However, most modern design methods and explanations of underlying design principles still remain ad hoc. The communication between design and safety assessment in practice is often characterised as an “over-the-wall” process. The problems are exacerbated by the uncertainty problem in the early development lifecycle. In this paper, we propose a Triple Peaks process framework, from which a system model, deviation model, mitigation model are proposed and linked together. The application of this framework is supported by the use of Bayesian Belief Networks and collation of relevant evidence. We elaborate the linkage between the three models by means of a case study. The central tenet in this paper is to address safety concerns based upon evidence available at an architectural level.

Keywords: software architecture, design decisions, software safety evidence.

1 Introduction

1.1 Motivation

For many years there has been an objective to improve software and system safety. Testing and inspection late in the system development lifecycle should no longer be relied upon as the primary line of defence for engineering software systems of significant size and complexity. Empirical experience shows that problems identified in the late lifecycle are often costly to fix and may introduce unexpected new problems [17]. Robust software and system architectures have been increasingly recognised as one of the keys to improving safety.

However, most modern architectural design methods and explanations of underlying design principles remain ad hoc. Architects or designers, who could claim in their defence that they adopted a specific design pattern or followed an industry standard, rarely articulate their design rationale and analyse the impact of their decisions along with design alternatives in a precise and sound manner. The communication between design and safety assessment in practice is often characterised as an “over-the-wall” process [19]. The problems are exacerbated by the presence of a high degree of uncertainty in the design detail that is available early in the system development lifecycle.

1.2 Software Safety Evidence

The development of software safety evidence is increasingly advocated in the safety community [38] to explicitly evaluate the safety of software, as opposed to relying on process prescription through safety standards such as IEC 61508 [3] and DO178B [5]. The tenet of using software safety evidence is straightforward: evidence shall be provided for assessors to demonstrate sufficient mitigation of risks associated with the use of software in safety-critical systems. The term “sufficiency” has been defined and deployed in a variety of risk acceptance regimes in the domain of risk management Risk mitigation has been generalised in terms of the following activities: hazard elimination, hazard reduction, hazard detection and control [35]. In principle, like other system components, software can only contribute to hazards in the system context by means of deviations from its intended behaviour [35]. Thus, it is possible to bring together the notion of “deviation”, “mitigation”, and “risk acceptance” with the aid of “evidence”. Here we define an item of software safety evidence to be an object encapsulating knowledge about potential deviations, plausible mitigation options and estimated risk reduction, along with reference to partial specification knowledge about a system and its environment.

Very often, safety evidence is produced after design completion. The need for incremental construction of safety evidence and corresponding safety arguments (a.k.a., safety cases) has been increasingly recognised. By utilising structured safety evidence explicitly from the very beginning of the system development lifecycle, the key issues such as loss of safety rationale and late discovery of safety flaws may be addressed. Figure 1 shows an evidence-oriented development process proposed by the Australian software safety standard DefAust 5679 [2], in which consideration of safety case development starts from the earliest stage of system development.

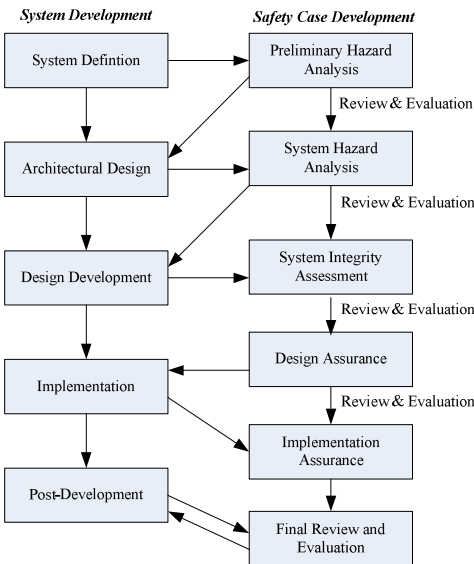


Fig. 1. The integrated development process (adapted from [2])

However, the linkage between the two processes (e.g., moving from preliminary hazard analysis to architectural design) still remains undefined, especially for software. Existing system safety approaches such as those advocated by ARP 4754 focus on the hazard analysis of purely functional requirements (i.e. Functional Failure Analysis – FFA), from which quantitative failure targets are defined and allocated, thereby driving the development of system and software architectures. Experience in application of FFA to engine controller development has revealed this technique is particularly vulnerable, as there is lack of rigorous techniques to identify and estimate controller-related failures with respect to levels of design detail [9]. The SEI (Software Engineering Institute) at Carnegie Mellon University has established a design method termed Attribute-Driven Design (ADD) [11] to emphasise the active role of quality attributes in architectural design. Yet there is little practical guidance on how to address safety concerns using ADD. Furthermore, the nature and amount of evidence changes as the design process progresses. Systematic techniques should be provided to handle these evolution issues within a structured evidence framework.

1.3 Scope of Paper

In attempting to integrate architecture design and safety evidence development processes in an effective manner, the interactions between the two must be elaborated – i.e. how requirements are evaluated by risk assessment, and design choices justified and design decisions driven by safety tradeoffs. In this paper, we introduce a Triple Peaks model as a framework for architectural design for safety. The process we present intertwines partial system specification, potential deviation concerns and plausible mitigation mechanisms incrementally and iteratively, through which incremental construction of safety arguments is facilitated and exploited. While risks may be communicated in a qualitative manner, they must be evaluated quantitatively, even in the early system development stages. We adopt Bayesian Belief Networks (BBNs) as the flexible medium for risk-based reasoning. The causal nature of BBNs allows reasoning about the propagation of software deviations and the effect of mitigation chosen. We illustrate the process by means of an aircraft wheel brake system (WBS) controller example extracted from ARP 4761 [1] and evaluate our process by comparison with the conventional ARP approach. We argue that systematic treatment of software safety evidence – guided by the Triple-Peaks model – holds the key to gaining confidence in safety-related architectural decision making.

This paper is organised in the following six sections. Section 2 reviews related work. Section 3 describes the process model within the evidence framework. Sections 4 and 5 elaborate the linkages between the three models by means of the WBS example. Finally, section 6 discusses the findings based upon the case studies conducted and section 7 presents the summary and future work.

2 Related Work

In order to inform architectural decisions, the first essential step for an architect is to interpret system and software requirements so that they can be understood. Goal-oriented methods, pioneered by van Lamsweerde [30, 31], proposed the use of goal

modelling languages such as Knowledge Acquisition in Automated Specification (KAOS) [31] to guide requirements elicitation and refinement. Alternatively, scenario-based approaches have been proposed in the form of use cases [24], Use Case Maps (UCMs) [14] and sequence diagrams [6] to elaborate requirements over a known system structure. Goals and scenarios are complementary to each other and can be combined in the design process [7]. Arguably, requirements to be addressed at an architectural level include both functional and quality requirements (e.g., timing, accuracy or reliability targets). Chung et al's Non-Functional Requirements (NFR) framework [40] and SEI's quality-attribute scenario framework [11] have been proposed for the purpose of formulating the quality requirements.

From the perspective of architecting dependable software, it is equally important to address all possible negative requirements. In contrast with (positive) requirements, negative requirements describe the system characteristics that are not allowed or desired. Nevertheless, the formulation of negative requirements and the relationship to positive requirements had not been explored until a decade ago. Potts et al informally proposed the notion of obstacles that might challenge the achievement of requirements within the Inquiry Cycle framework [45]. van Lamsweerde & Letier extended the KAOS language to incorporate the notion of obstacles as goal violation and provided heuristics on obstacle analysis over goals and resolution [32]. van Lamsweerde elaborated the obstacle framework further through anti-goals and anti-models in the context of security [29]. Rather than dealing with negative requirements at a goal level, complementary approaches have extended the notion of scenarios for the same purpose. Previous work at York [9] developed a method for deriving functional hazards from use cases. Alexander later proposed a unified view of deviation analysis over use cases in the form of misuse cases [8]. The safety community also turned their attention to extending hazard analysis at the requirements level to identify safety-related requirements errors. de Lemos et al [33] proposed an integrated framework that facilitates requirements analysis and hazard analysis iteratively and incrementally. Leveson et al [36] proposed to combine a set of hazard analysis techniques into an integrated safety analysis for checking safety-related requirements errors. All the approaches are defined solely in the context of requirements without considerations of architectural characteristics.

Given the positive and negative requirements formulated, the plausible design space must be elicited in order to capture the appropriate architectural choices. In the early 90s, Lane [48] proposed a multidimensional design space, each dimension representing relevant design choices to achieve a specific usability property. The SEI later developed a tree-form design space in terms of architectural tactics with respect to six common quality attributes [11]. The linkage between quality attributes and design space was also elaborated by SEI through the notion of quality-attribute reasoning frameworks [10]. A quality-attribute reasoning framework encapsulates knowledge about relevant analytic models for a quality attribute. For example, a performance reasoning framework imposes constraints on the relevant parameters for various performance measures such as a hard deadline. The collection of these frameworks thus offer an effective means of predicting system qualities and rationalising the selection of tactics. However, no specific techniques are provided for addressing the uncertainty and levels of design detail available in the early lifecycle.

From the viewpoint of safety, the reasoning framework should be built upon risk assessment. NASA have developed a probabilistic risk assessment (PRA) scheme [49] for more than two decades. The PRA approach is based upon the combination of fault tree analysis (FTA) [51] and event tree analysis (ETA) [35] and mandates a generic risk quantification and mitigation process that can be tailored to all phases of a project lifecycle. We believe that the burden of combining FTA and ETA can be relieved by the use of BBNs as a unified model. At the Jet Propulsion laboratory, a lightweight approach to risk assessment was developed, namely Defects Detection and Protection (DDP) [18]. The DDP scheme mandates the quantification of requirements, failure modes and mitigation by means of expert judgement. The underlying formal model of DDP is less justified, however.

Early work on argument-based design rationale (e.g., [16] and [46]) developed a set of generic models of design processes in terms of three common elements: issues/questions, positions/options, and arguments/criteria. The three elements are consistent with our design scheme as described in the section 1.2. The issues represent knowledge about deviations, the positions capture knowledge about possible mitigations, and arguments feature knowledge about reasoning. While the proposed design rationale models capture most common situations of design, they are too general to be configured for software architecture design problems. We believe there is a need to elaborate further the linkage of the three elements: i.e., how to generate issues, how to move from issues to positions, and then from positions to arguments.

The relationship between requirements engineering and architectures has recently been studied. Brandozzi and Perry [13] proposed the use of “architectural prescriptions” to describe the mappings between goals and architectural structures. Jackson et al [23] extended the problem frames approach to allow architectural decisions to be considered in requirements models in terms of “architectural frames”. Both approaches have explored the achievement of system functionality rather than system qualities. Nuseibeh [41] proposed a Twin Peaks model which explicitly features the challenges raised during the parallel development of requirements and architectures. Leveson [34] proposed the intent specification approach to deriving software safety requirements. An intent specification comprises two main dimensions: intent and part-whole dimensions. The two dimensions dictate the requirements and safety-related design decisions made respectively in the design process. Nevertheless, there is lack of practical guidance on how to generate intent specifications.

The notion of BBNs was developed by combining probability theory and graph theory [44]. A BBN represents a directed acyclic graph together with associated conditional probability distributions based upon explicit independence assumptions, thereby saving space of probabilistic computation [44]. In practice, BBNs are often interpreted as causal models [44], in which the directed edges are captured by knowledge about causal relations. Several tools such as Netica [4] are also available for public evaluation. BBNs have already been applied to solving software engineering problems. Fenton et al [50] developed generic BBN patterns to quantify software safety risks. Sutcliffe et al proposed a method of constructing generic BBNs to evaluate usability [20] and later developed an automated tool to evaluate reliability and performance through different configurations of BBNs [21]. Bosch and Gurf [22] proposed a generic BBN model as a software architecture evaluation framework.

However, most of the BBN models developed are generic and thus need to be tailored for a specific system domain.

At York, there has been a long-term objective to integrate software design and safety analysis. A decade ago Fenelon et al [19] proposed a prototype of compositional failure modelling language – Failure Propagation and Transformation Notation (FPTN). In our previous work, we developed a collection of safety tactics [55] as primitive building blocks for software safety design. In order to identify safety concerns, we have also proposed a method for deviation analysis over UCMs [52]. We further elaborated it by developing a negative scenario framework and mitigation action model [54] to help generate design options for the safety concerns formulated. We also examined the application of Communication Sequential Processes (CSP) as the implementation of FPTN for the purpose of architectural feedback [53]. Although CSP can capture nondeterminism in both a qualitative and quantitative manner [39], as a behaviour modelling language it is inadequate for capturing and evaluating evidence for the purpose of risk assessment. The work outlined in this paper is intended to offer a unified view of our previous work and address the need for risk-based quantitative reasoning through the application of BBNs.

3 Evidence-Oriented Method Construction

We treat design as an iterative and incremental process of producing evidence in which a system-to-be and its domain are *better* understood, *credible* deviation concerns are *exhaustively* identified and *sufficiently* mitigated by design options chosen, as Figure 2 suggests. Consequently, the design process comprises a number of design stages, each representing a cycle of moving from system modelling to deviation modelling and then mitigation modelling. The system model characterises system behaviours in terms of goals and scenarios and system structures with respect to viewpoints. The deviation model features the negative counterparts in terms of anti-goals and negative scenarios. The mitigation model captures possible design space in terms of mitigation actions to help inform decision-making. The proposed Triple-Peaks model is based upon Nuseibeh's Twin Peaks model and elaborates further the interactions between the requirements and architecture models by means of the deviation and mitigation models. From the viewpoint of evolutionary design, co-existent nature of the requirements and architecture models makes it possible to merge them into a single system model, thereby forming the Triple Peaks model.

At every single stage, appropriate evidence should be provided to justify the 'state' of the design progress – how safe the system-to-be would be given current knowledge and evidence. At York we have developed Goal Structuring Notation (GSN) [27] for communicating safety arguments. The items of evidence and their relationships to safety claims are described in terms of goal structures. Figure 3 shows the principal symbols of GSN. Goals can be refined by the aid of specific strategies. The goal refinement process stops when the goals can be satisfied by evidence available. Modular construction of GSN models is facilitated by the notation of 'Away Goal' and 'Module References' [26]: an away goal is a goal that is not defined (and supported) within the module where it is presented but is instead defined (and supported) in another module; a module reference is simply a goal structure packaged

in a module form. The development of the goal structures should proceed in parallel with the design process and reflect the progress of design. In other words, it is possible to use the goal structures to guide the development process. In GSN terminology, generalised goal structures can be captured by GSN patterns [27].

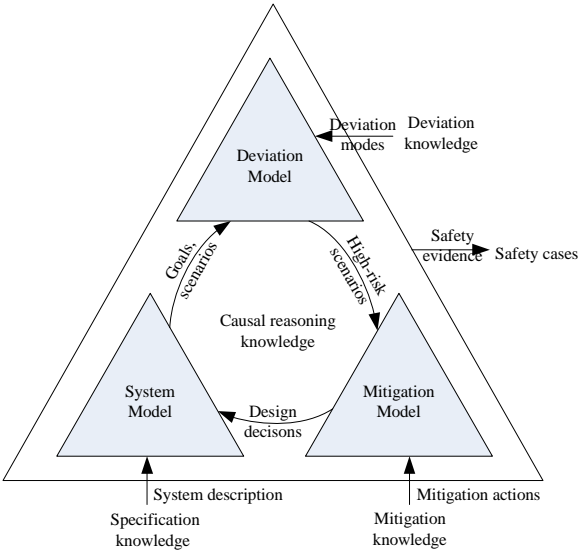


Fig. 2. The Triple-Peaks model

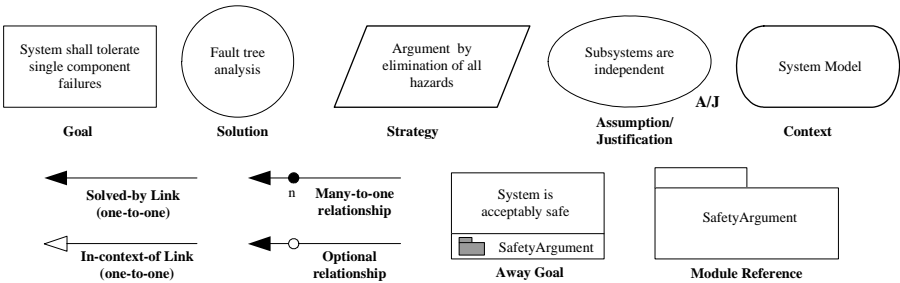


Fig. 3. The principal symbols of GSN

Figure 4 illustrates a sample GSN pattern for describing a system-independent design process when the architect is required to identify all anti-goals and relevant negative scenarios (as described in section 4.4 and 4.5) derived from a single system goal, and choose appropriate avoidance actions to mitigate them. The claims that the negative scenarios could not occur are satisfied by the analysis results of current development process. Justification of the completeness of the anti-goals is based upon the breadth of considerations of deviation modes. Credibility of the negative scenarios identified is evaluated through BBN modelling. By developing GSN patterns tailored

