Tiziana Margaria
Bernhard Steffen (Eds.)

LNCS 6416

Leveraging Applications of Formal Methods, Verification, and Validation

4th International Symposium
on Leveraging Applications, ISoLA 2010
Heraklion, Crete, Greece, October 2010, Proceedings, Part II

2 Part II

Springer

Lecture Notes in Computer Science 6416

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Tiziana Margaria Bernhard Steffen (Eds.)

Leveraging Applications of Formal Methods, Verification, and Validation

4th International Symposium
on Leveraging Applications, ISoLA 2010
Heraklion, Crete, Greece, October 18-21, 2010
Proceedings, Part II

Springer

Volume Editors

Tiziana Margaria
University of Potsdam
August-Bebel-Str. 89
14482 Potsdam
Germany
E-mail: margaria@cs.uni-potsdam.de

Bernhard Steffen
TU Dortmund University
Otto-Hahn-Str. 14
44227 Dortmund
Germany
E-mail: steffen@cs.tu-dortmund.de

Library of Congress Control Number: 2010936699

CR Subject Classification (1998): F.3, D.2.4, D.3, C.2-3, D.2, I.2

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

ISSN 0302-9743
ISBN-10 3-642-16560-5 Springer Berlin Heidelberg New York
ISBN-13 978-3-642-16560-3 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper 06/3180

Preface

This volume contains the conference proceedings of the 4th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation, ISoLA 2010, which was held in Greece (Heraklion, Crete) October 18–21, 2010, and sponsored by EASST.

Following the tradition of its forerunners in 2004, 2006, and 2008 in Cyprus and Chalchidiki, and the ISoLA Workshops in Greenbelt (USA) in 2005, in Poitiers (France) in 2007, and in Potsdam (Germany) in 2009, ISoLA 2010 provided a forum for developers, users, and researchers to discuss issues related to the adoption and use of rigorous tools and methods for the specification, analysis, verification, certification, construction, testing, and maintenance of systems from the point of view of their different application domains. Thus, the ISoLA series of events serves the purpose of bridging the gap between designers and developers of rigorous tools, and users in engineering and in other disciplines, and to foster and exploit synergetic relationships among scientists, engineers, software developers, decision makers, and other critical thinkers in companies and organizations. In particular, by providing a venue for the discussion of common problems, requirements, algorithms, methodologies, and practices, ISoLA aims at supporting researchers in their quest to improve the utility, reliability, flexibility, and efficiency of tools for building systems, and users in their search for adequate solutions to their problems.

The program of the symposium consisted of special tracks devoted to the following hot and emerging topics:

- Emerging services and technologies for a converging telecommunications/Web world in smart environments of the Internet of Things
- Learning techniques for software verification and validation
- Modeling and formalizing industrial software for verification, validation and certification
- Formal methods in model-driven development for service-oriented and cloud computing
- Tools in scientific workflow composition
- New challenges in the development of critical embedded systems—an "aeromotive" perspective
- Web science
- Leveraging formal methods through collaboration
- Resource and timing analysis
- Quantitative verification in practice
- Worst case traversal time (WCTT)
- Model transformation and analysis for industrial scale validation
- Certification of software-driven medical devices
- Formal languages and methods for designing and verifying complex engineering systems

- CONNECT: status and plan
- EternalS: mission and roadmap

and five co-located events

- Graduate/postgraduate course on "Soft Skills for IT Professionals in Science and Engineering"
- RERS—challenge on practical automata learning
- IT Simply Works—editorial meeting (ITSy)
- CONNECT internal meeting
- EternalS Task Force meetings

We thank the Track organizers, the members of the Program Committee and their subreferees for their effort in selecting the papers to be presented.

Special thanks are due to the following organization for their endorsement: EASST (European Association of Software Science and Technology), and our own institutions—the TU Dortmund, and the University of Potsdam.

August 2010

Tiziana Margaria
Bernhard Steffen

Organization

Committees

Symposium Chair

Tiziana Margaria	University of Potsdam, Germany

Program Chair

Bernhard Steffen	TU Dortmund, Germany

Program Committee

Yamine Ait Ameur	LISI/ENSMA, France
Frédéric Boniol	IRIT/ENSEEIHT, France
Anne Bouillard	ENS Cachan, France
Marc Boyer	ONERA, France
Karin Breitman	PUC-Rio, Brazil
Marco Antonio Casanova	PUC-Rio, Brazil
Samarjit Chakraborty	TU München, Germany
Noel Crespi	Institut Telecom, France
Rémi Delmas	ONERA, France
Howard Foster	City University London, UK
Pierre-Loïc Garoche	ONERA, France
Dimitra Giannakopoulou	CMU/NASA Ames, USA
Stefania Gnesi	ISTI-CNR, Pisa, Italy
Kevin Hammond	University of St Andrews, UK
Boudewijn Haverkort	ESI, The Netherlands
Michael Hinchey	LERO, Ireland
Valérie Issarny	INRIA, France
Visar Januzaj	TU Darmstadt, Germany
He Jifeng	East China Normal University, China
Joost-Pieter Katoen	RWTH Aachen University, Germany
Joost Kok	Leiden University, The Netherlands
Jens Knoop	Vienna University of Technology, Austria
Stefan Kugele	TU München, Germany
Anna-Lena Lamprecht	TU Dortmund, Germany
Kim G. Larsen	Aalborg University, Denmark
Boris Langer	Diehl Aerospace, Germany

Mark Lawford	McMaster University, Canada
Gyu Myoung Lee	Institut Télécom, France
Björn Lisper	Mälardalen University, Sweden
Zhiming Liu	UNU-IIST, Macao
Tom Maibaum	McMaster University, Canada
Steven Martin	LRI, France
Dominique Mery	University Nancy, France
Pascal Montag	Daimler AG, Germany
Alessandro Moschitti	University of Trento, Italy
Corina Pasareanu	CMU/NASA Ames, USA
Alexander K. Petrenko	ISPRAS, Moscow, Russia
Abhik Roychoudhury	NUS, Singapore
Christian Schallhart	Oxford University, UK
Jean-Luc Scharbarg	IRIT, France
Amal Seghrouchni	University Pierre and Marie Curie, France
Laura Semini	Pisa University, Italy
Giovanni Stea	Pisa University, Italy
Eric Thierry	ENS Lyon, France
Helmut Veith	Vienna University of Technology, Austria
Alan Wassyng	McMaster University, Canada
Virginie Wiels	ONERA, France
Mark D. Wilkinson	Heart and Lung Institute, and Canada
Rostislav Yavorskiy	Microsoft UK/Moscow, Russia
Lenore Zuck	University of Illinois at Chicago, USA

Table of Contents – Part II

EternalS: Mission and Roadmap

Formal Methods in Model-Driven Development for Service-Oriented and Cloud Computing

Quantitative Verification in Practice

CONNECT: Status and Plans

Certification of Software-Driven Medical Devices

Modeling and Formalizing Industrial Software for Verification, Validation and Certification

Resource and Timing Analysis

Table of Contents – Part I

New Challenges in the Development of Critical Embedded Systems – An "aeromotive" Perspective

Formal Languages and Methods for Designing and Verifying Complex Embedded Systems

Worst-Case Traversal Time (WCTT)

Tools in Scientific Workflow Composition

Emerging Services and Technologies for a Converging Telecommunications / Web World in Smart Environments of the Internet of Things

Web Science

Model Transformation and Analysis for Industrial Scale Validation

Learning Techniques for Software Verification and Validation

Introduction to the EternalS Track: Trustworthy Eternal Systems via Evolving Software, Data and Knowledge

Alessandro Moschitti

Department of Computer Science and Information Engineering
University of Trento
Via Sommarive 14, 38100 POVO (TN) - Italy
moschitti@disi.unitn.it

Latest research work within ICT has outlined that future systems must possess the ability of adapting to changes in user requirements and application domains. Adaptation and evolution depend on several dimensions, e.g., time, location, and security conditions, expressing the diversity of the context in which systems operate.

The Coordination Action (CA): Trustworthy Eternal Systems via Evolving Software, Data and Knowledge (EternalS) aims at coordinating research in the above-mentioned areas by means of a researcher Task Force and community building activities, where the organization of workshops and conferences is one of the important tools used for such purpose. EternalS aims at creating the conditions for mutual awareness and cross-fertilization among the four ICT-Forever Yours - FET projects (FP7-ICT-Call 3): LivingKnowledge, HATS, Connect and SecureChange. These projects are currently conducting research in key ICT areas: (i) automatic learning of systems capable of analyzing knowledge and diversity with respect to their complex semantic interactions and evolution over time, (ii) exploitation of formal methods for the design and networking of adaptive and evolving software systems; and (iii) design of security policies and fully connected environment. The above-mentioned projects will help EternalS to actively involve many researchers from both academic and industrial world in its action.

The EternalS track at ISOLA 2010 represents a first milestone on establishing task forces and in recruiting stakeholders of its research topics. For this issue, the track presents aims and results of three FET projects, HATS and SecureChange and LivingKnowledge, outlined in three different talks. Moreover, the 3D-Life project, which aims at fostering the creation of sustainable and long-term relationships between existing research groups in Media Internet, will be introduced in the fourth talk.

The work above represents the initial material on which the CA is working by means of three different task forces. These will be illustrated along with their current results and future plans in the following talks: (i) *Modeling and Analyzing Diversity*, (ii) *Modeling and Managing System Evolution* and (iii) *Self-adaptation and Evolution by Learning*. Moreover, since one of the most valuable contribution

T. Margaria and B. Steffen (Eds.): ISoLA 2010, Part II, LNCS 6416, pp. 1–2, 2010.

of EternalS will be the indications for future promising and needed research, the last talk will be devoted to an *Overview of Roadmapping by EternalS*.

Finally, although there is no talk about the Connect project in this track, it will be presented in many contributions of the other ISOLA tracks, e.g. in the afternoon meeting (following EternalS sessions), a series of six talks will detail the current research and results of Connect.

The exciting program of the EternalS track will be concluded with a general discussion on the presented ideas and topics, which aims at creating future research collaborations.

EternalS track chair
Alessandro Moschitti

HATS: Highly Adaptable and Trustworthy Software Using Formal Methods

Reiner Hähnle

Department of Computer Science and Engineering
Chalmers University of Technology, 41296 Gothenburg, Sweden
http://www.cse.chalmers.se/~reiner, http://www.hats-project.eu

Abstract. The HATS project develops a formal method for the design, analysis, and implementation of highly adaptable software systems that are characterized by high demand on trustworthiness. Existing modeling formalisms leave gap between highly abstract, largely structural models and executable code on the implementation level. HATS aims to close this gap with an object-oriented, executable modeling language for adaptable, concurrent software components. It comes with tool suite based on analysis methods developed hand in hand with the language.

1 Introduction

The HATS project develops a formal method for the design, analysis, and implementation of highly *adaptable* software systems that are at the same time characterized by a high demand on *trustworthiness*.

Adaptability includes two aspects: anticipated *variability* as well as *evolvability*, i.e., unanticipated change. The first is, to a certain extent, addressed in modern software architectures such as software product families (SWPF). However, increasing product complexity (features, deployment) is starting to impose serious limitations. Evolvability over time is an even more difficult problem.

Current development practices do not make it possible to produce highly adaptable *and* trustworthy software in a large-scale and cost-efficient manner. The crucial limitation is the *missing rigour* of models and property specifications: informal or semi-formal notations lack the means to describe precisely the *behavioural* aspects of software systems: concurrency, modularity, integrity, security, resource consumption, etc.

2 Mind the Gap

Existing formal notations for specification of systems at the modeling level such as UML or FDL are mainly *structural* and lack adequate means to specify detailed behavior including datatypes, compositionality, concurrency. But without a formal notation for the behavior of distributed, component-based systems it is impossible to achieve automation of consistency checking, enforcement of security, trusted code generation, test case generation, specification mining, etc.

T. Margaria and B. Steffen (Eds.): ISoLA 2010, Part II, LNCS 6416, pp. 3–8, 2010.

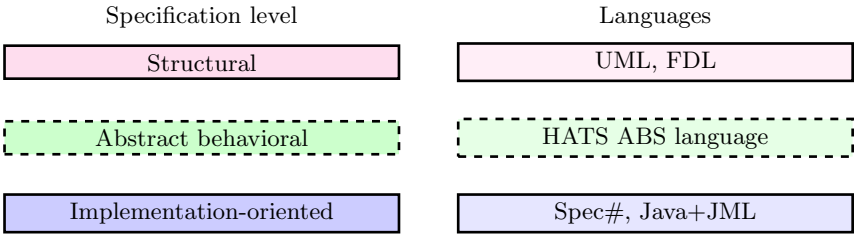


Fig. 1. Positioning of the HATS ABS language

At the same time, formal specification and reasoning about executable programs on the implementation level is by now well understood even for commercial languages such as JAVA or C and it is reasonably well supported by tools[1] The size and complexity of implementation languages, however, makes specification and verification extremely expensive. In addition, re-use is very hard to realize.

In conclusion, there is a gap between highly abstract modeling formalisms and implementation-level tools, visualized in Fig. 1. The HATS project addresses this specification gap by providing the following three ingredients:

1. An object-oriented, executable modeling language for adaptable, concurrent software components: the *Abstract Behavioral Specification* (ABS) language. Its design goal is to permit formal specification of concurrent, component-based systems on a level that abstracts away from implementation details but retains essential behavioral properties: concurrency model, component structure, execution histories, datatypes. The ABS language has a formal operational semantics which is the basis for unambiguous usage and rigorous analysis methods. The ABS language closes the mentioned gap, see Fig. 1.
2. A tool suite for analysis and development of ABS models as well as for analysis of executable code derived from these models:
   "Hard methods" typically strive for completeness or full coverage and require expert knowledge in the form of user interaction or detailed specifications. These include feature consistency, data integrity, security, property verification, and code generation.
   "Soft methods" typically are incomplete or non-exhaustive, but do not presuppose expert knowledge and are fully automatic. These include visualization, test case generation, specification mining, and type checking.

   One decisive aspect of the HATS project is to develop the analysis methods *hand in hand* with the ABS language to ensure *feasibility* of the resulting analyses.
3. A methodological and technological framework that integrates the HATS tool architecture and the ABS language.

[1] For example, KeY (www.key-project.org), Krakatoa (krakatoa.lri.fr), or VCC (research.microsoft.com/en-us/projects/vcc).

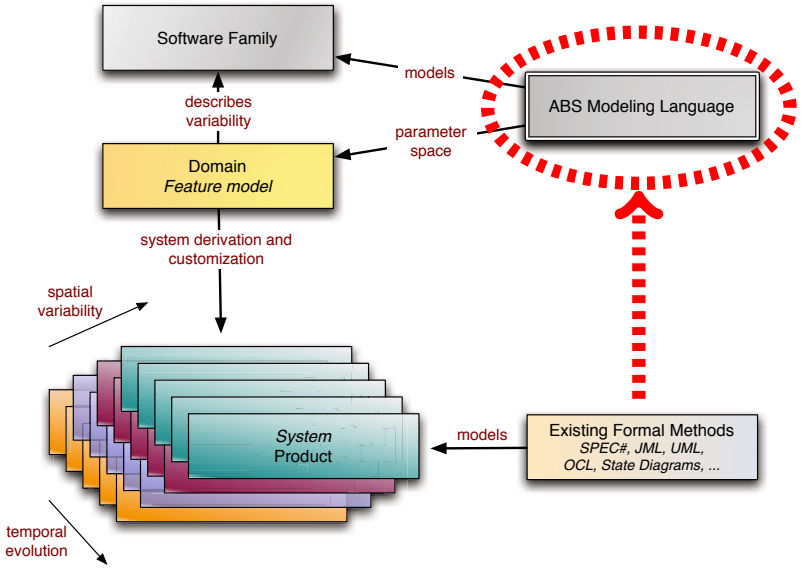


Fig. 2. Lifting formal methods to Family Engineering

As a main challenge in the development of the ABS language and HATS tool suite we identified (in addition to the technical difficulties) the need to make it *relevant* for industrial practice. Therefore, to keep the project firmly grounded, we orient the design of the ABS language and the HATS methodology along an empirically highly successful *informal* software development paradigm. In software product family-based development one separates *Family Engineering* which includes feature modeling and library development from *Application Engineering* where code is derived via selection, instantiation and composition.

In HATS we turn SWPF-based development into a rigorous approach based on formal specification. As visualized in Fig. 2, constructing a software family requires architecting both the *commonality*, that is, features, properties, and resources that are common to all products in the family, as well as the *adaptability*, that is, the varying aspects across the software family, in order to exploit them during the customization and system derivation process. Adaptability encompasses both anticipated differences among products in the family (*variability*), as well as the different products which appear over time due to evolving requirements (*evolvability*). Handling evolution is crucial as software undergoes changes during its lifetime; indeed, most future usages of a piece of software are not anticipated during its development. Therefore, variability and evolvability are the technological challenges that have to be addressed, when lifting formal modeling and analysis methods from the implementational level to the ABS level:

Variability. With this we mean functional variability and the invasive composition techniques used to instantiate feature models. A major challenge is

to understand which linguistic primitives are suited to formalize the desired concurrency-related aspects (including feature interactions but also failures, distribution, and parametrization on scheduling policies).

Evolvability. The key challenge here is to develop the theory, algorithms, and core mechanisms needed to build software systems that can be dynamically reconfigured—possibly even without service interruption—to adapt to changes that were *not anticipated* at the time the components which make up the running system were initially constructed.

Both aspects are strongly present in SWPF-based development where variability inherent to the feature space and deployment scenarios are an essential part of family engineering. Evolvability comes into play when new products with unanticipated features are to be created.

3 Main Results Achieved

During the first phase, the HATS project established a number of crucial results of which we highlight the most important ones:

Core ABS Language. The central achievement of HATS so far is the definition of the core of the ABS language. It consists of a (JAVA-like) syntax with a parser, a decidable type system with a compile-time type checker, and a formal operational semantics that permits simulation of ABS programs within the term rewriting engine Maude. To achieve maximal modularity and extensibility of the ABS language we decided a *layered* architecture with clearly defined tiers as depicted in Fig. 3. The core ABS is described in Deliverable D1.1A of the HATS project *Report on the Core ABS Language and Methodology: Part A*.[2]

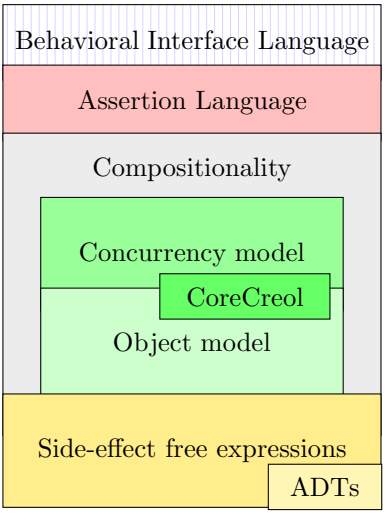


Fig. 3. ABS language layers

Delta Modeling. For the integration of features in the ABS language, we studied various formalisms and mechanisms; e.g., traits, context-oriented programming techniques, and delta modeling. We identified *delta modeling* as a promising approach to bridge the gap between features and ABS code. In delta modeling one distinguishes between a core implementation, containing the code common to each product, and deltas, containing code specific to some feature configuration(s). Deltas make changes to the core in order to integrate one or multiple features. Details are available in Deliverable D2.2.a *First Report on Feature Selection and Integration.*

[2] All HATS deliverables are public and are available from `www.hats-project.eu`

Incremental Verification. We started work on formal verification of ABS models: we developed a symbolic execution engine for the language CREOL whose concurrency model is the basis for the one in ABS. This will allow us to develop rapidly a formal verification framework for core ABS in the second project phase. We have studied three incremental verification techniques that are particularly suited to formal SWPF-based development with ABS: (i) combining symbolic execution and partial evaluation; (ii) proof reuse techniques in combination with the core/delta modeling methodology; (iii) lazy behavioral subtyping, based on combining syntax-driven inference systems such as type and effect systems with Hoare logics in order to track behavioral dependencies between classes.

Assertion Language. We investigated the foundation of an assertion language for the ABS which supports the specification and verification of properties of message sequences between objects, components, features. A major challenge is finding a proper formalization that is user-friendly, corresponds to the level of the ABS (clear object-oriented flavour) and lends itself easily to automated verification (both runtime testing and static checking). A promising framework for this formalization is provided by Attribute Grammars. We started to integrate histories of method calls and returns in JML and perform run-time testing of these assertions.

Behavior of Evolving Systems. In the evolution of systems, a central question is how to prove that the modification of a module does not affect the overall system behavior. We investigated this question with respect to object-oriented program modules as the modeling language was not fixed when we started. The first result is a compatibility criterion that allows to check whether a refactored module can be integrated into all possible contexts of the old module. We also worked on security monitor inlining for JAVA bytecode and started one of the first systematic investigations of monitor correctness for multi-threaded JVM. This work is reported in HATS Deliverable 3.1.a *First Report On Evolvable Systems.*

Resource Guarantees. Regarding trustworthiness we concentrated on resource guarantees. We took as a starting point an existing framework for the analysis of the *resource consumption* of JAVA and JAVA bytecode and its corresponding analyzer COSTA.[3] We obtained several extensions of this towards a cost analysis framework for HATS, including numeric fields, specifying and inferring *asymptotic* upper bounds, comparing cost expressions, and estimating the memory usage in the presence of *garbage collection.*

Requirements Analysis and Case Studies. In close collaboration with the HATS End-User Panel, we elicited high-level requirements of the HATS methodology. In addition, we described three representative case studies that will be refined and employed for evaluation purposes later on. The case studies should also be valuable for related projects. The case studies are reported in Deliverable 5.1 *Requirement Elicitation.*

[3] https://costa.ls.fi.upm.es/

In conclusion, a number of scalable and incremental approaches to analyze ABS models have been presented and are currently implemented/evaluated. Some of these generalize existing technology while others are completely new ideas that arose in the HATS context.

In order to keep this paper short, we decided not to supply references. A complete list of publications related to HATS is available from the www.hats-project.eu.

SecureChange: Security Engineering for Lifelong Evolvable Systems

Riccardo Scandariato[1] and Fabio Massacci[2]

[1] IBBT-DistriNet
Katholieke Universiteit Leuven
3001 Leuven, Belgium
[2] Dipartimento di Ingegneria e Scienza dell'Informazione
Università di Trento
38050 Trento, Italy

Abstract. The challenge of SecureChange is to re-invent security engineering for "eternal" systems. The project focuses on methods, techniques and tools to make change a first-class citizen in the software development process so that security is built into software-based systems in a resilient and certifiable way. Solving this challenge requires a significant re-thinking of all development phases, from requirements engineering to design and testing.

Keywords: Security, evolvability, software life-cycle.

1 Introduction

A reality-check on complex critical systems reveals that their life-cycle is characterized by a short design time (months) compared to the much longer operational time (many years or even decades). Due to these considerations, it is unlikely that such systems will remain unchanged during their life time. Hence, a primary need for those software-based systems is to be highly flexible and evolvable. At the same time, those systems expose strong security and dependability requirements as they handle sensitive data, impact our daily life, or put people's lives at stake. Current software engineering techniques are not equipped to deal with this conflicting situation. On one side, state-of-the-art research has unearthed methods and techniques to support *adaptability*. In this respect, autonomic systems represent, for instance, one of the most promising directions. However, the high flexibility of adaptable systems comes with the cost of little or no guarantees that the security properties are preserved with change. On the opposite side, very precise *verification* techniques work under the assumption that the properties to-be-verified and the system under verification are fixed over time. That is, they are conceived for rigid, inflexible systems.

The challenge of SecureChange is to re-invent security engineering for "eternal" systems. The project focusses on methods, techniques and tools to make *change* a first-class citizen in the software development process so that security is built and certified in software-based systems in a way that is resilient to

T. Margaria and B. Steffen (Eds.): ISoLA 2010, Part II, LNCS 6416, pp. 9–12, 2010.

change. Solving the above-mentioned challenge requires a significant re-thinking of all development phases, from requirements engineering to design and testing.

2 The Project at a Glance

All the activities carried out by the project consortium revolve around a common theme: understanding the effect of change on a given development artifact, conceive a way to trace such change, and develop techniques to incrementally deal with its impact. Evolution is categorized as being driven by (i) a change in the context (environment), (ii) a change in the behavioral or security requirements, and (iii) a change in the specification (design and implementation) of the system.

In order to achieve the ambitious objective stated in the previous section and in light of the above-mentioned flavors of change, the project has defined the following array of integrated activities, as summarized in Figure 1, which visualizes how these activities fit together in the life-cycle of an adaptive security-critical system.

Industrial Scenarios Validation (WP1). The techniques and tools developed within the project are validated in the context of three case studies: (i) evolvability of software running on portable devices like smart cards, (ii) evolvability of software running on a residential gateway providing digital home services, and (iii) evolvability of software running on the workstation of air traffic control operators.

Architecture and Process (WP2). As configuration management can no longer be separated from the development of the application, one objective is a configuration management process that is tightly intertwined with the development and life-cycle processes defined in the project. This process is supported by model-based tools that allow for automated reconfiguration as a reaction to change as well as verification of security properties in a changing context. To successfully validate this process in the proposed case-studies, it needs to be instantiated on the basis of a common architecture, which is rich enough to address the challenges of incompatible versions, hardware reconfiguration, scalability, and heterogeneity.

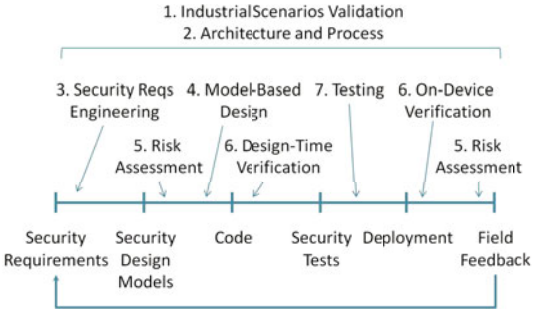


Fig. 1. Structure of the SecureChange project

Security Requirements Engineering (WP3). The project defines a new method that addresses the two possible interpretations of the concept of "evolving requirements". The first are the inventions that are necessary to consider in the current design, which are unknown but potentially vital future requirements. The second is how to redefine the flow from requirements to system, so that we can reconstruct how the system can evolve when requirements evolve.

Model-Based Design (WP4). The project develops a model-based design approach that is tailored to the needs of the secure development of adaptable systems. The approach glues requirement techniques to the final code and provides tools that allow the user to automatically analyze the models against its a-priori requirements, also taking into account the change scenarios that the risk analysis has defined likely. In particular, the approach takes into account various system views, including infrastructure elements, stakeholders, and hardware configurations—all of which may be subject to change during the system evolution.

Risk Assessment (WP5). The project also has the goal of generalizing existing methods and techniques for security assessment to cope with longevity. This addresses the issue of system documentation to facilitate assessment of security modulo changes as well as techniques, methods and tools for automatic or semi-automatic revalidation of existing assessment results modulo changes.

Design-Time and On-Device Verification (WP6). The project attempts to identify software programming models that can provably resist future errors, and how we can change the V&V process in order to cope with changes. Furthermore, the project provides embedded verification techniques targeting two goals. First, new code on an autonomous system must be verified in order to be safely loaded, and second, when new security requirements are provided, the verification process itself must be adapted.

Testing (WP7). The project also provides solutions for evaluating the impacts of changes, by means of model-based testing. By analyzing the differences between each model and their evolutions, testing strategies are defined that highlight the evolution or non-regression of the system.

3 Key Results

During the first project year, the SecureChange project obtained key results within all work packages. In the rest of this Section, a few highlights are presented:

1. Industrial Scenarios Validation. Based on change scenarios observed in the context of the industrial case studies, a taxonomy of change has been defined as a consortium-wide effort. The taxonomy provides the "scope" of the project activities. The classification has two sides: (i) how things change (problems) and (ii) how we deal with changes (solutions).
2. Architecture and Process. A methodology based on so-called Change Patterns has been defined to support adaptability to changing requirements at

the architectural level (i.e., co-evolution of security at the requirements and architectural level). Further, the framework for an artefact-centric change management process has been defined. The process keeps track of, and orchestrates the ripple effects among artifacts when a change is injected at any level.

3. Security Requirements Engineering. An approach to manage the evolution process of the requirements specification has been defined. The approach leverages so-called Change Rules. Evolution rules are specified as event-condition-action tuples. Events are changes in the requirements artifact (e.g., something is added and consequently a security inconsistency is detected) that trigger corrective actions, (e.g., the artefact is transformed to preserve the key security requirements).
4. Model-Based Design. Based on UMLsec, the UMLseCh extension has been defined. The extended notation allows to insert expected changes as predefined markers in the design models. These markers are leveraged for the formal security analysis carried out by a companion tool. The markers provide an indication of model elements that are added, deleted, or replaced. The tool performs an incremental analysis starting from both the original model (assumed correct) and the model of change.
5. Risk Assessment. A systematic approach has been defined (among others, as an extension of the CORAS method for security risk analysis) to identify the updates made to the target of evaluation, identify which security risks and parts of the security risk model are affected by the updates, and update the security risk model without having to change the unaffected parts.
6. Design-Time and On-Device Verification. At design-time, a programming model (failboxes) for verifying absence of dependency safety violations has been defined. At run-time, a programming model for writing safe programs that dynamically (un)load modules has been defined. Note that dynamically (un)loading modules is a key feature of adaptable systems.
7. Testing. An algorithm has been defined that, given the old test suite and the new design model, is able to compute: security tests that are outdated in light of the modeled change (obsolete), tests that are unchanged (reusable), tests that must be adapted or added (evolution).

The above results have led to several scientific publications that are available through the project portal[1].

[1] http://www.securechange.eu

3DLife: Bringing the Media Internet to Life

Ebroul Izquierdo, Tomas Piatrik, and Qianni Zhang

Queen Mary University of London, School of Electronic Engineering and Computer Science, Mile End Road, London

Abstract. "Bringing the Media Internet to Life" - or simply, 3DLife - is a European Union funded project that aims to integrate research conducted within Europe in the field of Media Internet. In this contribution, we give an overview of the project's main objectives and activities.

1 Introduction

It is widely argued that the next generation of Media Computing services will become the cornerstone of Information Society in our century. Its impact on the entertainment industry is already clear. However, its impact is expected to be much broader in changing the way that society delivers key services such as health care, learning and commerce.