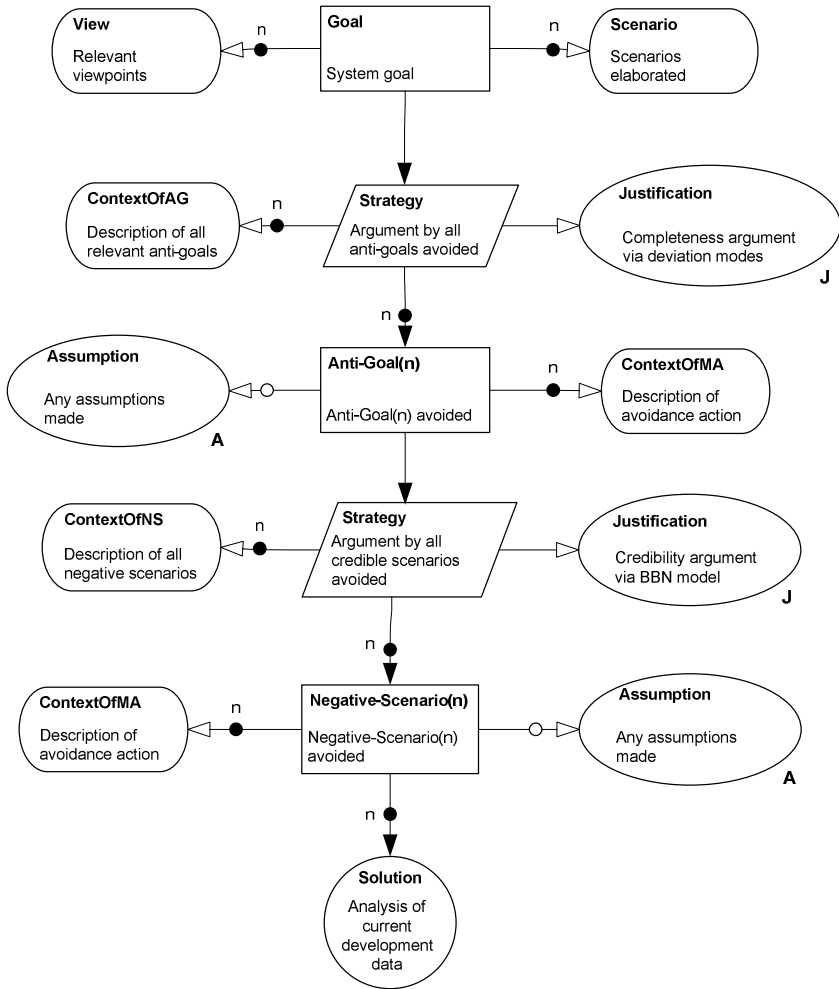


Fig. 4. A simple GSN pattern for the Triple Peaks process

for a specific system domain, system-specific development process model can be instantiated to guide architecting safety-critical software applications.

The proposed Triple Peaks framework represents much of the existing, but implicit, design practices in the dependability community. In order for the deployment of this framework to be successful, the transitions between the three models must be elaborated. Questions may be raised, for example:

- How do we capture and express the model elements such as goals and anti-goals within this framework?
- How do we reason about the properties of the model elements such as completeness, credibility and sufficiency?

Sections 4 and 5 elaborate the linkage between system model and deviation model, and between deviation model and mitigation model, respectively. The WBS example introduced in ARP 4761 is also used to illustrate our approach.

4 Moving from the System Model to Deviation Model

The system model comprises the description of the system under design. There are three essential elements of the system model within our framework: goals, scenarios and viewpoints. While a system model captures the desired properties of a composite system, the deviation model features the undesirable states of the system. In contrast with goals and scenarios in the system model, there are anti-goals and negative scenarios in the deviation model.

4.1 Goals

A goal is an objective that a composite system should meet. Through seeking goals explicitly from the requirements specification, the *core* system functionality and qualities can be effectively elicited and justified. Despite the diverse forms of goal formulation techniques (as discussed in section 2), there are four common elements of a goal we found:

- *Artefact*. The artefact is the composite system or its parts onto which a goal is applied.
- *Context*. The context are the pre-conditions that a goal refers to and evolves over.
- *Stimulus*. The stimulus is the trigger condition for the initiation of a goal.
- *Response*. The response captures the desired properties (i.e., postconditions) that the artefact should hold over time. Quality constraints (e.g., deadline or failure rate) can be specified in this part if they exist.

A sample goal can thus be expressed in the following form:

```
"The <artefact> shall <respond> upon <stimulus> when  
<context>"
```

This goal formulation is consistent with the SEI's quality attribute scenario framework and thus can be applicable to both functional and quality goals. As an example, consider the wheel braking system (WBS) of an aircraft [1]. We assume there are a number of top-level goals that can be stated in terms of aircraft functionality (e.g., controlling the aircraft on ground in this case) and qualities (e.g., safety). Each goal can be formulated in the stimulus-response form in spite of their high level of abstraction. Each functional goal can be decomposed further into a set of sub-goals and should evolve separately given that they are independent. The goal decomposition may be guided by the use of scenarios, as described in the next subsection. Goal structures can thus be constructed. Safety goals cannot simply be decomposed via functional goals or system structures; their refinement is based upon the results of deviation analysis (see sections 4.4 and 4.5) and the chosen mitigation. For example, deviation analysis may reveal that a 'late' output of a controller is safety

significant. A performance goal is thus derived and added into the safety goal structures.

Figure 5 shows a part of the goal structure in which the core functionality of WBS is elicited. All the goals in this structure are expressed using the above form. The expression language used is a structured natural language, and some expression can be very abstract at this level. For example, both the stimulus and response parts of the top-level goal *FnG1* are very general and need to be refined. This should be acceptable, however, in the early development lifecycle in which many requirements are volatile and unclear. Figure 6 shows the undeveloped goal structure of the aircraft-level safety goal. In most cases, the top-level safety goal (i.e., *SafeG1*) is simply derived from the certification authority. This achievement of this root goal is based upon the satisfaction of all the supporting safety-related functional and quality goals. Likewise, expressions of safety goals can be very abstract, as the concrete forms of deviations and mitigations are still unknown. It can be seen that the modular features of GSN makes it feasible to isolate development of different goal structures (e.g., functional and safety goal structures) and link them effectively via the notation of Away Goal and Module References.

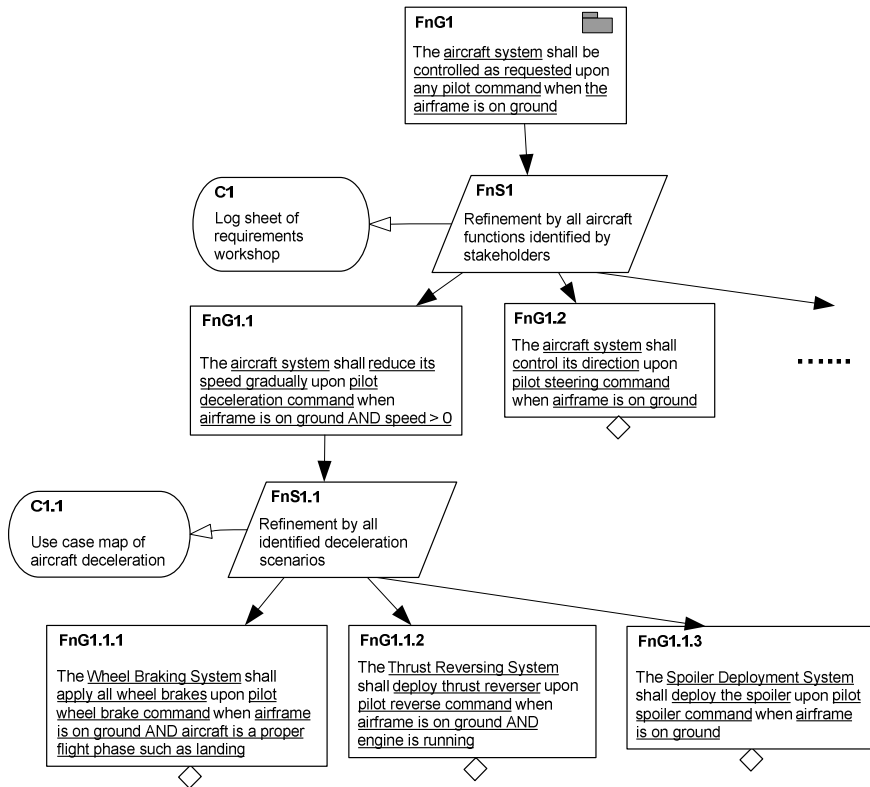


Fig. 5. A functional goal structure for the WBS example – Control the aircraft on ground

4.2 Scenarios

A scenario is a sequence of actions performed by objects instantiated within the known structure of the composite system. Scenarios provide an effective way to elaborate a goal in a white-box view. In other words, a goal defines a set of possible scenarios; a scenario defines a possible realisation of a goal. There are many forms of scenario formulation (see section 2). The use of UCMs is preferable based upon our experience, as it offers a clear-cut notation of causal relationships between architectural components in terms of responsibility points [14]. This is beneficial when we conduct deviation analysis over scenarios, as described in section 4.5. UCMs can be refined, as the underlying system structure is developed in more detail.

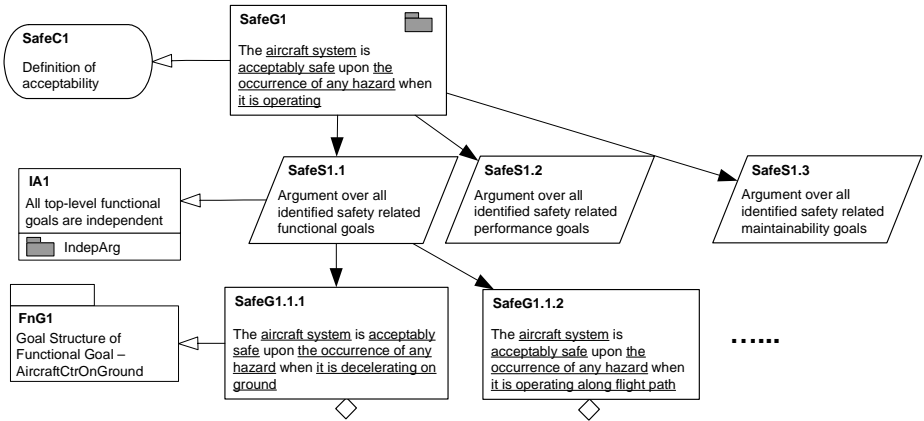


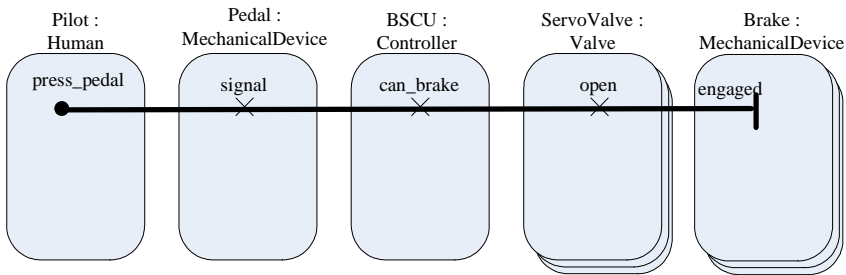
Fig. 6. The top-level safety goal structure for the WBS example

In order to define a scenario (i.e., a UCM model), a goal must be provided. A system structure (i.e., a *component context diagram* in UCM terminology) must also be defined as the context of elaborating the goal. The system structure should be derived from the predefined viewpoints as described in the next subsection. The system structures should be independent of the requirements allocated. This is another benefit of using UCMs, as component context diagrams and use case paths can be developed separately. For the WBS example, the wheel braking goal has been identified in Figure 5. The system architecture of the WBS has been defined in the system description (see [1]) as the input of the design process. Therefore we can elaborate the wheel braking goal through the system architecture. Figure 7 illustrates an example scenario for the wheel braking goal – manual braking in normal mode. As all relevant scenarios (in this example, manual braking in normal mode/alternative mode/emergent mode and auto braking) are elicited for a specific goal, the goal can be decomposed further given the responsibility points allocated. For instance, a sub-goal of the WBS goal will be the goal of BSCU expressed as follows:

The BSCU controller shall output the brake command upon arrival of pedal signal when airframe is on ground AND aircraft is in landing/taxing/RTO flight phase.

4.3 Viewpoints

In most cases, a system model has multiple structures due to the increasing size, complexity and heterogeneity of modern software systems [42]. In practice, these structures are classified in terms of viewpoints. An instance of a viewpoint is called a view (i.e., a system structure). Yet there is no consensus on the number of appropriate viewpoints in both research and practice that are considered necessary to describe software architectures adequately. From the viewpoint of embedded systems development, we define five essential viewpoints, as shown in Table 1. The first two are defined at system level to capture system boundaries and its physical structures. The remaining three viewpoints are defined at software level and consistent with common viewpoint approaches in the software community (e.g., SEI’s three viewtypes [15]). The recognition of multiple viewpoints has a significant impact on the completeness of deviation analysis, as viewpoints are interconnected and deviation arising from one view can propagate through another view (see section 4.5).



Preconditions:
- Aircraft is on ground
- Aircraft is moving
- Aircraft is in landing/takeoff/rejected takeoff phase

Postconditions:
- Eight wheel brakes are applied

Fig. 7. An example scenario – manual braking in a normal mode

Table 1. The information description of the five viewpoints

Viewpoint	Description	Intent
Contextual Viewpoint	How an embedded system interacts with its operating environment	To reason about the environmental properties of the system
System Architecture Viewpoint	How an embedded system is structured in terms of physical units. At least one of these units should be the controller or software	To reason about the physical characteristics of the system
Development Viewpoint	How the system’s software is structured in terms of implementation units	To reason about the software functions and maintenance
Run-Time Viewpoint	How the system’s software is structured in terms of run-time units	To reason about the runtime behaviours of the software
Allocation Viewpoint	How the system’s software is allocated onto non-software structures (e.g., hardware platform)	To reason about the impacts of the underlying hardware platform and development environment on software

4.4 Anti-goals

An anti-goal is a condition that, if true, would immediately prevent the composite system from achieving the corresponding goal. Both goals and anti-goals are complementary and thus capture the possible desired and undesired end states of a composite system respectively. A common example of an anti-goal is the loss of a system function where the function was a goal. Nevertheless, simple negation of a goal in terms of propositional logic cannot guarantee the *completeness* of the corresponding anti-goals. A less obvious but perhaps more severe anti-goal would be inadvertent application of that function. Given some goal formulation, it is important to ensure the exhaustiveness of deviations from that goal, at least from the viewpoint of safety. In the safety community, the possible deviations of a system are often characterised in terms of deviation or failure modes. Previous York’s work has developed a collection of deviation modes for software systems: SHARD guidewords [19]. We interpret the SHARD modes with respect to the goal formulation in the following Table 2.

Table 2. The anti-goal interpretation using SHARD guidewords

SHARD	Anti-Goal Interpretation
Omission	Response part does not hold while stimulus and environment parts hold
Commission	Stimulus or context parts do not hold while response part holds
Timing	Timing constraint specified in the response part is violated while the other parts hold
Value	Value constraint specified in the response part (e.g., accuracy or cost) is violated while the other parts hold

By allocating the SHARD modes onto a formulated goal and interpreting them using the above table, there can exist a high level of confidence on the exhaustiveness of the set of anti-goals elicited. Notably, not all anti-goals can have safety implications; anti-goals must be evaluated with respect to safety consequences (see section 5.1). Let us return to the WBS example. As soon as the system goals of WBS are formulated, the identification of anti-goals can start by considering the SHARD deviations first without information about the elaborated scenarios. In this example, only omission and commission modes are applicable. Table 3 illustrates an example anti-goal by negating the context part – wheel braking when the context is not as intended. The definition of the stimulus part is trivial in this case. The anti-goal elicited is abstract.

Table 3. An example anti-goal formulation

Portion of Goal	Possible Value
Artefact	WBS
Context	NOT (Airframe is on ground AND aircraft is in landing/taxiing/RTO flight phase)
Stimulus	N/A
Response	All wheel brakes are applied

By expanding the negation operation on the context part using Boolean logic, we can derive a set of well-refined anti-goals: e.g., wheel brakes applied when aircraft is taking off or when aircraft is in air. It must be stressed that the expansion here cannot

be achieved solely by formal Boolean logic and in many cases may need the help of domain experts. For the example of inadvertent wheel braking when the aircraft is taking off, we may need to distinguish further whether the aircraft is taking off before the decision speed $V1$, as the corresponding safety consequences would be different [1]. Obviously, this is impossible for formal logic alone to identify the two anti-goals. When all anti-goals are identified and refined (say, eight anti-goals for the WBS example), they should be linked to the anti-goals of the parent goal of the WBS (i.e., aircraft deceleration) in a bottom up manner, thereby forming an anti-goal structure. The anti-goal structure in the WBS example is shown in Figure 8.