3DLife is a Network of Excellence (NoE), funded under the "Cooperation" segment of the Seventh Framework Programme (FP7). It fosters the creation of sustainable and long-term relationships between existing national research groups and lays the foundations for a Virtual Centre of Excellence (VCE) in Media Internet. The key factors making the 3DLife consortium capable of reaching the posed objectives are reflected in the scientific quality of the partners and the diverse yet complementary research background they bring to the project. Those partners are Queen Mary, University of London (United Kingdom, Coordinator), Dublin City University (Ireland), Heinrich Hertz Institute, Fraunhofer (Germany), Informatics & Telematics Institute (Greece), Korea University (Korea), MIRALab, University of Geneva (Switzerland) and Telecom ParisTech (France). Since the 3DLife project comprehensively addresses several challenges of Media Internet, its impact is expected to be vast in several aspects of modern life including industry, academic and societal.

2 Highlights of 3DLife Activities

An important objective of the 3DLife NoE is to create sufficient momentum by integrating an existing large number of researchers and resources to enable realistic, efficient and to some extent autonomic media communication and interaction over the Internet. Three important integrative activities revolve around the project concept. A more detailed description of these activities is given in this section.

2.1 Integration and Sustainability

The 3DLife NoE promotes a series of collaborative activities of various types including Phd Student Exchanges, Researcher and Senior Researcher Visits, Industrial Placements, and Joint Postgraduate Courses. Furthermore, the ambition of 3DLife is

T. Margaria and B. Steffen (Eds.): ISoLA 2010, Part II, LNCS 6416, pp. 13–14, 2010.

to launch a VCE, namely the European Centre of Excellence in Media Computing and Communication (EMC^2), during the lifetime of the project. EMC^2 aims at nurturing Media Computing R&D in Europe and beyond by fostering integration and promoting cooperation between Academia and Industry. EMC^2 will promote additional collaborative activities such as Fellowships, Grand Challenges, Distinguished Lecture Series, Journal Special Issues, Workshop/Conference Series, and Short Term Collaborative Projects. It will bring together members' capabilities, knowledge and expertise to facilitate R&D funding through cooperative projects, joint research publications and technology transfer, while advancing the state of the art in Media Computing. Regarding scientific impact, EMC2 will endeavour to secure a dominant role in other well established scientific bodies, as IEEE Technical Committees, IET professional networks and EURASIP special interest groups. Several dissemination forums, standardization bodies, international conferences and exhibitions will be targeted and used to cement the ties between EMC^2 and the broad research community.

2.2 Cooperative Research

The 3DLife NoE is about media networking with enhanced interactivity and "autonomy". The project promotes cooperative research between its core partners on the following research areas: 3D Computer Graphics Methods, Media Analysis for 3D Data Generation, Virtual Humans Rendering and Animation, Distributed Immersive Virtual Worlds, Media Networking, Secure Networked Data Representations, etc. In the heart of this cooperative research lies the 3DLife software Framework for Integration, an internal section of 3DLife's web and network presence. It aims to enhance the collaboration between the project partners and to help compile and transfer their expert knowledge and technologies.

2.3 Spreading Excellence

3DLife plans to spread excellence in training, dissemination, and technology transfer. Spreading excellence of the scientific results and integrative efforts of this NoE targets three main groups: academics, industry/business and the non-specialist citizen. An important tool of dissemination is constituted by the 3DLife web site (www.3dlife-noe.eu). The website is part of the project's Collaboration Space aimed to be not only a mere repository of R&D results, but also a facilitator of the interaction between researchers, companies and experts, improving knowledge sharing, and supporting a culture of innovation among them. In addition, 3DLife has active social groups on popular social network platforms such as FaceBook, Twitter and LinkedIn. Latest activities and news in the project are constantly broadcasted through these channels to interested audience. Another important aspect of this activity is joint publications and coordination of special sessions at important international and well established conferences. These will help to spread excellence outside the network and to enlarge the network audience.

3 Conclusion

The 3DLife NoE aims at heavily impacting and influencing main constituents of the Media Internet by integrating, disseminating and sharing the different technologies. We believe that by helping you to understand the project activities, we are paving the way towards cooperation between your institution and the 3DLife project.

LivingKnowledge: Kernel Methods for Relational Learning and Semantic Modeling

Alessandro Moschitti

Department of Computer Science and Information Engineering
University of Trento
Via Sommarive 14, 38100 POVO (TN) - Italy
moschitti@disi.unitn.it

Abstract. Latest results of statistical learning theory have provided techniques such us pattern analysis and relational learning, which help in modeling system behavior, e.g. the semantics expressed in text, images, speech for information search applications (e.g. as carried out by Google, Yahoo,..) or the semantics encoded in DNA sequences studied in Bioinformatics. These represent distinguished cases of successful use of statistical machine learning. The reason of this success relies on the ability of the latter to overcome the critical limitations of logic/rule-based approaches to semantic modeling: although, from a knowledge engineer perspective, hand-crafted rules are natural methods to encode system semantics, noise, ambiguity and errors, affecting dynamic systems, prevent them from being effective.

One drawback of statistical approaches relates to the complexity of modeling world objects in terms of simple parameters. In this paper, we describe kernel methods (KM), which are one of the most interesting results of statistical learning theory capable to abstract system design and make it simpler. We provide an example of effective use of KM for the design of a natural language application required in the European Project LivingKnowledge[1].

Keywords: Kernel Methods; Structural Kernels; Support Vector Machines; Natural Language Processing.

1 The Data Representation Problem

In recent years, a considerable part of Information Technology research has been addressed to the use of machine learning for the automatic design of critical system components, e.g. automatic recognition/classification of critical data patterns. One of the most important advantages with respect to manually coded system modules is the ability of learning algorithms to automatically extract the salient properties of the target system from training examples. This approach can produce semantic models of system behavior based on a large number of attributes, where the values of the latter can be automatically learned. The

[1] http://livingknowledge-project.eu/

T. Margaria and B. Steffen (Eds.): ISoLA 2010, Part II, LNCS 6416, pp. 15–19, 2010.

statistically acquired parameters make the overall model robust and flexible to unexpected system condition changes. Unfortunately, while attribute values and their relations with other attributes can be learned, the design of attributes suitable for representing the target system properties (e.g. a system state) has to be manually carry out. This requires expertise, intuition and deep knowledge about the expected system behavior. For example, how can system module structures be converted into attribute-value representations?

Previous work on applied machine learning research (see the proceedings of ICML, ECML, ACL, SIGIR and ICDM conferences)[2] has shown that, although the choice of the learning algorithm affects system accuracy, feature (attribute) engineering more critically impacts the latter. Feature design is also considered the most difficult step as it requires expertise, intuition and deep knowledge about the target problem. Kernel methods is a research line towards alleviating the problem above.

2 Data Representation via Kernel Methods

Kernel Methods (KM) are powerful techniques developed within the framework of statistical learning theory [18]. They can replace attributes in learning algorithms simplifying data encoding. More specifically, kernel functions can define structural and/or semantic similarities between data objects at abstract level by replacing the similarity defined in terms of attribute matching.

The theory of KM in pattern analysis is widely discussed in [17] whereas an easier introduction can be grasped from the slides available at `http://disi.unitn.eu/~moschitt/teaching.html`. The main idea of KM is expressed by the following two points:

(a) directly using a similarity function between instances in learning algorithms, thus avoiding explicit feature design; and
(b) such function implicitly corresponds to attribute matching (more precisely scalar product) defined in huge feature spaces (possibly infinite), e.g. similarity between structures can be defined as substructure matching in the substructure space.

The first point states that instead of describing our data, e.g. a data stream, in terms of features (which aim at capturing the most interesting properties or behavior), it is enough to define a function capable to measure the similarity between any pair of different data objects, e.g. pairs of streams.

The second bullet emphasizes the great power of KM as the representations that can be modeled with them are extremely rich and are not computationally limited by the size of feature spaces.

[2] ICML and ECML are the International and European Conferences of Machine Learning, respectively. ACL is the Conference for Association of Computational Linguistics; IR is the most important conference for Information Retrieval and ICDM is the International Conference on Data Mining.

KM effectiveness has been shown in many ICT fields, e.g. in Bioinformatics [16], Speech Processing [1], Image Processing [5], Computational Linguistics [9], Data Mining [4] and so on. In particular, KM have been used to encode syntactic and/or semantic structures in the form of trees and sequences in learning algorithms, e.g. [2,3,7,19,10,20,14,11,13,12].

Given the wide and successful use of KM, they have been applied in the LivingKnowledge project to model several aspects of automatic knowledge acquisition and management, which are basic building blocks required by the project.

3 Using Kernels for Semantic Inference in LivingKnowledge

Judgements, assessments and opinions play a crucial role in many areas of our societies, including politics and economics. They reflect knowledge diversity in perspective and goals. The vision inspiring LivingKnowledge (LK) is to consider diversity as an asset and to make it traceable, understandable and exploitable, with the goal to improve navigation and search in very large multimodal datasets (e.g., the Web itself).

To design systems that are capable of automatically analyzing opinions in *free text*, it is necessary to consider syntactic/semantic structures of natural language expressed in the target documents. Although several sources of information and knowledge are considered in LK, we here illustrate an example focused on text. Given a natural language sentence like for example:

They called him a liar.

the opinion analysis requires to determine: (i) the opinion holder, i.e. *They*, (ii) the direct subjective expressions (DSEs), which are explicit mentions of opinion, i.e. *called*, and (iii) the expressive subjective elements (ESEs), which signal the attitude of the speakers by means of the words they choose, i.e. *liar*.

In order to automatically extract such data, the overall sentence semantics must be considered. In turn, this can be derived by representing the syntactic and shallow semantic dependencies between sentence words. Figure 1 shows a graph representation, which can be automatically generated by off-the-shelf syntactic/semantic parsers, e.g. [6], [8]. The oriented arcs, above the sentences, represent syntactic dependencies whereas the arcs below are shallow semantic

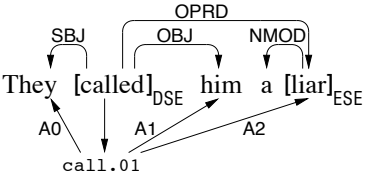


Fig. 1. Syntactic and shallow semantic structure

(or semantic role) annotations. For example, the predicate *called*, which is an instance of the PropBank [15] frame `call.01`, has three semantic arguments: the Agent (A0), the Theme (A1), and a second predicate (A2), which are realized on the surface-syntactic level as a subject, a direct object, and an object predicative complement, respectively.

Once the richer representation above is available, we need to encode it in the learning algorithm, which will be applied to learn the functionality (subjective expression segmentation and recognition) of the target system module, i.e. the opinion recognizer. Since such graphs are essentially trees, we exploit the ability of tree kernels [10] to represent them in terms of subtrees, i.e. each subtree will be generated as an individual feature of the huge space of substructures.

Regarding practical design, kernels for structures such us trees, sequences and sets of them are available in the SVM-Light-TK toolkit (`http://disi.unitn.it/moschitti/Tree-Kernel.htm`). This encodes several structural kernels in Support Vector Machines, which is one of the most accurate learning algorithm [18].

Our initial test on the LivingKnowledge tasks suggests that kernel methods and machine learning are an effective approach to model the complex semantic phenomena of natural language.

4 Conclusion

Recently, Information Technology research has been addressed to the use of machine learning for automatic design of critical system components, e.g. automatic recognition of critical data patterns. The major advantage is that the system behavior can be automatically learned from training examples. The most critical disadvantage is the complexity to model effective system parameters (attributes), especially when they are structured.

Kernel Methods (KM) are powerful techniques that can replace attribute-value representations by defining structural and/or semantic similarities between data objects (e.g. system states) at abstract level. For example, to encode the information in a data stream, we just define a function measuring the similarity between pairs of different streams: such function can be modeled in extremely rich and large feature spaces.

A considerable amount of previous work shows the benefit of employing KM and our initial study in LivingKnowledge, whose application domain requires to model complex textual and image information, further demonstrate their benefits.

Acknowledgements

This research has been supported by the EC project, EternalS: Trustworthy Eternal Systems via Evolving Software, Data and Knowledge, project number FP7 247758.

References

1. Campbell, W.M.: Generalized linear discriminant sequence kernels for speaker recognition. In: International Conference on Acoustics, Speech, and Signal Processing (2002)
2. Collins, M., Duffy, N.: New Ranking Algorithms for Parsing and Tagging: Kernels over Discrete Structures, and the Voted Perceptron. In: Proceedings of ACL 2002 (2002)
3. Culotta, A., Sorensen, J.: Dependency Tree Kernels for Relation Extraction. In: Proceedings of ACL 2004 (2004)
4. Gärtner, T.: A survey of kernels for structured data. SIGKDD Explor. Newsl. 5(1), 49–58 (2003)
5. Grauman, K., Darrell, T.: The pyramid match kernel: Discriminative classification with sets of image features. In: International Conference on Computer Vision (2005)
6. Johansson, R., Nugues, P.: Dependency-based syntactic–semantic analysis with PropBank and NomBank. In: Proceedings of the Shared Task Session of CoNLL 2008 (2008)
7. Kudo, T., Matsumoto, Y.: Fast methods for kernel-based text analysis. In: Proceedings of ACL 2003 (2003)
8. Moschitti, A., Coppola, B., Giuglea, A., Basili, R.: Hierarchical semantic role labeling. In: CoNLL 2005 shared task (2005)
9. Moschitti, A.: A study on convolution kernels for shallow semantic parsing. In: Proceedings of ACL 2004 (2004)
10. Moschitti, A.: Efficient convolution kernels for dependency and constituent syntactic trees. In: Fürnkranz, J., Scheffer, T., Spiliopoulou, M. (eds.) ECML 2006. LNCS (LNAI), vol. 4212, pp. 318–329. Springer, Heidelberg (2006)
11. Moschitti, A.: Making tree kernels practical for natural language learning. In: Proceedings of EACL 2006 (2006)
12. Moschitti, A.: Kernel methods, syntax and semantics for relational text categorization. In: Proceeding of CIKM 2008 (2008)
13. Moschitti, A., Quarteroni, S., Basili, R., Manandhar, S.: Exploiting syntactic and shallow semantic kernels for question/answer classification. In: Proceedings of ACL 2007 (2007)
14. Moschitti, A., Zanzotto, F.M.: Fast and effective kernels for relational learning from texts. In: ICML 2007 (2007)
15. Palmer, M., Gildea, D., Kingsbury, P.: The proposition bank: An annotated corpus of semantic roles. Comput. Linguist. 31(1), 71–106 (2005)
16. Schšlkopf, B., Guyon, I., Weston, J.: Statistical learning and kernel methods in bioinformatics. In: Artificial Intelligence and Heuristic Methods in Bioinformatics (2003)
17. Taylor, J.S., Cristianini, N.: Kernel Methods for Pattern Analysis. Cambridge University Press, Cambridge (2004)
18. Vapnik, V.N.: Statistical Learning Theory. Wiley-Interscience, Hoboken (1998)
19. Zhang, D., Lee, W.S.: Question classification using support vector machines. In: Proceedings of the 26th annual international ACM SIGIR conference on Research and development in informaion retrieval, pp. 26–32. ACM Press, New York (2003)
20. Zhang, M., Zhang, J., Su, J.: Exploring Syntactic Features for Relation Extraction using a Convolution tree kernel. In: Proceedings of NAACL (2006), `http://www.aclweb.org/anthology/N/N06/N06-1037`

Task Forces in the EternalS Coordination Action

Reiner Hähnle

Task Forces Coordinator of the EternalS Coordination Action
Department of Computer Science and Engineering
Chalmers University of Technology, 41296 Gothenburg, Sweden
http://www.cse.chalmers.se/~reiner, https://www.eternals.eu/

Abstract. We describe the scope, organization, and expected outcomes of the Task Forces of the EternalS Coordination Action. The goal of the Task Forces is to provide structure and focus to the activities pursued in EternalS while retaining essential openness to bottom-up initiatives.

1 Introduction

The EternalS Coordination Action (CA) is about the need to make software and knowledge-based systems capable of adapting to changes in user requirements and application domains. It aims to coordinate research in this key area comprising several dimensions including time, location, and security conditions all of which reflect the diversity of the context where systems operate.

The main instrument in EternalS to structure this rather wide field of interest into focused and manageable units is through the establishment of *Task Forces* (TFs). In this brief note we motivate and explain the adopted Task Force structure (Sect. 2), describe their organization (Sect. 3), and list expected outcomes (Sect. 4).

2 The EternalS Task Forces

The EternalS Task Forces are meant to give a coarse-grained structure to the EternalS theme of adaptability. As the aim of the CA is to encourage collaboration among the Integrated Projects (IPs) represented in EternalS we sought a structure that runs *cross-cutting* to the interests of the IPs. This excluded, for example, software-based vs. knowledge-based, software vs. middleware, or formal vs. informal as structuring criteria. Instead, we chose the dimensions of spatial vs. temporal adaptability as well as of planned vs. autonomous evolvability. We believe that this classification makes the Task Forces attractive for external stakeholders to join while providing the necessary focus and encouraging collaboration:

TF1. *Diversity awareness and management:* this can be seen as the "spatial" dimension of adaptability where we intend to gather expertise in mechanisms that model/handle diversity in IT-systems.

T. Margaria and B. Steffen (Eds.): ISoLA 2010, Part II, LNCS 6416, pp. 20–22, 2010.

TF2. *Time awareness and management:* this is one aspect of the "temporal" dimension of adaptability: that of planned evolvability of dynamic systems. We also put a special emphasis on security.

TF3. *Self-adaptation and evolution by learning:* here we collect expertise in learning/control techniques for autonomous evolution of systems.

A detailed description of the scope of each TF is found in *Task Force Descriptions* within the EternalS session of ISOLA.

3 Organization of EternalS Task Forces

The objective of a task force is to provide a concrete platform where stakeholders with a common interest in the development of "eternal" systems can meet, discuss, exchange ideas, take stock, suggest new initiatives, etc. With stakeholders we mean physical persons with a research or application interest in the area of each task force. The members of each task force are recruited from the four Integrated Projects of the "Forever Yours" cluster, from the institutions present in the EternalS CA, from the international research community at large, and from companies with an interest in the theme of EternalS. Typically, a TF could comprise a core of 6–10 members who actively drive its agenda, plus 10–15 members-at-large who contribute to discussions and the various outcomes (Sect. 4).

Membership is informal, free, and does not lead to any obligation for the TF member or his/her employer. Each TF has a leader who coordinates the work within a TF. The TF leader also maintains a list of TF members which is visible at the EternalS web portal `https://www.eternals.eu/`.

Specifically, each TF provides: (i) a fertile environment for facilitating the emergence of bottom-up research initiatives from the TF members themselves or from any group of stakeholders; (ii) a coarse-grained classification, where the above initiatives can be conceptually located; (iii) a pool of experts for discussing, developing and judging the emerging initiatives; and (iv) a platform to bootstrap communities and ideas that will constitute the initial seed for promoting bottom-up initiatives.

The classification of the work of the CA into task forces does not constrain in any way the type of the emerging initiatives as any new topic can be located according to the *nearest neighbor principle*. When an external (to the TFs) group of stakeholders proposes ideas (e.g., for collaboration or for discussing new topics), these will be considered by the most relevant TF.

To make the description of the TFs reasonably concrete, we defined a number of interesting *initial* topics along with possible collaborations between the already known stakeholders. These topics as well as the internal structure and a list of initial stakeholders are described separately in each *Task Force Description* within the EternalS session of ISOLA 2010.

4 Envisaged Outcomes of EternalS Task Forces

We list possible concrete outcomes of each TF. Except where explicitly noted, this list is neither inclusive nor mandatory. It is up to the initiative of the TF leader and the TF members to decide what they think is important. *The concrete work of the TFs is driven bottom-up.* Obviously, multiple activities and initiatives can be pursued simultaneously.

1. A survey of the state-of-art of research in the area of the TF. The aim is publication in a suitable journal. Such a survey is mandatory and is an EternalS Deliverable with due date January 2011.
2. Identification of research topics where the different IPs represented in EternalS can collaborate. Ideally, this would result in a decision of the Steering Committees of the involved IPs to collaborate on certain tasks.
3. TFs can be forums in which new project proposals are developed. Importantly, such initiatives are not limited to members of the IPs represented in EternalS or to the European area.
4. Organization of annual TF meetings as part of the plenary EternalS meetings in Spring 2011 and Spring 2012. The format, purpose, and composition of these meetings are completely open and can be adapted to the agenda of each TF.
5. Written summaries of the results of the Task Force after each annual EternalS workshop. The content of these is determined by what each TF decides to work on. It can be anything including (but not limited to) research papers, road maps, white papers, survey papers, inputs to standardizing committees, etc.

5 Conclusion

EternalS Task Forces provide an open and very flexible organizational framework for bottom-up initiatives within trustworthy, highly adaptable, and long-lived IT-systems. They structure the research within the EternalS theme along the dimensions of spatial/temporal adaptability and of planned/autonomous evolvability.

Modeling and Analyzing Diversity
Description of EternalS Task Force 1

Ina Schaefer

Department of Computer Science and Engineering
Chalmers University of Technology
41296 Gothenburg, Sweden
http://www.cse.chalmers.se/~schaefer

Abstract. We describe the objectives and vision of the EternalS Task Force 1 "Diversity Awareness and Management". We present its organization and workplan. The goal of the task force is to provide a platform for collaboration towards a unifying framework for modeling and analyzing diversity of system data and behavior in all development phases.

1 Introduction

Many, if not most, large digital systems are developed for a plethora of differing applications contexts and deployment scenarios. Hence, systems should embed into the context of their operation. This requires that the anticipated diversity in the application and deployment scenarios is taken into account during all phases of system development.

EternalS Task Force 1 "Diversity Awareness and Management" focuses on modeling and analyzing *anticipated diversity* in software systems. It considers specification and implementation techniques for diverse systems. Furthermore, efficient development processes for diverse systems and associated quality assurance techniques are in the scope of the task force. In this brief note, we present the objectives and vision of the task force (Section 2) and describe its organization and work plan (Section 3).

2 Objectives and Vision

In order to deal with system diversity during system design, appropriate modelling and specification techniques, for diversity in data and knowledge that is processed by the system as well as for diverse system behavior, are required. Diversity in terms of data and knowledge is in particular interesting, as this may provide insights into the causes for diversity also in the system's behavior. One objective of the task force is to find suitable abstractions from the plethora of application scenarios in which a system is deployed and from the different binding times of variability to capture relevant aspects of diverse systems.

One emphasis of the task force is the efficient development of diverse systems where a particular focus lies on model-centric development processes. Promising

T. Margaria and B. Steffen (Eds.): ISoLA 2010, Part II, LNCS 6416, pp. 23–25, 2010.

approaches for diverse system development are software product line engineering aiming at managed reuse of development artifacts, and model-driven engineering concentrating on platform-independent aspects during system development.

Another objective of the task force is the management of diversity in the system architecture, as well as the controlled integration of different components in a concurrent setting. Modularization techniques used in modern programming languages, such as traits, aspects, or mixins are considered regarding their potential to implement diverse systems. Additionally, context-oriented programing, domain-specific languages, and generative programming approaches are in the realm of the task force.

Along with the different modeling and implementation techniques at each design stage, the task force focuses on validation and verification methods to guarantee essential system qualities, such as security, consistency, correctness, and efficiency. These quality assurance techniques have to be devised to deal with the special requirements of diversity. Runtime monitoring as well as diversity model mining can ensure the quality of diverse systems at runtime.

The vision of the task force is to devise a unifying framework for modeling and analyzing diversity of system data and behavior in all development phases. This goal can be achieved by a model-centric development process for diverse systems relying on suitable modelling and implementation techniques for diversity in data and behavior at each design stage. Validation and verification techniques that can be integrated with the development process allow assuring essential system qualities, including security, correctness, consistency, and efficiency.

3 Organization and Work Plan

The EternalS Task Force 1 consists of 14 members that are from the FET-FY projects HATS and Connect and external experts from the software product line community. The HATS project is involved with Ralf Carbon (Fraunhofer IESE, Germany), Dave Clarke (KU Leuven, Belgium), Reiner Hähnle (Chalmers U, Sweden) and Ina Schaefer (TF Leader, Chalmers U, Sweden). Animesh Pathak (INRIA Paris-Rocquencourt, France) and Antonino Sabetta (ISTI-CNR Pisa, Italy) joined the task force from the Connect project.

The external experts of Task Force 1 with their affiliation and main research area are the following: Sven Apel (U Passau, Germany - Feature-oriented software development), David Benavides (U Seville, Spain - Feature model analysis), Lorenzo Bettini (U Torino, Italy - Programming languages for fine-grained code reuse), Götz Botterweck (Lero, Ireland - Software product line engineering), Pascal Costanza (VU Brussel, Belgium - Context-oriented programming), Christian Kästner (U Marburg, Germany - Preprocessor-based software product line development), Rick Rabiser (JKU Linz, Austria - Automated product line engineering), and Salvador Trujillo (IKERLAN, Spain - Model-driven development of software product lines).

The first activity of the task force is to compile a state-of-the-art survey on modelling and implementation approaches for diverse systems and accompaning

quality assurance techniques. Furthermore, the task force will organize an annual workshop to foster collaboration and discussions between the task force members. Each workshop will result in a document summarizing the state-of-the-art, identifying research challenges and outline future research directions that can be taken up by task force members in joint initiatives. These initiatives can result in new project proposals, in particular in the European context.

The task force will further initiate a scientific workshop on diverse system engineering, co-located with a major software engineering conference, to bring together the scientific communities working on diverse systems and to raise awareness on the particular challenges system diversity poses for system engineering.

4 Conclusion

EternalS Task Force 1 "Diversity Awareness and Management" focusses on anticipated diversity in software systems. It provides a platform to foster discussion and collaboration towards a unifying framework for modeling and analyzing system diversity in all software development phases.

Modeling and Managing System Evolution
Description of EternalS Task Force 2

Michael Hafner

Department of Computer Science and Engineering
University of Insbruck
6020 Innsbruck, Austria
http://www.qe-informatik.uibk.ac.at

Abstract. In this contribution we describe the scope, obectives, and outcomes of the EternalS Task Force 2 (TF2) - *Time Awareness and Management*. Task Force 2 will foster the discussion and the collection of novel ideas around the problem of maintaining the quality of software-based systems that evolve over time. Security is a topic of special interest.

1 Introduction

TF2 focuses on the monitored and managed *evolution* of security-critical systems from the three perspectives: development, deployment, and operation.

The question on how to cope with imposed or induced change is particularly challenging in the context of trends driving the so-called *Future Internet*. At the infrastructure level, the Future Internet will leverage new technologies and protocols to promote the convergence of traditional and small/portable devices on a much larger scale than present. At the service level, systems will no longer be able to address a closed universe of stakeholders. Additionally, market forces, technological innovation and new business models will push system fragmentation even further.

On the other hand, those complex, fragmented systems of the Future Internet (or at least parts of them) are expected to be operational for a very long time. Design and implementation decisions must be made in a broad context, considering long-term goals under the constraint of currently available resources and technologies.

To cope with these challenges, long-living Future Internet Systems need to be exceptionally flexible. They have to constantly evolve to adjust to the changing requirements. Evidently, evolution represents a constant threat to the system's quality. Since large-scale software-based systems increasingly pervade our daily life and put an ever rising number of digital assets at risk, security is a topic of special interest.

Task Force 2 will foster the discussion and the collection of novel ideas around the problem of maintaining the quality of security-critical software-based systems that evolve over time. In this short paper we sketch the vision driving TF2's activities (Sect. 2), define a set of objectives (Sect. 3) and operationalize them into a a workplan (Sect. 4). We close with a description of TF2's organization (Sect. 5).

T. Margaria and B. Steffen (Eds.): ISoLA 2010, Part II, LNCS 6416, pp. 26–29, 2010.

2 Vision

The vision of TF 2 is to contribute to the elaboration of an extended or revised software engineering process addressing the challenge of how to ensure the long-term compliance of long-living evolving software systems to quality requirements – especially security, privacy and dependability. The requirements themselves may also evolve over time.

TF2 is conceived as a forum to discuss and elaborate visions, scenarios, use cases, concepts and, where appropriate, to contribute to the development of methods, tools, and processes that support design and analysis techniques for the evolution, testing, verification, and configuration of evolving security-critical software.

3 Objectives

The emphasis of TF2 is on how to build and manage long-lived security-critical systems. This leads to a broad array of challenges. In the following two of them are mentioned exemplarily. The final set of objectives will be worked out during meetings and discussions of the task force members.

The first one refers to the engineering process: how should stakeholders (e.g., end users, business analysts, requirements analysts, system architects etc.) cope with the various aspects of change that may come with the evolution of long-lived systems. Current process models have considerable shortcommings: security is only integrated superficially, runtime adaptability is not addressed at all, model artefacts and runtime are hardly kept consistent. Artifact-centric software development processes represent a promising alternative.

The second challenge refers to the architecture and implementation of Future Internet Systems: how could such systems be designed and realized so that they are flexible enough to evolve over time accommodating the various changes. Promising approaches can be found in the area of pattern-driven and model-based engineering with a broad set of formal, semi-formal and informal techniques for the transformation of models, the deployment and configuration of components, functional and non-functional testing, and the verification of properties.

4 Work Plan

Obviously, many concerns affect the quality of software systems. Maintainability and security are key concerns in heterogeneous, diverse, and open environments. However, the interest for other qualities may emerge in a bottom-up way, according to the interests of the task force members. Thus, as a first activity the task force will identify areas of common interest and instigate discussions on selected key topics at the Isola 2010 Meeting.

Based thereupon, the second activity of the task force will be to compile a state-of-the-art survey on engineering approaches and quality assurance techniques for evolving, security-critical systems. The goal could be a publication in a suitable journal.

As a third activity, the task force will promote awareness on topics of interest through a set of complementary activities, such as e.g.,:

1. the organization of annual workshops. The aim is to foster collaboration between task force members, faciliate discussion about research challenges and outline potential directions of future research. Results are published in a document.
2. the active investigation of opportunities that may lead to project proposals (e.g., for the European ICT flagship initiative).
3. the publication of results of the Task Force after each annual EternalS workshop (e.g., research papers, position papers, road maps, white papers, survey papers, inputs to standardizing committees, etc.).
4. to promote awareness of the challenges linked to the topics of interest and engage in discussions with an international scientific community (e.g., through the organization of a workshop co-located with a major software engineering conference).

Activities operationalizing TF2's set of objectives will be discussed, planned and operationalized based ongoing discussions and during future meetings of the task force members.