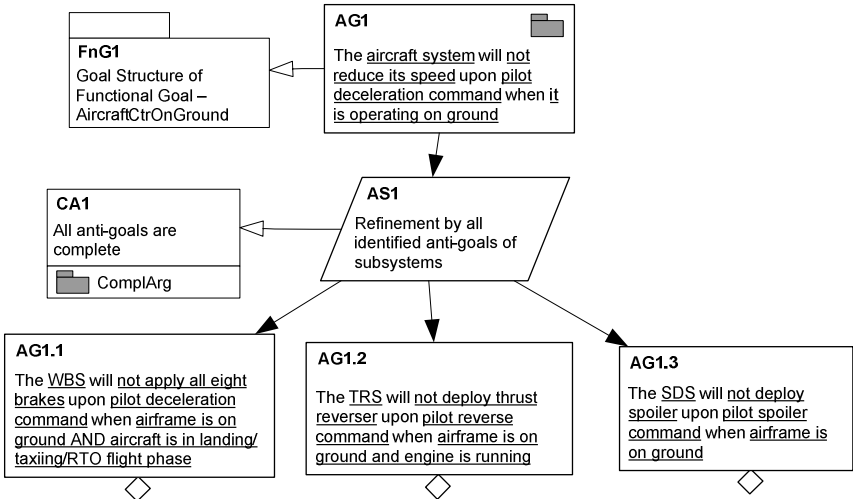


Fig. 8. The anti-goal structure – Total loss of aircraft deceleration

As seen in the Figure 8, the anti-goal structure should refer to the corresponding functional goal structure. It should be noted that the expression languages used in functional goals, safety goals and anti-goals are slightly different. The functional goals are simply requirements and thus ‘shall’ statements are suitable; the safety goals are claims in which ‘is/are’ statements are applicable; the anti-goals are hypotheses about states & events of the system and thus ‘will’ statements should be used. The construction of anti-goal structures will prompt the refinement of the safety-goal structure in which decisions need to be made regarding how to mitigate these identified anti-goals.

4.5 Negative Scenarios

Like a (positive) scenario, a negative scenario is a possible realisation of an anti-goal with respect to the known system structure. In other words, negative scenarios can be formulated using conventional scenario formulation techniques such as use case templates. We previously proposed a stimulus-effect form [54] to emphasise the status of causal propagation within a negative scenario. Not surprisingly, it can be seen as the negation of the stimulus-response form of goals and positive scenarios. To

ensure an exhaustive set of negative scenarios elicited, a broad spectrum of candidate stimuli must be considered. This spectrum should be based upon consideration of all *possible* deviations. Credibility of these deviations will be discussed in section 5.1. We here distinguish two classes of negative scenarios, which are consistent with the decomposition principle of fault tree construction [51]:

- *Primary negative scenarios.* Those stimuli are identified from deviation analysis over a positive scenario — a UCM model which is derived from a specific viewpoint. This is often done by inductive analysis or “what-if” questions conducted on every single action of a scenario.
- *Supporting negative scenarios.* Those stimuli are identified from deviation analysis over other viewpoints but with *possible* contribution to primary negative scenarios. For example, the failure of a run-time component in a runtime view can be caused by malfunction of underlying hardware platform in the deployment view.

Furthermore, the systematic identification of anti-goals offers an effective means of ensuring a broad spectrum of the end effects of the negative scenarios. By linking the candidate stimuli with anti-goals through the stimulus-effect framework, the exhaustiveness of negative scenarios can be justified. New anti-goals may also be identified during negative scenario development. Another concern is the effectiveness of the negative scenario elicitation. In practice, deviation analysis is aided by a set of pre-defined component deviation modes. For example, the common deviation modes for the components of valve type can be: stuck open, stuck close and leakage. Automated analysis is thus possible by extending system structures with component failure modes defined in the deviation knowledge base.

Let us continue the WBS example in which a wheel braking goal, four positive scenarios in a form of UCMs, and eight anti-goals have been defined. Now we need to perform deviation analysis over each of the four positive scenarios. Guidance on deviation analysis over UCMs can be found in our previous work [52]. Simply put, deviation analysis starts by forward search of each responsibility point across the use case path in order to identify possible primary negative scenarios. For a given responsibility point, deviation modes are allocated with respect to the type of the component in which the responsibility point resides, and the end effect of that deviation is identified along the use case path and linked to the identified anti-goals. The forward search procedure for a specific responsibility point is similar to ETA in which the initiating event is the deviation of that responsibility point and all possible event sequences are analysed along the use case path. Figure 9 illustrates the deviation analysis procedure for the scenario – the manual braking in normal mode.

The output of deviation analysis over the four UCM scenarios are fourteen primary negative scenarios. Supporting scenarios must be identified by deviation analysis over other views. To do this, we need to identify how the components in a UCM model can be mapped onto other views. In the WBS example, the UCM models are defined within the system architecture view, and all other views are still undefined. In this case, the architect needs to make some assumptions such as a uni-processor configuration and single monolithic software module in order for the remaining views to be produced. Put another way, the BSCU controller is first mapped to a single software module in both development and run-time views, and to a processor in the

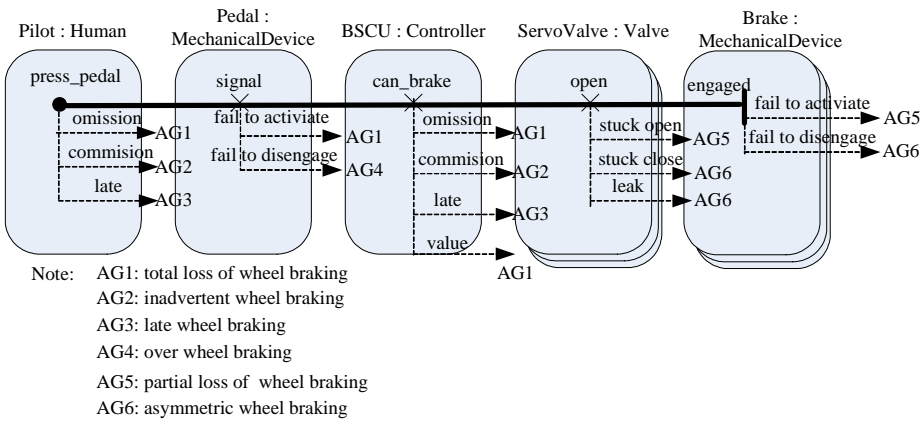


Fig. 9. Deviation analysis over the example scenario (manual wheel braking)

allocation view. Deviation analysis is then performed upon the two components by deviation mode allocation. Consequently, a set of supporting negative scenarios are identified. All identified negative scenarios should be formulated in the stimulus-effect form. Table 4 illustrates an example formulated negative scenario.

Table 4. An example negative scenario formulation

Source	Stimulus	Context	Course of Propagation	End Effect
BSCU	Fails to output brake command	Airframe is on ground AND aircraft is in landing/taxiing/RTO flight phase AND pilot presses the pedal	Eight servo valves, eight brakes	AG1: Total loss of wheel braking

5 Moving from the Deviation Model to Mitigation Model

All the negative scenarios and anti-goals are hypothesised on the basis of current knowledge about the system and its domain. Yet not all anti-goals identified are safety significant. Moreover, it is impractical to completely address all the identified safety concerns within a single design iteration. It must be possible to evaluate the deviation model in terms of safety risks. A negative scenario leading to a safety-significant anti-goal is a risk scenario. Only a small number of high-risk scenarios (say three) will be considered in the mitigation model for every single design iteration. As a result, the management of negative scenarios and assessment of acceptability must be a continuous process through the whole architectural design. The purpose of the mitigation model is to capture plausible mitigation space (i.e. a set of mitigation action candidates) against the high-risk scenarios identified through severity and credibility estimation. Cost-benefit analysis and design tradeoffs may be performed in order to make optimal decisions. The following subsections will describe our solutions to evaluating the deviation model using a BBN framework, identifying mitigation space and performing safety design tradeoffs.

5.1 Severity and Credibility

Like most risk assessment methods, the evaluation is performed in terms of severity and credibility estimation. Notably, we prefer the term “credibility” to the standard term “likelihood” or “frequency”, because the former is more consistent with the Bayesian interpretation of probability [25] that lies at the heart of our risk assessment framework. Severity estimation is conducted by considering the safety consequences of all identified anti-goals. Estimation proceeds from the anti-goal structures and takes into account of the contribution to their parent anti-goals if they exist. For the WBS example, the occurrence of the anti-goal *AG1.1* does not necessarily lead to the occurrence of its parent anti-goal *AG1* unless the anti-goals *AG1.2* and *AG1.3* also hold, as shown in Figure 8. In fact, the anti-goal *AG1.1* should be detectable by the pilot when the wheel braking is commanded and the pilot will be able to use spoiler and thrust reversal to the maximum extent possible in order to achieve deceleration. Hence, the severity classification of the anti-goal *AG1.1* should be hazardous rather than catastrophic.

Credibility estimation should be conducted over all the negative scenarios identified and formulated in a stimulus-effect form. There are two parts to be estimated: the stimulus and propagation parts – how credible is it for the occurrence of the hypothesised stimulus and its *propagation* to manifest the anti-goal? At the beginning of architectural design, it is plausible to make the worse-case assumption that the propagation is completely credible (i.e., its credibility is 1) given that no mitigation mechanisms are employed. Once mitigation actions are chosen against the propagation, the credibility of the propagation should be re-estimated with respect to the effectiveness of the chosen mitigation. Now our focus would be the stimuli of all negative scenarios. To do so, we first distinguish two classes of stimuli:

- Stimuli of a random nature. The sources of the stimuli will be non-agent objects such as hardware and natural environment.
- Stimuli of a systematic nature. The sources of the stimuli will be agents such as human and software.

For the first class of stimuli, the architect should seek historical data to justify their credibility. For the systematic stimuli, an analytic model should be constructed and evaluated through data collected from the real world. The selection of analytic models depends upon the classification of the stimulus: is it human operation error or software design fault? For the former, further investigation is required to check if it is a slip-related error or mistake-related error [47]. Task network analysis [28] may be chosen to predict the credibility of slip-related errors, for instance. For the software design faults, current design artefacts and the progress of development process must be considered. If fault data determined by testing are available, for example, Rome Laboratory’s software reliability prediction models [37] may be suitable. Alternatively, if product metrics (e.g., quality of requirements specification) and process metrics (e.g., competencies of the developers) can be estimated, Fenton’s defect prediction model may be applicable. Notably, no model is complete or even representative. One model may work well for a set of certain software, but may be completely off track for other kinds of problems. Assumptions and justification made during the selection procedure must be explicitly identified. Knowledge about historical data, the

classifications of analytic models as well as the applicability rules could be codified in a reasoning knowledgebase for the purpose of automation. If no applicable historical data and analytic models are available, expert judgements would be required.

Let us carry on the WBS example. An assumption of 100% propagation hold for all identified negative scenarios can be made. We then need to type-check the stimulus and source parts of every scenario. Analytic models are then selected for credibility estimation. Table 5 illustrates a portion of credibility estimation results for the WBS example. It should be noted that the results of credibility are by no means precise, as the level of design detail increases. In many cases, the credibility of the scenario will need to be updated as the design process progresses and subsequent design decisions are made upon the source of the stimulus. For example, the BSCU software module will inevitably be decomposed in more finer-grained modules in which the brake control responsibilities will be allocated. The update of evidence is possible to be incorporated within the BBN framework described in the next subsection. It must be stressed that the role of quantification is to prioritise scenarios instead of obtaining precise numerical data – through which dominant scenarios are identified and drive the design decision procedure.

Table 5. Example results of credibility estimation

Stimulus	Classification	Estimation Forms	Assumptions/ Justification	Credibility
Pilot fails to press pedal	Slip-related error	Task network analysis	Reason's human error classification [47]	1.5E-6
Power supply loss	Random failure	Historical data		1E-7
Existence of software fault	Design error	Fenton's defect prediction model [50]	No fault/failure data but some process and project management metrics are available at this stage.	2.5E-3

5.2 Causal Bayesian Modelling

A negative scenario is inherently a causal chain starting from a stimulus and leading to undesired end states (i.e., anti-goals). Through composing all identified negative scenarios together, a causal structure can thus be formed. In BBN terminology, a causal structure C is defined in a directed acyclic graph (DAG) form: $C = (V, E)$, where V is defined as a set of nodes, and E is defined as a set of directed edges among V . We distinguish further between two subsets of V : $V1$ and $V2$, where $V1$ corresponding to the set of all the leaf nodes, $V2$ represents the set of the remaining nodes such that $V = V1 \cup V2$ and $V1 \cap V2 = \emptyset$. Each element of the set $V1$ corresponds to a *distinct* anti-goal identified, whilst each element of the set $V2$ corresponds to a *distinct* architectural component identified from the source of stimulus and propagation parts in the negative scenario framework. Each element of E captures a *distinct* causal relation identified by knowledge about the sequence of propagation of a stimulus as specified in the negative scenario framework. Figure 10 illustrates a portion of a causal structure for the WBS example. To simplify the BBN computation, we remove the nodes *WheelBrake(i)* and *ServoValve(i)*, as our main focus here is the controller BSCU.

To form a causal model, the first step is to transfer all the nodes in V into variables in which their value domains must be defined. For elements in V_1 , their value domain will be simply Boolean true and false to indicate if an anti-goal occurs or not. For elements in V_2 , their value domains will be their deviation modes allocated and the normal mode (say, ok) during deviation analysis, as described in section 4.5. Finally, we need to define a conditional probability table (CPT) for each variable with respect to the credibility estimation results. Often, this would be a tedious step. However, many BBN tools such as Netica allow us to define probability table by equation. Figure 11 shows the equation definition for the BSCU variable using Netica expression language. In general, careful justification is required when defining a CPT. These judgements need to be captured in the safety arguments (i.e., GSN forms). The following are a number of rules learnt from our experience:

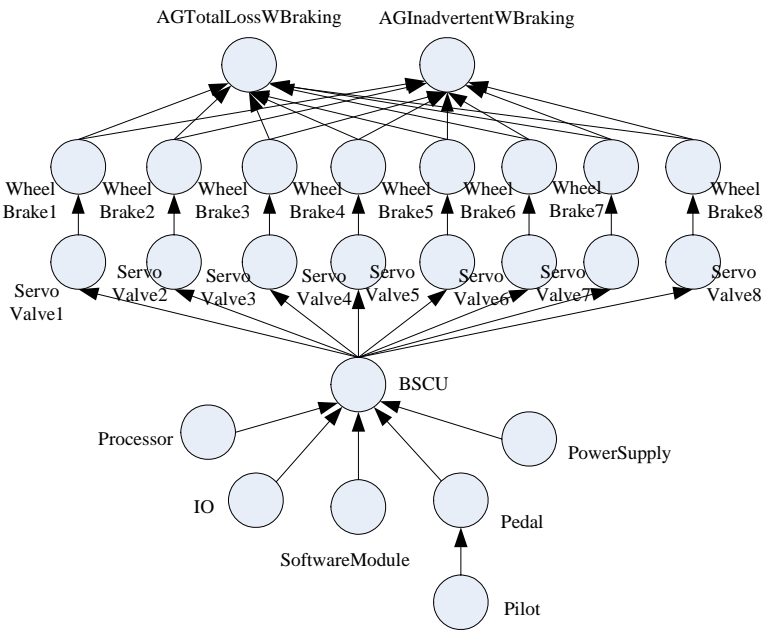


Fig. 10. The causal structure for WBS example

- During the elicitation of negative scenarios, a single deviation is considered as the stimulus for one scenario. But when composing scenarios to form a causal model, the occurrence of multiple deviations must be taken into account. For the WBS example, what if both power supply loss and processor failure happen simultaneously? Obviously, the effect of process failure will not be exhibited by the occurrence of power supply loss, thereby leading to no output of BSCU.
- Some deviations such as transient failure of processor or software faults may have multiple effects in a non-deterministic manner [53]. In those cases, the architect needs to make some assumptions: e.g., all chances are equal.

```
P (BSCU | CPU, IO, SW, Pedal, Power) =
(Power == ok && CPU == ok && Pedal == ok && SW == ok && IO == ok) ? (BSCU == ok ? 1.0 :
0.0) :
(Power == loss) ? (BSCU == fail_to_output_brake ? 1.0 : 0.0) :
(Power == ok && Pedal == fail_to_activate) ? (BSCU == fail_to_output_brake ? 1.0 : 0.0) :
(Power == ok && Pedal == fail_to_disengage) ? (BSCU == ok ? 1.0 : 0.0) :
(Power == ok && Pedal == activate_inadvert) ? (BSCU == output_brake_inadvert ? 1.0 : 0.0) :
(Power == ok && Pedal == ok && CPU == crash) ? (BSCU == fail_to_output_brake ? 1.0 : 0.0) :
(Power == ok && Pedal == ok && IO == crash) ? (BSCU == fail_to_output_brake ? 1.0 : 0.0) :
(Power == ok && Pedal == ok && IO == transient_failure) ?
(BSCU == fail_to_output_brake ? 0.3 : BSCU == output_brake_late ? 0.5 : BSCU == ok ? 0.1 : 0.1) :
(Power == ok && Pedal == ok && (CPU == transient_failure || SW == faulty)) ?
(BSCU == fail_to_output_brake ? 0.3 : BSCU == output_brake_inadvert ? 0.3 : BSCU ==
output_brake_late ? 0.2 : 0.2) : 0
```

Fig. 11. The equation expression for the BSCU variable

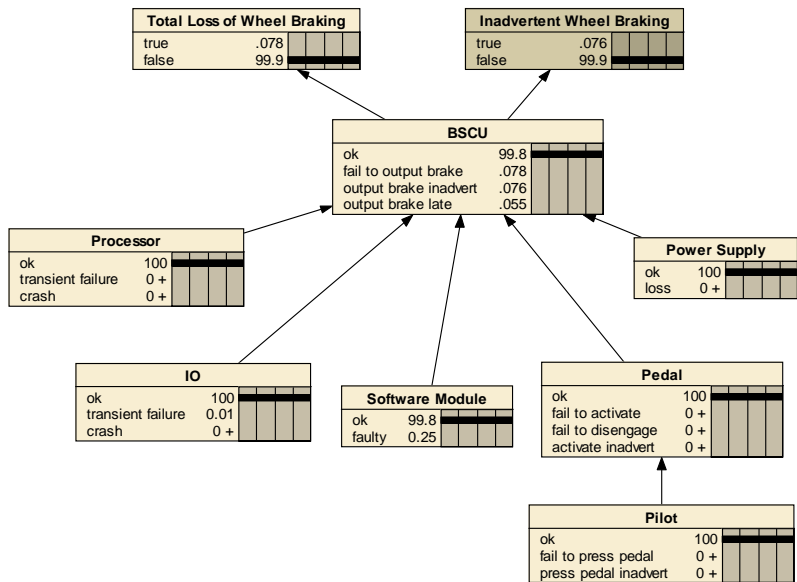


Fig. 12. The BBN evaluation results using Netica

- Like programming, comments for each statement (though not shown in the figure) will be provided to enhance understanding and readability.