Table 1. Task Force 2 Members

Name	**Affiliation**	**Project**	**Research Interest**
Michael Hafner	Univ. of Innsbruck	SECURECHANGE	security usability, software engineering, SOA
Richard Bubel	Chalmers University	HATS	software engineering, formal methods
Jim Clarke	Tssg.org	*External*	software engineering, security, trust, dependability
Qi Ju	Univ. of Trento	LIVINGKNOWLEDGE	natural language processing
Fabio Martinelli	Univ. of Pisa	CONNECT	security, distributed systems
Riccardo Scandariato	Cath. Univ. Leuven	SECURECHANGE	software architectures, security
Prof. Martin Steffen	Univ. of Oslo	HATS	software engineering, formal methods
Massimo Tivoli	University of L'Aquila	CONNECT	software engineering, formal methods
Prof. Ruth Breu	Univ. of Innsbruck	SECURECHANGE	software engineering, modeling
Margareth Stoll	Univ. of Innsbruck	*External*	security management
Prof. Patrick Hung	Univ. of Ontario	*External*	software engineering, SOA, security
Prof. Jon Whittle	Lancaster University	*External*	software engineering, modeling
N.N.	--	*External*	security, formal methods

5 Organization

The members of each task force are recruited from the four Integrated Projects of the "Forever Yours" cluster, as well as from the international research community with an interest in the themes of EternalS. Table 1 gives an overview of Task Force 2 members.

The process of recruiting more members is ongoing. One more candidate with an academic background may be interested in joining Task Force 2.

6 Conclusion

Task Force 2 is primarily conceived as a forum to discuss and elaborate visions, scenarios, use cases, and concepts around the problem of maintaining the quality of software-based systems that evolve over time. Security is a topic of special interest.

The final set of objectives as well as matching activities will depend on areas of common interest identified and key topics selected at the Isola 2010 Meeting.

Self-adaptation and Evolution by Learning
Description of EternalS Task Force 3

Richard Johansson

DISI, University of Trento, Italy
johansson@disi.unitn.it
http://disi.unitn.it/~johansson

Abstract. The EternalS Task Force 3 focuses on *self-adaptation and evolution* of software systems and knowledge resources; this includes evolution over time and adaptation to new domains. The primary method for achieving this will be the application of modern *machine learning* methods.

1 Overview

Task Force 3 will be a collector of ideas and research initiatives about *self-adaptation and evolution* of systems from a knowledge, software, and application viewpoint. TF3 will be generally related to dynamic system research and will deal with semantics of evolution and adaptation, knowledge management, software adaptation and security policies with respect to different dimensions, e.g., time and domain diversity.

To make the TF3 areas more concrete, we also characterize it by collecting expertise in *automatic learning*, which is a promising research direction. Accordingly, TF3 will include experts from the machine learning (ML) community and will be able to support, develop and advice initiatives, which aim at exploiting tools, techniques and know-how of that community. Regarding the latter, TF3 has already identified the most general purpose and flexible ML techniques that will facilitate the emerging of bottom-up collaborations in the area of self-adaptation and evolution, namely statistical learning theory and kernel methods. These allow for accurate learning and flexible and easy/automatic feature design which are essential characteristics when the application domain is novel and unknown.

2 Objectives and Vision

The cross-fertilization of the participating projects in Eternals brings the opportunity to open up unexplored research areas. For instance, TF3 proposes to apply automatic learning techniques to model virtues and techniques of control systems such as stability analysis and self-stabilization to maintain system integrity. Another research opportunity will be to investigate the feasibility and consequences in terms of research impact when combining logic-based formalisms as, e.g., often used in security, with statistical ones such as learning theory.

T. Margaria and B. Steffen (Eds.): ISoLA 2010, Part II, LNCS 6416, pp. 30–31, 2010.

A concrete example is the detection of obsolete software components (e.g., in SecureChange) by means of automatic classifiers (as for example defined in LivingKnowledge). It will also be interesting to compare the ML approaches with those typical of continuous systems from control theory.

Finally, a rather interesting topic that will be managed by TF3 is the definition, automatic generation and use of *semantics* in software systems and applications. In particular, several research communities with whom LivingKnowledge is connected have developed expertise and techniques for the representation and management of natural language semantics. This expertise will be compared with expertise and problems in different CA areas, e.g. the semantics of communication process in Connect or the semantics of security, consistency and correctness in SecureChange, Connect and HATS.

3 Organization and Work Plan

As the first dissemination activity of TF3, the TextGraphs-5 workshop on graph-based methods for natural language processing has been successfully organized. This event was held as a satellite event of the widely attended 2010 conference of the Association for Computational Linguistics, which took place on July 16, 2010, in Uppsala, Sweden. The workshop included a special session on opinion mining, and the LivingKnowledge project was represented in this special sessions by two submissions.

As a next step, the task force will compile a survey of the state of the art in machine learning and its application in fields related to the areas of interest in the projects that constitute EternalS. A special focus of this survey will be to demonstrate how modern machine learning techniques handle *structured objects* in input and output spaces, and how these techniques are relevant to the projects participating in EternalS. Other topics of interest include learning methods that make use of *logic-based representations*, which may be used as a bridge to connect the various viewpoints represented in EternalS, as well as methods for *domain adaptation* of statistical models, which may be helpful in the evolutionary aspect.

4 Conclusion

The activities of Task Force 3 of EternalS focus on *evolution*: software components, knowledge bases, or statistical models may need to evolve over time as the world and the vocabulary to describe it are constantly evolving, or they may need to be adapted to new domains and application scenarios for which they were not originally intended. To handle the complexities of evolutionary systems, we will make use of *machine learning* methods that are adapted to this scenario. We envision a number of synergies between the EternalS projects in this area. The dissemination activities in this task force are already under way: an EternalS-sponsored workshop was held that included a special LivingKnowledge-related session on opinion mining.

Overview of Roadmapping by EternalS

Jim Clarke and Keith Howker

TSSG, Waterford Institute of Technology, Ireland

Abstract. The roadmapping activity in EternalS project focuses on developing and contributing a vision and direction for future Framework Programme research, arising from the work of a cluster of current projects[1] towards *Trustworthy Eternal Systems*.

Keywords: research roadmap, trust, dependability.

1 Introduction

EternalS has set up Task forces that will develop and focus on three characteristics of the dynamics of the *Eternal system*: TF1 - Diversity awareness and management; TF2 - Time awareness and management; TF3 -Self-adaptation and evolution by learning. The expertise – and the actual work – of the task forces will derive principally from the cluster of projects [1][2][3][4] of which EternalS partners are strongly associated. A specific activity of the project is to take the findings of the taskforces to build a roadmap that will recommend and influence future action and research in this particular field of the long-lived system, but will also contribute to the more general requirements for Framework Programme (FP) research that will support the Future Internet and Information Society.

2 Goals for the Roadmap

The goal of the roadmap is two-fold: it will develop a vision of the future research requirements in this field with respect to aspects of trust, security, and dependability, that will provide guidance to the planning and Strategic Research Agenda (SRA) for the future EFFECTS+; it will also inform the more local direction and agenda of the four ICT-Forever Yours projects and their possible follow-on.

The roadmap will seek to identify the principal challenges, and to establish a timeline for future work and results, with recommendations for technical action, and requirements for any supporting or consequent policy measures.

3 Approach

Earlier work by EternalS participants in Support and Coordination actions has already contributed to the SRA, and forms a foundation for the roadmapping here. The work

[1] The "ICT-Forever Yours" FET projects: *LivingKnowledge* [1], *HATS* [2], *Connect* [3], and *SecureChange* [4].

T. Margaria and B. Steffen (Eds.): ISoLA 2010, Part II, LNCS 6416, pp. 32–34, 2010.

of SecurIST [5] road mapping activities has informed the thinking towards the later calls of FP7 regarding Trust and Security programmes. The report of the Think-Trust RISEPTIS group [6] focused on the need to convey the fundamental needs for trust and security to the, often non-technical, policy maker, and set out high-level recommendations for action by the European Commission. The Think-Trust (T-T) recommendations [7] will provide further, mainly technical, elaboration of RISEPTIS. These are based on the findings of the T-T Working Groups, so there are strong parallels with the envisaged approach in EternalS, with the information that will be generated by its Task Forces.

3.1 Initial Roadmap Mindmap

The current ideas for structuring the *roadworks* are to take three vectors or perspectives: initial conditions – the 'givens'; considerations – the context; and the outputs – findings and recommendations. These are shown in Figure 1 below.

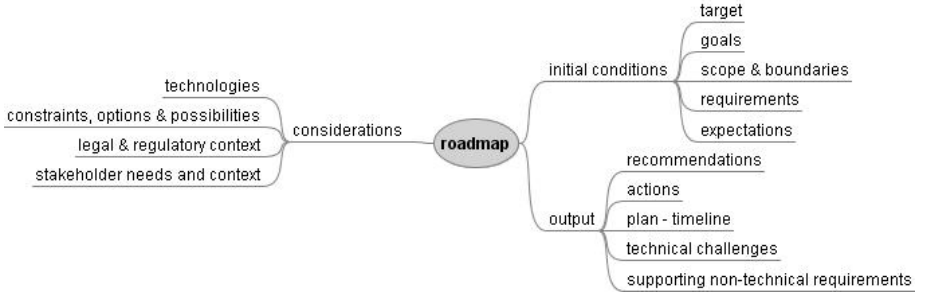


Fig. 1. Initial proposals for structuring the roadmapping work

The *initial conditions* will seek to take in a wide range of input views, from informal stakeholder expectations to formal objectives of the FP and the FIA. The *considerations* will take into account various aspects of the current and envisaged context. *Output* will be wide-ranging, but will provide specific appropriate results for targeted recipients, including the SRA and the *local* project cluster.

3.2 Relationship to Other Roadmapping Activity

The principal sources and inputs to the EternalS roadmap will be derived from the EternalS Task Forces that will internally gain their contributions and sights from the four clustered projects. However, it is anticipated that there will also be close liaison between the roadmapping work and parallel activities that will shortly start on the compilation of a roadmap for future FP research to support the Future Internet activities. This will be supported by a group of coordination and support actions that make up the Future Internet Support Activity (FISA). Current thinking is that this will also rely heavily on clustering mechanisms amongst the RTD projects to gather together their intelligence and insights and feed into road mapping activities.

4 Outlook

The Future Internet roadmap activity is planned to begin in September 2010. EternalS will have representation into the early considerations of this, and will be generating input to a first iteration of its own roadmap during the first half of 2011 according to the time schedule of the project. EternalS will both benefit from the structural and methodological development of the FI roadmap, and will be able to contribute to its further development.

References

1. LivingKnowledge, http://livingknowledge.europarchive.org/index.php
2. HATS – Highly adaptable and trustworthy software using formal models, http://www.cse.chalmers.se/research/hats/
3. Connect – Emergent Connectors for Eternal Software Intensive Networked Systems, http://connect-forever.eu/
4. SecureChange – Security engineering for lifelong evolvable systems, http://www.securechange.eu
5. SecurIST project, Deliverable D3.3 – ICT Security & Dependability Research beyond 2010: Final strategy (2010), http://www.securitytaskforce.eu/dmdocuments/d3_3_final_strategy_report_v1_0.pdf
6. Trust in the Information Society, A Report of the Advisory Board *RISEPTIS*, http://www.think-trust.eu/riseptis.html
7. Towards a Trustworthy Information Society: The Research Challenges, Think-Trust Deliverable D3.1C, http://www.think-trust.eu/RecommendationsReport.html

Adaptive Composition of Conversational Services through Graph Planning Encoding

Pascal Poizat[1,2] and Yuhong Yan[3]

[1] University of Evry Val d'Essonne, Evry, France
[2] LRI UMR 8623 CNRS, Orsay, France
pascal.poizat@lri.fr
[3] Concordia University, Montreal, Canada
yuhong@encs.concordia.ca

Abstract. Service-Oriented Computing supports description, publication, discovery and composition of services to fulfil end-user needs. Yet, service composition processes commonly assume that service descriptions and user needs share the same abstraction level, and that services have been pre-designed to integrate. To release these strong assumptions and to augment the possibilities of composition, we add adaptation features into the service composition process using semantic structures for exchanged data, for service functionalities, and for user needs. Graph planning encodings enable us to retrieve service compositions efficiently. Our composition technique supports conversations for both services and user needs, and it is fully automated thanks to a tool, pycompose, which can interact with state-of-the-art graph planning tools.

1 Introduction

Task-Oriented Computing envisions a user-friendly pervasive world where *user tasks* corresponding to a (potentially mobile) user would be achieved by the automatic assembly of resources available in her/his environment. Service-Oriented Computing [1] (SOC) is a cornerstone towards the realization of this vision, through the abstraction of heterogeneous resources as services and automated composition techniques [2,3,4]. However, services being elements of composition developed by different third-parties, their reuse and assembly naturally raises composition mismatch issues [5,6]. Moreover, Task-Oriented Computing yields a higher description level for the composition requirements, *i.e.*, the user task(s), as the user only has an abstract vision of her/his needs which are usually not described at the service level. These two dimensions of interoperability, namely *horizontal* (communication protocol and data flow between services) and *vertical matching* (correspondence between an abstract user task and concrete service capabilities) should be supported in the composition process.

Software adaptation is a promising technique to augment component reusability and composition possibilities, thanks to the automatic generation of software pieces, called adaptors, solving mismatch out in a non-intrusive way [7]. More recently, adaptation has been applied in SOC to solve mismatch between

T. Margaria and B. Steffen (Eds.): ISoLA 2010, Part II, LNCS 6416, pp. 35–50, 2010.

services and clients (*e.g.*, orchestrators) [8,9,10]. In this article we propose to add adaptation features in the service composition process itself. More precisely, we propose an automatic composition technique based on *planning*, a technique which is increasingly applied in SOC [11,12] as it supports automatic service composition from underspecified requirements, *e.g.*, the data one requires and the data one agrees to give for this, or a set of capabilities one is searching for. Such requirements do not refer to service operations or to the order in which they should be called, which would be ill-suited to end-user composition.

Outline. Preliminaries on planning are given in Section 2. After introducing our formal models in Section 3, Section 4 presents our encoding of service composition into a planning problem, and Section 5 addresses tool support. Related work is discussed in Section 6 and we end with conclusions and perspectives.

2 Preliminaries

In this section we give a short introduction to AI planning [13].

Definition 1. *Given a finite set* $L = \{p_1, \ldots, p_n\}$ *of proposition symbols, a* planning problem *[13] is a triple* $P = ((S, A, \gamma), s_0, g)$*, where:*

- $S \subseteq 2^L$ *is a set of states.*
- A *is a set of actions, an* action *a being a triple* $(pre, effect^-, effect^+)$ *where* $pre(a)$ *denotes the preconditions of* a*, and* $effect^-(a)$ *and* $effect^+(a)$*, with* $effect^-(a) \cap effect^+(a) = \emptyset$*, denote respectively the negative and the positive effects of* a*.*
- γ *is a state transition function such that, for any state* s *where* $pre(a) \subseteq s$*,* $\gamma(s, a) = (s - effect^-(a)) \cup effect^+(a)$*.*
- $s_0 \in S$ *and* $g \subseteq L$ *are respectively the initial state and the* goal*.*

Two actions a and b are *independent* iff they satisfy $effect^-(a) \cap [pre(b) \cup effect^+(b)] = \emptyset$ and $effect^-(b) \cap [pre(a) \cup effect^+(a)] = \emptyset$. An action set is independent when its actions are pairwise independent. A *plan* is a sequence of actions $\pi = a_1; \ldots; a_k$ such that $\exists s_1, \ldots, s_k \in S$, $s_1 = s_0$, $\forall i \in [1, k]$, $pre(a_i) \in s_{i-1} \wedge \gamma(s_{i-1}, a_i) = s_i$. The definition in [13] takes into account predicates and constant symbols which are then used to define states (ground atoms made with predicates and constants). We directly use propositions here.

Graph Planning [14] is a technique that yields a compact representation of relations between actions and represent the whole problem world. A planning graph G is a directed acyclic leveled graph. The levels alternate proposition levels P_i and action levels A_i. The initial proposition level P_0 contains the initial propositions (s_0). The planning graph is constructed from P_0 using a polynomial algorithm. An action a is put in layer A_i iff $pre(a) \subseteq P_{i-1}$ and then $effect^+(a) \subseteq P_i$. Specific actions (no-ops) are used to keep data from one layer to the next one, and arcs to relate actions with used data and produced effects. Graph planning also introduces the concept of mutual exclusion (mutex) between non independent actions. Mutual exclusion is reported from a layer to the next one

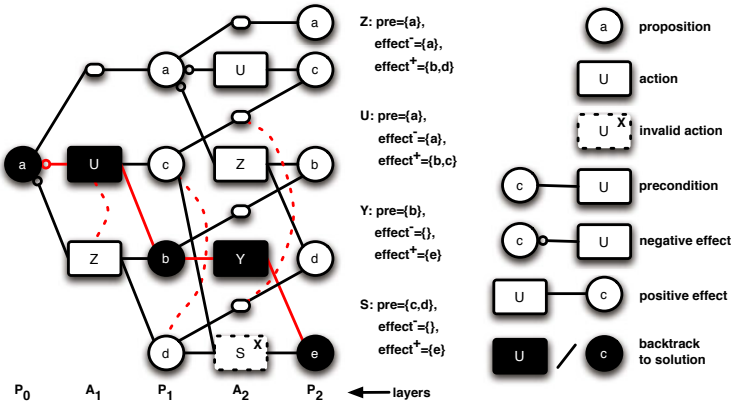


Fig. 1. Graphplan example

while building the graph. The planning graph actually explores multiple search paths at the same time when expanding the graph, which stops at a layer A_k iff the goal is reached ($g \subseteq A_k$) or in case of a fixpoint ($A_k = A_{k-1}$). In the former case there exists at least a solution, while in the later there is not. Solution(s) can be obtained using backward search from the goal. Planning graphs whose computation has stopped at level k enable to retrieve all solutions up to this level. Additionally, planning graphs enable to retrieve solutions in a concise form, taking benefit of actions that can be done in parallel (denoted ||).

An example is given in Figure 1 where we suppose the initial state is {a} and the objective is {e}. Applying U in the first action layer, for example, is possible because a is present; and this produces b and c. The extraction of plans from the graph is performed using a backward chaining technique over action layers, from the final state (objective) back to the initial one. In the example, plans U;Y, Z;Y, (U||Z);Y and (U||Z);S can be obtained (see bold arcs in Fig. 1 for U;Y). However, U and Z are in mutual exclusion. Accordingly, since there is no other way to obtain c and d than with exclusive actions, these two facts are in exclusion in the next (fact) layer, making S impossible. Note that other nodes are indeed in mutual exclusion (such as the no-op and U in A_1, or two no-ops in A_2 but we have not represented this for clarity).

3 Modeling

In this section, we present our formal models, grounding service composition. Both services and composition requirements support conversations. Therefore, we begin with their definition. We then present the structures supporting the definition of semantic data and capabilities. Finally, we present models for services and service composition requirement.

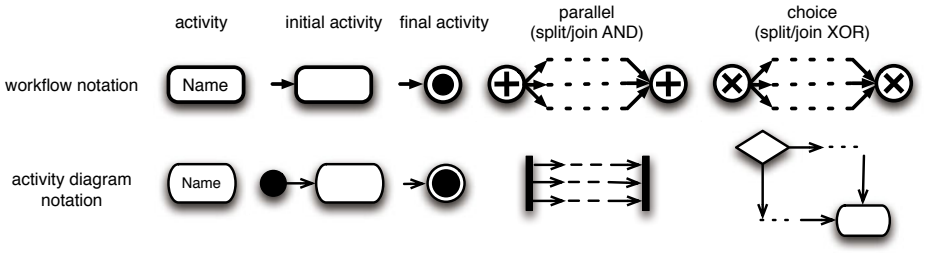


Fig. 2. Workflow notation and relation to the UML activity diagrams

3.1 Conversation Modelling

Different models have been proposed to support service discovery, verification, testing, composition or adaptation in presence of service conversations [15,16,9]. They mainly differ in their formal grounding (Petri nets, transition systems, or process algebra), and the subset of service languages being supported. Since we target centralized composition (orchestration) with possible parallel service invocation, we choose the workflow model from [17]. An important benefit of workflow models is that they can be related via model transformation to graphical notations that are well-known by the software engineers, *e.g.*, UML activity diagrams (Fig. 2) or BPMN. Additionally, workflows are more easily mastered by a non-specialist through pre-defined patterns (sequence, alternative choice, parallel tasks). Transition systems models could yield a simpler encoding as a planning problem but raise issues when it comes to implement the composition models, requiring model filtering to remove parts in the composition models which are not implementable in the target language [9].

Definition 2. *Given a set of activity names N, a Workflow (WF) [17] is a tuple $WF^N = (P, \rightarrow, Name)$. P is a set of process elements (or workflow nodes) which can be further divided into disjoint sets $P = P_A \cup P_{so} \cup P_{sa} \cup P_{jo} \cup P_{ja}$, where P_A are activities, P_{so} are XOR-Splits, P_{sa} are AND-splits, P_{jo} are OR-Joins, and P_{ja} are AND-Joins. $\rightarrow \subseteq P \times P$ denotes the control flow between nodes. $Name : P_A \rightarrow N$ is a function assigning activity names to activity nodes.*

We note $\bullet x = \{y \in P | y \rightarrow x\}$ and $x\bullet = \{y \in P | x \rightarrow y\}$. We require that WF are well-structured [17] and without loop. A significant feature of well-structured workflows is that the XOR-splits and the OR-Joins, and the AND-splits and the AND-splits appear in pairs (Fig. 2). Moreover, we require $|\bullet x| \leq 1$ for each x in $P_A \cup P_{sa} \cup P_{so}$ and $|x \bullet| \leq 1$ for each x in $P_A \cup P_{ja} \cup P_{jo}$.

3.2 Semantic Structures

In our work we use semantic information to enrich the service composition process and its automation. We have two kinds of semantic information. Capabilities represent the functionalities that are either requested by the end-users or provided by services. They are modelled using a Capability Semantic Structure

Table 1. eTablet buying – DSS relations: $d_1 \sqsubset d_2$ (left), $d_1 \lhd_x d_2$ (right)

d_1	d_2
etablet	pear_product
etelephone	pear_product
pear_product	product
product_price	order_amount
user_address	shipping_addr
user_address	billing_addr
user_address	address

d_1	x	d_2
pear_product_info	price	product_price
pear_product_info	details	product_technical_information
user_info	name	user_name
user_info	address	user_address
user_info	cc	credit_card_info
user_info	pim	pim_wallet
pim_wallet	paypal	paypal_info
pim_wallet	amazon	amazon_info
paypal_info	login	paypal_login
paypal_info	pwd	paypal_pwd
amazon_info	login	amazon_login
amazon_info	pwd	amazon_pwd
credit_card_info	number	credit_card_number
credit_card_info	name	credit_card_holder_name

(CSS). Further, service inputs and outputs are annotated using a Data Semantic Structure (DSS).

We define a *Data Semantic Structure (DSS)* as a tuple $(\mathcal{D}, \lhd, \sqsubset)$ where $\mathcal{D}$ is a set of concepts (or semantic data type[1]) that represent the semantics of some data, $\lhd$ is a composition relation ($(d_1, x, d_2) \in \lhd$, also noted $d_1 \lhd_x d_2$ or simply $d_1 \lhd d_2$ when x in not relevant for the context, means a d_1 is composed of an x of type d_2), and $\sqsubset$ is a subtyping relation ($d_1 \sqsubset d_2$ means d_1 can be used as a d_2). We require there is no circular composition. DSSs are the support for the automatic decomposition (of d into D if $D = \{d_i \mid d \lhd d_i\}$), composition (of D into d if $D = \{d_i \mid d \lhd d_i\}$) and casting (of d_1 into d_2 if $d_1 \sqsubset d_2$) of data types exchanged between services and orchestrator. We also define a *Capability Semantic Structure (CSS)* as a set $\mathcal{K}$ of concepts that correspond to capabilities.

Application. We will illustrate our composition technique on a simple, yet realistic, case study: the online buying of an eTablet. A DSS describes concepts and relations for this case study. For place matters, we only give the relations here (Tab. 1) since concepts can be inferred from these and from the service operation signatures, below.

3.3 Services

A service is a set of operations described in terms of capabilities, inputs, and outputs. Additionally, services have a conversation. We define services as follows.

Definition 3. *Given a CSS $\mathcal{K}$ and a DSS $\mathcal{D} = (\mathcal{D}, \lhd, \sqsubset)$, a* service *is a tuple $w = (O, WF^O)$, where O is a set of operations, an operation being a tuple (in, out, k) with $in \subseteq \mathcal{D}$, $out \subseteq \mathcal{D}$, $k \in \mathcal{K}$, and WF^O is a workflow built over O.*

For a simple service (without a conversation) w, a trivial conversation can be obtained with a workflow where $P_A = O(w)$ (one activity for each operation),

[1] In this paper, the concepts of semantics and type of data are unified.

Table 2. eTablet buying – services' operations

service	operation	profile
w_1	order	pear_product → pear_product_info, as_sessionid :: product_selection
w_1	cancel	as_sessionid → ∅ :: nil
w_1	ship	shipping_addr, as_sessionid → ∅ :: shipping_setup
w_1	bill	billing_addr, as_sessionid → ∅ :: billing_setup
w_1	charge	credit_card_info, as_sessionid → ∅ :: payment
w_1	gift_wrapper	giftcode, as_sessionid → ∅ :: payment
w_1	ack	as_sessionid → tracking_num :: order_finalization
w_2	order	product → e_sessionid :: product_selection
w_2	ship	shipping_addr, e_sessionid → order_amount :: shipping_setup
w_2	charge_pp	paypal_trans_id, e_sessionid → ∅ :: nil
w_2	charge_cc	credit_card_info, e_sessionid → ∅ :: payment
w_2	bill	billing_addr, e_sessionid → ∅ :: billing_setup
w_2	finalize	e_sessionid → tracking_num :: order_finalization
w_3	login	paypal_login, paypal_pwd → p_sessionid :: nil
w_3	get_credit	order_amount, p_sessionid → paypal_trans_id :: payment
w_3	ask_bill	address, p_sessionid → ∅ :: billing_setup
w_3	logout	p_sessionid → ∅ :: nil

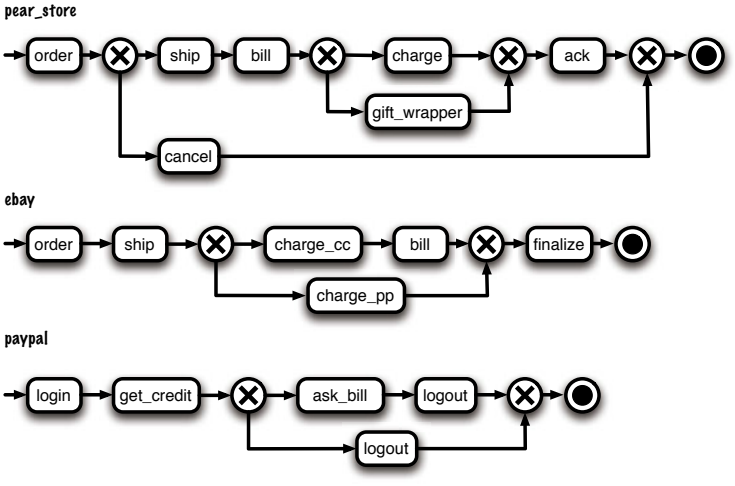


Fig. 3. eTablet buying – services' workflows

$P_{so} = \{\otimes\}$, $P_{jo} = \{\overline{\otimes}\}$, $P_{sa} = P_{ja} = \emptyset$, and $\forall o \in P_A$, $\{(\otimes, o), (o, \overline{\otimes})\} \subseteq \rightarrow$. This corresponds to a generalized choice between all possible operations. An operation may not have a capability (we then let $k = \mathsf{nil}$). $o = (in, out, k)$ is also noted $o : in \rightarrow out :: k$.

Application. To fulfil the user need, we have three services: pear_store (w_1, online store for pear products), ebay (w_2, general online shop) and paypal (w_3, online payment facilities). Their operations are given in Table 2 and their workflows are given in Figure 3.

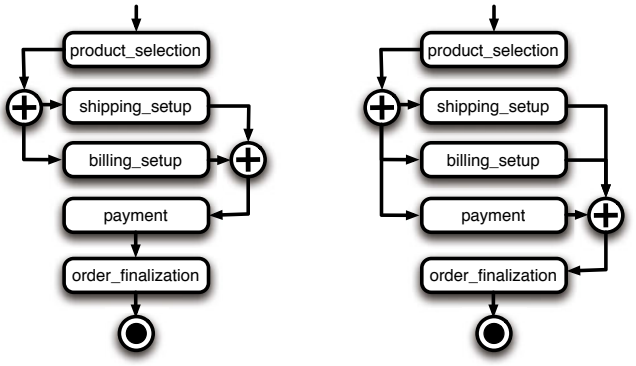


Fig. 4. eTablet buying – requirement workflows

3.4 Composition Requirements

A service composition requirement is given in terms of the inputs the user is ready to provide and the outputs this user is expecting. Additionally, the capabilities that are expected from the composition are specified, and their expected ordering given under the form of a workflow.

Definition 4. *Given a CSS* $\mathcal{K}$ *and a DSS* $\mathcal{D} = (\mathcal{D}, \lhd, \sqsubset)$*, a* composition requirement *is a tuple* $(D_{\text{in}}, D_{\text{out}}, WF^{\mathcal{K}})$ *where* $D_{\text{in}} \subseteq \mathcal{D}$*,* $D_{\text{out}} \subseteq \mathcal{D}$*, and* $WF^{\mathcal{K}}$ *is a workflow build over* $\mathcal{K}$*.*

Application. The user requirement in our case study is $(\{etablet, user\_info\}, \{tracking\_num\}, wfc)$. As far as the wfc requirement workflow is concerned, we have two alternatives for it. The first one (Fig. 4, left) requires that payment is done after shipping and billing have been set up (which can be done in parallel). The second one (Fig. 4, right) is less strict and enables the payment to be done in parallel to shipping and billing setup.

4 Encoding Composition as a Planning Problem

In this section we present how service composition can be encoded as a graph planning problem. We will first explain how DSS can be encoded (to solve out horizontal adaptation). Then we will present how a generic workflow can be encoded. Based on this, we will then explain how services and composition requirements are encoded (the workflow of the later solving out vertical adaptation).

4.1 DSS Encoding

For each $d \lhd \{x_i : d_i\}$ in the DSS we have an action $comp_d(\bigcup_i \{d_i\}, \emptyset, \{d\})$ and an action $dec_d(\{d\}, \emptyset, \bigcup_i \{d_i\})$ to model possible (de)composition. Moreover, for each $d \sqsubset d'$ in the DSS we have an action $cast_{d,d'}(\{d\}, \emptyset, \{d'\})$ to model possible casting from d to d'.