Figure 12 shows a fragment of the compiled BBN model produced by Netica. Note that the belief numbers shown in the figure are based upon percentage and some numbers are not shown completely (i.e., 0+) due to limitations of display. For example, the probability of the anti-goal “Total Loss of Wheel Braking” calculated is 0.0007846, which falls short of civil aviation target 1E-7. Therefore, mitigation is required. To do so, we need to identify the high-risk scenarios. This can be done

automatically by sensitivity analysis through Netica. There are three high-risk scenarios, which must be considered in the mitigation model:

- S1. Residual faults in the BSCU software leading to malfunctions of the BSCU
- S2. I/O (transient or permanent) failures leading to malfunctions of the BSCU
- S3. Processor (transient or permanent) failures leading to malfunctions of the BSCU

5.3 Mitigation Space and Safety Tradeoffs

We previously developed a mitigation action model in which mitigation actions are organised in a tree form of five branches (i.e., *elimination*, *reduction*, *detection*, *resistance* and *minimisation*) and codified by a template, which can be implemented in a mitigation knowledgebase [54]. If we treat all the codified mitigation actions as the whole design space, our concern then lies at how to identify the *most appropriate* subset of these actions with respect to a specific negative scenario. Since negative scenarios are formulated in a form of causal chains, a plausible mitigation space will be defined by means of *controllable* parts of the causal chain for the purpose of stopping the propagation. Therefore, the identification of the mitigation space is an iterative procedure of locating the reachable BBN non-leaf nodes (i.e., architectural components) with respect to a given anti-goal and searching applicable mitigation action branches. For the WBS example, consider the scenario S1 identified in the previous subsection. Clearly, the mitigation options would lie within the *software module* and *BSCU* nodes. Mitigation actions applied to the BSCU can also help address scenarios S2 and S3. We then need to determine which of the five branches of mitigation will be applicable. For instance, the *minimisation* action branch (i.e. minimisation of the effect of total loss of wheel braking) is irrelevant, as there is nothing to act when BSCU fails to output the brake command as requested. The procedure continues until all action candidates are located. Table 6 illustrates the mitigation options for S1, accompanied by their rationale.

Table 6. A mitigation space for the negative scenari S1

ID	Node	Branch	Mitigation	Intent
S1	Soft-ware module	Eliminat-ion	Simplification	Correctness of software design can be verified
		Reduct-ion	Rigorous testing	A amount of faults can be detected by testing
			Following safety standard	The process for high Development Assurance Level (DAL) [5] produces ‘better’ software
			Functional redundancy	The likelihood of faults in different designs is sufficiently low
	BSCU	Detection	Timeout	No response of the BSCU is assumed to fail
			Comparison	Deviations can be detected in case of discrepancy
			Voting	Deviations can be detected and tolerated in case of discrepancy
		Resistan-ce	Recovery	Any error detected can be fixed
			Reconfiguration	Any error detected can be removed by replacement
			Degradation	Any error detected can be removed by removal of the faulty component

To make decisions in response to the mitigation space identified, cost-benefit-risk analysis must be performed. The benefit of each action candidate is determined in terms of its impact upon the credibility of the scenario that it is intended to address. The cost of each option is estimated in terms of orders of magnitude. Both cost and benefit estimates usually require the aid of domain experts and past experience. The known vulnerabilities and side effects of each option are identified by the use of the codified mitigation knowledge. As an example in Table 6, the effectiveness of deploying functional redundancy to reduce software faults lies at a high degree of diversity between module designs, which may be hard to implement in practice. This can be done by means of tables [54] and the architect is free to choose specific options based upon judicious considerations of the mitigation space. It must be stressed that our method does not make decisions for the architects but aids them in eliciting and rationalising their design decisions. For the WBS example, we chose the simplification tactic against the dominant scenario S1 because of the limited number of software input, and comparison and reconfiguration tactics against S2 and S3 by assuming a stringent cost budget. Once mitigation actions are chosen, the system model needs to be refined in the following possible ways:

- Add new components or remove existing components in specific view(s).
- Add new responsibilities (a.k.a., derived requirements) in specific view(s).
- Re-allocate existing responsibilities in specific view(s)

For the WBS example, the correctness of the monolithic software module in the development view must be verified. In this case, no refinement of software module is required by this decision. However, non-safety such as modifiability-related design decisions can drive the decomposition of the monolithic module into a control function module and compiler module so that the BSCU software can be portable to different compilers. Two dual processors and buses are introduced in the allocation view and the output of the BSCU is arbitrated, as indicated by the chosen comparison tactic. Transient processor/bus failures can be repaired by rebooting the processor and reloading its copy of software. Behaviours regarding the run-time comparison and reboot behaviours must be captured in the scenario forms. At this point, both the structures of functional goals and safety goals can be decomposed further to reflect the refinement of the system model and derivation of safety requirements.

Likewise, a new deviation model will be generated upon the refinement of the system model. In most cases, the step of identification of anti-goals can be skipped unless new system goals (e.g., system monitoring) are identified. The main focus is thus the elicitation of new negative scenarios. For the WBS, example negative scenarios elicited can be failure of both processors and the use of potentially faulty compiler. The process loops until all the core system requirements have been elicited and all identified anti-goals are mitigated sufficiently with respect to the risk acceptance. A stable architecture is therefore formed at the end of the design process.

6 Discussion

We have so far applied the proposed framework to a number of medium-size case studies such as AGV [54] and WBS systems. The WBS example was selected for the

purpose of comparison with the ARP process which has been widely applied in practice. We discuss our results in terms of the following three aspects:

Exhaustiveness. Establishing an argument of exhaustive identification of the design issues (i.e., negative requirements) and design alternatives is recognised as the key to the robustness of dependability design. Within the ARP framework, negative requirements are identified through deviation analysis purely on system functions (i.e., FFA) and refined through FTA during architecture definition. The transitions from FFA to FTA and from system level to software level are undefined, however. The exhaustiveness argument is thus implicit and cannot be validated. The techniques presented in our approach provide semi-formal and multi-viewpoint support for deviation identification: deviation analysis is started at goal level through top-down goal formulation and refined at scenario level and component level through pre-defined architectural viewpoints in a bottom-up manner. Moreover, there is no step for identification and justification of design alternatives in the ARP process. The exhaustiveness argument is inherently limited and subject to the competencies of domain experts and the past experience of architects.

Effectiveness. Most software standards advocate heavyweight development approaches in which the upfront specification of all system requirements is mandatory before the design proceeds [12]. For the ARP example, a sequential ‘waterfall-like’ process is defined, starting from aircraft-level FFA, system-level FFA to system-level FTA and software-level FTA, from which system and software safety requirements are derived and architectures are validated. This form of waterfall model has been known to be inadequate for handling the volatility and uncertainty commonly involved in the real-world problems. The Triple-Peaks process presented in this paper inspired from the Twin-Peaks model addresses requirements specification, design issues and corresponding design alternatives iteratively and incrementally. The process is receptive to requirements change, as only core system requirements are analysed and achieved by a stable architecture defined. Incremental construction of safety evidence is facilitated by means of GSN in a top-down manner, though it remains to be seen whether the explicit recording of safety arguments is best done in order to ‘fake’ a rational design process as described by Parnas and Clements [43]. The effectiveness of the design process is also enhanced by available knowledge sources such as deviation modes, component deviation types and mitigation tactics, though there is still lack of tool support available to integrate these techniques.

Rationality. Existing software design approaches often rely upon implicit reasoning, through which design decisions are mainly promoted by design intuition. The linkage between design decisions and requirements is largely undefined. Though the design decisions are clearly identified within the ARP process, the steps moving from the identified safety requirements to design decisions are still unclear. Our approach elaborates the linkage between safety requirements and mitigation options chosen by means of requirements formulation and prioritisation, design space analysis and cost-benefit-risk analysis. With the aid of BBN tools, the notion of credibility can be deployed in design and a level of confidence can thus be established. Deciding the

stopping rules of the design process (i.e., when design issues identified are complete and mitigation is adequate) is based upon risk acceptability criteria.

7 Summary

In this paper, we have presented an integrated approach to architectural design for safety-critical software applications through a Triple Peaks framework. In particular, we have demonstrated that how it is practical to conduct deviation analysis simultaneously at both the requirements and architecture level. The key principle underlying this paper is that software safety evidence must be collected in the early development lifecycle, and architectural design decisions must be informed based upon this evidence & quantitative risk assessment. Our future work includes seeking possible automation of the linkage between BBN models and architectural choices, and expanding the proposed method into other critical domains such as security-critical software applications.

References

1. ARP 4761: Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment, Society of Automotive Engineers, Inc. (1996)
2. Australian Defence Standard Def(Aust) 5679: Procurement of Computer-based Safety Critical Systems, Australian Department of Defence (1998)
3. IEC 615038 – Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems, International Electrotechnical Commission (1998)
4. Netica, Norsys Software Corp. (2006), <http://www.norsys.com/>
5. RTCA/DO-178B: Software Considerations in Airborne Systems and Equipment Certification, Radio Technical Commission for Aeronautics (1992)
6. The United Modelling Language (UML) Specification. The Object Management Group (2005)
7. Achour, C.B., Rolland, C., Souveyet, C.: Guiding Goal Modelling Using Scenarios. *IEEE Trans. on Software Engineering* 24(2), 1055–1071
8. Alexander, I.: Misuse Cases: Use Cases with Hostile Intent. *IEEE Software* 20(1), 58–66
9. Allenby, K., Kelly, T.: Deriving Safety Requirements using Scenarios. In: the 5th IEEE International Symposium on Requirements Engineering(RE'01), p. 228. IEEE Computer Society Press, Los Alamitos (2001)
10. Bachmann, F., Bass, L., Klein, M.: *Deriving Architectural Tactics: A Step Toward Methodical Architectural Design*, SEI (2003)
11. Bass, L., Clements, P., Kazman, R.: *Software Architecture in Practice*, 2nd edn. Addison Wesley, Reading, MA, USA (2003)
12. Boehm, B., Turner, R.: *Balancing Agility and Discipline: A Guide for the Perplexed*. Addison-Wesley Professional, Reading (2003)
13. Brandozzi, M., Perry, D.E.: From Goal-Oriented Requirements to Architectural Prescriptions: The Preskriptor Process. In: *Proceedings of Third International Workshop From Software Requirements to Architectures (STRAW'03)*, pp. 107–113 (2003)
14. Buhr, R.J.A., Casselman, R.S.: *Use Case Maps for Object-Oriented Systems*. Prentice-Hall, Englewood Cliffs (1996)
15. Clements, P.: *Documenting software architectures: views and beyond*. Addison-Wesley, Boston (2003)

16. Conklin, J., Begeman, M.L.: gIBIS: A Hypertext Tool for Exploratory Policy Discussion. *ACM Transactions on Office Information Systems* 6(4), 303–331
17. Easterbrook, S., Lutz, R., Covington, R., Kelly, J., Ampo, Y., Hamilton, D.: Experiences Using Lightweight Formal Methods for Requirements Modeling. *IEEE Trans. on Software Engineering* 24(1), 4–14
18. Feather, M.S., Cornford, S.L.: Quantitative risk-based requirements reasoning. *Requirements Engineering* 8(4), 248–265
19. Fenelon, P., McDermid, J., Nicholson, M., Pumfrey, D.: Towards Integrated Safety Analysis and Design. *ACM Computing Reviews* 2(1), 21–32
20. Galliers, J., Sutcliffe, A., Minocha, S.: An impact analysis method for safety-critical user interface design. *ACM Transactions on Computer-Human Interaction (TOCHI)* 6(4), 341–369
21. Gregoriades, A., Sutcliffe, A.: Scenario-Based Assessment of Nonfunctional Requirements. *IEEE Trans. on Software Engineering* 31(5), 392–409
22. Gorp, J.v., Bosch, J.: SAABNet: Managing Qualitative Knowledge in Software Architecture Assessment. In: 7th IEEE International Symposium on Engineering of Computer-Based Systems (ECBS 2000), IEEE Computer Society, Los Alamitos (2000)
23. Hall, J., Jackson, M., Laney, R., Nuseibeh, B., Rapanotti, L.: Relating Software Requirements and Architectures using Problem Frames. In: *Proceedings of the 10th International Conference on Requirements Engineering*, IEEE Computer Society, Los Alamitos (2002)
24. Jacobson, I., Christerson, M., Jonsson, P., Oevergaard, G.: *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison Wesley, Reading, Mass (1992)
25. Jaynes, E.T.: *Probability Theory: The Logic of Science*. Cambridge University Press, Cambridge (2003)
26. Kelly, T.: Using Software Architecture Techniques to Support the Modular Certification of Safety-Critical Systems. In: *Proceedings of Eleventh Australian Workshop on Safety-Related Programmable Systems* (2006), <http://www-users.cs.york.ac.uk/~tpk/scs2006.pdf>
27. Kelly, T.P.: *Arguing Safety - A Systematic Approach to Safety Case Management*. Department of Computer Science, DPhil Thesis, University of York, York (1999)
28. Kirwan, B., Ainsworth, L.K. (eds.): *A Guide to Task Analysis: The Task Analysis Working Group*. Taylor & Francis, Abington (1992)
29. Lamsweerde, A.v.: Elaborating Security Requirements by Construction of Intentional Anti-Models. In: *Proceedings of the 26th International Conference on Software Engineering*, pp. 148–157. IEEE Computer Society, Los Alamitos (2004)
30. Lamsweerde, A.v.: Goal-Oriented Requirements Engineering: A Guided Tour. In: Lamsweerde, A. (ed.) *Proceedings of 5th IEEE International Symposium on Requirements Engineering (RE'01)*, pp. 249–263. IEEE Press, Los Alamitos (2001)
31. Lamsweerde, A.v., Dardenne, A., Fickas, S.: Goal-directed Requirements Acquisition. *Science of Computer Programming* 20, 3–50
32. Lamsweerde, A.v., Letier, E.: Integrating Obstacles in Goal-Driven Requirements Engineering. In: Lamsweerde, A. (ed.) *Proceedings of the 20th International Conference on Software Engineering*, pp. 53–62. IEEE Computer Society Press / ACM Press, Los Alamitos (1998)
33. Lemos, R.d., Saeed, A., Anderson, T.: On the Safety Analysis of Requirements Specifications. In: *Proceedings of the 13th International Conference on Computer Safety, Reliability and Security*, Instrument Society of America, pp. 217–227 (1994)
34. Leveson, N.G.: Intent Specifications: An Approach to Building Human-Centered Specifications. *IEEE Trans. on Software Engineering* 26(1), 15–35

35. Leveson, N.G.: *Safeware: System Safety and Computers*. Addison-Wesley, Reading (1995)
36. Leveson, N.G., Modugno, F., Reese, J.D., Partridge, K., Sandys, S.D.: Integrated Safety Analysis of Requirements Specifications. In: *Proceedings: 3rd International Conference on Requirements Engineering* (1997)
37. Lyu, M.R. (ed.): *Handbook of Software Reliability Engineering*. McGraw-Hill, New York (1996)
38. McDermid, J.A.: Software Safety: Where's the Evidence? In: McDermid, J.A. (ed.) *The 6th Australian Workshop on Industrial Experience with Safety Critical Systems and Software (SCS'01)* (Brisbane, 2001), Australian Computer Society (2001)
39. Morgan, C.: Of Probabilistic Wp and SP-and Compositionality. In: *Symposium on the Occasion of 25 Years of CSP* (London, 2004), pp. 220–241. Springer, Heidelberg (2004)
40. Mylopoulos, J., Chung, L.: B.N. Representing and Using Non-Functional Requirements: A Process-Oriented Approach. *IEEE Trans. on Software Engineering* 18(6), 497–497
41. Nuseibeh, B.: Weaving Together Requirements and Architectures. *IEEE Computer* 34(3), 115–114
42. Nuseibeh, B., Kramer, J., Finkelstein, A.: Expressing the relationships between multiple views in requirements specification. In: *Proceedings of the 15th international conference on Software Engineering*, pp. 187–196. IEEE Computer Society Press, Los Alamitos (1993)
43. Parnas, D.L., Clements, P.C.A.: rational design process: How and why to fake it. *IEEE Trans. on Software Engineering* 12(2), 251–257
44. Pearl, J.: *Causality: models, reasoning, and inference*. Cambridge University Press, Cambridge (2000)
45. Potts, C., Antón, A.I.: Inquiry-based Requirements Analysis. *IEEE Software*. 21–32.
46. Ramesh, B., Dhar, V.: Supporting systems development by capturing deliberations during requirements engineering. *IEEE Trans. on Software Engineering* 18(6), 498–510
47. Reason, J.: *Human Error*. Cambridge University Press, Cambridge (1990)
48. Shaw, M., Garlan, D.: *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, Englewood Cliffs (1996)
49. Stamatelatos, M., Apostolakis, G., Dezfuli, H., Everline, C., Guarro, S., Moieni, P., Mosleh, A., Paulos, T., Youngblood, R.: *Probabilistic Risk Assessment Procedures Guide for NASA Managers and Practitioners*, NASA Office of Safety and Mission Assurance (2002)
50. The SERENE Partners: CSR, E., ERA, OT, TUV. *The SERENE Method Manual Safety and Risk Evaluation using bayesian NETs: SERENE*, ERA Technology Ltd. (1999)
51. Vesely, W.E.: *Fault Tree Handbook*. Nuclear Regulatory Commission (1987)
52. Wu, W., Kelly, T.: Deriving Safety Requirements as Part of System Architecture Definition. In: *Proceedings of 24th International System Safety Conference*, System Safety Society (2006)
53. Wu, W., Kelly, T.: Failure Modelling in Software Architecture Design for Safety. *SIGSOFT Softw. Eng. Notes* 30(4), 1–7
54. Wu, W., Kelly, T.: Managing Architectural Design Decisions for Safety-Critical Software Systems. In: *Proceedings of the 2nd International Conference on the Quality of Software Architectures*, Springer, Heidelberg (2006)
55. Wu, W., Kelly, T.: Safety Tactics for Software Architecture Design. In: *Proceedings of the 28th International Computer Software and Applications Conference*, IEEE Computer Society, Los Alamitos (2004)

Extending Failure Modes and Effects Analysis Approach for Reliability Analysis at the Software Architecture Design Level*

Hasan Sozer, Bedir Tekinerdogan, and Mehmet Aksit

Department of Computer Science, University of Twente,
P.O. Box 217 7500 AE Enschede, The Netherlands
{sozerh,bedir,aksit}@ewi.utwente.nl

Abstract. Several reliability engineering approaches have been proposed to identify and recover from failures. A well-known and mature approach is the Failure Mode and Effect Analysis (FMEA) method that is usually utilized together with Fault Tree Analysis (FTA) to analyze and diagnose the causes of failures. Unfortunately, both approaches seem to have primarily focused on failures of hardware components and less on software components. Moreover, for utilizing FMEA and FTA very often an existing implementation of the system is required to perform the reliability analysis. We propose extensions to FMEA and FTA to utilize them for the reliability analysis of software at the architecture design level. We present the software architecture reliability analysis approach (SARAH) that incorporates the extended FMEA and FTA. The approach is illustrated using an industrial case for analyzing reliability of the software architecture of a Digital TV.

Keywords: reliability analysis, FMEA, FTA, software architecture evaluation.

1 Introduction

A number of important trends can be observed in the development of embedded systems. First, due to the high industrial competition and the advances in hardware and software technology, there is a continuous demand for products with more functionality. Second, the functionality provided by embedded systems is shifting from hardware to software. Third, the functionality of embedded systems is not solely developed by just one manufacturer only but it is host to multiple parties. Finally, embedded systems are more and more integrated in networked environments that affect these systems in ways that might not have been foreseen during their construction. Altogether, these trends complicate the design and implementation of embedded systems. As a result, major steps in technology are required to keep the *reliability* [2] of these systems at the current level. Obviously, since embedded

* This work has been carried out as part of the TRADER project under the responsibility of the Embedded Systems Institute. This project is partially supported by the Netherlands Ministry of Economic Affairs under the Bsik program.

systems are now largely defined and controlled by software, it is also expected that the software failures form a major threat for reliability. Therefore, appropriate reliability analysis and design techniques should be provided to support the anticipation and prevention of potential failures.

In the literature, Failure Mode and Effect Analysis (FMEA) together with Fault Tree Analysis (FTA) are well-known and mature approaches to identify failure modes of system components and evaluate their impact on the system reliability. It appears though that these approaches have primarily focused on failures in hardware components and less on software components. In fact this is not so strange because historically, software formed only a small part of embedded systems and as such hardware components primarily defined the quality of the system. The developments and trends in embedded systems have provided an increasing awareness that reliability analysis should not be limited to hardware components but should also cover software components [6,7,15,16,21]. Further, for applying FMEA and FTA very often a running system is required and less focus has been given to reliability analysis before the system is actually implemented. However, it is of importance to analyze the failures earlier in the life cycle, at the software architecture design level. In this way, potential risks can be identified earlier, before committing organizational resources for implementing the system.

Obviously, FMEA and FTA have provided their clear merits for reliability analysis and it is worthwhile to extend and integrate these mature approaches in reliability analysis for software components. Hence, we propose extensions to FMEA and FTA and integrate them in our novel software architecture reliability analysis method (SARAH). Hereby, we first construct a *failure domain model* by analyzing the domain of failures and the relevant categories. The failure domain model is utilized to derive general failure scenarios that can be defined as possible types of failures. Failure scenarios are described using (adapted) FMEA worksheets. The prioritization of failure scenarios is different from the conventional use of FMEA and FTA where safety has been usually the main criteria for prioritizing failure scenarios. In the industrial project that we are working in [19], we focus on the consumer electronics domain, where safety is less or not an issue. Instead, it is the perception of the user for a failure that defines the severity of that failure. Therefore, we prioritize the failure scenarios based on their severity from the end-user perspective. Failure scenarios are then utilized to derive a *Fault Tree Set* (FTS), which shows the causal and logical connections among the failures. FTS is used to perform *severity analysis*, in which we measure the impact and contribution of the other failures to the user perceived failures. In FTA, severity analysis uses the probabilities of failure occurrences [6] that are usually obtained from an already developed system. We introduce an impact model to estimate the overall impact of a failure even if the probability values are not known. To sum up, the extensions that we propose for FMEA and FTA relate to using a *failure domain model* for systematic derivation of failures, the prioritization of failure scenarios based on *user perception*, and finally an impact analysis model for FTA that does not explicitly require a running system. The extended approaches are utilized to perform *sensitivity analysis*, where we identify the sensitive components of the architecture. These are the components that are associated with the most severe failures. SARAH results in a failure analysis report that defines the sensitive

components of the architecture and provides information on the type of failures that might happen frequently.

The remainder of this paper is organized as follows. Section 2 provides the background information on conventional FMEA and FTA methods. In Section 3, we give an overview of the analysis method. In Section 4, we present the industrial case, in which software architecture for a Digital TV is introduced. This example will be used throughout the paper to illustrate the steps of SARAH and to discuss our experiences in the industrial context. Section 5 presents the specification and derivation of the failure scenarios. In section 6, severity analysis based on fault trees is explained. Section 7 presents the analysis of the architecture. Section 8 provides the related work. Finally, in Section 9 we provide the conclusions.

2 FMEA and FTA

FMEA [17] is a well-known and mature approach for reviewing the causes and effects of system failures systematically. The basic operations of the method are 1) to question the ways that each component fails (failure modes) and 2) to consider the reaction of the system to these failures (effects). The analysis results are organized by means of a worksheet, which comprises information about each failure mode, its causes, its local and global effects (concerning other parts of the product and the environment), the associated component and its severity. A simplified FMEA worksheet template is presented in Table 1.

Table 1. An example FMEA worksheet based on MIL-STD-1629A [12]

System: <i>Car Engine</i> Date: <i>10-10-2000</i> Compiled by: <i>J. Smith</i> Approved by: <i>D. Green</i>					
ID	Item ID	Failure Mode	Failure Causes	Failure Effects	Severity Class
<i>1</i>	<i>CE5</i>	<i>fails to operate</i>	<i>Motor shorted</i>	<i>Motor overheats and burns</i>	<i>V</i>
<i>2</i>	<i>...</i>	<i>...</i>	<i>...</i>	<i>...</i>	<i>...</i>

FMEA can be employed for risk assessment and for discovering potential single-point failures. Systematic analysis increases the insight in the system and the analysis results can be used for guiding the design, its evaluation and improvement. At the downside, some failure modes can be overlooked [14] and some information (e.g. failure probability, risk) regarding the failure modes can be incorrectly estimated at early design phases. Since the technique focuses on individual components at a time, combined effects and coordination failures can also be missed. In addition, FMEA process is effort and time consuming.

FMEA is usually applied together with FTA. FTA [6] is based on a graphical model (fault tree), which defines causal relationships between faults (a set of example fault trees can be seen in Figure 5). Faults, which are assumed to be provided, are

defined as undesirable system states or events that can lead to a system failure. The top node (i.e. root) of the fault tree represents the system failure and the leaf nodes represent faults. The nodes of the fault tree are interconnected with logical connectors (e.g. AND, OR gates) that infer propagation and contribution of faults to the failure. Once the fault tree is constructed, it can be processed in a bottom-up manner to calculate the probability that a failure would take place. This calculation is done based on the probabilities of fault occurrences and interconnections between the faults and the failure [6]. Additionally, the tree can be processed in a top-down manner for diagnosis to determine the potential faults that may cause the failure.

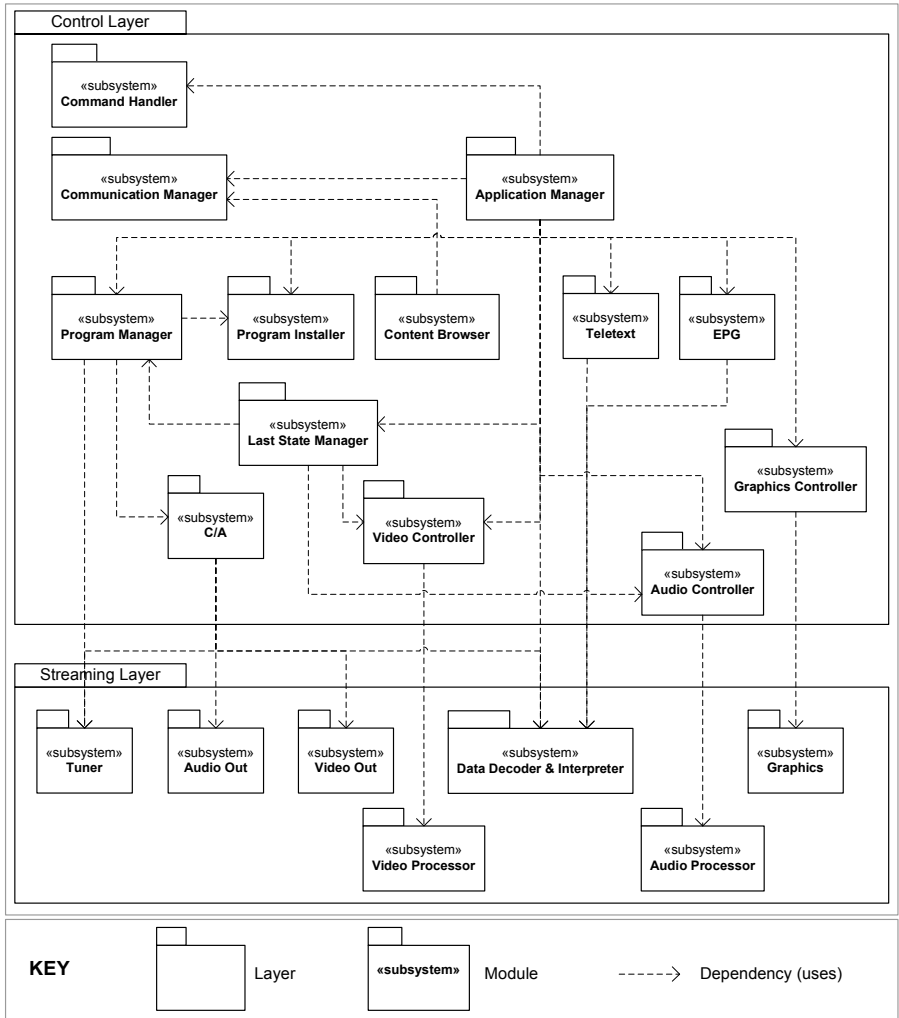
3 Industrial Case: Digital TV (DTV) Software Architecture

At the Embedded Systems Institute (ESI, Netherlands), the TRADER (Television Related Architecture and Design to Enhance Reliability) project is carried out together with NXP Semiconductors and several other academic and industrial partners [19]. The objective of the project is to develop methods and tools for ensuring reliability of digital television (DTV) sets. Due to the increasing size and complexity of software that is embedded in TVs, in practice, it is not feasible to design a perfect system that is fault-free. To cope with the faults that cannot be detected, appropriate fault tolerance mechanisms are required. In the DTV design in the TRADER project, faults of which effects can be directly observed by the user require a special attention. Such faults are considered to be important and they should be tolerated. Because TRADER aims to anticipate also on faults in future releases, it is also important that reliability analysis techniques are defined at the design level. From this perspective one of the key aims in TRADER is the design of fault tolerant software architecture with respect to the perception of TV users.

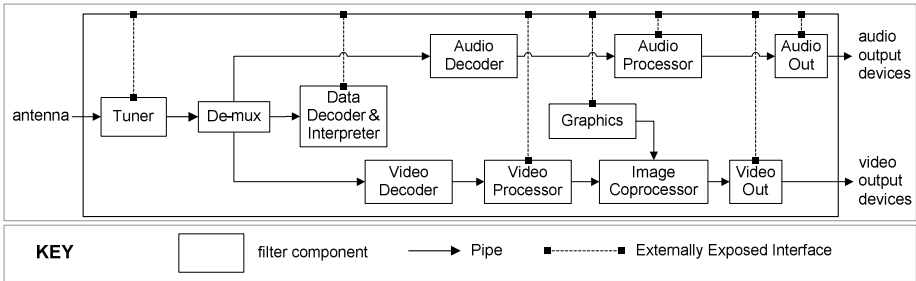
A conceptual architecture of DTV is depicted in Figure 1, which will be referred to throughout the paper. Figure 1(a) presents a module view [4] of the whole DTV structure with implementation units and direct dependencies among them. It mainly comprises two layers. The bottom layer, namely the streaming layer, involves modules taking part in streaming of audio/video information. The upper layer consists of application-related and utility modules that control the streaming process. Figure 1(b) presents a view of the streaming layer in pipe-and-filter style [4]. The streaming layer modules shown in Figure 1(a) correspond to the streaming layer components in Figure 1(b) with the same names.

In the following, we briefly explain some of the important modules and components that are shown in the architecture. For brevity, the components for decoding and processing audio/video signals are not explained here.

Application Manager (AMR) initiates and controls execution of both resident and downloaded applications in the system. It keeps track of application states, user modes and redirects commands/information to specific applications or controllers accordingly. *Audio Controller* (AC), controls audio features like volume level, bass and treble based on commands received from AMR. *Command Handler* (CH) interprets externally received signals (i.e. through keypad or remote control) and sends corresponding commands to AMR. *Communication Manager* (CMR) employs



(a) Module View of the Digital TV Architecture



(b) Component-Connector View (pipe-and-filter style) of the Streaming Layer

Fig. 1. Conceptual Architecture of a Digital TV

protocols for providing communication with external devices. *Conditional Access* (C/A) authorizes information that is presented to the user. *Content Browser* (CB) presents and provides navigation of content residing in a connected external device. *Electronic Program Guide* (EPG) presents and provides navigation of electronic program guide. *Graphics Controller* (GC) is responsible for generation of graphical images corresponding to user interface elements. *Last State Manager* (LSMR) keeps track of last state of user preferences such as the volume level and the selected program. *Program Installer* (PI) searches and registers programs together with the channel information (e.g. frequency). *Program Manager* (PM) tunes to a specific channel. *Teletext* (TXT) handles acquisition, interpretation and presentation of the teletext pages. *Video Controller* (VC) controls the video features like scaling of the video frames based on commands received from AMR.

4 Fundamental Steps of the Analysis Method

Software architecture forms one of the key artifacts in the software development life cycle since it embodies the earliest design decisions and includes the gross-level components that directly impact the subsequent analysis, design and implementation. Accordingly, it is important that the architecture design supports the required qualities of the software system. In general static analysis of formal architectural models is applied or a set of architecture analysis methods as described in [5] are used to analyze and predict the quality of the system. In this paper, we focus on software architecture analysis methods that utilize scenarios for evaluating architectures. In general, these analysis methods take as input the architecture design and measure the impact of predefined scenarios on it to identify the potential risks and the sensitive points of the architecture. This helps to predict the quality of the system before it is built, thereby reducing unnecessary maintenance costs. A scenario is considered to be a brief description of some anticipated or desired use of the system.