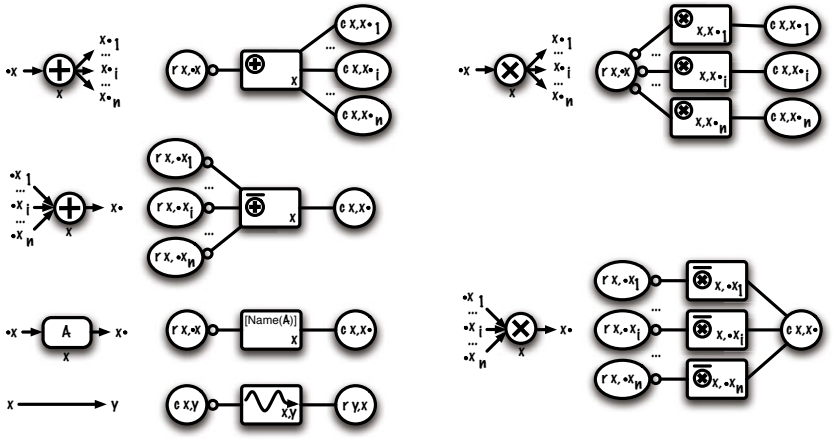


Fig. 5. Workflow encoding

4.2 Workflow Encoding

We reuse here a transformation from workflows to Petri net defined in [17]. Instead of mapping a workflow $(P, \rightarrow, Name)$ to a Petri net, we map it to a planning problem. Let us first define the set of propositions that are used. The behavioural constraints underlying the workflow semantics (*e.g.*, an action being before/after another one) are supported through two kinds to propositions: $r_{x,y}$ and $c_{x,y}$. We also have a proposition I for initial states, and a proposition F for correct termination states. F will be used both for final states and for initial states (in this case to denote that a service can be unused). We may then define the actions that are used (Fig. 5):

- for each $x \in P_{sa}$, we have an action $a = \oplus x$, for each $x \in P_{ja}$, we have an action $a = \bar{\oplus} x$, and for each $x \in P_A$, we have an action $a = [Name(x)]x$. In all three cases, we set $pre(a) = effect^-(a) = \bigcup_{y \in \bullet x} \{r_{x,y}\}$, and $effect^+(a) = \bigcup_{y \in x\bullet} \{c_{x,y}\}$.
- for each $x \in P_{so}$, for each $y \in x\bullet$, we have an action $a = \otimes x, y$ and we set $pre(a) = effect^-(a) = \bigcup_{z \in \bullet x} \{r_{x,z}\}$, and $effect^+(a) = \{c_{x,y}\}$.
- for each $x \in P_{jo}$, for each $y \in \bullet x$, we have an action $a = \bar{\otimes} x, y$, and we set $pre(a) = effect^-(a) = r_{x,y}$, and $effect^+(a) = \bigcup_{z \in \bullet x} \{c_{x,z}\}$.
- for each $x \rightarrow y$, we have an action $a = \rightsquigarrow x, y$ and we set $pre(a) = effect^-(a) = \{c_{x,y}\}$, and $effect^+(a) = \{r_{y,x}\}$.
- additionally, for any initial action a we add $\{I, F\}$ in $pre(a)$ and $effect^-(a)$.
- additionally, for any final action a we add $\{F\}$ in $effect^+(a)$.

4.3 Composition Requirements Encoding

A composition requirement $(D_{\text{in}}, D_{\text{out}}, WF^{\mathcal{K}})$ is encoded as follows. First we compute the set of actions resulting from the encoding of $WF^{\mathcal{K}}$ (see 4.2). Then

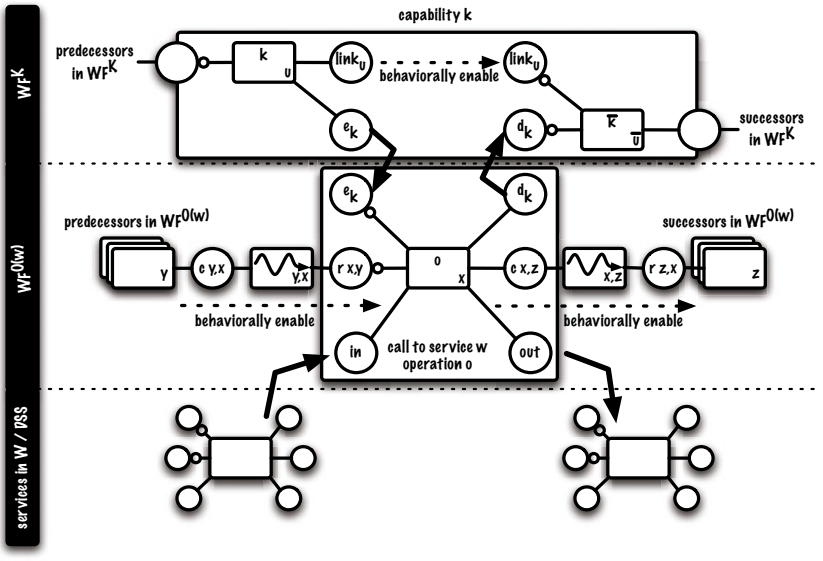


Fig. 6. Principle of interaction between service and requirement encodings

we have to encode the fact that capabilities in the composition requirement encoding should interoperate with operations in service encodings. The idea is the following. Taking a service w, when a capability k is enabled at the current state of execution by $WF^{\mathcal{K}}$ then we should invoke an operation of capability k that is enabled at the current state by $WF^{O(w)}$ before any one of the capability possibly following k could be enabled. Moreover, an operation o with capability k of w can be invoked only iff this is enabled by the current state of execution in $WF^{O(w)}$ and k is enabled in $WF^{\mathcal{K}}$. To achieve this, we replace any action $a = [k]x$ in the encoding of $WF^{\mathcal{K}}$ by two actions, $a' = [k]x$ and $\overline{a'} = [k]\overline{x}$, and we set:

- $pre(a') = pre(a)$, $effect^-(a') = effect^-(a)$, $effect^+(a) = \{e_k, link_x\}$.
- $pre(\overline{a'}) = effect^-(\overline{a'}) = \{link_x, d_k\}$, $effect^+(\overline{a'}) = effect^+(a)$.

e_k and d_k enforce the synchronizing rules between capability workflow (defining when a capability k can be done) and service workflows (defining when an operation with capability k can be done) as presented in Figure 6. $link_k$ ensure that two actions $a_1 = [k]x_1$ and $a_2 = [k]x_2$ with the same capability will not interact incorrectly when x_1 and x_2 are in parallel in a workflow.

4.4 Service Encoding

Each service $w = (O, WF^O)$ is encoded as follows. First we encode the workflow WF^O as presented in 4.2. Then, for each action $a = [o]x$ in this encoding we add:

- $in(o)$ in $pre(a)$ to model the inputs required by operation o and $out(o)$ in $effect^+(a)$ to model the outputs provided by operation o.
- $e_{k(o)}$ in $pre(a)$ and in $effect^-(a)$ and $d_{k(o)}$ in $effect^+(a)$ to implement the interaction with capabilities presented in 4.3 and in Figure 6.

4.5 Overall Encoding

Given a DSS $\mathcal{D}$, a set of services W, and a composition requirement (D_{in}, D_{out}, $WF^{\mathcal{K}}$), we obtain the planning problem $((S, A, \gamma), s_0, g)$ as follows:

- $s_0 = D_{\text{in}} \cup \{wfc : I, wfc : F\} \bigcup_{w \in W} \{w : I, w : F\}$.
- $g = D_{\text{out}} \cup \{wfc : F\} \bigcup_{w \in W} \{w : F\}$.
- $A = dss : ||\mathcal{D}|| \cup wfc : ||WF^{\mathcal{K}}||. \bigcup_{w \in W} w : ||WF^{O(w)}||$.
- S and γ are built with the rules in Definition 1.

where $||x||$ means the set of actions resulting from the encoding of x. Prefixing (denoted with $prefix :$) operates on actions and on workflow propositions (I, F, $r_{x,y}$, and $c_{x,y}$) coming from encodings. It is used to avoid name clashes between different subproblems. We suppose that, up to renaming, there is no service identified as dss or wfc.

4.6 Plan Implementation

Solving the planning problem, we may get a failure when there is no solution satisfying both that (i) a service composition exists to get D_{out} from D_{in}, (ii) using operations/capabilities in an ordering satisfying both used service conversations and capability conversation, (iii) leaving used services in their final state. In other cases, we obtain (see Sect. 2) a plan $\pi = L_1; \ldots; L_i; \ldots; L_n$ where ; is the sequence operator and each L_i is of the form $(P_{i,1}|| \ldots ||P_{i,j}|| \ldots ||P_{i,m_i})$ where $||$ is the parallel operator and each $P_{i,j}$ is a workflow process element. First of all, we begin by filtering out π by removing from it all $P_{i,j}$ that is not of the form $dss : \ldots$ or $w : [o]x$, *i.e.*, that is a purely structuring item, not corresponding to data transformation or service invocation. Given the filtered plan, we can generate a WS-BPEL implementation for it as done for transitions systems in [9]. Still, we may benefit here from the fact that actions that can be done in parallel are explicited in a graph planning plan (using operation $||$), while in transition systems we only have interleaving semantics (finding out which actions can be done in parallel is much more complex). Therefore, for the main structure of the <process> ... < /process> element we replace the [9] state machine encoding by a more efficient version using sequence and flows. For π we get:

$\langle$`sequence`$\rangle modeltrans(L_1) \ldots modeltrans(L_i) \ldots modeltrans(L-n) \langle$/`sequence`$\rangle$

and for each $L_i = (P_{i,1}|| \ldots ||P_{i,j}|| \ldots ||P_{i,m_i})$ we have:

$\langle$`flow`$\rangle modeltrans(P_{i,1}) \ldots modeltrans(P_{i,j}) \ldots modeltrans(P_{i,m_i})) \langle$/`flow`$\rangle$

where *modeltrans* is the transformation of basic assignment / communication activities defined in [9].

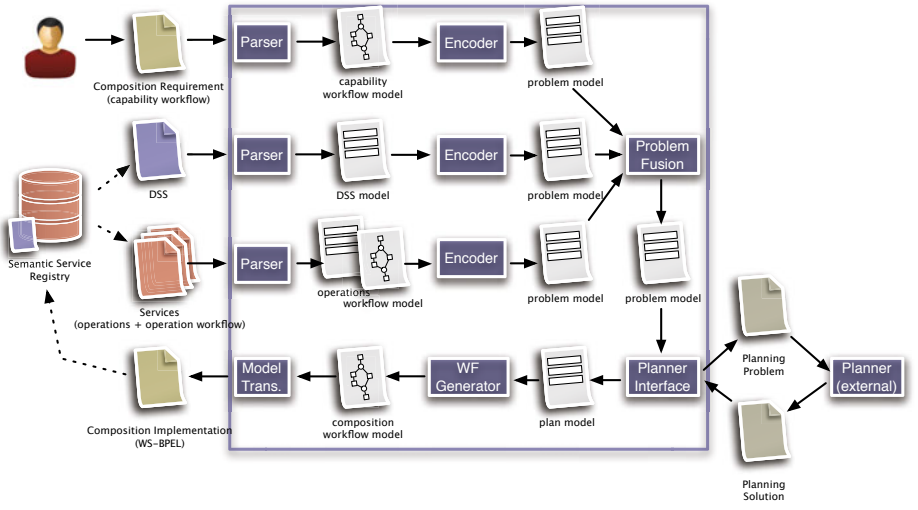


Fig. 7. Architecture of the pycompose tool

5 Tool Support

Our composition approach is supported with a tool, pycompose (Fig. 7), written in the Python language. This tool takes as input a DSS file, several service description files (list of operations and workflow), and the composition requirement (input list, output list, and a workflow file). It then generates the encoding of this composition problem. pycompose supports through a command-line option the use of several planners: the original C implementation of graph planning, graphplan[2], a Java implementation of it, PDDLGraphPlan[3], and Blackbox[4], a planner combining graphplan building and the use of SAT solvers to retrieve plans. The pycompose architecture enables to support other planners through the implementation of a class with two methods: problemToString and run, respectively to output a problem in planner format and to run and parse planner results.

Application. If we run pycompose on our composition problem with the first requirement workflow (Fig. 4, left), we get one solution (computed in 0.11s on a 2.53 GHz Mac Book Pro, including 0.03s for the planner to retrieve the plan):

(pear_product:=*cast*(etablet) || {user_name,user_address,credit_card_info,pim_wallet}
:=*dec*(user_info)) ;
(shipping_addr:=*cast*(user_address) || billing_addr:=*cast*(user_address) || w_1:order) ;
w_1:ship ; w_1:bill ; w_1:charge ; w_1:ack

The workflow representation of this solution is presented in Figure 8.

[2] http://www.cs.cmu.edu/~avrim/graphplan.html
[3] http://www.cs.bham.ac.uk/~zas/software/graphplanner.html
[4] http://www.cs.rochester.edu/~kautz/satplan/blackbox/

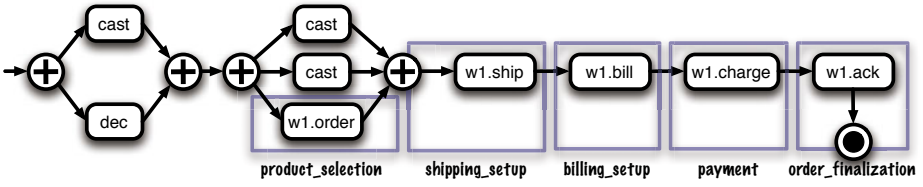


Fig. 8. eTablet buying – composition solution

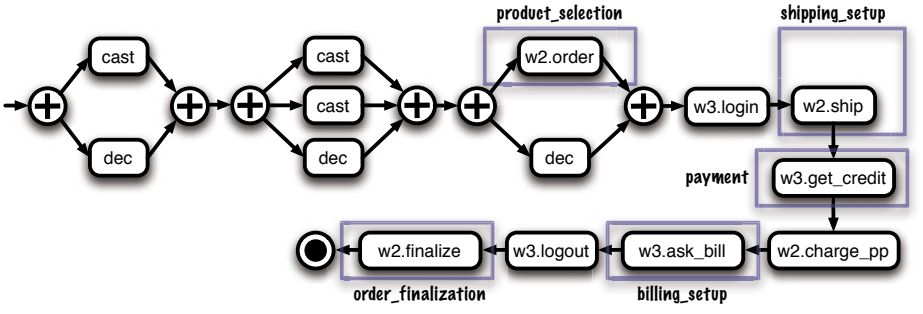


Fig. 9. eTablet buying – alternative composition solution

However, let us now suppose that the user does not want to give his credit card (user_info $\lhd_{cc}$ credit_card_info is removed from DSS, or the user input is replaced with {etablet,user_name,user_address,pim_wallet}). There is no longer any possible composition: w_1 cannot proceed with payment (no credit card information), moreover, w_2 and w_3 cannot interact since this would yield that capability *payment* is done before capability *billing_setup* (see w_3 workflow in Fig. 3 and its operations in Tab. 2) while the requirement workflow forbids it. However, if we let a more permissive requirement workflow (Fig. 4, right) then we get a composition (computed in 0.11s on a 2.53 GHz Mac Book Pro, including 0.04s for the planner to retrieve the plan) where w_2 and w_3 interact:

(pear_product := *cast*(etablet) || {user_name,user_address,credit_card_info,pim_wallet} := *dec*(user_info)) ;
(product := *cast*(pear_product) || shipping_addr := *cast*(user_address) || {paypal_info,amazon_info} := *dec*(pim_wallet)) ;
(w_2:order || {paypal_login,paypal_pwd} := *dec*(paypal_info)) ;
w_3:login ; w_2:ship ; w_3:get_credit ; w_2:charge_pp ; w_3:ask_bill ; w_3:logout ; w_2:finalize

The workflow representation of this second solution is given in Figure 9.

6 Related Work

Our work is at the intersection of two domains: service composition and software adaptation. Automatic composition is an important issue in Service-Oriented Computing and numerous works have addressed this over the last years [2,3,4].

Planning-based approaches have particularly been studied due to their support for underspecified requirements [11,12]. Automatic composition has also been achieved using matching and graph/automata-based algorithms [18,19,20] or logic reasoning [21]. Various criteria could be used to differentiate these approaches, yet, due to our Task-Oriented Computing motivation, we will focus on issues related to service and composition requirement models, and to adaptation.

While both data input/output and capability requirements should be supported, as in our approach, to ensure composition is correct wrt. the user needs, only [22,19] do, while [23,24,25,18,20,26] support data only and [21] supports capabilities only. As far as adaptation is concerned, [24,25,19,20] support a form of horizontal (data) adaptation, using semantics associated to data; and [23] a form of vertical (capability abstraction) adaptation, due to its hierarchical planning inheritance. We combined both techniques to achieve both adaptation kinds. Few approaches support expressive models in which protocols can be described over capabilities – either for the composition requirement [21] or for both composition and services [22,19] like us. [23,24,25,18,20] only support conversations over operations (for a given capability).

As opposed to the aforementioned works dealing with orchestration, in [27], the authors present a technique with adaptation features for automatic service choreography. It supports a simple form of horizontal adaptation, however their objective is to maximize data exchange between services but they are not able to compose services depending on an abstract user task.

Most software adaptation works, *e.g.*, [28,29,30] are pure model-based approaches whose objective is to solve protocol mismatch between a fixed set of components, and that do not tackle service discovery, composition requirements, or service composition implementation. Few works explicitly add adaptation features to Service-Oriented Computing [8,9,10]. They adopt a different and complementary view wrt. ours since their objective is not to integrate adaptation within composition in order to increase the orchestration possibilities, but to tackle protocol adaptation between clients and services, *e.g.*, to react to service replacement.

In an earlier work [31] we already used graph planning to perform service composition with both vertical and horizontal adaptation. With reference to this work, we add support for conversations in both service descriptions and composition requirements. Moreover, adaptation was supported in an ad-hoc fashion, yielding complexity issues when backtracking to get composition solutions. Using encodings, we are able in our work to support adaptation with regular graph planning which enables us to use state-of-the-art graph planning tools.

7 Conclusion

Software adaptation is a promising approach to augment service interoperability and composition possibilities. In this paper we have proposed a technique to integrate adaptation features in the service composition process. With reference to related work, we support both horizontal (data exchange between services

and orchestrator) and vertical adaptation (abstraction level mismatch between user need and service capabilities). This has been achieved combining semantic descriptions (for data and capabilities) and graph planning. We also support conversations in both service descriptions and composition requirements.

The approach at hand is dedicated to deployment time, where services are discovered and then composed out of a set of services that may change. Yet, in a pervasive environment, services may appear and disappear also during composition execution, *e.g.*, due to the user mobility, yielding broken service compositions. We made a first step towards repairing them in [32], still with a simpler service and composition requirement model (no conversations). A first perspective concerns extending this approach to our new model. Further, we plan to study the integration of our composition and repair algorithms as an optional module in existing runtime monitoring and adaptation frameworks for services composition such as [33].

Acknowledgement. This work is supported by project "Building Self-Manageable Web Service Process" (`RGPIN/298362-2007`) of Canada NSERC Discovery Grant, and by project "PERvasive Service cOmposition" (`ANR-07-JCJC-0155-01`, PERSO) of the French National Agency for Research.

References

1. Papazoglou, M.P., Georgakopoulos, D.: Special Issue on Service-Oriented Computing. Communications of the ACM 46(10) (2003)
2. Rao, J., Su, X.: A survey of automated web service composition methods. In: Cardoso, J., Sheth, A.P. (eds.) SWSWPC 2004. LNCS, vol. 3387, pp. 43–54. Springer, Heidelberg (2005)
3. Dustdar, S., Schreiner, W.: A survey on web services composition. Int. J. Web and Grid Services 1(1), 1–30 (2005)
4. Marconi, A., Pistore, M.: Synthesis and Composition of Web Services. In: Proc. of the 9th International School on Formal Methods for the Design of Computer, Communications and Software Systems: Web Services (SFM)
5. Canal, C., Murillo, J.M., Poizat, P.: Software Adaptation. L'Objet 12, 9–31 (2006)
6. Becker, S., Brogi, A., Gorton, I., Overhage, S., Romanovsky, A., Tivoli, M.: Towards an Engineering Approach to Component Adaptation. In: Reussner, R., Stafford, J.A., Szyperski, C. (eds.) Architecting Systems with Trustworthy Components. LNCS, vol. 3938, pp. 193–215. Springer, Heidelberg (2006)
7. Seguel, R., Eshuis, R., Grefen, P.: An Overview on Protocol Adaptors for Service Component Integration. Technical report, Eindhoven University of Technology (2008) BETA Working Paper Series WP 265
8. Brogi, A., Popescu, R.: Automated Generation of BPEL Adapters. In: Dan, A., Lamersdorf, W. (eds.) ICSOC 2006. LNCS, vol. 4294, pp. 27–39. Springer, Heidelberg (2006)
9. Mateescu, R., Poizat, P., Salaün, G.: Adaptation of service protocols using process algebra and on-the-fly reduction techniques. In: Bouguettaya, A., Krueger, I., Margaria, T. (eds.) ICSOC 2008. LNCS, vol. 5364, pp. 84–99. Springer, Heidelberg (2008)

10. Nezhad, H.R.M., Xu, G.Y., Benatallah, B.: Protocol-aware matching of web service interfaces for adapter development. In: Proc. of WWW, pp. 731–740 (2010)
11. Peer, J.: Web Service Composition as AI Planning – a Survey. Technical report, University of St.Gallen (2005)
12. Chan, K.S.M., Bishop, J., Baresi, L.: Survey and comparison of planning techniques for web service composition. Technical report, Dept Computer Science, University of Pretoria (2007)
13. Ghallab, M., Nau, D., Traverso, P.: Automated Planning: Theory and Practice. Morgan Kaufmann Publishers, San Francisco (2004)
14. Blum, A.L., Furst, M.L.: Fast Planning through Planning Graph Analysis. Artificial Intelligence Journal 90(1-2), 225–279 (1997)
15. ter Beek, M.H., Bucchiarone, A., Gnesi, S.: Formal Methods for Service Composition. Annals of Mathematics, Computing & Teleinformatics 1(5), 1–10 (2007)
16. Bozkurt, M., Harman, M., Hassoun, Y.: Testing Web Services: A Survey. Technical Report TR-10-01, Centre for Research on Evolution, Search & Testing, King's College London (2010)
17. Kiepuszewski, B.: Expressiveness and Suitability of Languages for Control Flow Modelling in Workflow. PhD thesis, Queensland University of Technology, Brisbane, Australia (2003)
18. Brogi, A., Popescu, R.: Towards Semi-automated Workflow-based Aggregation of Web Services. In: Benatallah, B., Casati, F., Traverso, P. (eds.) ICSOC 2005. LNCS, vol. 3826, pp. 214–227. Springer, Heidelberg (2005)
19. Ben Mokhtar, S., Georgantas, N., Issarny, V.: COCOA: COnversation-based Service Composition in PervAsive Computing Environments with QoS Support. Journal of Systems and Software 80(12), 1941–1955 (2007)
20. Benigni, F., Brogi, A., Corfini, S.: Discovering Service Compositions that Feature a Desired Behaviour. In: Krämer, B.J., Lin, K.-J., Narasimhan, P. (eds.) ICSOC 2007. LNCS, vol. 4749, pp. 56–68. Springer, Heidelberg (2007)
21. Berardi, D., Giacomo, G.D., Lenzerini, M., Mecella, M., Calvanese, D.: Synthesis of Underspecified Composite e-Services based on Automated Reasoning. In: Proc. of ICSOC (2004)
22. Bertoli, P., Pistore, M., Traverso, P.: Automated composition of web services via planning in asynchronous domains. Artif. Intell. 174(3-4), 316–361 (2010)
23. Klush, M., Gerber, A., Schmidt, M.: Semantic Web Service Composition Planning with OWLS-Xplan. In: Proc. of the AAAI Fall Symposium on Agents and the Semantic Web (2005)
24. Constantinescu, I., Binder, W., Faltings, B.: Service Composition with Directories. In: Löwe, W., Südholt, M. (eds.) SC 2006. LNCS, vol. 4089, pp. 163–177. Springer, Heidelberg (2006)
25. Liu, Z., Ranganathan, A., Riabov, A.: Modeling Web Services using Semantic Graph Transformation to Aid Automatic Composition. In: Proc. of ICWS. (2007)
26. Zheng, X., Yan, Y.: An Efficient Web Service Composition Algorithm Based on Planning Graph. In: Proc. of ICWS, pp. 691–699 (2008)
27. Melliti, T., Poizat, P., Ben Mokhtar, S.: Distributed Behavioural Adaptation for the Automatic Composition of Semantic Services. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE 2008. LNCS, vol. 4961, pp. 146–162. Springer, Heidelberg (2008)
28. Bracciali, A., Brogi, A., Canal, C.: A Formal Approach to Component Adaptation. Journal of Systems and Software 74(1), 45–54 (2005)

29. Canal, C., Poizat, P., Salaün, G.: Model-based Adaptation of Behavioural Mismatching Components. IEEE Transactions on Software Engineering 34(4), 546–563 (2008)
30. Tivoli, M., Inverardi, P.: Failure-free coordinators synthesis for component-based architectures. Science of Computer Programming 71(3), 181–212 (2008)
31. Beauche, S., Poizat, P.: Automated Service Composition with Adaptive Planning. In: Bouguettaya, A., Krueger, I., Margaria, T. (eds.) ICSOC 2008. LNCS, vol. 5364, pp. 530–537. Springer, Heidelberg (2008)
32. Yan, Y., Poizat, P., Zhao, L.: Repairing service compositions in a changing world. In: Proc. of SERA (2010)
33. Moser, O., Rosenberg, F., Dustdar, S.: Non-intrusive monitoring and service adaptation for ws-bpel. In: Proc. of WWW, pp. 815–824 (2008)

Performance Prediction of Service-Oriented Systems with Layered Queueing Networks

Mirco Tribastone, Philip Mayer, and Martin Wirsing

Institut für Informatik
Ludwig-Maximilians-Universität München, Germany
{tribastone,mayer,wirsing}@pst.ifi.lmu.de

Abstract. We present a method for the prediction of the performance of a service-oriented architecture during its early stage of development. The system under scrutiny is modelled with the UML and two profiles: UML4SOA for specifying the functional behaviour, and MARTE for the non-functional performance-related characterisation. By means of a case study, we show how such a model can be interpreted as a layered queueing network. This target technique has the advantage to employ as constituent blocks entities, such as threads and processors, which arise very frequently in real deployment scenarios. Furthermore, the analytical methods for the solution of the performance model scale very well with increasing problem sizes, making it possible to efficiently evaluate the behaviour of large-scale systems.

1 Introduction

Service-oriented architectures (SOAs) pose challenging problems regarding the evaluation of their performance. Approaches based on field measurements are problematic when systems distributed on a large scale are to be assessed. In many cases, parts of the system are not directly accessible to the engineer, perhaps because they employ third-party services. Even if the whole system was accessible, profiling may turn out to be an unduly costly exercise. However, during early stages of the development process, the engineer may content herself with some, perhaps approximate and less expensive, prediction of the performance of the system. A predictive model is often expressed as a mathematical problem. This has the advantage that one can easily tune its parameters so as to carry out analyses such as *capacity planning*, i.e., optimising the amount of processing capacity to satisfy some assigned quality-of-service guarantees.

This is the topic addressed in this paper. Specifically, we are concerned with situations which employ model-driven development techniques for the specification of the functional behaviour of SOAs. We consider a system modelled with UML4SOA [14], a UML profile which allows behavioural specifications of services and service orchestrations by using activities and composite structure descriptions, modelling the interconnections between services. The model is augmented with elements that specify the (intended or predicted) performance characteristics of the system. To this end, we adopt MARTE [16], another UML profile

T. Margaria and B. Steffen (Eds.): ISoLA 2010, Part II, LNCS 6416, pp. 51–65, 2010.

which extends behavioural UML specifications by adding timing properties. Furthermore, the model is accompanied by a deployment specification which emphasises the processing capacity of the computing platform on which the SOA is run. In this manner, a UML4SOA model becomes amenable to translation into a performance model as a layered queueing network (LQN) [7].

The motivation for the choice of LQNs is twofold. First, the LQN model features basic elements which have semantics close to corresponding elements of UML activities and behaviours in general. Indeed, research pursued in this direction has led to an abstract model of UML behaviours which can be used to automate the process of extracting an underlying LQN performance model [18]. Second, the analytical methods available for LQNs are very scalable with respect to increasing problem sizes. This makes this approach particularly convenient when the modeller wishes to predict the performance of large-scale SOAs, whose analysis would be otherwise computationally difficult when using approaches such as simulation or discrete-state models with explicit state-space enumeration.

The approach taken for the extraction of the LQN model is discussed by means of a case study and a numerical evaluation gives examples of the kinds of indices of performance that can obtained and how those can be interpreted in terms of the elements of the UML4SOA model.

Related Work. The general line of research followed by the present work is that on early-stage prediction of performance characteristics of software systems (see [1] for an excellent survey), with focus on designs of SOAs within the context of the EU SENSORIA project [12]. In particular, it is most closely related to [8], where UML4SOA was translated into the stochastic process algebra PEPA [10]. In that paper, the profile for MARTE was also used for performance annotation although the translation did not take into account deployment information. In effect, the resulting model was based on an *infinite-server* assumption, i.e., it was assumed that the system had as much processing capacity as it required. In this context, delays were only due to the clients contenting for a limited number of software threads. Conversely, the translation proposed here does model processing capacity explicitly — in fact its crucial role in the overall performance of the system will be exemplified in the numerical evaluation of the case study. LQNs were also considered in [11] as the target performance description of the Palladio Component Model [4].

Paper Organisation. Section 2 gives a brief overview of UML4SOA and Section 3 discusses the case study. The main concepts of the LQN model are overviewed in Section 4. Section 5 illustrates the translation of UML4SOA into LQN and Section 6 provides an example of a numerical evaluation of the case study. Finally, Secion 7 concludes the paper with pointers for future research.

2 Modelling Services in UML4SOA

The Unified Modelling Language (UML) [15] is a well-known and mature language for modelling software systems with support ranging from requirement

modelling to structural overviews of a system down to behavioural specifications of individual components. However, the UML has been designed with object-oriented systems in mind, thus native support and top-level constructs for service-oriented computing such as participants in a SOA, modelling service communication, and compensation support are missing. As a consequence, modelling SOA systems with plain UML requires the introduction of technical helper constructs, which degrade usability and readability of the models.

Adding service functionality to UML is currently under investigation in both academia and industry. Static aspects of service architectures are addressed in SoaML [17], an upcoming standard of the OMG for the specification of service-oriented architectures. For describing the behaviour of services, we have introduced the UML4SOA profile [14] which allows the description of service behaviour in the form of specialised UML activities.