We propose a scenario-based architecture analysis method, SARAH that incorporates FMEA and FTA. In contrast to general-purpose architectural analysis methods in which scenarios can refer to different quality factors, we utilize the notion of *failure scenario* to analyze the impact of failures. A failure scenario represents the possible occurrence of a failure in a given component. The steps of SARAH are presented in a UML activity diagram in Figure 2. The approach consists of two basic processes: (1) *Definition* (2) *Analysis*.

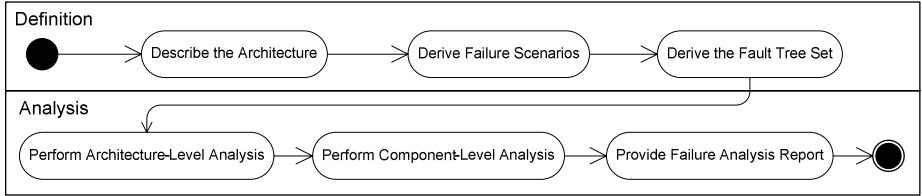


Fig. 2. The Steps of the Analysis Method

In the *definition process*, first the software architecture is described (section 3). Failure scenarios are then derived from a failure domain model that includes a categorization of failures. Failure scenarios are described using a template like FMEA worksheet (section 5). As a subsequent step the failure scenarios in the FMEA are utilized in the FTA (section 6). Here, we introduce the Fault Tree Set (FTS) concept, which is basically a set of possibly connected fault trees. We use an impact model based on FTS to estimate the impact of a failure scenario on the occurrence of the user-perceived failures. By combining the impact model together with the failure prioritization, we measure the severity of a failure scenario based on the type of user-perceived failures it can lead to and its contribution on the occurrence of these failures. Section 6 describes in detail our analysis based on fault trees.

Based on the input from the definition process, in the *analysis process*, an architectural level analysis (section 7.1) and a component level analysis (section 7.2) is performed. The results are presented in the failure analysis report (section 8). In the following sections the main steps of the method will be explained in detail using the industrial case study.

5 Specification and Derivation of Failure Scenarios

In this section we define the process for deriving and specifying failure scenarios. Section 5.1 introduces the scenario template that is used for describing failure scenarios. Section 5.2 explains the derivation of the failure domain model, which provides a categorization for failure scenarios. Section 5.3 explains the derivation of general failure scenarios. Finally, Section 5.4 explains the derivation of concrete failure scenarios based on general failure scenarios.

5.1 Failure Scenario Template

We define the concept of *failure scenario* to analyze the architecture with respect to reliability. Reliability is the ability of the system to function without a *failure*, which is as an event that occurs when the delivered service of a system deviates from a correct service [2]. A correct service is delivered when the service implements the required system function. An *error* is defined as the system state that may lead to a failure and the cause of an error is called a *fault* [2]. Figure 3 depicts the fundamental chain of threats that leads to a failure.

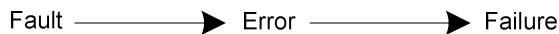


Fig. 3. The Fundamental Chain of Reliability Threats Leading to a Failure

A failure scenario specification includes a description of the *fault*, the *error* and the *failure* (See Table 2). Additionally, the *failure ID* is used to uniquely identify a scenario and the *component ID* defines the component for which the failure scenario applies. Note that this template is similar to the FMEA worksheet (See Table 1). To specify failure scenarios, we use *FID*, *CID*, *fault*, *error* and *failure* instead of the

Table 2. Template for Defining the Failure Scenarios

Attribute	Explanation
FID	A numerical value to identify the failures (i.e. Failure ID).
CID	An acronym defining the component for which the failure scenario applies (i.e. Component ID).
Fault	The description and the features of the cause of the error.
Error	The description and the features of the component state that leads to the failure.
Failure	The description and the features of the deviation of the component function from the required function.

concepts *failure ID*, *related component*, *failure cause*, *failure mode* and *failure effect*, respectively.

5.2 Derivation of the Failure Domain Model

The failure scenario template can be used to describe scenarios. However, there exist too many fault, error and failure types that can be considered. Hence, there is a high risk that several potential and relevant failure scenarios are missed or that other irrelevant failure scenarios are included. To define the relevant failures SARAH defines relevant domain model for faults, errors and failures using a systematic domain analysis process. This domain model provides a first scoping of the potential scenarios. In fact, several researchers have already focused on modeling and classifying failures for embedded systems. Avizienis et al [2], for example, provide a nice overview of this related work and provide a comprehensive classification of faults, errors and failures. The provided domain classification by Avizienis et al., however, is rather broad¹, and one can assume that for a given reliability analysis project not all the potential failures in this overall domain are relevant. Therefore, the given domain is further contracted by focusing only on the faults, errors and failures that are considered relevant for the actual project. Figure 4, for example, defines the derived domain model that is considered relevant for the DTV project. Figure 4 includes three feature diagrams that depict a categorization for faults, errors and failures.

In the feature diagram of fault (see Figure 4), faults are identified according to their *source*, *dimension* and *persistence*. In SARAH, failure scenarios are defined per component. For that reason, the source of the fault can be either (1) internal to the component in consideration, (2) caused by other component(s) of the system or (3) caused by the external entities with respect to the system. Faults can be introduced by software or hardware, and be transient or persistent. The relevant features of an error comprise the *type* of error together with its *detectability* and *reversibility* properties. The features for failure include the *type* and the *target*. The *target* of a failure can be the user or the other component(s) of the system.

¹ Due to space limitations we do not show this domain model and refer the interested reader to the corresponding publication.

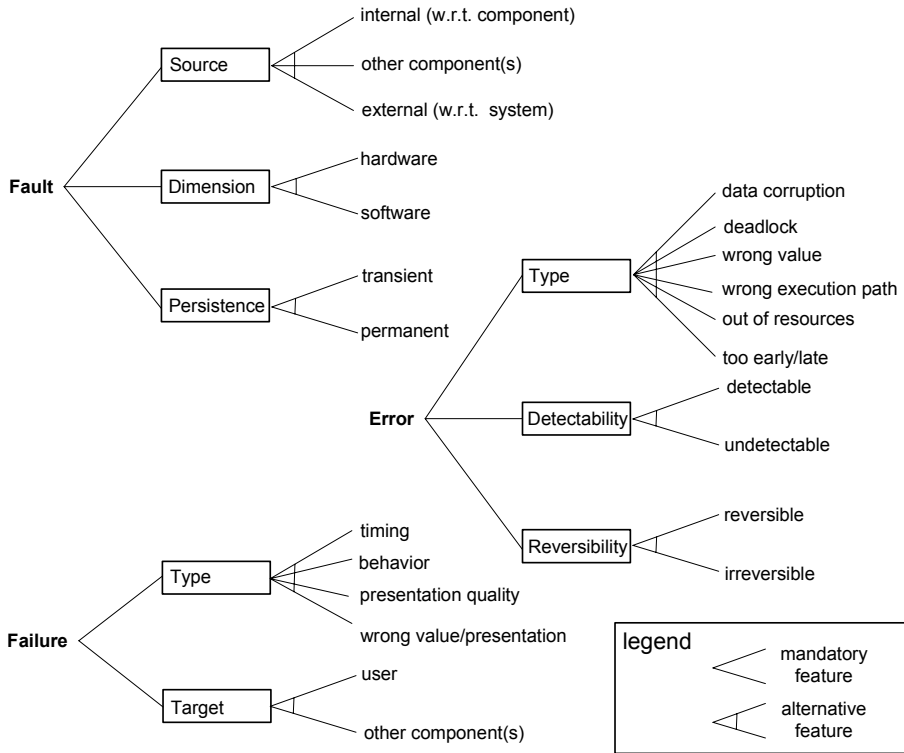


Fig. 4. Failure Domain Model: Classification of (relevant) Faults, Errors and Failures

In principle, for different project requirements one may come up with a slightly different domain model, but as we will show in the next sections this does not impact the steps in the analysis method itself. The key issue here is that the failure domain model is derived in accordance with the project requirements and the elements of a failure scenario are defined as instances of this model.

5.3 Derivation of the General Failure Scenarios

The domain model defines a system-independent specification of the type of faults, error and failures that could occur and is utilized for deriving so-called *general failure scenarios*². A general failure scenario is a system-independent failure scenario that includes selected sub-features from the fault, error and failure features. The number and type of general failure scenarios is implicitly defined by the failure domain model.

In the fault classification (see Figure 4), for example, we can define faults based on three features, namely *Source*, *Dimension* and *Persistence*. The feature *Source* can

² The notion of general failure scenario is a specialization of the general scenario as defined by Bachmann et al. [3].

have 3 different values, the features *Dimension* and *Persistence* 2 values. This means that the fault classification captures $3 \times 2 \times 2 = 12$ different faults. Similarly, from the error classification we can derive $6 \times 2 \times 2 = 24$ different errors, and $4 \times 2 = 8$ different failures are captured by the failure classification. Since a general scenario is a composition of selection of features from the failure domain model, we can state that for the given failure domain model in Figure 4, $12 \times 8 \times 24 = 2304$ general failure scenarios can be in principle defined. One of these general failure scenarios is, for example, shown in Table 3.

Table 3. Example General Failure Scenario Derived from the Failure Domain Model

Fault	Error	Failure
<i>source</i> : internal	<i>type</i> : too early/late	<i>type</i> : timing
<i>dimension</i> : software	<i>detectability</i> : detectable	<i>target</i> : user
<i>persistence</i> : transient	<i>reversibility</i> : irreversible	

5.4 Derivation of the Concrete Failure Scenarios

Although general failure scenarios provide an indication of possible failure scenario *types*, these do not reflect instantiated real failure scenarios. For example, the failure scenario specified in Table 3 does not specify the associated component, the actual fault, the error and the failure. SARAH includes additional steps to identify the relevant failure scenarios and to change the general failure scenarios to *concrete failure scenarios*. Unlike general failure scenarios, the concrete failure scenarios relate to specific components in the architecture. The steps for deriving concrete failure scenarios include (1) analyzing the domain (2) analyzing problems of existing systems, and (3) consulting domain experts. We can illustrate this for the adopted example case on the DTV architecture.

In the first step, we have analyzed the domain on corresponding digital TVs, including requirements specifications, design documentation of existing systems, literature on digital TV systems and real-time embedded systems.

In the second step, we have made use of the available DTV *Problem Database* (PRDB) that had been developed to report on failures of earlier TV systems. The primary goal of PRDB by recording failures was to solve the existing problems in current TV sets and not to support the analysis of the system. This had a clear impact on the PRDB. First of all, the reported problems were not in the format that we required, and we had to interpret these first. Further, the PRDB also included outdated information and we had to filter these out. Some problems were related to concrete and older products but since existing TV systems share common properties with next generation products, they have to cope with similar type of failures. For example, *out-of-spec signal* is a typical fault leading to numerous errors in extraction of Teletext information. Usually, specific instances of this type of fault have been reported in PRDB and appropriate solutions were provided. However, in general, reception of out-of-spec signals can pop up as a problem for every new TV system. By collecting such regular problems we could derive useful failure scenarios. Despite of the problems that we encountered in analyzing this PRDB it definitely provided valuable input.

Finally, in the last step for deriving concrete failure scenarios we have consulted digital TV domain experts to derive additional scenarios and to cross-validate the previously identified scenarios.

Based on these steps, we have derived and specified 44 concrete failure scenarios for the reliability analysis of the DTV. As an example, Table 4 presents a list of nine selected concrete failure scenarios that have been derived. In Table 4 the five attributes *failure ID*, *component ID*, *fault*, *error* and *failure* are represented as columns headings. Failure scenarios are represented in rows. The failure ID (*FID*) does not imply a specific ordering but it is only used to identify the concrete failure scenarios. The column component ID (*CID*) includes acronyms of component names from Figure 1 to which the identifiers refer. Note that the separate features of the failure domain model are represented as keywords in the cells. For example, *Fault* includes the features *source*, *dimension* and *persistence* as defined in Figure 4. Such features are represented as keywords, which distinguish the fault, error and failure categories. For example, failure scenario F5 indicates a *permanent* fault (wrong implementation of a protocol). It leads to a *timing* error since a communicating party can not receive an expected response on time. This error turns out to be a failure that leads to a *transient* fault in F2 because the communication temporarily ceases. We consider *timing* errors as *irreversible* whereas a *wrong value*, for instance, can be rolled back (*reversible*). Apart from the different features, every column also includes the keyword *description*, which is used to denote the domain specific details of the concrete failure scenarios. Typically these descriptions are derived from domain experts.

6 Fault Tree Set and Severity Analysis

A close analysis of the failure scenarios in Table 4 shows that they are connected to each other. The columns *Fault* and *Failure* include the fields *source* and *target* respectively. These fields represent the propagation and the links between failure scenarios. For example, in failure scenario F2, the fault source is defined as CMR(F5), indicating that the fault in F2 occurs due to a failure in component CMR as defined in F5. The source of the fault can be caused by a combination of failures. This is expressed by logical connectives. For example, the source of F1 is defined as CH(F4) OR CMR(F6) indicating that F1 occurs due to a failure in component CH as defined in F4 or due to a failure in component CMR as defined in F6. To make all these connections explicit, in SARAH *fault trees* are defined. A fault tree is a model for representing the cause-effect relations of failures and faults. The root of a fault tree represents a failure and the leaf nodes represent faults. Since a failure can be logically caused by a set of faults, the nodes of the fault tree are interconnected with logical connectors.

Normally, a fault tree has one root (top) node, which represents the failure of the system. However, a system may fail in many different ways each of which has a different effect (i.e. annoyance caused) on the user. For example, a failure can be transient, permanent or catastrophic. A failure scenario can contribute to one or more user perceived failures. Hence, we define a *Fault Tree Set (FTS)* based on a given set

Table 4. Selected Failure Scenarios Derived for the DTV Architecture

FID	CID	Fault	Error	Failure
F1	AMR	<i>description:</i> Reception of irrelevant signals. <i>source:</i> CH(F4) OR CMR(F6) <i>dimension:</i> software <i>persistence:</i> transient	<i>description:</i> Working mode is changed when it is not desired. <i>type:</i> wrong path <i>detectability:</i> undetectable <i>reversibility:</i> reversible	<i>description:</i> Switching to an undesired mode. <i>type:</i> behavior <i>target:</i> user
F2	AMR	<i>description:</i> Can not acquire information. <i>source:</i> CMR(F5) <i>dimension:</i> software <i>persistence:</i> transient	<i>description:</i> Information can not be acquired from the connected device. <i>type:</i> too early/late <i>detectability:</i> detectable <i>reversibility:</i> irreversible	<i>description:</i> Can not provide information. <i>type:</i> timing <i>target:</i> CB(F3)
F3	CB	<i>description:</i> Can not acquire information. <i>source:</i> AMR(F2) <i>dimension:</i> software <i>persistence:</i> transient	<i>description:</i> Information can not be presented due to lack of information. <i>type:</i> too early/late <i>detectability:</i> detectable <i>reversibility:</i> irreversible	<i>description:</i> Can not present content of the connected device. <i>type:</i> behavior <i>target:</i> user
F4	CH	<i>description:</i> Software fault. <i>source:</i> internal <i>dimension:</i> software <i>persistence:</i> permanent	<i>description:</i> Signals are interpreted in a wrong way. <i>type:</i> wrong value <i>detectability:</i> undetectable <i>reversibility:</i> reversible	<i>description:</i> Provide irrelevant information. <i>type:</i> wrong value/presentation <i>target:</i> AMR(F1)
F5	CMR	<i>description:</i> Protocol mismatch. <i>source:</i> external <i>dimension:</i> software <i>persistence:</i> permanent	<i>description:</i> No communication with the connected device. <i>type:</i> too early/late <i>detectability:</i> detectable <i>reversibility:</i> irreversible	<i>description:</i> Can not provide information. <i>type:</i> timing <i>target:</i> AMR(F2)
F6	CMR	<i>description:</i> Software fault. <i>source:</i> internal <i>dimension:</i> software <i>persistence:</i> permanent	<i>description:</i> Signals are interpreted in a wrong way. <i>type:</i> wrong value <i>detectability:</i> undetectable <i>reversibility:</i> reversible	<i>description:</i> Provide irrelevant information. <i>type:</i> wrong value/presentation <i>target:</i> AMR(F1)
F7	DDI	<i>description:</i> Out-of-spec signals. <i>source:</i> external <i>dimension:</i> software <i>persistence:</i> transient	<i>description:</i> Scaling information can not be interpreted from meta-data. <i>type:</i> wrong value <i>detectability:</i> detectable <i>reversibility:</i> reversible	<i>description:</i> Can not provide data. <i>type:</i> wrong value/presentation <i>target:</i> IC(F8)
F8	IC	<i>description:</i> Inaccurate scaling ratio information. <i>source:</i> DDI(F7) AND VC(F9) <i>dimension:</i> software <i>persistence:</i> transient	<i>description:</i> Video image can not be scaled appropriately. <i>type:</i> wrong value <i>detectability:</i> undetectable <i>reversibility:</i> reversible	<i>description:</i> Provide distorted video image. <i>type:</i> presentation quality <i>target:</i> user
F9	VC	<i>description:</i> Software fault. <i>source:</i> internal <i>dimension:</i> software <i>persistence:</i> permanent	<i>description:</i> Wrong scaling ratio calculation. <i>type:</i> wrong value <i>detectability:</i> detectable <i>reversibility:</i> reversible	<i>description:</i> Provide inaccurate information. <i>type:</i> wrong value/presentation <i>target:</i> IC(F8)

of failure scenarios. Here, we introduce a set of definitions related to FTS, which will be further used in the remainder of the paper. FTS is a graph $G(V,E)$ with the following properties:

1. $V = F \cup A$
2. F is the set of failure scenarios each of which is associated with an architectural component.
3. $F_u \subseteq F$ is the set of failure scenarios comprising failures that are perceived by the user (i.e. system failures). Vertices residing in this set constitute root nodes of fault trees.
4. A is the set of gates representing the logical connectors.
5. $\forall g \in A$,
 $\text{outdegree}(g) = 1 \wedge$
 $\text{indegree}(g) \geq 1$
6. $A = A_{\text{AND}} \cup A_{\text{OR}}$ such that,
 A_{AND} is the set of AND gates,
 A_{OR} is the set of OR gates.
7. E is the set of directed edges (u, v)
 where $u, v \in V$.

For the example case, based on the failure scenarios in Table 4 we can derive the FTS as depicted in Figure 5.

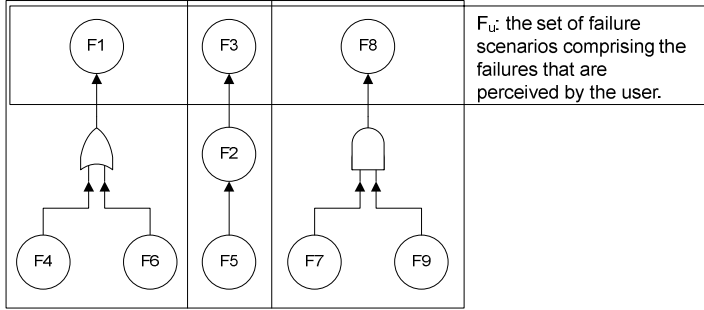


Fig. 5. Fault Trees Derived from the Failure Scenarios in Table 4

Here, the FTS consists of three fault trees. On the left the fault tree shows that F1 is caused by F4 or F6. The middle column indicates that failure scenario F3 is caused by F2 which is on its turn caused by F5. Finally, in the last column the fault tree shows that F8 is caused by both F7 and F9.

In conventional FMEA and FTA, every fault is assigned a severity value to denote how severe a fault is (e.g. faulty component can be repaired or not). In our model, we take a user-centric approach and define *severity* based on the user-perception. We assign severities to all failures based on their impact on the user. System failures that we consider are not restricted to complete crash-down of the system. For instance, a minor distortion in the brightness level of the video image and absence of any image

are both failures. However, the impact on the user perception for both failures is different. For example, a complete black screen will upset the user more than a temporary distortion in the image.

Severities based on user perception can be only defined for user perceived failures (elements of the set F_u). We need to propagate these severity values to the other failures (elements of the set F) as well. For this, we need to consider the impact of such intermediate failures to the user perceived failures. In conventional FTA, this impact is known as the *sensitivity* of the system failure (i.e. user perceived failure) with respect to a fault (i.e. intermediate failure). Sensitivity analysis is based on cut-sets in a fault tree and the probability values of fault occurrences [6]. However, this analysis leads to complex formulas and it requires that the probability values are known priori. We propose the following impact model by means of which one can reason about and estimate the overall impact of a failure even if the probability values are not known.

$$\begin{aligned}
 & \text{root} \in F_u, \text{node} \in F, \\
 & \forall n \in F \text{ s.t. } n \neq \text{node} \wedge n \neq \text{root}, P(n) = p, \\
 & P(\text{node}) = p', P(\text{root}) = f(p', p), \\
 & \text{impact}(\text{node}) = \int_0^1 \left(\frac{\partial}{\partial p'} P(\text{root}) \right) dp
 \end{aligned} \tag{1}$$

In equation (1) above, the impact calculation of an intermediate failure (*node*) on a user-perceived failure (*root*) is presented. The probability of occurrence of *root* ($P(\text{root})$) is represented in terms of the occurrence probabilities of all other nodes as it is done in FTA [6]. For example, the following probabilities can be defined for user-perceived failures considering the fault trees presented in Figure 5: $P(F1) = P(F4) + P(F6) - (P(F4) \times P(F6))$, $P(F3) = P(F2) = P(F5)$, $P(F8) = P(F7) \times P(F9)$. We assign p' to the probability of occurrence of *node* and fix the probability values of all other nodes to p in $P(\text{root})$. Thus $P(\text{root})$ turns out to be a function of p and p' . Recalling the example in Figure 6 again, if we are interested in impact of F4 on F1, we transform $P(F1)$ to $P(F1) = p' + p - (p' \times p)$. Then, we take a partial derivation of $P(\text{root})$ with respect to p' . This gives the rate of change of $P(\text{root})$ with respect to p' . For our example, this will yield to $\partial/\partial p' P(F1) = 1 - p$. Finally, the result of the partial derivation is integrated with respect to p for all possible probability values ([0-1]) to calculate the overall impact. For the example case, $\int (\partial/\partial p' P(F1)) dp = \int (1 - p) dp = p - p^2/2$. So, the result of the integration from 0 to 1 will be 0.5. When the probability values are known, they can be included in the model instead of fixing them to p and ranging them from 0 to 1. The basic shortcoming of this model is the equality assumption for the probability values that are fixed to p . There exist similar approaches in the literature where sensitivity/impact analyses are applied by changing a parameter one at a time and fixing the others [7, 21]. However, such analyses are applied for a specific architecture and the associated fault tree without providing a generic model. In those studies, basically the parameters are varied and results obtained from the reliability model are observed to assess the sensitivity.

Combining the impact model introduced above together with the severity degrees assigned to the user perceived failures leads us to the following severity model, which assigns severity to each node of a fault tree.

$$\begin{aligned}
& root \in F_u, node \in F, \\
& \forall n \in F \text{ s.t. } n \neq node \wedge n \neq root, P(n) = p, \\
& P(node) = p', \forall root \in F_u : P(root) = f(p', p), \\
& severity(node) = \sum_{root \in F_u} severity(root) \times \int_0^1 \left(\frac{\partial}{\partial p'} P(root) \right) dp
\end{aligned} \tag{2}$$

In equation (2), $severity(root)$ gives the severity of a user-perceived failure. This can be based on a set of values in ordinal scale that defines user annoyance levels or it can be based on a more sophisticated model. The severity of other failure ($severity(node)$) is calculated based on the impact model introduced in equation (1) and the severity values assigned to the user perceived failures they lead to. As a result, the severity of a failure depends on the type of failures it leads to and to how much it contributes (the impact) to these failures. Note in equation (2) that if there is no connection between $node$ and a $root \in F_u$ in the fault tree, then the result of the partial derivation and hence the whole severity calculation will yield to 0 (since there will be no term in the function with p' as the multiplier).

For the analysis presented in this paper, we adopt a simpler model for propagation of severity values. As a diversion from the usual approach, we process the fault trees in a top-down manner to assign severities to intermediate failures based on severities of user-perceived failures. Given the FTS, the calculation of severities of failure scenarios can be defined as follows.

$$\begin{aligned}
& \forall f \in F_u, s(f) = s_u \quad \text{s.t.} \quad 1 \leq s_u \leq 5 \\
& \forall f \in F \wedge f \notin F_u, \\
& s(f) = \sum_{\substack{\forall v \text{ s.t.} \\ (u,v) \in E \wedge \\ (v \in F \vee v \in A_{OR})}} s(v) + \sum_{\substack{\forall v \text{ s.t.} \\ (u,v) \in E \wedge \\ v \in A_{AND}}} s(v) / INDEGREE(v)
\end{aligned} \tag{3}$$

In the first part of the equation, we take into account the failure scenarios with the target *user*. A severity value (s_u) is assigned for each such failure based on their severities with respect to the user-perception. These are the failures F1, F3 and F8. We apply a prioritization of the failure scenarios based on the severity values that range from 1 to 5 (See Table 5).

Table 5. The Severity Levels for the Prioritization of Failure Scenarios

Severity	Type	Annoyance Description
1.	Very low	User hardly perceives failure.
2.	low	A failure is perceived but not really annoying.
3.	moderate	Annoying performance degradation is perceived.
4.	high	User perceives serious loss of functions.
5.	very high	Basic functions fail. System locks up and does not respond.

The severity values for failures F1, F3 and F8 are depicted in Figure 6. These values are then used in order to determine the severities of other failures as shown in the second part of the equation (3). If a failure directly leads to another one or if it is connected through an OR gate in which some other failures may exclusively cause the same failure also, we add the severity of the resulting failure to the severity of failure under consideration. If there is a connection through an AND gate where a combination of failures results in another one, we share the severity value of the resulting failure among contributing failures. In other words, we add the severity of resulting failure divided by the number of contributing failures to the severity of each contributing failure. For example, since F1 has the assigned severity value of 3, this is also assigned to the failures F6 or F8 that are connected through an OR logic to F1. In case of F7 and F9, the severity value is $4/2$ because F8 has the severity value of 4 and it has an AND gate with 2 failure scenarios connected to it. A failure scenario can be connected to multiple gates and other failures in which the severity is derived as the sum of severities calculated for all these connections.

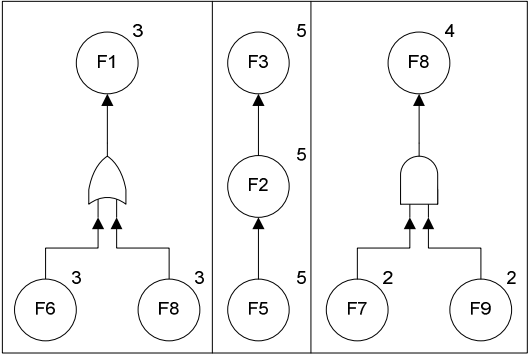


Fig. 6. Fault Trees with the Severity Values

7 Analysis of the Software Architecture

In our example case, we defined a total of 44 failure scenarios including the scenarios presented in Table 4. We completed the definition process by deriving the corresponding fault tree set and calculating severity values as explained in Section 6. The results that were obtained during the definition process are utilized by the analysis process as described in the subsequent sub-sections.

7.1 Architecture-Level Analysis

The first step of the analysis process is the architecture-level analysis in which we pinpoint the sensitive points of the architecture with respect to reliability. As a primary and straightforward means of comparison, we consider the percentage of failures (PF) that are associated with the components. For each component c the value for PF is calculated as follows.

$$PF_c = \frac{\#of\ failures\ associated\ with\ c}{\#of\ failures} \times 100 \quad (4)$$

This means that simply the number of failures related to a component is divided by the total number of failures (in this case 44). The results are shown in Figure 7. A first analysis of this figure already shows that the *Application Manager (AMR)* and *Teletext (TXT)* components have to cope with a higher number of failures than the other components.

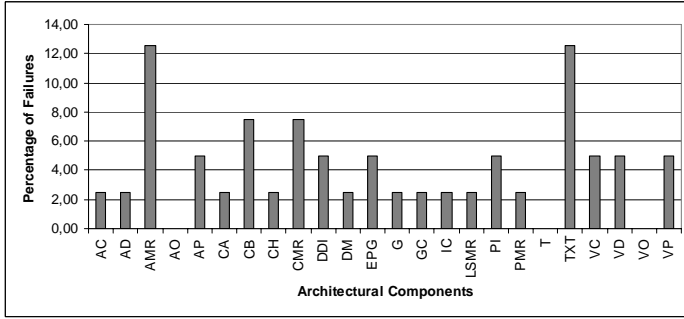


Fig. 7. Percentage of the Failure Scenarios Impacting the Components

This analysis treats all failures equally. To take the severities of the failures into account we define the Weighted Percentage of Failures (WPF) as the following.

$$WPF_c = \frac{\sum_{\substack{u \in F \\ s.t. \\ comp(u)=c}} s(u)}{\sum_{u \in F} s(u)} \times 100 \quad (5)$$

For each component, we collect the set of failures associated with it and we add up their severity values. After averaging this value with respect to the all failures and calculating the percentage, we obtain the WPF value. The result of the analysis is shown in Figure 8.

The weighted percentage presents different results compared to the previous one. Two components can be associated with the same amount of failures but these failures can have different severities. For example, AP and EPG have the same values in Figure 7 but EPG has a greater value than AP in Figure 8. Nevertheless, the *Application Manager (AMR)* and *Teletext (TXT)* components again appears to be the most critical.

From the project perspective it is not always possible to focus on the total set of possible failures due to the time constraints and the cost of fault tolerance. To optimize the cost usually one would like to consider the failures that have the largest impact on the system. For this, in SARAH the architectural components are ordered in

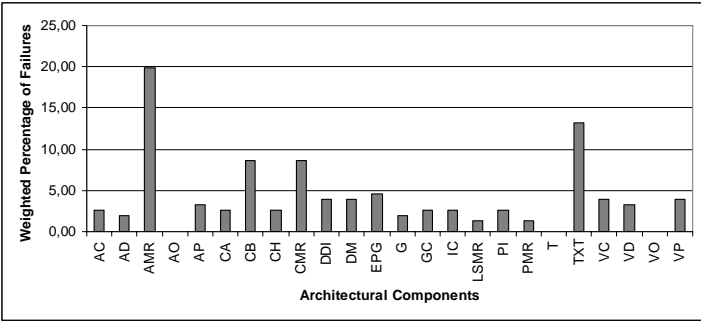


Fig. 8. Percentage of the Failure Scenarios Impacting the Components

ascending order with respect to their WPF values. The components are then categorized based on the proximity of their WPF values. Accordingly, components of the same group have WPF values that are close to each other. The results of this prioritization and grouping are provided in Table 6, which also shows the sum of the WPF values of the components for each group. Here we can see that, for example, group 4 consists of two components AMR and TXT. The reason for this is that their WPF values are the highest and close to each other. The sum of their WPF values is $20+13=33\%$.

Table 6. Components Grouped Based on the WPF Values

Group #	Components	WPF
1	AC, CA, CH, GC, IC, PI, AD, G, LSMR, PMR, AO, T, VO	23%
2	EPG, VC, VD, DDI, DM, VD, AP	27%
3	CB, CMR	17%
4	AMR, TXT	33%

To highlight the difference in impact of the component groups we define a Pareto chart as presented in Figure 9.

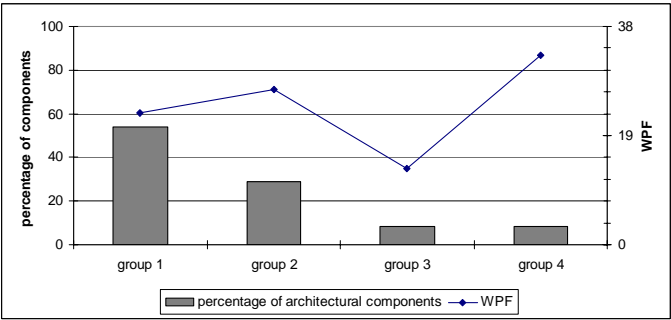


Fig. 9. Pareto Chart Showing the Largest Impact of the Smallest Set of Components

In the Pareto chart, the component groups shown in Table 6 are ordered along the x-axis with respect to the number of components they include. The percentage of components that each group includes is depicted with bars. The y-axis on the left hand shows the percentage values from 0 to 100 and is used for scaling the percentages of the architectural components whereas the y-axis on the right hand side scales the WPF values. The plotted line represents the WPF value for each group. In the figure we can, for example, see that group 4 (consisting of two components) represents about 8% of the components but has a WPF of 33%. Here, we group the components according to their WPF values and assign importance to these components according to the amount and severities of failures they are associated with (not according to their functionality or other aspects). The components with the highest WPF values are considered to be the most important ones to focus on.

7.2 Component-Level Analysis

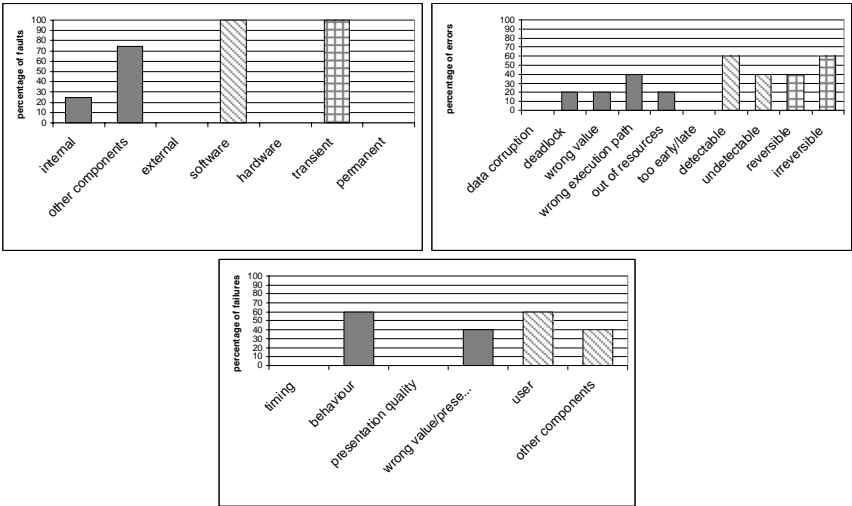
The architectural level analysis provides only a quantitative analysis of the impact of failure scenarios on the given architecture. However, for failure management and recovery it is also necessary to define the *type* of failures that might occur in the identified sensitive components. This is analyzed in the component-level analysis in which the features of faults, errors and failures that impact the component are determined. For the example case, in the architectural-level analysis it appeared that components residing in the 4th group (see Table 6) had to deal with largest set of failure scenarios. Therefore, in the component-level analysis, we will focus on the members of this group, namely *Application Manager* and *Teletext* components.