UML4SOA is built on top of the Meta Object Facility (MOF) metamodel and is defined as a conservative extension of the UML2 metamodel. For the new elements of this metamodel, a UML profile is created using the extension mechanisms provided by the UML. The principle followed is that of minimal extension, i.e. to use UML constructs wherever possible and only define new model elements for specific service-oriented features and patterns, thus increasing readability and conciseness of the resulting models and diagrams.

The core of the UML4SOA profile considered in this paper is based on the the following two concepts:

Communication Primitives. UML4SOA extends the classic UML communication actions with four specialised actions for service communication: «send», «receive», «send&receive», and «reply». As the names suggest, these actions are used to model sending a call to a remote service, receiving a call from a remote service, performing both in one step, and replying to a previous call from a remote service, respectively. Specialised pins may be added to each action. Most importantly, the link (lnk) pin indicates the remote service (or more specifically, the port where the service is attached) the current action is targeted at. The send (snd) and receive (rcv) pins indicate from which variables or variable contents to retrieve, or in which variable to place data sent or received.

Compensation. Services are often used to implemented long-running transactions, such as (parts of) business processes. Compensation is a mechanism for undoing *successfully completed work* of a service if, in the course of later transactions, an error occurs and the complete process must be rolled back. UML4SOA lifts the specification and invocation of compensation handlers up to a first-level entity of UML activities. Instead of using standard activities and structured activity nodes, UML4SOA introduces the concept of a «serviceActivity» to which handlers can be attached using edges; in the case of compensation, of a «compensationEdge». A compensation handler can be invoked with the new actions «compensate» and «compensateAll».

In this paper, the UML4SOA extensions to the UML are used to model the mobile payment case study in the next section. In order to keep the example

small and simple, only the first of the above-mentioned three core features is used (communication primitives). For more information on UML4SOA, the interested reader is referred to the UML4SOA specification [13].

3 Mobile Payment Case Study

The case study in this paper is taken from the domain of financial transactions. It models a mobile payment gateway which allows customers to pay with their mobile communication device, such as a phone. The system is implemented using a collection of services which interact with one another to complete a payment request from a customer. The architecture of the system is shown in Figure 1.

The main service is the MobilePaymentGateway shown in black. A client (MobileDevice) can use this service to perform a payment operation. The gateway first authenticates the customer using the AuthenticationService and, if the authentication is successful, performs the payment using the PaymentService. The last two services use additional services to fulfill their tasks.

Three of the services present in the system are actually orchestrations of services. These services have been modelled using UML4SOA as shown in Figure 2; from left to right, the modelled services are the MobilePaymentGateway, the AuthenticationService, and the PaymentService.

Mobile Payment Gateway. As the name suggests, the gateway is the main entrance point for the customer to the payment service. An instance of this service is started when a paymentRequest call from the customer is received; the link pin indicates that the call must come through the the device port. In the receive pin, the call payload target is specified. In this case, the data attached to the call is to be placed in the variable payment.

The gateway proceeds to authenticating the customer by using the AuthenticationService which is attached to the authService port. If authentication fails, an error is returned to the customer. Otherwise, the payment is delegated to

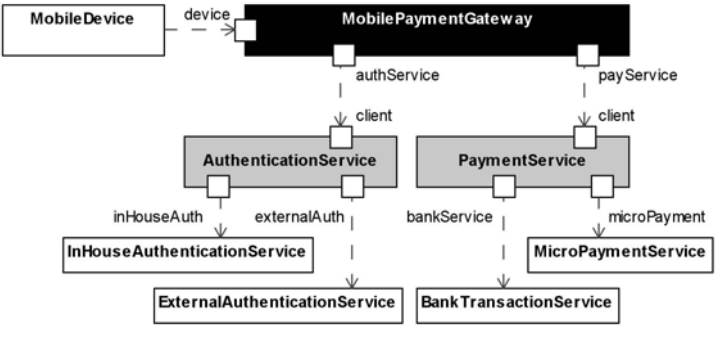


Fig. 1. Architecture of the Case Study

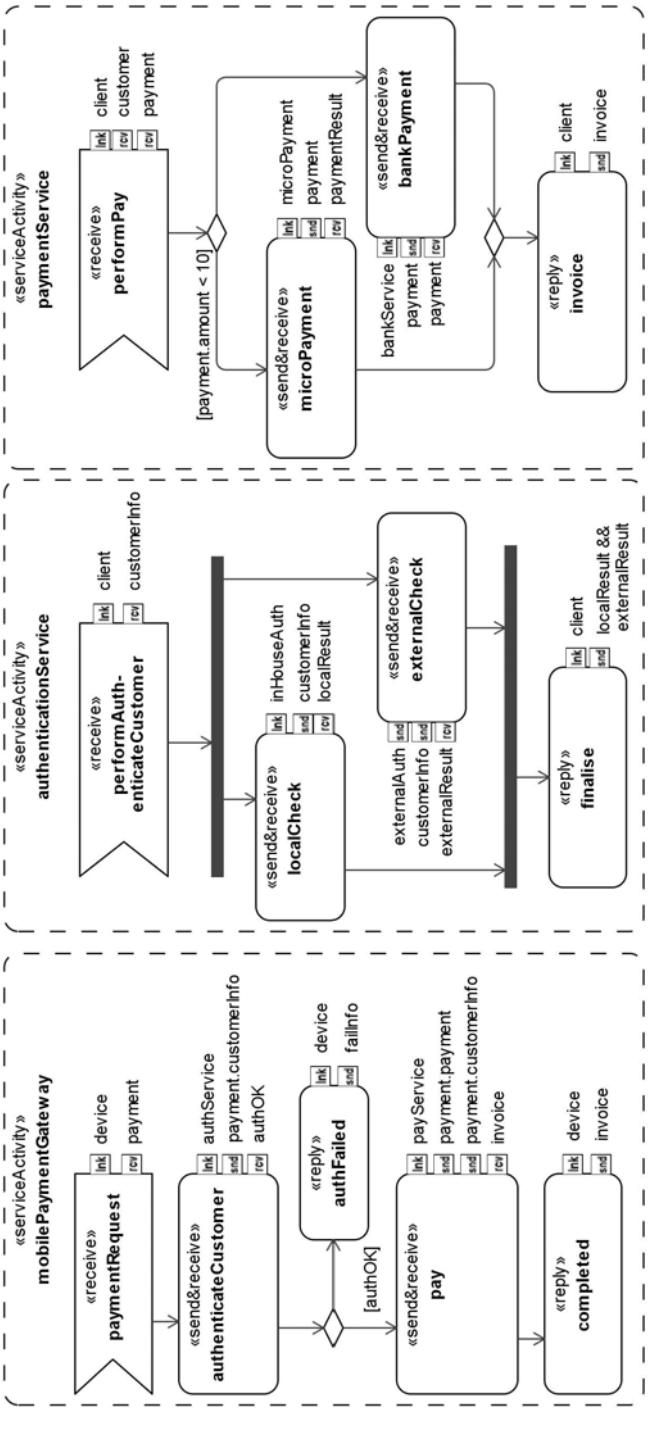


Fig. 2. Behaviour of the three orchestrations in the Payment Case Study

the PaymentService attached to the payService port: This service requires both the customer and the payment information. Finally, the result of the payment operation is returned to the client.

Authentication Service. The authentication service attempts to authenticate the customer with the system to ensure that the customer is allowed to use the service. For maximum security, both an in-house and an external authentication service are contacted in parallel: The result is only positive if both services agree. Note that the client port used in the link pins of the first and last method refers to the mobile gateway service on the left.

Payment Service. The payment service is invoked last, and tries to withdraw the appropriate amount of money from the customer's account. Depending on the amount, two different services are used. If the amount is less than 10, a micro-payment service is used, which aggregates several small payments until the customer is billed. If the amount is larger than 10, the amount is directly drawn from the customer's account via the bank. The micro-payment service has the advantage of performing faster, but cannot be used for large payments.

Summarising, the mobile payment gateway enables customers to perform a payment operation. The gateway and its invoked services have to consider a series of constraints to carry out this task. Overall, seven services and one client take part in this SOA.

4 The Layered Queueing Model

The Layered Queueing Network (LQN) model is an extension of classical queueing networks (e.g., [9,3]). Its primitives are entities that re commonly present in distributed computing systems, such as multi-threaded servers, multi-processor hardware, and inter-process communication via synchronous and asynchronous messages. The model can be solved via stochastic simulation, or more efficiently through approximate techniques which are known to be accurate (i.e., usually within 5%) and scalable with increasing problem sizes (e.g., [5,2]). The analytical methods will be preferred over stochastic simulation in the numerical evaluation conducted in Section 6. In general, they seem more appropriate when evaluating large-scale models such as SOAs. The remainder of this section gives an informal and brief overview of the LQN model, with particular emphasis on the notions that will be employed to analyse the case study presented in this paper. This description makes references to the graphical notation for LQNs. The reader may wish to consult Figure 5 for an example of such a representation. For more details on the LQN model, in addition to [7], the interested reader is referred to [6] which provides a tutorial and documents the functionality implemented in the *Layered Queueing Network Solver* tool.

The LQN model comprises the following elements.

Task. A task usually represents a software component that is capable of serving different kinds of requests. It is represented graphically as a stacked parallelogram and is associated with a multiplicity which models the number of concurrent instances of the task available at runtime. For instance, if the task models a multi-threaded application, then its multiplicity is the size of the thread pool.

Processor. A task is deployed onto a processor element, which is denoted graphically by a circle connected to the task. Its multiplicity represents the number of parallel processors available. Different tasks may be deployed on the same processor.

Entry. An entry is a kind of service exposed by a task. It is depicted as a small parallelogram in the top area inside the task. In the remainder of this paper we shall be concerned with single-entry tasks only.

Activity. An activity may be regarded as the basic unit of computation of the LQN model. A directed graph whose nodes are activities (drawn as rectangles) expresses the dynamic behaviour of an entry. This graph — called the *execution graph* — is shown inside the task's parallelogram where the entry resides. An activity denotes some computation time required on the processor on which its task is deployed. This time is assumed to be exponentially distributed with expectation specified within square brackets in the activity's rectangle. Activities may be connected through nodes to model the following behaviour:

- Sequential behaviour is specified by two activities being connected through an edge.
- Probabilistic choice (denoted by a small + circle) performs one of the activities connected by the outgoing edges according to some probability mass function, specified through labels on the outgoing edges.
- Fork/Join (denoted by a small & circle): a fork executes in parallel all the activities reached by its outgoing edges whereas a join waits until all activities connected through its incoming edges terminate.

Inter-process Communication. Activities within one task may invoke entries of another task. Service invocation is represented graphically by an solid arrow which connects the activity with the invoked entry. The arrow is labelled with a number within parentheses which denotes the number of invocations performed per execution of one activity. The semantics for *synchronous* invocation is that the calling activity suspends until the execution graph of the callee terminates. Termination of an execution graph is denoted by a dotted arrow pointing back to the graph's entry. For an *asynchronous* invocation, the invoked activity is executed concurrently with the execution graph of the calling activity. Synchronous invocations are depicted with a solid arrowhead whereas asynchronous ones are depicted with an open arrowhead.

Reference Task. A task with no incoming service invocation is called a reference task and models the system's workload.

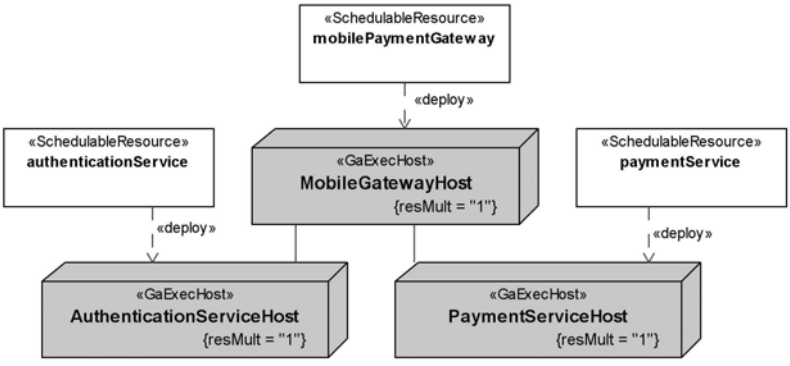


Fig. 3. Annotated deployment diagram for the case study

5 LQN Models for UML4SOA

5.1 Performance Annotations with MARTE

As discussed in Section 1, the LQN model requires further quantitative information on the required execution times of the activities, which is not available in the UML nor in the UML4SOA profile. To this end, we adopt the same approach taken in [8] where the profile for MARTE (cfr. [16]) is employed to augment the model the timing specifications.

Deployment Information. In the following, we assume that the UML4SOA model contains a deployment specification in which each of the services is associated with a node. Each node must be stereotyped with «GaExecHost», which indicates a host that is capable of carrying out computation. In particular, the property `resMult` will be used to extract the processor multiplicity in the LQN performance model. The artifacts deployed on these nodes are stereotyped with «SchedulableResource». The name of the artifact is referenced by the service activity which implements its behaviour. An example of a suitable deployment diagram is shown in Figure 3. In this scenario each service is run on a separate single-processor machine. Therefore, if the services are executed as multithreaded applications, all threads will contend for the CPU time of the same processor. This is one potential source of delays (i.e., queueing effects), as will be discussed in more detail in Section 6.

Stereotyping of Service Activities. We now discuss how each service activity is to be annotated with MARTE stereotypes. Figure 4 shows an excerpt of the complete model regarding the annotations on the MobilePaymentGateway service. The other two services are annotated similarly and are not shown due to space constraints. Each UML4SOA service activity is stereotyped with «PaRunTInstance», indicating that the activity is an instance of a runtime object. The

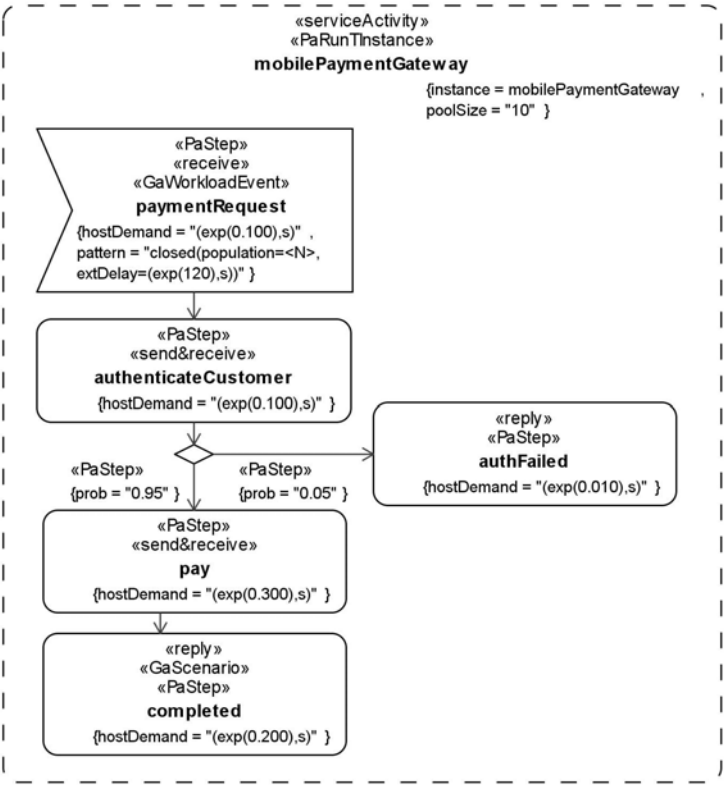


Fig. 4. The service activity for the mobile payment gateway annotated with the profile for MARTE

crucial property of this stereotype is `poolSize` which holds an integer that indicates the number of available threads at runtime. The property `instance` is set to the corresponding name in the deployment specification. In the example, MobilePaymentGateway is executed as a ten-thread application which runs on MobileGatewayHost. Each node of a UML4SOA service activity is stereotyped with MARTE's «PaStep» with the property `hostDemand` set to `(exp(<time>), s)`, indicating the exponential distribution associated with that action. Because of its probabilistic interpretation, as discussed later in this section, the outgoing edges of a decision node must be stereotyped with «PaStep» with the property `prob` set such that the sum across all edges equals one.

Workload specification. The «receive» action node which triggers the whole system must be stereotyped with «GaWorkloadEvent», which describes the dynamics of the arrival of the requests by clients. Thus, paymentRequest of MobilePaymentGateway is stereotyped with «GaWorkloadEvent», whereas authenticateCustomer and pay are not because these activities are not triggered

directly by users. The property `pattern` of this «GaWorkloadEvent» is set to `closed(population=<N>, extDelay=(exp(1/<think>),s))` to model a population of `N` users of the service-oriented architecture which make payment requests cyclically, interposing a think time of `think` seconds between successive requests.

The uses of the profile for MARTE specified above are sufficient to derive the LQN performance model from the UML4SOA specification, as discussed next.

5.2 Extracting the LQN Model

We will focus on the extraction of the LQN model from the specific case study presented in the paper. In doing so, we will describe some patterns of translation that can find a wider applicability. A more formal and general specification of the meta-model transformation of UML4SOA (and the MARTE profile) to LQN is the subject of ongoing work. For the sake of clarity, we find it more convenient to present first the overall LQN model of the case study (in Figure 5) and then guide the reader through the steps taken to obtain it.

Each «serviceActivity» is modelled as an LQN task, conveniently denoted by the same name. The task multiplicity is inferred from the application of the stereotype of «PaRunTInstance», as discussed above. The task is associated with an LQN processor which is named after the node in the deployment specification on which the «serviceActivity» is deployed. Each task has a single entry, named after the «receive» action node that triggers the service. The execution graph of an entry resembles closely the service activity in the UML model. Any action node — except for the triggering «receive» node — is translated into an LQN activity with execution demand taken from the «PaStep» stereotype application. For instance, the «send&reply» node authenticateCustomer is modelled as a basic LQN activity with execution demand equal to 0.100. (For ease of reference, basic activities are named after their corresponding action nodes as well.) Decision nodes are interpreted as probabilistic choices, with probabilities taken from the «PaStep» application to their outgoing edges. UML's forks and join are simply translated into their LQN analogues.

The exchange of messages between services is modelled as an invocation of external entries in the LQN model. In this paper we focus our attention on synchronous messages which return a reply to the caller. In the UML4SOA model, this corresponds to having action nodes stereotyped with «send&receive» in the invoking service and a «reply» node which (in this case study) terminates the service that is invoked. Therefore, the LQN activities corresponding to the «reply» nodes present dotted arrows directed to the parent entry (e.g., see paymentCompleted and authenticateCustomer).

The necessary information to translate the pattern of synchronous communication is gathered from the names of the lnk pins in the service activities and from the architectural specification. Given a «send&receive» node, the name of the lnk pin is used to find the corresponding edge which connects the two communicating services in the composite structure specification. For example, the lnk pin of the authenticateCustomer node is authService, which connects

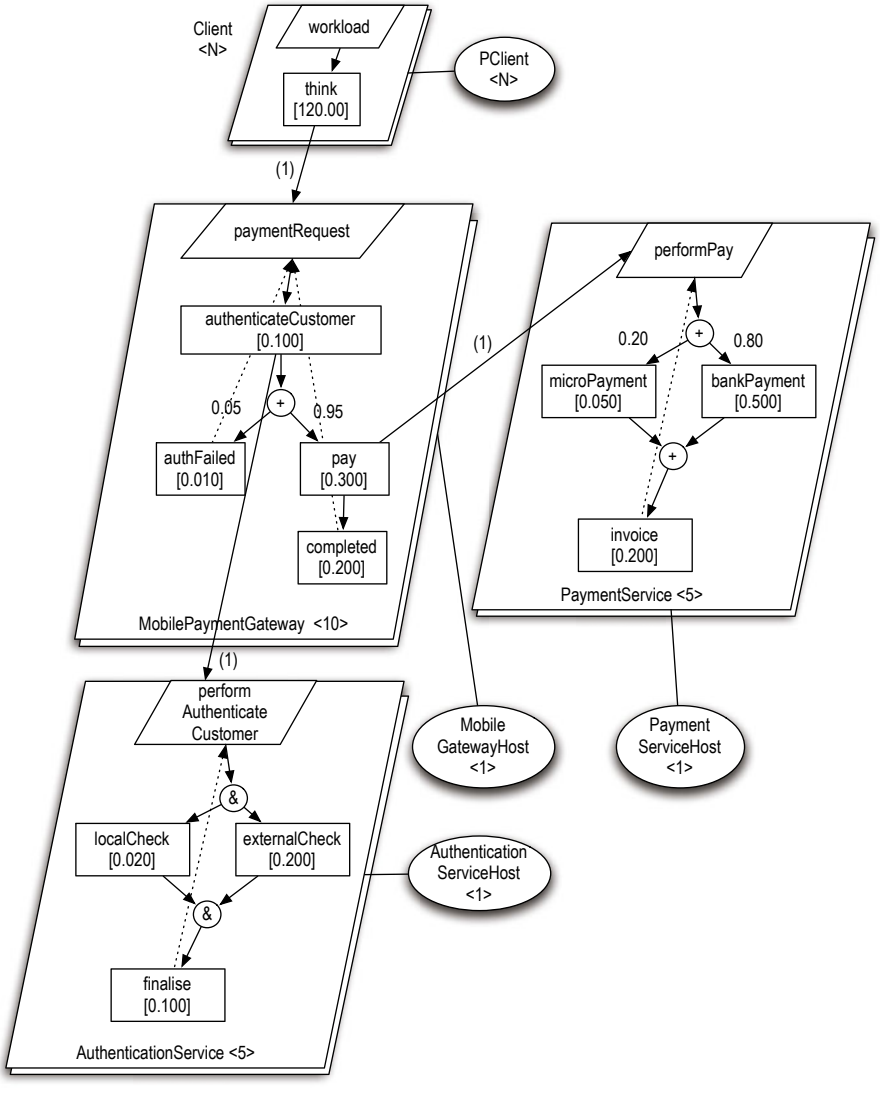


Fig. 5. Layered Queueing Network model of the Payment case study

MobilePaymentGateway to AuthenticationService. If the connected component has an explicit behavioural specification in terms of «serviceActivity» — as is the case for AuthenticationService — then the LQN model will feature a synchronous call (with multiplicity one) from the activity that models the «send&receive» node to the entry of the invoked activity. Using the same example, the LQN model has a synchronous invocation of the entry performAuthenticateCustomer from the activity node authenticateCustomer.

There may be in the UML4SOA model invocations of external services which are not given a behavioural specification — e.g., the externalCheck «send&receive» node in AuthenticationService invokes some ExternalAuthenticationService of unspecified behaviour. Such nodes will be translated as basic LQN activities which do not make calls to other entries. In effect, this external invocation is abstracted away in the LQN model and its impact on the performance of the system in encompassed in the execution demand, as specified in the «send&receive» node.

The specification of the system workload through the «GaWorkloadEvent» is translated into an LQN reference task as follows. A task named *Client* is created with one entry called *workload*. Its execution graph consists of a single activity named *think* with execution demand equal to `<think>`. The task has multiplicity `N` and is deployed on a processor with multiplicity `N` named *PClient*. A synchronous message (with multiplicity one) connects *think* with the entry corresponding to the action node with «GaWorkloadEvent», i.e., paymentRequest in the case study.

5.3 Indices of Performance

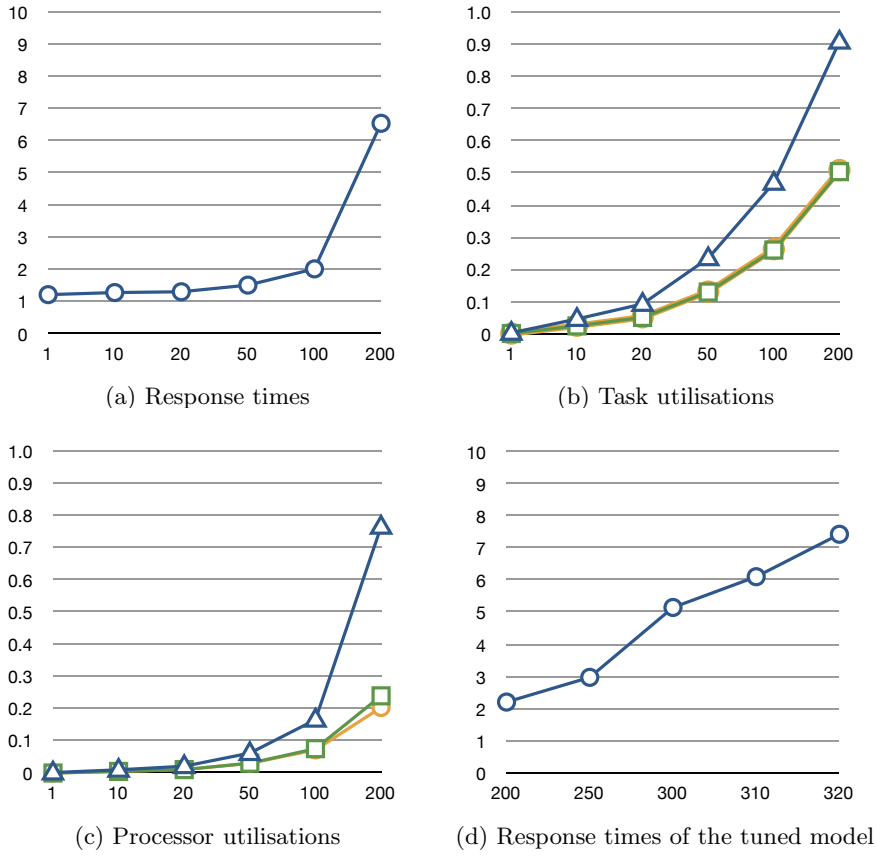
The analysis techniques available for LQNs provide the modeller with a wide range of quantitative estimates on the *long-run* (or *steady-state*) behaviour of the system, i.e., the performance attained after a sufficiently long period of time that the system was started. This appears to be an appropriate characterisation of the performance of real-life service-oriented architectures, which are usually on-line continuously. In this paper we put emphasis on two such indices:

- The *response time* for a client, measured as the average time it takes for the system to process a payment request. The response time does not include the think time by the client, but it does include all the execution times of the basic activities that are involved during the processing of a request and the time (if any) spent while waiting for the system resources (e.g., threads and processors) to be available.
- The *processor utilisation*, which measures the average number of processors in a multi-processor component that are busy. Alternatively, this value, if normalised with respect to the total multiplicity of the processor, can be interpreted as the percentage of time that a processor is busy. Analogously, the *task utilisation* measures the average number of threads that are busy.

In the following section, these two indices will be used in a numerical evaluation of the performance of our case study. Another notable performance metric — not discussed further in this paper due to space constraints — is *throughput*, i.e., the rate at which an entry (or an activity) is executed.

6 Numerical Example

The performance study addressed in this section is a typical *dimensioning* problem, in which the modeller wishes to find a suitable configuration of the system



(a) Response times

(b) Task utilisations

(c) Processor utilisations

(d) Response times of the tuned model

Fig. 6. Numerical results. x-axis: population sizes. The y-axis for response times are in time units, whereas utilisation is a dimensionless metric between 0 and 1. The markers in (b) and (c) are related with the UML4SOA components as follows. Triangle: MobileGatewayHost; Square: AuthenticationServiceHost; Circle: PaymentServiceHost.

in order to satisfy some quality-of-service criteria. In the following, we assume for the sake of simplicity that the execution demands are given and fixed, as shown in Figure 5. Thus, the parameters that may be changed are the multiplicities of the tasks and of the processors in the model. Perhaps the most intuitive index of performance is the average response time experienced by a client; this index is shown in Figure 6a for increasing system workload, represented by the property of `population = N` in the UML model.

The baseline $N = 1$ is of interest because it gives the minimum response time attainable, since there is no contention for resources in the system. The curve shows a typical profile, characterised by increasing response times as function of N, with sharp deterioration after some critical point. In this example, the response time at $N = 200$ is about six times as much as the baseline value.

The normalised utilisation profiles for the tasks and the processors, shown in Figures 6b and 6c, respectively, offer more insight into where the degradation of performance arises from. Clearly, the utilisations increase with increasing workload, however those related with AuthenticationServiceHost and PaymentServiceHost do not appear to be problematic since they are at most about 50% in the worst case $N = 200$. Instead, the processor utilisation of MobileGatewayHost is about 91%, indicating a heavy utilisation of this resource.

Taken together, these results suggest that an effective route toward performance improvement is to add more processing capacity to MobileGatewayHost. Figure 6d shows the response times when the node is deployed on a two-processor machine (instead of the original single-processor one). The response time at $N = 200$ is now about one third of the original model, and the system can sustain up to 310 clients with an average response time that would be delivered with only 200 clients in the original configuration.

7 Conclusion

We discussed a methodology for extracting a layered queueing network performance model from a service-oriented architecture description in UML4SOA. The level of abstraction of LQNs appears convenient for the prediction of the quantitative behaviour of a system under scrutiny. The services are modelled as multithreaded applications communicating with each other. The explicit modelling of the deployment scenario puts constraints on the level of threading and on the processing power of the hardware on which the application is run. A numerical evaluation of the case study has shown how marginal changes to the deployment can have a significant impact on the predicted performance. It is our opinion that the possibility of effortless experimentation with different configurations and a generally deeper insight into the system's dynamics outweigh the additional modelling effort required to augment the model with performance-related annotations.

In order to widen the applicability of this methodology and make it available to practitioners in SOAs, further research needs to be carried out. It is our plan to provide a precise, formal characterisation of the meta-model transformation presented in this paper, which would also support other UML4SOA constructs, such as compensations and event handling which were not considered here. On a more practical level, we would like this transformation to be implemented as a module in leading UML modelling tools to be able to experiment with larger, real-world service-oriented applications.

Acknowledgement. This work was supported by the EU-funded project SENSORIA, IST-2005-016004.