Following the derivation of the set of failure scenarios impacting the component, we group them in accordance with the features presented in Figure 4. This grouping results in the distribution of fault, error and failure categories of failure scenarios associated with the component.

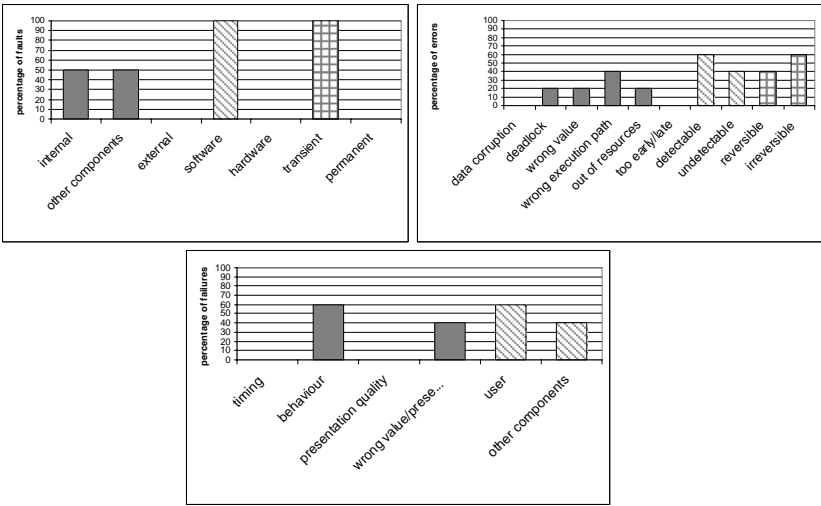
For example, the results obtained for the *Application Manager* and *Teletext* components are shown in Figure 10(a) and Figure 10(b), respectively. If we take a look at the fault features presented on those figures for instance, we see that most of the faults impacting the *Application Manager* Component are caused by the other components. On the other hand, *Teletext* Component has internal faults as much as faults stemming from the other components. As such, distribution of features reveals characteristics of faults, errors and failures associated with the individual components of the architecture.

7.3 Failure Analysis Report

SARAH defines a detailed description of the fault tree set, the failure scenarios, the architectural level analysis and the component level analysis. These are described in the *failure analysis report* that summarizes the previous analysis results and provides hints for recovery. Sections comprised by the failure analysis report are listed in Table 7, which are in accordance with the steps of SARAH.



(a) Fault, Error and Failure Features of the Failure Scenarios Associated with the *Application Manager* Component



(b) Fault, Error and Failure Features of the Failure Scenarios Associated with the *Teletext* Component

Fig. 10. Fault, Error and Failure Features of the Failure Scenarios Associated with the Components in the 4th Group of the Pareto Chart

Table 7. Sections of the Failure Analysis Report

Section #	Heading
1	Introduction
2	Software Architecture
3	Failure Domain Model
4	Failure Scenarios
5	Fault Tree Set
6	Architecture Level Analysis
7	Component Level Analysis

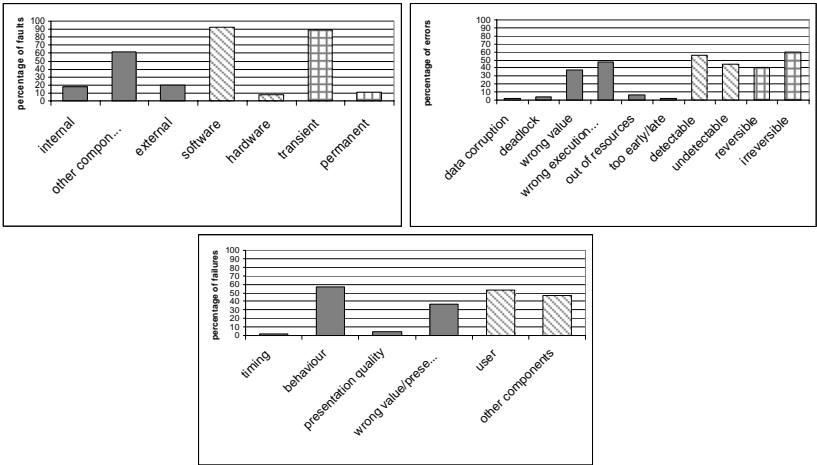


Fig. 11. Feature Distribution of Fault, Error and Failures for all Failure Scenarios

The first section describes the project context, information sources and specific considerations (e.g. cost-effectiveness). The second section describes the software architecture that is used as an input for the analysis. The third section presents the failure domain model including the fault, error and failure features that are of interest to the project. The fourth section contains list of failure scenarios annotated based on this domain model. The fifth section depicts the fault tree set generated from the failure scenarios together with the severities assigned to each. The sixth and seventh sections include analysis results as presented in sections 7.1 and 7.2 of this paper. Additionally, the sixth section includes the distribution of fault, error and failure features for all failure scenarios as depicted in Figure 11.

8 Related Work

FMEA has been widely used in various industries such as automotive and aerospace. It has also been extended to make it applicable to the other domains or to achieve more analysis results. For instance, Failure Modes, Effects and Criticality Analysis (FMECA) is a well-known method that is built upon FMEA. Additionally, FMECA

[12] incorporates severity and probability assessments for faults. The probabilities of occurrence of faults (assumed to be known) are utilized together with the fault tree in order to calculate the probability that the system will fail. On the other hand, a severity is associated with every fault to distinguish them based on the cost of repair. Note that the severity definition in our method is different from the one that is employed by FMECA. We take a user-centric approach and define the severities for failures based on their impact on the user.

Almost all reliability analysis techniques have primarily devised to analyze failures in hardware components. These techniques have been extended and adapted to be used for the analysis of software systems. For instance, in his book [15], Redmill shows how Hazard and Operability Studies (HAZOP) can be applied to computer-based systems. Application of FMEA to software has a long history [16]. Both FMEA and FTA have been employed for the analysis of software systems and named as Software Failure Modes and Effects Analysis (SFMEA) and Software Fault Tree Analysis (SFTA), respectively. In SFMEA, failure modes for software components are identified such as computational, logic and data I/O. This classification resembles the failure domain model of SARAH. However, SARAH separates fault, error and failure concepts and provides a more detailed categorization for each. Also, note that the failure domain model can vary depending on the project requirements and the system. In [11], SFTA is used for the safety verification of software. However, the analysis is applied at the source code level rather than the software architecture design level. It utilizes a set of failure-mode templates that outlines the failure modes of programming language elements like assignments and conditional statements. These templates are composed according to the control flow of the program to derive a fault tree for the whole software.

In general, efforts for applying reliability analysis to software [10] mainly focus on the safety-critical systems, whose failure may have very serious consequences such as loss of human life and large-scale environmental damage. In our case, we focus on consumer electronics domain, where the systems are usually not safety-critical. FMEA has been used in other domains as well, where the methodology is adapted and extended accordingly. To use FMEA for analyzing the dependability of Web Services, new failure taxonomy, intrusions and various failure effects (data loss, financial loss, denial of service, etc.) are taken into account in [9]. Utilization of FMEA is also proposed in [22] for early robustness analysis of Web-based systems. The method is applied together with the Jacobson's method [18], which identifies three types of objects in a system: 1) boundary objects that communicate with actors in a use-case, 2) entity objects that are objects from the domain and 3) control objects that serve as a glue between boundary objects and entity objects. In our method, we do not presume any specific decomposition of the software architecture and we do not categorize objects or components. However, we categorize failure scenarios based on the failure domain model and each failure scenario is associated with a component.

Jacobson's classification [18] is aligned with our failure domain model with respect to the propagation of failures (fault.source, failure.target). The target feature of failure, for instance, can be the user (i.e. actor) or the other components. In [20], a modular representation called Fault Propagation and Transformation Calculus (FPTC) is introduced. FPTC is used to specify the failure behavior of each component (i.e.

how a component introduces or transforms a failure type). This facilitates the automatic derivation of the propagation of the failures throughout the system. In our method, we represent the propagation of failure scenarios with fault trees. The semantics of the transformation is captured in the “type” tags of failure scenarios.

In this work, we made use of spreadsheets that define the failure scenarios and automatically calculate the severity values in the fault trees (after initial assignment of the user-perceived failure severities). This is a straightforward calculation and as such we have not elaborated on this issue. On the other hand, there is a body of work focusing on tool-support for FMEA and FTA. In [9], for example, FMEA tables are being integrated with a web service deployment architecture so that they can be dynamically updated by the system. In [13], fault trees are synthesized automatically. Further, multiple failures are taken into account, where a failure mode can contribute to more than one system failure. The result of the fault tree synthesis is a network of interconnected fault trees, which is analogous to the fault tree set in our method.

An advanced Failure Modes and Effect Analysis (AFMEA) is introduced in [8], which also focuses on the analysis at the early stages of design. However, the aim of this work is to enhance FMEA in terms of the number and range of failure modes captured. This is achieved by constructing a behavior model for the system.

9 Conclusion

FMEA and FTA are well known techniques that have been successfully applied in various domains like automotive and aerospace for reviewing the causes and effects of system failures systematically. In this paper we have utilized these techniques for early reliability analysis of software architectures. It appears that the techniques can not be directly used as-is for analyzing reliability of software earlier in the life cycle and as such need to be extended.

For the FMEA, we have utilized scenarios as the basic means to analyze the software architecture [5]. We have separated fault, error and failure concepts in a failure scenario as defined in the software reliability engineering domain [2]. We have provided a systematic means for deriving failure scenarios based on a failure domain model. For the analysis presented in this paper, we have also derived the fault, error and failure categorizations through domain analysis techniques [1].

For the FTA, we have introduced a quantitative impact model based on fault trees to estimate the impact of faults on the occurrence of a system failure. This model can be used to reason about how sensitive the system reliability is with respect to occurrence of a fault, even if the occurrence rates of faults are not known. Furthermore, we have distinguished and prioritize failures based on their effect (i.e. annoyance caused) on the user. We have incorporated this measure to our severity calculation.

We have integrated the extended FMEA and FTA techniques with our software architecture reliability analysis method (SARAH) and illustrated their application for analyzing the reliability of the software architecture of a Digital TV. In our future work we will experiment with the method to identify the potential failures of embedded systems and as such improve their reliability.

Acknowledgments. We thank all members of the TRADER project, for their feedback on this work and their input about the TV domain knowledge and reliability issues. Particularly, we thank Rob Golsteijn from NXP Semiconductors, Paul L. Janson from Philips Research, Iulian Nitescu from Philips TASS for their contribution in deriving the conceptual architecture of DTV and possible failures. We also specially thank Christian Hofmann from University of Twente, Teun Hendriks and Jozef Hooman from ESI, the editors and anonymous reviewers for reviewing and providing useful feedback to this paper.

References

1. Arrango, G.: Domain Analysis Methods. In: Schafer, R., Prieto-Diaz, R., Matsumoto, M. (eds.) *Software Reusability*, pp. 17–49. Ellis Horwood, New York (1994)
2. Avizienis, A., Laprie, J.-C., Randell, B., Landwehr, C.: Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Trans. on Dependable and Secure Computing* 1(1), 11–33 (2004)
3. Bachman, F., Bass, L., Klein, M.: *Deriving Architectural Tactics: A Step Toward Methodical Architectural Design*. CMU/SEI-2003-TR-004, Pittsburgh, PA (2003)
4. Clements, P., Bachman, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., Stafford, J.: *Documenting Software Architectures*. Addison-Wesley, Reading (2002)
5. Dobrica, L., Niemela, E.: A Survey on Software Architecture Analysis Methods. *IEEE Trans. on Software Engineering* 28(7), 638–654 (2002)
6. Dugan, J.B.: Software System Analysis Using Fault Trees. In: Lyu, M.R. (ed.) *Handbook of Software Reliability Engineering*, ch. 15, pp. 615–659. McGraw-Hill, New York (1996)
7. Dugan, J.B., Lyu, M.R.: Dependability Modeling for Fault-Tolerant Software and Systems. In: Lyu, M.R. (ed.) *Software Fault Tolerance*, ch. 5, pp. 109–138. John Wiley & Sons, New York (1995)
8. Eubanks, C.F., Kmenta, S., Ishil, K.: Advanced Failure Modes and Effects Analysis using Behavior Modeling. In: *Proceedings of the ASME Design Theory and Methodology Conference*, New York (1997)
9. Gorbenko, A., Kharchenko, V., Tarasyuk, O.: FMEA- technique of Web Services Analysis and Dependability Ensuring. In: Butler, M., Jones, C., Romanovsky, A., Troubitsyna, E. (eds.) *Rigorous Development of Complex Fault-Tolerant Systems*. LNCS, vol. 4157, pp. 153–168. Springer, Heidelberg (2006)
10. Isaksen, U., Bowen, J.P., Nissanke, N.: *System and Software Safety in Critical Systems*. Technical Report RUCS/97/TR/062/A, The University of Reading, UK (1997)
11. Leveson, N.G., Cha, S.S., Shimeall, T.J.: Safety Verification of Ada Programs using Software Fault Trees. *IEEE Software* 8(4), 48–59 (1991)
12. MIL-STD-1629A: Procedures for Performing a Failure Modes, Effects and Criticality Analysis. Department of Defense, Washington, DC (1980)
13. Papadopoulos, Y., Parker, D., Grante, C.: Automating the Failure Modes and Effects Analysis of Safety Critical Systems. In: *Proceedings of HASE'04, FL*, pp. 310–311 (2004)
14. Redmill, F.: Exploring Subjectivity in Hazard Analysis. *Engineering Management Journal (IEE)* 12(3) (2002)
15. Redmill, F., Chudleigh, M., Catmur, J.: *System Safety: HAZOP and Software HAZOP*. John Wiley & Sons Ltd, Chichester (1999)
16. Reifer, D.J.: Software Failure Modes and Effects Analysis. *IEEE Transactions on Reliability* R-28(3), 247–249 (1979)

17. Roland, E., Moriarty, B.: Failure Mode and Effects Analysis. In: System Safety Engineering and Management, 2nd edn. John Wiley & Sons, Chichester (1990)
18. Rosenberg, D., Scott, K.: Use Case Driven Object Modeling with UML: A Practical Approach. Addison-Wesley, Reading (1999)
19. Trader project web site (2006), <http://www.esi.nl/site/projects/trader>
20. Wallace, M.: Modular Architectural Representation and Analysis of Fault Propagation and Transformation. In: Proceedings of FESCA, ENTCS vol. 141(3) (2005)
21. Yakoub, S., Cukic, B., Ammar, H.: Scenario-based Reliability Analysis of Component Based Software. In: Proceedings of ISSRE'99, Boca Raton, FL, pp. 22–31 (1999)
22. Zhou, J., Stalhane, T.: Using FMEA for early robustness analysis of Web-based systems. In: Proceedings of COMPSAC'04, Washington, DC, pp. 28–29 (2004)

Author Index

- Aksit, Mehmet 409
- Baresi, Luciano 337
- Batista, Thais V. 237
- Bondarev, Egor 188
- Buskens, Rick 163
- Casimiro, António 287
- Chen, DeJiu 39
- Cheng, Betty H.C. 115
- Coulson, Geoff 237
- Cuenot, Philippe 39
- Cukic, Bojan 89
- Dan, Asit 262
- de Lemos, Rogério 142
- Desovski, Dejan 89
- Dumitraş, Tudor 262
- Ebnenasir, Ali 115
- Gérard, Sébastien 39
- Gomes, Antônio Tadeu A. 237
- Gonzalez, Oscar 163
- Graydon, Patrick J. 362
- Grunske, Lars 188
- Guinea, Sam 337
- Heller, Christoph 316
- Inverardi, Paola 210
- Joolia, Ackbar 237
- Kaâniche, Mohamed 14
- Kaiser, Jörg 287
- Kanoun, Karama 14
- Kelly, Tim 383
- Knight, John C. 362
- Kolagari, Ramin Tavakoli 39
- Lindsay, Peter 188
- Lönn, Henrik 39
- Mostarda, Leonardo 210
- Narasimhan, Priya 262
- Paige, Richard 66
- Papadopoulos, Yiannis 188
- Parker, David 188
- Plebani, Marco 337
- Radjenovic, Alek 66
- Reiser, Mark-Oliver 39
- Roşu, Daniela 262
- Rugina, Ana-Elena 14
- Schalk, Josef 316
- Schneele, Stefan 316
- Servat, David 39
- Sorea, Maria 316
- Sozer, Hasan 409
- Strunk, Elisabeth A. 362
- Tekinerdogan, Bedir 409
- Tokar, Joyce L. 1
- Törngren, Martin 39
- Verissimo, Paulo 287
- Voss, Sebastian 316
- Weber, Matthias 39
- Wu, Weihang 383