References

1. Balsamo, S., Di Marco, A., Inverardi, P., Simeoni, M.: Model-Based Performance Prediction in Software Development: A Survey. IEEE Trans. Software Eng. 30(5), 295–310 (2004)

2. Bard, Y.: Some extensions to multiclass queueing network analysis. In: Third International Symposium on Modelling and Performance Evaluation of Computer Systems, pp. 51–62. North-Holland, Amsterdam (1979)
3. Baskett, F., Mani Chandy, K., Muntz, R.R., Palacios, F.G.: Open, closed, and mixed networks of queues with different classes of customers. J. ACM 22(2), 248–260 (1975)
4. Becker, S., Koziolek, H., Reussner, R.: Model-based performance prediction with the palladio component model. In: Proceedings of the 6th international workshop on Software and performance, vol. 65. ACM, New York (2007)
5. Mani Chandy, K., Neuse, D.: Linearizer: A heuristic algorithm for queueing network models of computing systems. Commun. ACM 25(2), 126–134 (1982)
6. Franks, G., Maly, P., Woodside, M., Petriu, D., Hubbard, A.: Layered Queueing Network Solver and Simulator User Manual (2005), http://www.sce.carleton.ca/rads/lqns
7. Franks, G., Omari, T., Murray Woodside, C., Das, O., Derisavi, S.: Enhanced modeling and solution of layered queueing networks. IEEE Trans. Software Eng. 35(2), 148–161 (2009)
8. Gilmore, S., Gönczy, L., Koch, N., Mayer, P., Tribastone, M., Varró, D.: Nonfunctional properties in the model-driven development of service-oriented systems. Software and System Modeling (2010)
9. Gordon, W.J., Newell, G.F.: Closed queuing systems with exponential servers. Operations Research 15(2), 254–265 (1967)
10. Hillston, J.: A Compositional Approach to Performance Modelling. Cambridge University Press, Cambridge (1996)
11. Koziolek, H., Reussner, R.: A model transformation from the palladio component model to layered queueing networks. In: Kounev, S., Gorton, I., Sachs, K. (eds.) SIPEW 2008. LNCS, vol. 5119, pp. 58–78. Springer, Heidelberg (2008)
12. Wirsing, M., et al.: Sensoria: Engineering for Service-Oriented Overlay Computers. MIT Press, Cambridge (2009)
13. Mayer, P., Koch, N., Schroeder, A., Knapp, A.: The UML4SOA Specification. Specification, LMU Munich (2009), http://www.uml4soa.eu/wp-content/uploads/uml4soa_spec.pdf.
14. Mayer, P., Schroeder, A., Koch, N.: MDD4SOA: Model-Driven Service Orchestration. In: EDOC, pp. 203–212. IEEE Computer Society, Los Alamitos (2008)
15. Object Management Group (OMG): UML Superstructure Specification 2.1.2. Technical report, OMG (2007), http://www.omg.org/spec/UML/2.1.2/Superstructure/PDF/ (last accessed on May 5, 2009)
16. Object Management Group (OMG). A UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded systems, Beta 2. Technical report, Object Management Group (2008)
17. Object Management Group (OMG). Service oriented architecture Modeling Language(SoaML), Beta 1. Technical report, Object Management Group (2009)
18. Murray Woodside, C., Petriu, D.C., Petriu, D.B., Shen, H., Israr, T., Meseguer, J.: Performance by unified model analysis (PUMA). In: WOSP, pp. 1–12 (2005)

Error Handling: From Theory to Practice*

Ivan Lanese[1] and Fabrizio Montesi[2]

[1] Focus Team, Università di Bologna/INRIA, Bologna, Italy
lanese@cs.unibo.it
[2] Focus Team, Università di Bologna/INRIA,
Bologna and italianaSoftware s.r.l., Italy
fmontesi@italianasoftware.com

Abstract. We describe the different issues that a language designer has to tackle when defining error handling mechanisms for service-oriented computing. We first discuss the issues that have to be considered when developing error handling mechanisms inside a process calculus, i.e. an abstract model. We then analyze how these issues change when moving from a process calculus to a full-fledged language based on it. We consider as an example the language Jolie, and the calculus SOCK it is based upon.

1 Introduction

Nowadays computing systems are made of different interacting components, frequently heterogeneous and distributed. Components exploit each other functionalities to reach their goals, communicating through some network middleware. Components may belong to different companies, and they are not always reliable. Also, the underlying network infrastructure may be unreliable too, thus connections may break and components may disconnect. Nevertheless, applications should provide reliable services to their users. For these reasons, it becomes more and more important to deal with unexpected events, so to be able to manage them and get correct results anyway. That is, error (or fault) handling is today a major concern.

Service-oriented computing is a programming paradigm for developing complex distributed applications by composing simpler, loosely coupled services. This is implemented as a set of standards allowing to describe service interface and behavior, to look for services to perform the task at hand, to invoke them and to compose them so to produce the desired result. What said above about unexpected events holds, in particular, in the case of services. Thus different techniques and primitives for fault handling have been proposed in this field. For instance, WS-BPEL [21], the de-facto standard for web service composition, provides scopes, fault handlers, compensation handlers and termination handlers to deal with unexpected events.

However, the problem of finding good programming abstractions and primitives for programming reliable applications out of unreliable services is far from being solved. Take WS-BPEL for instance. Its specification is informal and unclear, and the interactions between the different primitives not clarified. This is

* Research supported by Project FP7-231620 HATS.

T. Margaria and B. Steffen (Eds.): ISoLA 2010, Part II, LNCS 6416, pp. 66–81, 2010.

witnessed by the fact that different implementations of WS-BPEL behave in different ways on many programs [14]. To avoid ambiguities, to clarify the expected behavior of programs, and to prove properties of the available mechanisms, formal methods are needed. Thus, there have been many proposals trying to specify WS-BPEL semantics in a formal way [16,15,22], and more in general proposing primitives for modeling web services and their fault handling mechanisms (see the related work paragraph below).

However, most of these proposals are at a very abstract level, and quite far from real programming languages. Thus the problem of exporting primitives and techniques from high-level theoretical models to full-fledged programming languages usable to program service-oriented applications in an industrial context has rarely been tackled, even less solved. This paper aims at describing Jolie [20,9], a language for programming service-oriented applications built on top of the calculus SOCK [4], which has been developed and exploited in practice by company italianaSoftware s.r.l. In particular, we will concentrate on its mechanisms for error handling, detailing the reasoning that drove their development, from the theoretical calculus SOCK to the full language Jolie.

SOCK and Jolie are a good choice to exemplify our ideas. First, they propose a novel approach to error handling with original features such as dynamic handler update and automatic fault notification. Also, Jolie has been developed closely following the semantics of SOCK, in particular as far as its error handling mechanisms are concerned.

Related works. There are many works in the literature on error handling for concurrent systems, such as service-oriented computing ones. Many of them are based on process calculi. The mechanisms they propose range from basic constructs such as the interrupt of CSP [8] and the try-catch found in most programming languages, to complex proposals such as the ones of Webπ [13], StAC [5], SAGAs calculi [3], dcπ [24], SOCK [6], ... Those models differ on many respects, ranging from flow composition models, where basic activities are composed and compensated (e.g., StAC or SAGAs calculi), to calculi taking into account communication and distribution, aiming at modeling distributed error handling (e.g., Webπ or SOCK). Another thread of research [2,11,12] tries to compare the expressive power of different models.

Our work has however a different perspective: we are not proposing new mechanisms nor comparing existing ones, but analyzing the requirements that all those mechanisms have to satisfy. We are not aware of other papers on this topic.

Structure of the paper. Section 2 introduces the basics of SOCK, without considering error handling. Section 3 discusses the main aims that should be reached when developing error handling mechanisms, considering both the cases of a calculus and of a full-fledged language. Section 4 presents the main features of SOCK for error handling and illustrates how it has tried to fulfill the requirements in Section 3. Section 5 does the same analysis for the peculiar mechanisms of Jolie. Finally, Section 6 concludes the paper.

Table 1. Service behavior syntax

$$\overline{\epsilon} ::= \overline{o}@z(\boldsymbol{y}) \mid \overline{o_r}@z(\boldsymbol{y},\boldsymbol{x}) \qquad \epsilon ::= o(\boldsymbol{x}) \mid o_r(\boldsymbol{x},\boldsymbol{y},P)$$

$P,Q,\ldots ::= \mathbf{0}$	null process			
$\overline{\epsilon}$	output	ϵ		input
$x := e$	assignment	if χ then P else Q		conditional
$P;Q$	sequential comp.	$P \mid Q$		parallel comp.
$\sum_{i \in W} \epsilon_i; P_i$	non-det. choice	while χ do (P)		iteration

2 SOCK

In this section we introduce SOCK [4], the calculus underlying Jolie [20,9]. We leave to next sections the description of its approach to fault handling, concentrating here on its standard behavior.

SOCK is a three-layers calculus. The behavior layer describes how single services act and communicate. The engine layer describes how the state of services is stored and how their sessions are instantiated and managed. The network layer allows to compose different engines in a network.

Error handling is mostly dealt with at the behavior layer, thus we will concentrate on it. We refer to [4] for a description of the other layers.

The language for defining behaviors in SOCK is inspired both from concurrent calculi, featuring for instance a built-in parallel composition operator, and imperative languages, providing for instance assignment (SOCK is stateful) and sequential composition.

A main point in SOCK behaviors concerns communication. SOCK behaviors communicate with each other using two modalities (inspired by WSDL [25]): one-way, where one message is sent, and request-response, where a message is sent and a response is computed and sent back.

To define the syntax of SOCK behaviors we consider the following (disjoint) sets: Var, ranged over by x, y, for variables, Val, ranged over by v, for values, $\mathcal{O}$, ranged over by o, for one-way operations, and $\mathcal{O}_R$, ranged over by o_r for request-response operations. Also, we use z to range over locations. The syntax for processes is defined in Table 1. There, $\mathbf{0}$ is the null process. Outputs can be notifications (for one-way communication) $\overline{o}@z(\boldsymbol{y})$ or solicit-responses (for request-response communication) $\overline{o_r}@z(\boldsymbol{y},\boldsymbol{x})$ where $o \in \mathcal{O}$, $o_r \in \mathcal{O}_R$ and z is a location. Notification $\overline{o}@z(\boldsymbol{y})$ sends a one-way communication to operation o located at location z (locations of behaviors are defined at the network layer), and variables $\boldsymbol{y}$ contain the data to be sent. Similarly, solicit-response $\overline{o_r}@z(\boldsymbol{y},\boldsymbol{x})$ sends a request-response communication to operation o_r located at location z, communicating values in variables $\boldsymbol{y}$, and then waits for an answer. When the answer is received, received values are assigned to variables in $\boldsymbol{x}$. Dually, inputs can be one-ways $o(\boldsymbol{x})$ or request-responses $o_r(\boldsymbol{x},\boldsymbol{y},P)$ where the notations are as above. Additionally, P is the process to be executed upon request to produce the response. Assignment $x := e$ assigns the result of the expression e to

the variable x. We do not present the syntax of expressions: we just assume that they include the arithmetic and boolean operators, values, variables and arrays. Conditional is written as $\mathsf{if}\ \chi\ \mathsf{then}\ P\ \mathsf{else}\ Q$. $P;Q$ and $P|Q$ are sequential and parallel composition respectively, whereas $\sum_{i\in W}\epsilon_i;P_i$ is input-guarded non-deterministic choice. Finally, $\mathsf{while}\ \chi\ \mathsf{do}\ (P)$ models iteration.

Example 1. Let us consider a (very simplified) service for performing money transfers between two accounts. Such a service can be invoked by:

$$\overline{pay_r}@bank(\langle src, dest, amount\rangle, \langle transId\rangle)$$

where *src* is the source account, *dest* the destination account, *amount* the amount of money to be moved and $transId$ a transaction Id to be used for later referring to the transaction.

A possible implementation for the service (again, very simplified) could be:

$$\begin{array}{l} pay_r(\langle src, dest, amount\rangle, \langle transId\rangle,\\ \quad acc[src] := acc[src] - amount;\\ \quad \underline{acc[dest]} := acc[dest] + amount;\\ \quad \overline{gen - id_r}@bank(\langle src, dest, amount\rangle, \langle transId\rangle)) \end{array}$$

We assume that this behavior is located at location *bank* and that there is another service at the same location, $gen - id$, which takes care of generating the transaction Id. Also, *acc* is an array containing accounts' credit.

3 The Quest for Error Handling Primitives

As described in the Introduction, the problem of finding good programming primitives for error handling is hard, as witnessed by the huge number and variety of proposals that have been put forward in the literature. Even considering a unique kind of model, many variants exist. SAGAs calculi [3,2,10] for instance may differ on whether parallel flows of computation are interrupted when an error occurs, on whether the compensation is centralized or distributed, and on whether the order of compensations depends on the static structure of the term or on the dynamic execution.

The difficulty in finding the best model for error handling in service-oriented computing is due to the many concerns such a model has to answer:

full specification: the model should define the behavior of error handling primitives in all the possible cases, including when the management of different errors interfere;

expressiveness: the available primitives should allow to specify all the error handling policies that may be necessary to program complex applications;

intuitiveness: the behavior of the provided primitives should match the intuition of programmers, allowing to understand the behavior of the applications;

minimality: we look for the simplest possible set of primitives able to model the required behavior, and, in particular, the different proposed mechanisms should be as much orthogonal as possible.

We describe in the next section how the error handling mechanisms proposed for SOCK have tried to satisfy the requirements above.

However, when moving from theoretical models to full-fledged languages, while most of the concerns above remain (actually, intuitiveness becomes even more important, while minimality is less critical), new ones emerge. The main ones are the following:

usability: the proposed primitives should be easy to use for the programmers when developing complex applications: while this is connected to intuitiveness and expressiveness, this goes beyond. For instance, this includes the development of suitable macros or additional primitives to simplify the writing of common patterns. Note that this is in contrast with the concern for minimality, which is more important in theoretical models than in real languages;

robustness: most theoretical models assume perfect communications, and do not consider low level failures, however these failures may happen in practice, and should be taken into account;

compatibility: in practice applications do not live alone, but they are immersed in a world including the network middleware and other applications, possibly developed using different languages and technologies: thus, in real languages, mechanisms should be provided to interact with other entities, which may follow different policies for error handling.

These additional concerns force the error handling approaches used in practice to be different, and in general more complex, w.r.t. the ones considered in theoretical models. This makes difficult to export the results obtained working on theoretical models (expressiveness results, property guarantees) to full-fledged languages. We will describe in Section 5 how those practical concerns have influenced the development of Jolie, discussing whether properties of SOCK are preserved in Jolie.

4 Error Handling in SOCK

Error handling in SOCK has been inspired by error handling in WS-BPEL, but explored some new directions, in particular concerning dynamic handler update and automatic fault notification.

As in WS-BPEL, error handling in SOCK is based on the concepts of fault, scope, fault handler, termination handler and compensation handler. A fault is an unexpected event that causes the interruption of the normal flow of computation. A scope defines the boundaries for error handling activities. In particular, each scope defines handlers specifying how to manage internal and external faults. A handler is a piece of code specifying how to react to particular faults. We consider three kinds of handlers: fault handlers specify how to deal with

Table 2. Service behavior syntax

$P, Q, \ldots ::= \ldots$		standard processes		
	$\{P\}_q$	scope	$\mathsf{inst}(\mathcal{H})$	install handler
	$\mathsf{throw}(f)$	throw	$\mathsf{comp}(q)$	compensate
	cH	current handler	$\overline{o_r}@z(\boldsymbol{y}, \boldsymbol{x}, \mathcal{H})$	solicit-response

internal faults, termination handlers specify how to smoothly terminate a scope when an external fault reaches it, compensation handlers specify how to undo the activities performed by a scope that already terminated with success, if this is needed for error recovery. All these concepts are realized by extending the syntax of SOCK with the primitives in Table 2. There f denotes a fault name and q a scope name. Furthermore, $\mathcal{H}$ denotes a function from fault and scope names to processes. We refer to [7] for a detailed description of the behavior of the primitives, including the formal semantics.

As already said, a scope $\{P\}_q$ is the main mechanism for structuring error handling. It has name q, and executes process P taking care of its faults. At the beginning, it defines no policy for error handling: policies are specified dynamically by installing handlers using operation $\mathsf{inst}(\mathcal{H})$. The intended semantics is that assigning a process P to a fault name f defines P as fault handler for f, while assigning P to the name q of the scope defines P as its termination handler. Handlers may replace or update previously defined handlers with the same name. This is done using the placeholder cH (for current handler), that during installation is replaced by the code of the old handler. Thus for instance $\mathsf{inst}([f \mapsto cH|Q])$ adds Q in parallel to the old handler for fault f.

Primitive $\mathsf{throw}(f)$ throws fault f: the fault propagates by killing the ongoing activities around itself until it reaches a scope. Then the handler for the fault is looked for: if it is available then it is executed, and fault propagation is stopped. Otherwise the scope is killed, and the fault propagates to the outside. During propagation, the fault may kill sibling scopes: in this case their termination handler is executed to ensure smooth termination. All error handling activities are executed in a protected way, thus ensuring that they are completed before taking care of successive errors. During error handling, it may be necessary to undo previously completed activities. To this end, compensation handlers are used. Compensation handlers are defined when a scope successfully terminates, and they correspond to the last defined termination handler. Thus, they are available and they can be executed using primitive $\mathsf{comp}(q)$.

The last main point concerning error handling is related to the request-response communication pattern. This communication pattern enforces a strong relationship between the caller and the callee: for instance, if the callee gives back no answer then the caller remains stuck. For this reason we want that errors on the callee are notified to the caller. In particular, if there is a local fault f in the callee, the same fault is sent back to the caller, where it is re-raised as a local fault, triggering management of the remote fault. In particular, the caller is guaranteed to receive either a successful answer or a fault notification and thus do not get stuck (unless the callee diverges).

Example 2. Consider a slightly more refined version of the bank service in Example 1, which checks first whether there is enough money in the source account, and throws fault f otherwise.

$$\begin{array}{l} pay_r(\langle src, dest, amount\rangle, \langle transId\rangle, \\ \quad \mathsf{if}\ acc[src] \geq amount\ \mathsf{then}\ \mathbf{0}\ \mathsf{else}\ \mathsf{throw}(f); \\ \quad acc[src] := acc[src] - amount; \\ \quad acc[dest] := acc[dest] + amount; \\ \quad \overline{gen - id_r}@bank(\langle src, dest, amount\rangle, \langle transId\rangle)) \end{array}$$

Thus, in case there is not enough money in the source account, the operation will fail and fault f is thrown both at the callee and at the caller sides.

Faults in the caller may influence the communication pattern too: if there is a failure which is concurrent to the solicit-response, different cases may occur. If the fault happens before the solicit-response is started, the solicit-response is not executed at all, and the remote partner is unaffected. If it is after instead, the answer for the partner is waited for. If this is successful, meaning that the remote partner has performed its task, then the local handler is updated according to the handler update defined in the solicit-response primitive. Thus, this handler update can take care of undoing the remote computation. If instead the remote computation has failed, an error notification is received, and the local handler update is not performed, since the remote computation had no effect. Also, the remote fault is not propagated locally, since the local computation has already failed.

Example 3. Consider again the service in Example 2. The client for such a service has to manage fault f. It can be written for instance as:

$$\begin{array}{l} \{\mathsf{inst}([f \mapsto ErrorMsg := \text{"Not enough money for the transfer"}; \\ \quad \overline{print}@user(\langle ErrorMsg\rangle)]); \\ \overline{pay_r}@bank(\langle src, dest, amount\rangle, \langle transId\rangle, [q \mapsto \overline{undo}@bank(\langle transId\rangle)])\}_q \end{array}$$

Now, if everything goes fine, upon receipt of the answer, the handler for q is installed, thus if later this scope has to be compensated, the undo of the payment operation is requested. If instead a fault occurs on the remote side, the handler is not updated and the undo will never be required. Instead, an error message is sent to the user. Even if there is a local fault, the answer will be waited for, and the handler update will be performed only if the answer is successful, thus the termination handler for q will undo the payment iff it has actually been performed.

4.1 Full Specification

The definition of the semantics of error handling (as well as of normal processing), should cover all the possible cases. Questions such as:

1. What happens if, while a fault is being managed, an external fault occurs?
2. What happens if both the caller and the callee of a request-response fail?
3. What happens if a fault handler causes a fault?

should not be left unanswered. Notice however that for informal specifications such as WS-BPEL one [21], it is very difficult to check whether all the cases have been specified. Instead, this is not normally a problem for formal specifications: the only possible transitions are the ones defined by the model, and the model fully describes what happens in each case (at worst, it specifies that no transition is possible). This is for instance the case for SOCK semantics [6]. Considering the questions above, it is easy to deduce the following answers:

1. The internal fault handler is executed in a protected way, thus the management of the external fault has to wait for the completion of local recovery.
2. A fault notification is sent back from the callee to the caller, the handler update specified by the solicit-response is not applied, but the remote fault is not propagated to the caller.
3. The fault is propagated as usual, and dealt with by existing handlers. Note that when the handler for fault f is executed, its definition is removed, thus further faults with the same name should be dealt with by external scopes.

4.2 Expressiveness

The available primitives should be able to express all the policies that may be necessary for programming applications. As stated, this is a very vague goal, since it is quite difficult to guess which kinds of policies may be necessary. For formal models, such a constraint is usually checked by relying on case studies and on encodings. As for SOCK, it has been applied to the specification of the Automotive [26] and Financial [1] case studies of the European project Sensoria [23], and the derived language Jolie is applied every day for programming service-oriented applications such as, for instance, a web portal for managing employer time sheets, VOIP service monitoring, and others. The results of these tests may trigger refinements of the language for improving its expressive power.

Another way of assessing the expressive power of SOCK is via encodings. By showing that another calculus can be encoded into SOCK, one shows that SOCK is at least as expressive as the other calculus. This has been done [12] for instance in the case of SAGAs. The results therein show that both static SAGAs with interruption and centralized compensations [2] and dynamic SAGAs [12] can be modeled into SOCK preserving some notion of behavior. This guarantees that each policy that can be expressed in these flavors of SAGAs can also be expressed in SOCK.

Another result concerning expressiveness is related to dynamic handler update: it is easy to show that SOCK dynamic handler update can easily model WS-BPEL static scopes. In WS-BPEL, each scope has statically associated a fault handler F_i for each fault f_i, a termination handler T and a compensation handler C. Using dynamic handler installation, this can be simulated as follows:

$$\{\mathsf{inst}([f_1 \mapsto F_1, \ldots, f_n \mapsto F_n, q \mapsto T]); P; \mathsf{inst}([q \mapsto C])\}_q$$

In [11] it is shown that dynamic handler update is strictly more expressive than both static recovery (as in WS-BPEL), and parallel recovery (where additional pieces of handlers can be added only in parallel). Albeit this result can not directly be applied to SOCK, since it is proved on a stateless calculus, this is another hint of the expressive power of dynamic handler update.

4.3 Intuitiveness

This is one of the most important, yet difficult to reach, goals for a programming language, and, in particular, for error handling mechanisms. Intuitiveness means that the behavior of the primitives follows the intuition of the programmer (or, better, of a programmer that has understood the basics of the approach). While the formal specification of the calculus is normally quite complex to understand for a programmer without specific background on formal methods, it is required that such a programmer can learn how to program in the language by reading some informal description (one has to resort anyway to the formal specification to work out the behavior in the most complex cases). This becomes much easier if the specification of the language is built on top of a few clear and orthogonal concepts. Having a formal specification, one may guarantee that those intuitive properties are actually valid in all the cases.

Let us consider as an example the case of SOCK scopes. Their behaviors can be characterized by Property 1 below.

Property 1. *A scope may either succeed and thus install its compensation handler, or fail by raising a (unique) fault. Furthermore, if it succeeds, it will never throw faults, and if it raises a fault it will never install its compensation.*

Such a property clearly describes the intuition about scope outcomes, and in [6] it is proved to hold for each SOCK process.

Other sample interesting properties of this kind valid for SOCK follows.

Property 2. *A request-response that terminates its execution always sends back an answer, either a successful one or an error notification.*

Property 3. *When a fault is triggered, there is no handler update that is ready to be installed but has not been installed yet.*

4.4 Minimality

When developing a calculus, one has to look for simplicity and minimality, avoiding for instance redundant or overlapping primitives. This makes the calculus more understandable and simplifies and shortens the proofs. In fact, some of the most successful calculi in the literature such as CCS [17] and π-calculus [18], are the most simple and compact way for modeling the desired features, interaction for CCS and mobility for π-calculus.

SOCK is different w.r.t. those calculi, since it is nearer to current technologies (e.g., it is the only calculus featuring request-response), and thus more complex than other calculi. However, each of its error handling primitives has a well-defined and non-overlapping role. Take for instance the three kinds of handlers. They take care of orthogonal features: internal faults for fault handlers, external faults for termination handlers and undoing of complete activities for compensation handlers.

Also, SOCK provides a unified way to deal with installation of fault and termination handlers (and, indirectly, of compensation handlers), and this dynamic installation is (probably, since this has not been proved for SOCK yet) needed to ensure the expressive power of the language. If SOCK would only allow to add pieces of code in parallel to existing handlers, as happens in dcπ [24], then it would not be minimal, since it has been shown in [11] that such a mechanism can be defined as a macro by exploiting the other constructs.

5 From SOCK to Jolie

As said before, when moving from a theoretical calculus like SOCK to a full-fledged language such as Jolie, new concerns have to be taken into account. Before analyzing those new concerns in detail, we give a general description of Jolie.

Jolie, Java Orchestration Language Interpreter Engine, is an open-source project released under the LGPL license. Its reference implementation is an interpreter written in Java. We refer to [9] for a detailed description of its features, concentrating here on the ones more useful for our discussion (some of them are also outlined in [19]). Jolie refines and extends SOCK so to offer to the programmer a powerful and intuitive environment, suitable to build both complex applications and single services.

One of the most prominent advantages of Jolie is the elegant separation between the program behavior and the underlying communication technologies. The same behavior can be used with different communication mediums (such as bluetooth, local memory, sockets, etc.) and protocols (such as HTTP, REST, SOAP, etc.) without being changed. This can be obtained since Jolie basic data structures are XML-like trees, which are automatically translated from and to XML files (or other suitable formats) for communication. Thus a Jolie variable is a tree, with java-style field access used to denote subtrees, and the array notation used to distinguish different subtrees with the same name. Thus, for instance, *var.subtree*[1] denotes the first subtree of variable *var* named *subtree*.

Jolie may also perform type checking on communicated data: each operation may specify types constraining the kind of data that may be sent or received, and checks are made at runtime to verify that those constraints are satisfied. Constraints are published in the service interface, so that remote partners may know the typing constraints to be satisfied for interacting with a service. We refer to [9] for more details on the type system.

We can now move to the description of how Jolie tries to satisfy the requirements in Section 3.

5.1 Usability

While features such as intuitiveness and expressiveness are fundamental for usability, other needs emerge. In particular, SOCK and its error handling mechanisms have been developed concentrating on issues such as synchronization of different entities and interaction between different error handling activities, but there has been scarce emphasis on data management. However, this aspect becomes fundamental in a real language, where applications managing possibly complex data structures are common. The major importance of data handling in Jolie w.r.t. SOCK has influenced also its mechanisms for error handling, as detailed below.

First, faults in Jolie include also a datum, which is normally used to carry information about the error itself (for instance, an error message, or a stack trace). Thus the throw primitive in Jolie has the syntax `throw(f,v)` where `f` is the fault and `v` a value. The handler can access the data with the special syntax *scopename.faultname*. The prefix *scopename* is needed to avoid interferences in case different scopes manage the same kind of fault concurrently (the scope of variables is the whole behavior). Note that such a modification in the throw primitive does not change the possible error handling policies (e.g., the properties described in Section 4.3 are unaffected), but makes the generation of error messages much easier.

Example 4. Consider the client in Example 3. In Jolie, one can exploit data attached to fault to simplify error handling. Now the server can specify the desired error message together with the fault, including for instance how much money is missing to perform the transfer[1]:

```
pay(varIn)(transId){
    if (acc[varIn.src] >= amount) {nullProcess} else
        {msg = "Missing "+string(amount-acc[varIn.src])+" euros";
        throw(f,msg)}
...
```

The client may use this information to present a more detailed error message to the user.

```
scope(q) {
install(f => print@user(q.f));
...
```

Another important point concerns data management inside handlers. Handlers in SOCK contain variables whose value is looked for when the handler is executed. However, sometimes one wants to use the values that variables had when the handler has been installed, to keep track of information concerning the computation that caused handler installation. This concern has been tackled in Jolie by adding a freeze prefix ^ to variables: if a variable `x` in a handler occurs

[1] The Jolie syntax should be rather intuitive, but we refer to [9] for details.

freezed, i.e. as `^x`, then its value is looked for and fetched at handler update time. Consider for instance Example 3. Assume that many invocations are performed inside a while loop. In case of later error one wants all the transactions to be canceled. Thus the correct handler update would be:

```
this => cH;undo@bank(^transId)
```

Without the freeze operator for `transId`, the value of `transId` in all the calls would be the last one.

As before, this is a mechanism that does not change the error handling properties, but that comes in handy when writing actual programs.

5.2 Robustness

Many calculi, and SOCK in particular, do not model network or node failures, while, in practice, these events may occur. Jolie has faced this problem by adding system faults. A system fault is a fault that is not generated by the throw primitive, but it is generated by the Jolie runtime system to notify the behavior of some problem. In particular, Jolie defines the system fault `IOException`, which is generated when an error occurs during communication. Such a fault can be managed in the same way of other faults, by defining and installing suitable handlers. For instance the Jolie code:

```
scope(q) {install( IOException => ... );
    pay@bank(...)(...)
```

allows to manage network failures in our payment example.

5.3 Compatibility

SOCK mechanisms have been devised to work in a close world, i.e. a world composed only by SOCK processes. However, Jolie applications are aimed at being executed over the net, interacting with other applications developed using different technologies and adhering to different standards.

In Jolie, this is mainly taken care by the communication module, that allows for specifying the protocol to be associated with each communication, and automatically translates messages to and from the desired format. However, a few aspects influence also error handling.

First, while Jolie guarantees remote error notifications inside the request-response pattern, most of the other technologies do not. However, even when interacting with other technologies, communication in Jolie is implemented by connection-oriented technologies such as tcp/ip, unix sockets or bluetooth connections. Thus the Jolie engine is notified when the connection is broken, and can react by generating system fault `IOException`. This is less informative w.r.t. the usual Jolie error notification, which describes exactly the kind of fault that happened on the remote client, but it is however enough to preserve Property 2 (or better its dual).

Another compatibility issue concerns typing. Assuming that each service correctly exposes its typing information, it would be enough to check types when messages are sent. However, when interacting with non Jolie applications there is no guarantee that they check types of communicated messages, thus Jolie services may receive ill-typed messages. For this reason, type checking is also performed on incoming messages. Type errors are managed in different ways according to where they happen. In one-way operations, a type mismatch of an outgoing message generates locally a system fault `TypeMismatch`. Instead, incoming messages that do not respect typing are discarded. The management is similar for request-responses, but, in case of type mismatch in receptions, the sender is also notified with a `TypeMismatch` fault, thus ensuring the preservation of the properties of the request-response pattern.

5.4 Property Preservation

As we have seen in the previous sections, Jolie is an extension and a refinement of SOCK. Also, some of the assumptions that are used to prove SOCK properties do not always hold for Jolie programs in a real environment. Thus, proving that a property of SOCK programs, such as one of those in Section 4.3, holds also for Jolie applications is non trivial.

We discuss now a few of the reasons that make this happen, analyzing their effect on a few sample properties.

Low level errors: SOCK, and theoretical models in general, rely on some basic assumptions ensuring the correct behavior of the system itself. Thus global failures due for instance to end of memory, to system crashes or to programming errors in the Jolie implementation are not considered. It is clear that these kinds of errors break most of the interesting properties, thus one has to assume that these events do not occur. One can exploit formal methods to ensure that these assumptions are satisfied, but this requires dedicated techniques whose description goes far beyond the aim of this paper. For instance, end of memory can be checked and avoided by a suitable resource analysis, system crashes superseded via techniques for reliability such as the use of redundant engines, and errors in the Jolie implementation avoided by using certified compilers and correctness proofs.

Jolie added features: as discussed above, Jolie includes features that are not available in SOCK, such as data in faults. Other additional features not related to error handling are described in [9]. Those features are normally related to aspects which are abstracted away in models, thus they do not affect global properties such as the ones in Section 4.3 (this has however to be checked for each property and each extension). However, because of this, not all Jolie programs are correct SOCK processes, thus it becomes much more difficult to prove properties of specific programs. To this end one has to find a SOCK process which is equivalent to the Jolie one, trying to implement Jolie additional features as macros. When this is not possible, one has to extend the theory to match the practice. For instance, a typed theory of

SOCK is not yet available, but it is on our research agenda. This will allow to prove properties of Jolie type system.

Assumptions on the environment: we refer here to the fact that network failures are not modeled in SOCK, and that interaction with non Jolie programs may raise new issues, as described in Section 5.3. In these cases, one may think to extended models taking care of this, but, mainly for interaction with non Jolie programs, it becomes quite difficult because of the huge variety in their behaviors. Thus, the simplest approach is to analyze their impact on each property, as outlined in Section 5.3, and introduce in Jolie mechanisms to deal with these problems in a uniform way w.r.t. similar issues in SOCK programs. An example of this is the introduction of system faults, which can be managed similarly to normal faults, and can enjoy (most of) their properties. Clearly, local properties such as Property 1 are largely unaffected by these issues, while properties concerning communication such as Property 2 are less robust.

6 Conclusion and Future Works

In this paper we have discussed the main concerns that should be kept into account when designing error handling mechanisms for service-oriented computing. We have considered both the design of a theoretical calculus and of a full-fledged language. We have considered the language Jolie and the underlying calculus SOCK as an example.

Concerning future work, the relations between formal models and practically relevant languages for service-oriented computing are still largely unexplored. Even in the case of SOCK/Jolie, which have been developed in a strongly connected way, many mismatches exist. Theory should be developed so to match interesting aspects of Jolie applications such as the type system, or network failures. For other differences instead, analysis should be carried out so to better understand the effect that they have on formal properties. However, Jolie is continuously evolving to face new programming challenges, thus making it a moving target. For instance, timeouts are an important aspect in practice, to break deadlocks, and work for introducing them in Jolie is ongoing.

Acknowledgments. We thank Gianluigi Zavattaro for his useful comments.

References

1. Banti, F., Lapadula, A., Pugliese, R., Tiezzi, F.: Specification and analysis of SOC systems using COWS: A finance case study. In: Proc. of WWV 2008. ENTCS, vol. 235, pp. 71–105. Elsevier, Amsterdam (2009)
2. Bruni, R., et al.: Comparing two approaches to compensable flow composition. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005. LNCS, vol. 3653, pp. 383–397. Springer, Heidelberg (2005)

3. Bruni, R., Melgratti, H., Montanari, U.: Theoretical foundations for compensations in flow composition languages. In: Proc. of POPL 2005, pp. 209–220. ACM Press, New York (2005)
4. Busi, N., Gorrieri, R., Guidi, C., Lucchi, R., Zavattaro, G.: SOCK: a calculus for service oriented computing. In: Dan, A., Lamersdorf, W. (eds.) ICSOC 2006. LNCS, vol. 4294, pp. 327–338. Springer, Heidelberg (2006)
5. Butler, M.J., Ferreira, C.: An operational semantics for StAC, a language for modelling long-running business transactions. In: De Nicola, R., Ferrari, G.-L., Meredith, G. (eds.) COORDINATION 2004. LNCS, vol. 2949, pp. 87–104. Springer, Heidelberg (2004)
6. Guidi, C., Lanese, I., Montesi, F., Zavattaro, G.: On the interplay between fault handling and request-response service invocations. In: Proc. of ACSD 2008, pp. 190–199. IEEE Computer Society Press, Los Alamitos (2008)
7. Guidi, C., Lanese, I., Montesi, F., Zavattaro, G.: Dynamic error handling in service oriented applications. Fundamenta Informaticae 95(1), 73–102 (2009)
8. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Englewood Cliffs (1985)
9. Jolie website, http://www.jolie-lang.org/
10. Lanese, I.: Static vs dynamic sagas. In: Proc. of ICE 2010 (to appear, 2010)
11. Lanese, I., Vaz, C., Ferreira, C.: On the expressive power of primitives for compensation handling. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 366–386. Springer, Heidelberg (2010)
12. Lanese, I., Zavattaro, G.: Programming Sagas in SOCK. In: Proc. of SEFM 2009, pp. 189–198. IEEE Computer Society Press, Los Alamitos (2009)
13. Laneve, C., Zavattaro, G.: Foundations of web transactions. In: Sassone, V. (ed.) FOSSACS 2005. LNCS, vol. 3441, pp. 282–298. Springer, Heidelberg (2005)
14. Lapadula, A., Pugliese, R., Tiezzi, F.: A formal account of WS-BPEL. In: Lea, D., Zavattaro, G. (eds.) COORDINATION 2008. LNCS, vol. 5052, pp. 199–215. Springer, Heidelberg (2008)
15. Lohmann, N.: A feature-complete petri net semantics for WS-BPEL 2.0. In: Dumas, M., Heckel, R. (eds.) WS-FM 2007. LNCS, vol. 4937, pp. 77–91. Springer, Heidelberg (2008)
16. Lucchi, R., Mazzara, M.: A pi-calculus based semantics for WS-BPEL. J. Log. Algebr. Program. 70(1), 96–118 (2007)
17. Milner, R.: A Calculus of Communicating Systems. LNCS, vol. 92. Springer, Heidelberg (1980)
18. Milner, R., Parrow, J., Walker, J.: A calculus of mobile processes, I and II. Information and Computation 100(1), 1–40, 41–77 (1992)
19. Montesi, F., Guidi, C., Lanese, I., Zavattaro, G.: Dynamic fault handling mechanisms for service-oriented applications. In: Proc. of ECOWS 2008, pp. 225–234. IEEE Computer Society Press, Los Alamitos (2008)
20. Montesi, F., Guidi, C., Zavattaro, G.: Composing services with JOLIE. In: Proc. of ECOWS 2007, pp. 13–22. IEEE Computer Society Press, Los Alamitos (2007)
21. Oasis: Web Services Business Process Execution Language Version 2.0 (2007), http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html
22. Ouyang, C., Verbeek, E., van der Aalst, W.M.P., Breutel, S., Dumas, M., ter Hofstede, A.H.M.: Formal semantics and analysis of control flow in WS-BPEL. Sci. Comput. Program. 67(2-3), 162–198 (2007)

23. Sensoria Project. Public web site, http://sensoria.fast.de/
24. Vaz, C., Ferreira, C., Ravara, A.: Dynamic recovering of long running transactions. In: Kaklamanis, C., Nielson, F. (eds.) TGC 2008. LNCS, vol. 5474, pp. 201–215. Springer, Heidelberg (2009)
25. W3C: Web services description language (wsdl) version 2.0 part 0: Primer (2007), http://www.w3.org/TR/wsdl20-primer/
26. Wirsing, M., et al.: Semantic-based development of service-oriented systems. In: Najm, E., Pradat-Peyre, J.-F., Donzeau-Gouge, V.V. (eds.) FORTE 2006. LNCS, vol. 4229, pp. 24–45. Springer, Heidelberg (2006)

Modeling and Reasoning about Service Behaviors and Their Compositions

Aida Čaušević, Cristina Seceleanu, and Paul Pettersson

Mälardalen Real-Time Research Centre (MRTC), Mälardalen University, Västerås, Sweden
{aida.delic,cristina.seceleanu,paul.pettersson}@mdh.se

Abstract. Service-oriented systems have recently emerged as context-independent component-based systems. Unlike components, services can be created, invoked, composed, and destroyed at run-time. Consequently, all services need a way of advertising their capabilities to the entities that will use them, and service-oriented modeling should cater for various kinds of service composition. In this paper, we show how services can be formally described by the resource-aware timed behavioral language REMES, which we extend with service-specific information, such as type, capacity, time-to-serve, etc., as well as boolean constraints on inputs, and output guarantees. Assuming a Hoare-triple model of service correctness, we show how to check it by using the strongest postcondition semantics. To provide means for connecting REMES services, we propose a hierarchical language for service composition, which allows for verifying the latter's correctness. The approach is applied on an abstracted version of an intelligent shuttle system.

1 Introduction

Service-oriented systems (SOS) assume *services* as their basic functional units, with capabilities of being published, invoked, composed and destroyed at runtime. Services are loosely coupled and enjoy a higher level of independence from implementation specific attributes than components do.

An important problem is to ensure the *quality-of-service* (QoS) that can be expected when deciding which service to select out of a number of available services delivering similar functionality. Some of the existing SOS standards support formal analysis [3, 12, 14, 15] to ensure QoS, but usually it is not straightforward to work out the exact formal analysis model.

In order to fully understand the ways in which services evolve and impact on QoS attributes, a *service behavioral description* is required [6]. Such behavior is assumed to be internal to the service, and hidden from the user. It should include the representation of a service functionality, enabled actions, resource annotations, and possible interactions with other services.

To meet the above demands, in this paper, concretely in Section 3, we extend the existing resource-aware, timed hierarchical language REMES [19], recalled in Section 2, to become fit for service behavioral modeling. In REMES, a service is modeled by an atomic or composite *mode*, which we enrich with attributes such as service type, capacity, time-to-serve etc., pre- and postconditions, which are exposed at the mode's

T. Margaria and B. Steffen (Eds.): ISoLA 2010, Part II, LNCS 6416, pp. 82–96, 2010.

interface. Still in Section 3, we introduce a synchronization mechanism for REMES modes, which enables modeling and verification of synchronized services.

By exploiting the pre-, postcondition annotations, we show how to describe the service behavior in Dijkstra's guarded command language [8], and how to check the service correctness by employing Dijkstra's and Scholten's strongest postcondition semantics [9].

Since services can be composed at run-time, analyzing the correctness of a service in isolation does not suffice. To exemplify, let us consider a service that is composed of several navigation services, out of which some return the route length in miles, whereas others in kilometers. If the developer has omitted to introduce a service that converts length from one metric to the other, it is desirable to uncover such an error right away, by formally checking the correctness of the actual service composition, at run-time.

To address the dynamic aspects of services, in Section 4, we propose a hierarchical language for dynamic service composition (HDCL) that allows creating new services, via binary operators, as well as adding and/or deleting services from lists. In the same section, we also give the semantics of sequential, parallel, and parallel with synchronization service composition, respectively. Next, we apply the approach on an abstracted version of an intelligent shuttle system, for which we show the use of REMES language to model the system and apply HDCL language to check the correctness of service compositions. In Section 6, we compare to some of the relevant related work, before concluding the paper in Section 7.

2 Preliminaries

2.1 REMES Modeling Language

The REsource Model for Embedded Systems REMES [19] is intended as a meaningful basis for modeling and analysis of resource-constrained behavior of embedded systems. REMES provides means for modeling of both continuous (i.e., power) and discrete resources (i.e., memory access to external devices). REMES is a state-machine behavioral language that supports hierarchical modeling, continuous time, and a notion of explicit entry and exit points, making it fit for component-based system modeling.

To enable formal analysis, REMES models can be transformed into timed automata (TA) [1], or priced timed automata (PTA) [2], depending on the analysis type.

The internal component behavior in REMES is given in terms of modes that can be either *atomic* (do not contain submode(s)), or *composite* (contain submode(s)). The data transfer between modes is done through the *data interface*, while the control is passed via the *control interface* (i.e., entry and exit points). REMES assumes *local* or *global* variables that can be of types boolean, natural, integer, array, or clock (continuous variable evolving at rate 1). Each (sub)mode can be annotated with the corresponding continuous resource usage, if any, modeled by the first derivative of the real-valued variables that denote resources, and which evolve at positive integer rates.

The control flow is given by the set of directed lines (i.e., *edges*) that connect the control points of (sub)modes. Modes may also be annotated with *invariants*, which bound

from above the current mode's delay/execution time. For a more thorough description of the REMES model, we refer the reader to [19].

The REMES language benefits from a set of tools[1] for modeling, simulation and transformation into PTA, which could assist the designer during system development.

2.2 Guarded Command Language

The Guarded Command Language (GCL) was introduced and defined by Dijkstra for predicate transformers semantics [8]. The basic element of the language is the guarded command, a statement list prefixed by a boolean expression, which can be executed only when the boolean expression is initially true.

The syntax of the GCL is given in Backus-Naur Form (BNF) extended with braces "{..}", where the braces mean: "followed by zero or more instances of the enclosed".

$$
\begin{array}{ll}
< \text{guarded command} > & ::= < \text{guard} > \; - > < \text{guarded list} > \\
< \text{guard} > & ::= < \text{boolean expression} > \\
< \text{guarded list} > & ::= < \text{statement} > \{; < \text{statement} >\} \\
< \text{guarded command set} > & ::= < \text{guarded command} > \{ [\!] \; < \text{guarded command} >\} \\
< \text{alternative construct} > & ::= \textbf{if} \; < \text{guarded command set} > \; \textbf{fi} \\
< \text{statement} > & ::= < \text{alternative construct} > \mid \text{"other statements"} \\
< \text{repetitive construct} > & ::= \textbf{do} < \text{guarded command set} > \textbf{od}
\end{array}
$$

The semicolons in the guarded list denote that whenever the guarded list is selected for execution, its statements will be executed successively in the order from the left to the right. A guarded command is not a statement but a component of a guarded command set from which statements can be constructed. The separator " [] " is used for mutual separation of guarded commands in guarded command set.

The alternative construct is written using special bracket pair: "**if ... fi**". The program aborts if none of the guards is true, otherwise an arbitrary guarded list with a true guard will be executed. Similarly, the repetitive construct "do ... od" means that the program runs as long as one of the guards is true, and terminates if none of the guards is true.

Semantics and Correctness of Guarded Commands. Let us assume the Hoare triple, $\{p\}\ S\ \{q\}$, where p, q are predicates, denoting the partial correctness of guarded command S with respect to precondition p and postcondition q. Introduced by Dijkstra and Sholten [9], the *strongest postcondition predicate transformer* (a function that maps predicates to predicates), denoted by sp.S.p, holds in those final states for which there exists a computation controlled by S, which belongs to class "initially p". Proving the Hoare triple, that is, the correctness of a guarded command, reduces to showing that $(\mathsf{sp.S.p} \Rightarrow q)$ holds. The strongest postcondition rules for the assignment statement, for sequential composition, and for the non-deterministic conditional are as follows:

$$\mathsf{sp.}(x := e).p(x) \equiv x = e \wedge (\exists x \cdot p(x)) \tag{1}$$

$$\mathsf{sp.}(S_1; S_2).p \equiv \mathsf{sp.}S_2.(\mathsf{sp.}S_1.p), \forall p \tag{2}$$

$$\mathsf{sp.}(\mathsf{if}\ g_1 \rightarrow S_1\ [\!]\ \ldots\ [\!]\ g_n \rightarrow S_n\ \mathsf{fi}).p \equiv \mathsf{sp.}S_1.(g_1 \wedge p) \vee \ldots \vee \mathsf{sp.}S_n.(g_n \wedge p), \forall p \tag{3}$$

[1] The REMES tool-chain is available at http://www.fer.hr/dices/remes-ide

3 Behavioral Modeling of Services in REMES

In REMES, a service is represented by a mode (be it atomic or composite). The service may have a special Init entry point, visited when the service first executes, and where all variables are initialized. In order for a service to be published and later discovered, a list of attributes should be exposed at the interface of a REMES mode/service (see Fig. 1).

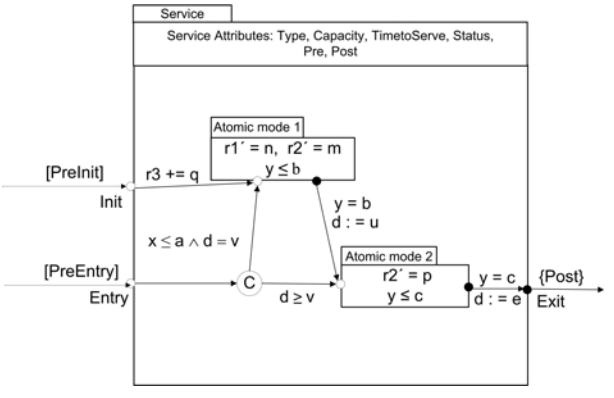


Fig. 1. A service modeled in REMES

The attributes depicted in Fig. 1 have the following meaning:

- service type - specifies whether the given service is a web service (i.e., weather report), a database service (i.e., ATM services), a network service, etc.;
- service capacity - specifies the service's maximum ability to handle a given number of messages per time unit (i.e., the maximum service frequency)($\in$ N);
- time-to-serve - specifies the worst-case time needed for a service to respond and serve a given request ($\in$ N);
- service status - describes the current service status (that is, passive (not invoked), idle, active);
- service precondition - is a predicate ($Pre : \sum \rightarrow Bool$, $Pre \equiv (PreInit \vee PreEntry)$) that conditions the start of service execution, and must be true at the time a REMES service is invoked. In this expression $\sum$ is the polymorphic type of the state that includes both local and global variables, and predicates PreInit, PreEntry are the initial, and the entry precondition of the service, respectively;
- service postcondition - is a predicate (Post) that must hold at the end of a REMES service execution.

The attributes are used to discover Service; they are specified by an interested party and, based on the specification, the service is either retrieved or not.

The formal specification of a service, modeled as the composite mode of Fig. 1, is the Hoare triple $\{p\}$Service$\{q\}$, where Service is described in terms of the guarded command language, and the mode's precondition p, and postcondition (requirement) q are as follows:

$$
\begin{aligned}
& p \\
\equiv & \\
& y \leq c \wedge c > b \wedge (d = 0 \vee v \leq d \leq e) \wedge r1 = r2 = r3 = 0 \wedge (h = 0 \vee h = 1) \\
& q \\
\equiv & \\
& y \leq c \wedge d \leq e \wedge (\forall i, 1 \leq i \leq 3 \cdot ri \leq val_i)
\end{aligned}
$$

where val_i are the given upper bounds on each resource usage, respectively.

Below, we give the GCL description of the REMES composite mode Service:

Service ::=

IF

$\neg u1 \wedge h = 0 \wedge y \leq b$ $\rightarrow r3 := r3 + q;$ $sm :=$ Atomic mode 1; $u1 :=$ true; Update(now)	Init $\rightarrow$ Atomic mode 1
$[\!] \neg u2 \wedge h = 1 \wedge (x \leq a \wedge d = v) \wedge y \leq b$ $\rightarrow sm :=$ Atomic mode 2; $u2 :=$ true; Update(now)	Entry $\rightarrow$ Atomic mode 1
$[\!] (\neg u3 \wedge (h = 1 \wedge d \geq v) \vee d = u) \wedge y \leq c$ $\rightarrow sm :=$ Atomic mode 2; $u3 :=$ true; Update(now)	(Entry or Atomic mode 1) $\rightarrow$ Atomic mode 2
$[\!] \neg u4 \wedge sm =$ Atomic mode 1 $\wedge y \leq b$ $\rightarrow r1(t) := r1(now) + n * (t - now);$ $r2(t) := r2(now) + m * (t - now);$ $\{y \leq b\}; u4 :=$ true; Update(now) $[\!] \neg u5 \wedge sm =$ Atomic mode 1 $\wedge y = b$ $\rightarrow d := u; u5 :=$ true; Update(now)	Delay in Atomic mode 1
$[\!] \neg u6 \wedge sm =$ Atomic mode 2 $\wedge y \leq c$ $\rightarrow r2(t) := r2(now) + p * (t - now);$ $\{y \leq c\}; u6 :=$ true; Update(now)	Delay in Atomic mode 2
$[\!] \neg u7 \wedge sm =$ Atomic mode 2 $\wedge y = c$ $\rightarrow d := e;$ $h := 1; u7 :=$ true; Update(now); $u1, \ldots, u7 :=$ false	Atomic mode 2 $\rightarrow$ Exit

FI

(4)

In the GCL description (4), the variables x, y are clocks, h is the history variable that is used to decide where to enter the composite mode, sm is the variable ranging over submodes, and $r1 : \mathsf{Real}_+ \rightarrow \mathsf{T}_1$, $r2 : \mathsf{Real}_+ \rightarrow \mathsf{T}_2$ are the continuous resources of the model, defined as functions over the non-negative reals that are used as the time domain. In addition, u_i are local variables used for preventing executing the same action more than once, at the same time point. These variables are reset each time the mode Service exits. Similar to the approach taken for action system models [18], the variable now shows the current time, and it is explicitly updated by statement Update(now).

The assertions $\{y \leq b\}, \{y \leq c\}$ model the invariants (Inv) of Atomic mode 1, and Atomic mode 2, respectively.

We define Update(now) as follows:

$$Update(now) \triangleq now := \mathsf{next}.now$$

The submodes can be urgent (no delays are allowed), or non-urgent (where delays can happen, until an invariant Inv is violated); also, guarded actions can annotate edges connecting the entry points of the composite mode with submodes, via some conditional connector (denoted by encircled C in Figure 1). Given these, and assuming that gg is the disjunction of the action guards of the edges leaving a mode (or a conditional connector), and that Inv is the invariant of the respective mode, next is defined by:

$$\mathsf{next}.t \triangleq \begin{cases} min\{t' \geq t \mid \neg Inv \vee gg\}, & \text{if exists } t' \geq t \text{ such that } \neg Inv \vee gg \\ +\infty, & \text{otherwise.} \end{cases}$$

If a mode is urgent, or the guards correspond to a conditional connector, then $I \equiv$ false, so the next moment of time is identical to the current one, no delay being possible.

The mode Service, modeled by (4), can be iterated for as long as needed, so the complete specification is: $\mathsf{status}_{\mathsf{Service}} := \mathsf{active}$; (DO g $\rightarrow$ Service $[\!]$ $\neg$g $\rightarrow$ $\mathsf{status}_{\mathsf{Service}} :=$ idle OD). According to rule (3), the strongest postcondition of the conditional statement is:

$$\begin{array}{l} sp.Service.p \\ \equiv \\ sp.(r3 := r3 + q; sm := \text{Atomic mode 1}; \mathsf{Update(now)}).(h = 0 \wedge y \leq b \wedge p) \\ \vee \\ \ldots \\ \vee \\ sp.(d := e; h := 1; \mathsf{Update(now)}).(sm = \text{Atomic mode 2} \wedge y = c \wedge p) \end{array}$$

Assuming that sp.{y $\leq$ c}.p $\equiv y \leq c \wedge (\exists$x $\cdot$ p(x)), the above sp can be mechanically computed by successively applying rules (1) - (2). The correctness proofs reduce to checking whether each of the strongest postconditions of the above disjunction implies the requirement q, given earlier.

In service-oriented systems, there is often the case that services need to synchronize their behaviors. In order to model synchronized behavior, we introduce a special kind of REMES mode, given in Figure 2, which can act either as an AND mode, or as an OR mode, depending on whether the services need to be entered simultaneously, or not.

The composite mode of Figure 2 contains as sub-modes the services that need to be synchronized. For AND modes, both Service a, and Service b are entered at the same time (through their entry point). This means that the edges marked with (*) do not have guards. In case of OR modes, one or all constituent services are entered, so the edges marked with (*) are annotated with guards. If some of the edges need to be taken at the same time in both services, the communication between Service a and b is realized via synchronization variables, chan (in x), (out x), which are used similarly to the PTA channels $x?, x!$, respectively. Depending on the required synchronization type

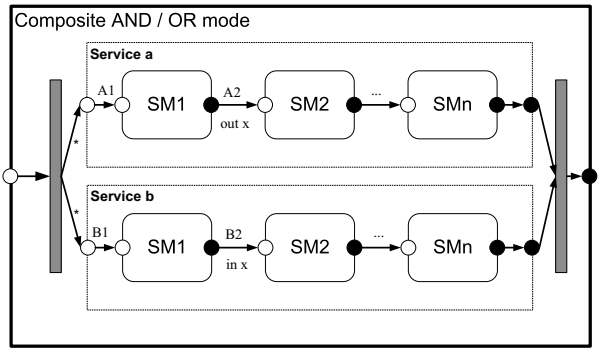


Fig. 2. AND/OR REMES mode

and starting time of the constituent services' execution, AND modes, but also OR can be employed when either "**and**" synchronization (both services should finish execution at the same time), or "**max**" synchronization (the composite mode finishes when the slowest service finishes) is required.

In Figure 2, Service a, Service b need to synchronize actions $A2$, $B2$. This can be done by decorating the respective edges with channel variables $out\ x$ for $A2$, and $in\ x$ for $B2$, meaning that the respective edges are taken simultaneously in both services, $A2$ writing variables that $B2$ is reading. The same applies if the services need to "and"-synchronize at the end of their execution. The exit edge of each service, respectively, needs to be annotated with chan variables.

The GCL representation of such synchronization requires strengthening the guards of the respective synchronized commands of the conditional statement, as follows: $(in\ x) \wedge g_{A2} \rightarrow S_{A2}$, $(out\ x) \wedge g_{B2} \rightarrow S_{B2}$, where S_{A2}, S_{B2} are the action bodies of $A2$, $B2$, respectively. The actions can then be executed in a sequence, with the one writing variables, first. The "**max**" synchronization can be represented in GCL by using a virtual selector (variable sel) [18], which selects for execution the modes $SM1, \ldots, SMn$, according to the control flow, marks them as executed after they finish their execution, and keeps the time values of now in a copy variable now_c, which is updated only after the slowest service finishes executing; the latter translates in exiting the composite AND, or OR mode.

4 Hierarchical Language for Dynamic Service Composition: Syntax and Semantics

Service compositions may lead to complex systems of concurrently executing services. An important aspect of such systems is the correctness of their temporal and resource-wise behavior. In the following, we propose an extension to the REMES language, which provides means to define and support creation, deletion, and composition of fine-grained or coarser-grained services, applicable to different domains. We also investigate a formal way of ensuring the correctness of the composition, based on the strongest postcondition semantics of services.

Let us assume that a service, whose behavior is described by a REMES mode, is denoted by $service\_name_i$, $i \in [1..n]$; then, a service list, denoted by *s_list*, is defined as follows:

$$\mathsf{s\_list} ::= [\mathsf{service\_name}_1, ..., \mathsf{service\_name}_n]$$

In order to support run-time service manipulation, we define a set of REMES interface operations, by a pre- postcondition specification. We denote by Σ the set of service states, respectively, that is, the current collection of variable values.

- **Create service:** *create service_name*
  $[pre] : service\_name = \mathsf{NULL}$
  $\mathsf{create} : Type \times \ N \times N \times \ \ ''passive'' \times (\Sigma \rightarrow bool) \times (\Sigma \rightarrow bool) \rightarrow service\_name$
  $\{post\} : service\_name \neq \mathsf{NULL}$

- **Delete service:** *del service_name*
  $[pre] : service\_name \neq \mathsf{NULL}$
  $\mathsf{del} : service\_name \rightarrow \mathsf{NULL}$
  $\{post\} : service\_name = \mathsf{NULL}$

- **Create service list:** *create s_list*
  $[pre] : s\_list = \mathsf{NULL}$
  $\mathsf{create\_list} : s\_list \rightarrow s\_list,\ s\_list = List()$
  $\{post\} : s\_list \neq \mathsf{NULL}$

- **Delete service list:** *del s_list*
  $[pre] : s\_list \neq \mathsf{NULL}$
  $\mathsf{del\_list} : s\_list \rightarrow \mathsf{NULL}$
  $\{post\} : s\_list = \mathsf{NULL}$

- **Add service to a list:** *add service_name, s_list*
  $[pre] : service\_name \notin s\_list$
  $\mathsf{add} : s\_list \rightarrow s\_list$
  $\{post\} : service\_name \in s\_list$

- **Remove service from the list:** *del service_name, s_list*
  $[pre] : service\_name \in s\_list$
  $\mathsf{del} : s\_list \rightarrow s\_list$
  $\{post\} : service\_name \notin s\_list$

- **Replace service in the list:** $replace\ service\_name_1, service\_name_2$
  $[pre] : s\_list(p) = service\_name_1$
  $\mathsf{replace} : s\_list \rightarrow s\_list$
  $\{post\} : s\_list(p) = service\_name_2$

– **Insert service at a specific position:** $insert\ service\_name_i, s\_list$
$[pre] : s\_list(p) \neq service\_name_i$
$\mathsf{add} : s\_list \rightarrow s\_list$
$\{post\} : s\_list(p) = service\_name_i$

Note that a new service list can be created by using the constructor *List()*, which holds list values of any type. Such a constructor enables the creation of both empty list and also list with some initial value (*s_list* = List : String([“Shuttle1″, “Shuttle2″])). Also, adding a service to a list means, in this context, appending that service, that is, adding it at the end of the list. Replacing a service with another one, and inserting a service at a specific position requires the use of parameter p, which specifies the position at which the service is replaced or inserted.

Most often, services can be perceived as independent and distributed functional units, which can be composed to form new services. The systems that result out of service composition have to be designed to fulfill requirements that often evolve continuously and therefore require adaptation of the existing solutions.

Alongside the above operations, we also define a hierarchical language that supports dynamic REMES service composition (HDCL), that is, facilitates modeling of nested sequential, parallel or synchronized services:

$$\mathsf{DCL} \ ::= \ (\mathsf{s\_list}, \mathsf{PROTOCOL}, \mathsf{REQ})$$

$$\mathsf{HDCL} ::= \ (((\mathsf{DCL}^+, \mathsf{PROTOCOL}, \mathsf{REQ})^+, \mathsf{PROTOCOL}, \mathsf{REQ})^+, \ \dots)$$

The formula above allows a theoretically infinite degree of nesting. The positive closure operator is used to express that one or more DCLs (Dynamic Composition Languages) are needed to form an HDCL. The PROTOCOL defines the way services are composed, that is, the type of binding between services, as follows:

$$\mathsf{PROTOCOL} ::= \mathsf{unary\_operator}\ service\_name \mid service_m\ \mathsf{binary\_operator}\ service_n$$

The requirement REQ is a predicate ($\Sigma \rightarrow \mathsf{Bool}$) that can include both functional and extra-functional properties/constraints of the composition. It identifies the required attribute constraints, capability, characteristics, or quality of a system, such that it exhibits the value and utility requested by the user. The above unary and binary operators are defined as follows:

$$\mathsf{Unary\_operator} ::= \ \mathsf{exec-first}$$
$$\mathsf{Binary\_operator} ::= \ ; \ | \ \| \ | \ \|_{SYNC-and} \ | \ \|_{SYNC-or}$$

Let us assume that two services s_1, s_2 are invoked at some point in time, and their instances are placed in the service list s_list. Also, we assume that $s_i.Pre_i$ is the strongest postcondition of s_i, $i \in 1, 2$, w.r.t. precondition Pre_i. Then, the semantics of the unary and binary protocol operators, as well as the correctness conditions for such compositions are given as follows.

– **Exec-first** (specifies which service should be initially executed in a composition) - below we formalize the fact that s_1 should execute first, and only when it finishes and establishes its postcondition, service s_2 can become active:

$$status_{s_1} = active \ \wedge \ status_{s_2} \ = \ idle \ \wedge \ Post_{s_1} \Rightarrow (status_{s_2} \ = \ active)$$

If we assume n services $s_1, \ldots, s_n$ of a list, executing s_1 first is defined as:

$$\mathsf{Exec-first\ s_1} \triangleq \mathsf{s_1} \ [\!] \ \neg \mathsf{g_{s_1}} \rightarrow (\mathsf{s_2\ Binary\_operator} \ldots \mathsf{Binary\_operator\ s_n})$$

This means that, even if any other service (or service composition) could be executed, it will be executed only after s_1 has finished execution.

– **Sequential composition** - two services are executed in a sequence, uninterrupted, e.g., $s_1; s_2$. The correctness condition of $s_1; s_2$ is:

$$(sp.s_2.(sp.s_1.Pre_{s_1}) \Rightarrow Post_{s_2}) \wedge (Post_{s_2} \Rightarrow REQ)$$

– **Parallel composition's** $(s_1 \parallel s_2)$ correctness condition is:

$$(sp.s_1.Pre_{s_1} \ \vee \ sp.s_2.Pre_{s_2}) \ \Rightarrow \ REQ$$

– **Parallel composition with synchronization** - we denote by S-AND the set of services belonging to an AND mode, which need to synchronize their executions in the end. Then, the "and" synchronization of such services is defined as:

$$\begin{aligned}(s_1 \parallel_{SYNC-and} s_2) \triangleq (s_1, s_2 \in \mathsf{S-AND} \Rightarrow ((\forall now \cdot status_{s_1} = status_{s_2} = active) \\ \wedge (start_{s_1} + TimetoServe_{s_1} = start_{s_2} + TimetoServe_{s_2})))\end{aligned}$$

The correctness condition of the "and-AND" synchronization is given below:

$$(sp.(s_1 \parallel_{SYNC-and} s_2).Pre_{AND} \Rightarrow (Post_{s_1} \wedge Post_{s_2})) \wedge (Post_{s_1} \wedge Post_{s_2} \Rightarrow REQ)$$

A service user, but also a developer of services, might need to replace a service with one with possibly better QoS. It follows that one needs to be able to check whether the new service still delivers the original functions, while having better time-to-serve or resource-usage qualities. Verifying such a property reduces to proving refinement of services. Either weakening the service precondition or strengthening its postcondition qualifies as service refinement.

5 Example: An Autonomous Shuttle System

In this section, we consider an example, previously modeled and analyzed in the PTA framework, in our recent work [5].

We consider a simplified version of a three train system that provides transportation service to three different locations. The system has been developed at University of Paderborn within the Railcab project [11]. While in our previous work [5], we have focused on resource effective design, in the current example, we extract parts of the behavior described by Giese and Klein [11], to show how services are created, invoked, composed, and idled, by using the REMES extended interface and behavioral language.

Each of the trains has a well-defined path to follow, as shown in Fig. 3. During the transport, the shuttles might meet at point B, in which they are forced to create a convoy. In order to enter the convoy, they have to respect given speed and acceleration limits,

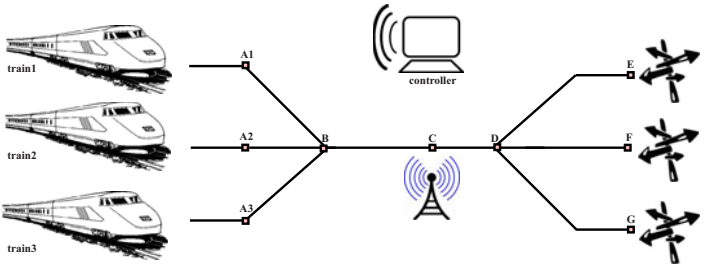


Fig. 3. An example overview

measured in points A1, A2, and A3, respectively, otherwise they may stop to let others that fulfil the given requirements join the convoy. After a convoy is formed and has left, those that were stopped are allowed to continue their journey to previously assigned destination, if the sensor at point C, in Fig. 3, has sent the "safe to continue" signal.

After the destination point is being reached, a shuttle is free to turn to the idle state, and wait for new orders. The system described above is equipped with one central controller, as shown in Fig. 3, which decides when and which shuttle to invoke, based on the service descriptions for each shuttle, respectively.

5.1 Modeling the Shuttle System in REMES

We model the behavior of the Autonomous shuttle system services as modes in the extended REMES. The composite mode of Shuttle1 is depicted in Fig. 4, yet, due to lack of space, we do not show here the constituent submodes, but we briefly explain them instead (for more details we refer reader to [5]).

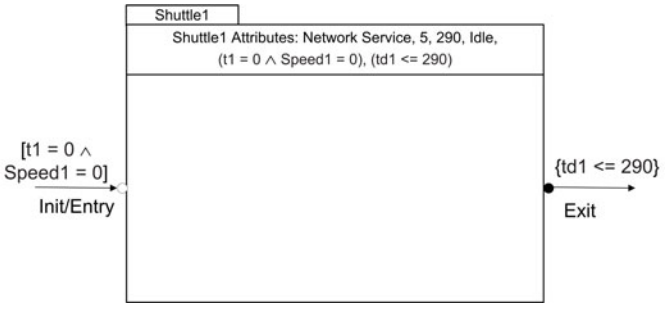


Fig. 4. The model of Shuttle1 given as a REMES service

The mode consists of the *atomic* modes (i.e., Acceleration1, STOP, and Destination). They communicate data between each other using the global variables: speed_i, status_i, t_i, and StatusConvoy. The control interfaces are used to expose mode attributes relevant for mode discovery. Shuttle1 and Shuttle3 have the same behavior, while Shuttle2 is an

older shuttle than the other two, and therefore it requires more time to start, accelerate, slow down.

5.2 Applying the Hierarchical Language

Below, we illustrate the use of our proposed hierarchical language for modeling service composition, as depicted in Table 1, on the example described in Section 5.

Table 1. An illustration of the REMES language

00 **declare** Shuttle1 ::=< network service,	18 **create** Shuttle1
01 5,	19 **create** Shuttle2
02 290,	20 **create** Shuttle3
03 passive,	21 **create** list_Convoy
04 $(t1 = 0 \wedge speed = 0)$,	22 **add** Shuttle1 list_Convoy
05 $(t1 \leq 290) >$	23 **add** Shuttle2 list_Convoy
06 **declare** Shuttle2 ::=< network service,	24 **DCL_**Convoy ::= (list_Convoy, ; , $t \leq 300$)
07 7,	25 **HDCL_**Convoy ::= ((DCL_Convoy, Shuttle3), $\|\|$, $t \leq 300$)
08 300,	26 **check**(sp.(Shuttle1; Shuttle2).$(t1 = 0 \wedge speed = 0) \wedge (t = t1 \vee t = t2)) \Rightarrow (t \leq 300)$
09 passive,	27 **check**(sp.Shuttle3.$(t3 = 0 \wedge speed = 0) \wedge (t = t3)) \Rightarrow (t \leq 300)$
10 $(t2 = 0 \wedge speed = 0)$,	28 **del** HDCL_Convoy
11 $(t2 \leq 300) >$	
12 **declare** Shuttle3 ::=< network service,	
13 5,	
14 290,	
15 passive,	
16 $(t3 = 0 \wedge speed = 0)$,	
17 $(t3 \leq 290) >$	

The needed services are introduced through the declarative part (lines 00-17 in Table 1). A service declaration contains the service name, type, status, TimeToServe, precondition and postcondition. The corresponding requirement is matched against such attribute information, when choosing a service. After the selection, the instances of the selected services are created (lines 18-20 in Table 1), and added to the service list using the *add* command (lines 22-23 in Table 1). Finally, the chosen services are composed by DCL. The list of services, employed protocol (type of service binding), and DCL requirements are given as parameters. Moreover, the language provides means to compose the existing DCLs with other services, through HDCL, as shown in line 25 of Table 1. If not anymore needed, the composition can be deleted.

The advantage of this language is that, after each composition, one can check whether the given requirement is satisfied, by forward analysis, e.g., by calculating the strongest postcondition of a given composition w.r.t. a given precondition. Due to space limitation, we show only the final computed result. Below, $p1 \equiv (t1 = 0 \wedge speed = 0)$.

By applying the sp rules (1) - (3), we get the following:

$$
\begin{aligned}
sp.(Shuttle1; Shuttle2).p1 &\equiv sp.Shuttle2.(sp.Shuttle1.p1)\\
sp.Shuttle1.p1 &\equiv (t1 = 0 \wedge 245 \leq t \leq 266 \wedge speed1 = 0 \wedge\\
&\quad \wedge mode = Destination \wedge r1 = 0 \wedge\\
&\quad \wedge status1 = end1 = idle)\\
sp.Shuttle2.(sp.Shuttle1.p1) &\equiv (t1 = t2 = 0 \wedge 264 \leq t \leq 285 \wedge\\
&\quad \wedge speed1 = speed2 = 0 \wedge r1 = r2 = 0 \wedge\\
&\quad \wedge status1 = end1 = idle \wedge status2 = end2 = idle)
\end{aligned}
$$

One can notice that the requirement $\mathsf{REQ} \equiv (\mathsf{t} \leq 300)$ is implied by the calculated strongest postcondition to which the condition $(\mathsf{t} = \mathsf{t1} \vee \mathsf{t} = \mathsf{t2})$ is added. This is actually what the command check should return as a main proof obligation, provided that the method is implemented in the REMES tool-chain.

For the second check, we have Shuttle3 composed in parallel with the sequential composition of the other two shuttles, with $\mathsf{p3} \equiv (\mathsf{t3} = 0 \wedge \mathsf{speed} = 0)$. Then, according to the composition semantics of section 4, proving the correctness of the $(\mathsf{Shuttle3} \,||(\mathsf{Shuttle1}; \mathsf{Shuttle2}))$ composition reduces to showing that:

$$(\mathsf{sp.Shuttle3.p3} \vee \mathsf{sp.Shuttle2.(sp.Shuttle1.p1)}) \Rightarrow \mathsf{REQ}$$

As already shown, the sequential composition of the first two shuttles implies the requirement. What is left to be proven is that the strongest postcondition of Shuttle3, w.r.t p3, also implies the requirement. The calculated strongest postcondition of the latter is as follows:

$$\begin{aligned} \mathsf{sp.Shuttle3.p3} \equiv\ & (\mathsf{t3} = 0 \wedge 245 \leq \mathsf{t} \leq 266 \wedge \mathsf{speed3} = 0 \wedge \\ & \wedge\ \mathsf{mode} = \mathsf{Destination} \wedge \mathsf{r3} = 0 \wedge \mathsf{status3} = \mathsf{end3} = \mathsf{idle}) \end{aligned}$$

It is easy to check that the requirement REQ is actually implied by $\mathsf{sp.Shuttle3.p} \wedge \mathsf{t} = \mathsf{t3}$. This concludes our service composition correctness verification.

6 Discussion and Related Work

Based on the level of details that are provided through the behavioral description, all approaches related to services and SOS can be in principle divided into three groups.

Code-level behavioral description approaches are mostly based on XML language (e.g., BPEL, WS-CDL). BPEL [3] is an orchestration language whose behavioral description includes a sequence of project activities, correlation of messages and process instances, and recovery behavior in case of failures and exceptional conditions. Approaches like BPEL are useful when services are intended to serve a particular model or when the access to the service implementation exists. The drawback of such approaches is the lack of formal analysis support, which forces the designer/developer to master not only the specification and modeling processes, but also the techniques for translating models into a suitable analysis environment.

When compared to the above group, BPMN [14] can be seen as a higher-level language. It relies on a process-oriented approach, and supports a graphical representation to be used by both designers and analysts. The lack of a formal behavioral description does not provide means for detailed analysis, as the one supported by REMES.

The third group includes approaches with formal background. Rychlý describes the service behavior as a component-based system for dynamic architectures [16]. The specification of services, their behavior, and hierarchical composition are formalized within the π-calculus. Similar to our approach, this work emphasizes the behavior in terms of interfaces, (sub)service communication, and bindings, while we can also cater for service descriptions including timing and resource annotations [5]. Foster et al. present an approach for modeling and analysis of web service compositions [10].

The approach takes BPEL4WS service specification and translates it into Finite State Processes (FSP), and Labeled Transition Systems (LTS), for analysis purposes.

A comprehensive survey on several approaches that are accommodating service composition, and are checking the correctness of compositions [3, 12, 15, 14] is given by Beek et al. [20]. Regarding service modeling, all these approaches are solid; however, w.r.t. service compositions and their correctness checking [7, 13, 17] (usually by employing formal methods), such approaches show limited capabilities to automatically support these processes. In comparison, as shown in this paper, compositions of REMES models can be mechanically reasoned about (although, as for now, we still miss the interface correctness tool support), or can be automatically translated to timed- [1] or priced timed automata [2], and analyzed with UPPAAL , or UPPAAL CORA tools[1], for functional but also extra-functional behaviors (timing and resource-wise behaviors).

7 Conclusions

In this paper, we have presented an approach for formal service description by extending the resource-aware timed behavioral language REMES. Attributes such as type, time-to-serve, capacity, etc., together with precondition and postcondition are added to REMES to enable service discovery, as well as service interaction. Even if the original semantics of REMES [19] is given in terms of Priced Timed Automata (PTA), here, we have chosen to use Hoare triples and the strongest postcondition semantics to prove service correctness, motivated by the lack of decidability results for computing simulations relations on PTA. We have also proposed a hierarchical language for service composition, which allows for the verification of, e.g., service composition correctness. The approach is demonstrated on a simplified version of an intelligent shuttle system.

As future work, we plan to look into the algorithmic computation of strongest postconditions of priced timed automata, by building on preliminary results of Badban et al. [4]. We also intend to extend the REMES tool-chain with a postcondition calculator.

References

[1] Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science 126(2), 183–235 (1994), citeseer.nj.nec.com/alur94theory.html

[2] Alur, R.: Optimal paths in weighted timed automata. In: Di Benedetto, M.D., Sangiovanni-Vincentelli, A.L. (eds.) HSCC 2001. LNCS, vol. 2034, pp. 49–62. Springer, Heidelberg (2001)

[3] Andrews, T., Curbera, F., Dholakia, H., Goland, Y., Klein, J., Leymann, F., Liu, K., Roller, D., Smith, D., Thatte, S., Trickovic, I., Weerawarana, S.: BPEL4WS, Business Process Execution Language for Web Services Version 1.1. IBM (2003)

[4] Badban, B., Leue, S., Smaus, J.-G.: Automated predicate abstraction for real-time models. EPTCS 10, 36 (2009), doi:10.4204/EPTCS.10.3

[5] Causevic, A., Seceleanu, C., Pettersson, P.: Formal reasoning of resource-aware services. Technical Report ISSN 1404-3041 ISRN MDH-MRTC-245/2010-1-SE, Mälardalen University (June 2010)

[1] For more information about the UPPAAL and UPPAAL CORA tool, visit the web page www.uppaal.org

[6] Causevic, A., Vulgarakis, A.: Towards a unified behavioral model for component-based and service-oriented systems. In: 2nd IEEE International Workshop on Component-Based Design of Resource-Constrained Systems (CORCS 2009). IEEE Computer Society Press, Los Alamitos (July 2009)
[7] Díaz, G., Pardo, J.J., Cambronero, M.E., Valero, V., Cuartero, F.: Automatic translation of ws-cdl choreographies to timed automata. In: Bravetti, M., Kloul, L., Zavattaro, G. (eds.) EPEW/WS-EM 2005. LNCS, vol. 3670, pp. 230–242. Springer, Heidelberg (2005)
[8] Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. ACM Commun. 18(8), 453–457 (1975)
[9] Dijkstra, E.W., Scholten, C.S.: Predicate calculus and program semantics. Springer, New York (1990)
[10] Foster, H., Emmerich, W., Kramer, J., Magee, J., Rosenblum, D., Uchitel, S.: Model checking service compositions under resource constraints. In: ESEC-FSE 2007: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, pp. 225–234. ACM, New York (2007)
[11] Giese, H., Klein, F.: Autonomous shuttle system case study. In: Leue, S., Systä, T.J. (eds.) Scenarios: Models, Transformations and Tools. LNCS, vol. 3466, pp. 90–94. Springer, Heidelberg (2003)
[12] Kavantzas, N., Burdett, D., Ritzinger, G., Fletcher, T., Lafon, Y., Barreto, C.: Web services choreography description language version 1.0. World Wide Web Consortium, Candidate Recommendation CR-ws-cdl-10-20051109 (November 2005)
[13] Narayanan, S., McIlraith, S.A.: Simulation, verification and automated composition of web services. In: WWW 2002: Proceedings of the 11th international conference on World Wide Web, pp. 77–88. ACM, New York (2002)
[14] Object Management Group (OMG): Business Process Modeling Notation (BPMN) version 1.1 (January 2008), http://www.omg.org/spec/BPMN/1.1/
[15] Roman, D., Keller, U., Lausen, H., de Bruijn, J., Lara, R., Stollberg, M., Polleres, A., Feier, C., Bussler, C., Fensel, D.: Web service modeling ontology. Applied Ontology 1(1), 77–106 (2005)
[16] Rychlý, M.: Behavioural modeling of services: from service-oriented architecture to component-based system. In: Software Engineering Techniques in Progress, pp. 13–27. Wroclaw University of Technology (2008)
[17] Salaün, G., Bordeaux, L., Schaerf, M.: Describing and reasoning on web services using process algebra. In: ICWS 2004: Proceedings of the IEEE International Conference on Web Services, p. 43. IEEE Computer Society Press, Washington (2004)
[18] Seceleanu, C.: A Methodology for Constructing Correct Reactive Systems. Ph.D. thesis, Turku Centre for Computer Science (TUCS) (December 2005)
[19] Seceleanu, C., Vulgarakis, A., Pettersson, P.: Remes: A resource model for embedded systems. In: Proc. of the 14th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2009). IEEE Computer Society, Los Alamitos (June 2009)
[20] Ter Beek, M.H., Bucchiarone, A., Gnesi, S.: Formal methods for service composition. Annals of Mathematics, Computing & Teleinformatics 1(5), 1–10 (2007), http://journals.teilar.gr/amct/

Design and Verification of Systems with Exogenous Coordination Using Vereofy

Christel Baier[1], Tobias Blechmann[1], Joachim Klein[1], Sascha Klüppelholz[1], and Wolfgang Leister[2,⋆]

[1] Faculty of Computer Science, Technische Universität Dresden, Germany
[2] Norsk Regnesentral (Norwegian Computing Center), Norway

Abstract. The feasibility of formal methods for the analysis of complex systems crucially depends on a modeling framework that supports compositional design, stepwise refinement and abstractions. An important feature is the clear separation of coordination and computation which permits to apply various verification techniques for the computation performed by components and interactions as well as dependencies between the components. We report here on a model-checking approach using the tool Vereofy that is based on an exogenous coordination model, where the components are represented by their behavioral interfaces. Vereofy supports the verification of the components and their communication structure. Our approach is illustrated by means of a case study with a sensor network where Vereofy has been used to establish several properties of the sensor nodes and their routing procedures.

1 Introduction

Component-based software engineering strives to divide complex systems into smaller logical components with well-defined interfaces. A variety of coordination models and languages have been introduced which support the separation between computations inside the components and the interactions between components, e.g., an aspect-oriented approach [6], a variant of the π-calculus with anonymous peer-to-peer communication [13], and formalisms that rely on the construction of component connectors, such as interaction systems [12,18] or the declarative channel-based language Reo [2]. Tool support for the verification of such systems relies on operational models that are powerful enough to describe both the coordination imposed by the connectors and the behavioral interfaces of the connected components and can serve as the structures for temporal logics and model-checking algorithms.

For this purpose, constraint automata, a special variant of labeled transition systems using constraints on the active I/O ports and transmitted data values, have been introduced [5]. Constraint automata are adequate to represent any kind of synchronous and asynchronous peer-to-peer communication, modeling exogenous and endogenous coordination mechanisms and have been used as a

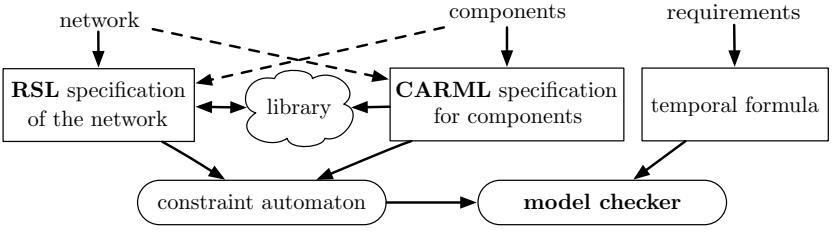
⋆ The authors are supported by the EU-project Credo and the DFG-project Syanco.

T. Margaria and B. Steffen (Eds.): ISoLA 2010, Part II, LNCS 6416, pp. 97–111, 2010.

compositional semantics for the exogenous coordination language Reo. In the exogenous setting the components are not aware of the context in which they are used. The interactions are handled by a network from outside the components.

Our toolkit Vereofy (`http://www.vereofy.de/`) allows hierarchical, compositional modeling of complex systems based on the constraint automata semantics and provides symbolic model-checking algorithms for linear-time and branching-time properties. It supports two modeling languages, CARML (Constraint Automata Reactive Module Language) as a guarded-command language for specifying component interfaces and behavior and RSL (Reo Scripting Language) for hierarchically building systems from components, connectors and channels. It supports complex, structured data types both as messages and state variables of the components, which facilitates the modeling of complex, data-dependent behavior. Features like template instantiation, scripting and conditional compilation ease the generation of variants and abstractions of models.

Vereofy supports the model checking of individual components and the composite system against properties specified in Linear Time Logic (LTL [22,4]), the branching-time logic BTSL [15] and the alternating-time logic ASL [16], tailored to the constraint automata setting. Vereofy also supports checking bisimulation equivalence. Vereofy varies from other model checkers such as [14,8,1] as the main focus is the verification of coordination aspects and the communication and interactions between components on the level of the behavioral interfaces. Vereofy uses a symbolic representation of the components and the composite system and offers verification algorithms that are applied on this symbolic representation which is based on binary decision diagrams (BDDs). For more details about the modeling languages and verification techniques of Vereofy, see [3,4].



In this paper we will use CARML and RSL for modeling and `LTL` and `BTSL` for analyzing a sensor network. The routing protocol implemented by each node of the network is an AODV-like protocol [21]. In contrast to [7,20,19] the goal of this paper is to illustrate the usefulness of Vereofy rather than to find errors in the protocol itself. As the behavior of the network nodes is highly data dependent, the analysis is a challenging task even when restricted to simple network structures with fixed topology [17].

Organization. Section 2 presents the definition of constraint automata and related notations. The syntax and semantics of the temporal logics `LTL` and `BTSL` are presented in Section 3. Section 4 illustrates the modeling and analysis of the case study before Section 5 concludes the paper.

2 Constraint Automata

Constraint automata (CA) provide a generic operational model to formalize the behavioral interfaces of the components, the network that coordinates the components (i.e., the glue code or connector), and the composite system consisting of the components and the glue code. Constraint automata are variants of labeled transition systems (LTS) where the labels of the transitions represent the (possibly data-dependent) I/O-operations of the components and the network. They support any kind of synchronous and asynchronous peer-to-peer communication.

The port names of a CA play the role of the I/O-ports of the components or the network. The states represent the local states of components and/or configurations of a connector. The transitions in a CA describe the potential one-step behavior. Fig. 1 shows a component with I/O-ports send, receive and failure together with its CA implementing an abstract communication protocol. The transition labels are constraints encoding which ports are active and what data is observed at the active ports when the transition fires. In the sequel, let Data be a finite nonempty data domain and $\mathcal{N}$ a finite, nonempty set of port names.

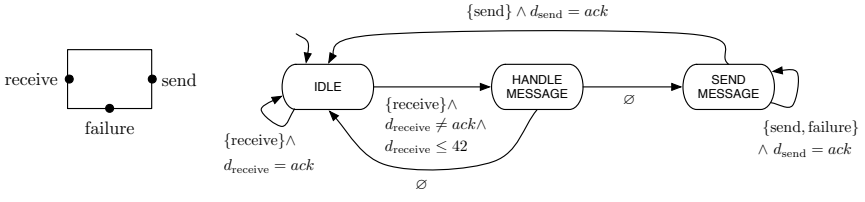


Fig. 1. Interface and CA for a component implementing a simple protocol

Concurrent I/O-operations (CIO). A *concurrent I/O-operation* consists of a (possibly empty) set of port names $N \subseteq \mathcal{N}$ together with data items for each $A \in N$ that are written or received at port A. When c is executed there is no data flow (i.e., no read or write operation) at the ports $A \in \mathcal{N} \setminus N$. Formally, a concurrent I/O-operation is a partial function assigning data values to the ports, i.e., a function $c : \mathcal{N} \to \mathsf{Data}$. We write $\mathsf{Ports}(c)$ for the set of ports $A \in \mathcal{N}$ such that $c(A) \in \mathsf{Data}$. $\mathsf{CIO}_{\mathcal{N}}$, or briefly CIO, denotes the set of all concurrent I/O-operations. CIO is finite, as $\mathcal{N}$ and Data are finite.

I/O-constraints (IOC). The transitions of a constraint automata are labeled with *I/O-constraints*. These are propositional formulas that stand for *sets* of concurrent I/O-operations. The I/O-constraints may impose conditions on the ports that may or may not be involved and on the data items written on or read from them. Formally, I/O-constraints (IOC) are Boolean combinations of atomic formulas A and $(d_{A_1}, \ldots, d_{A_k}) \in D$ where $A, A_1, \ldots, A_k \in \mathcal{N}$ and $D \subseteq \mathsf{Data}^k$. Each IOC g stands for a set $\mathsf{CIO}(g)$ of concurrent I/O-operations. For the atoms, we define $\mathsf{CIO}(A) \stackrel{\text{def}}{=} \{\, c \in \mathsf{CIO} : A \in \mathsf{Ports}(c) \,\}$ The I/O-constraints $(d_{A_1}, \ldots, d_{A_k}) \in D$ impose conditions for the written and read data items. That is, $\mathsf{CIO}((d_{A_1}, \ldots, d_{A_k}) \in D)$ agrees with the set

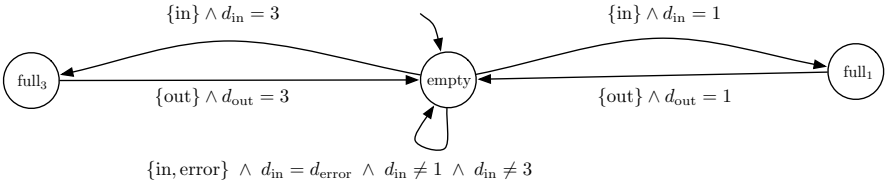


Fig. 2. Constraint automaton: filtering buffer

$$\big\{\, c \in \mathsf{CIO} \,:\, \{A_1,\ldots,A_k\} \subseteq \mathsf{Ports}(c), (c(A_1),\ldots,c(A_k)) \in D \,\big\}.$$

The semantics of the Boolean operators is the standard one, e.g., $\mathsf{CIO}(g_1 \wedge g_2) = \mathsf{CIO}(g_1) \cap \mathsf{CIO}(g_2)$. We often use simplified notations for the I/O-constraints of the form $(d_{A_1},\ldots,d_{A_k}) \in D$. E.g., the notation $d_A = d_B$ is a shorthand for $(d_A, d_B) \in \{(d_1,d_2) \in \mathsf{Data}^2 : d_1 = d_2\}$. The notation $\{A,B\}$ is used as a shorthand for any I/O-constraint g with $\mathsf{CIO}(g) = \{c \in \mathsf{CIO} : \mathsf{Ports}(c) = \{A,B\}\}$. We use the symbol tt for any valid I/O-constraint, i.e., $\mathsf{CIO}(tt) \stackrel{\text{def}}{=} \mathsf{CIO}$.

Our logical framework refers to two kinds of labels, the I/O-constraints referring to the I/O-operations in the network and atomic propositions referring to the states of constraint automata, which can be regarded as unary state predicates. For example, if the network (i.e., connector) contains a FIFO channel then there might be atomic propositions stating, e.g., that all buffer cells are empty or that the first buffer cell contains a value d in some set $D \subseteq \mathsf{Data}$.

Constraint automata [5]. A constraint automaton (CA) is a tuple $\mathcal{A} = \langle Q, \mathcal{N}, \longrightarrow, Q_0, \mathsf{AP}, L\rangle$ where Q is a finite and nonempty set of states, $\mathcal{N}$ a finite set of ports, $\longrightarrow$ is a subset of $Q \times \mathsf{IOC} \times Q$, called the transition relation of $\mathcal{A}$, $Q_0 \subseteq Q$ a nonempty set of initial states, AP a finite set of atomic propositions, and $L : Q \to 2^{\mathsf{AP}}$ a labeling function. The meaning of a transition $q \xrightarrow{g} p$ is that in configuration q, all concurrent I/O-operations c satisfying I/O-constraint g are enabled and state p is a possible successor state of q executing the CIO c.

Example 1 (CA for a filtering buffer). Fig. 2 depicts a constraint automaton $\mathcal{A}$ with three ports in, out and error and a finite integer data domain. The incoming messages are filtered, in the case of 1 and 3 stored in a buffer (represented by the "full$_v$" states for buffer value v) and can be subsequently read at the out port. In case the incoming message does not match the filter criterion, an error is signaled via the error port. □

To simplify the presentation of the paper, we describe here our logical approach under the assumption that there are no terminating behaviors, i.e., every state has at least one outgoing transition.

Executions, paths, I/O-streams. As in standard LTSs an *execution* in $\mathcal{A}$ is a finite or infinite sequence $\eta = q_0 \xrightarrow{c_1} q_1 \xrightarrow{c_2} \ldots$ built by consecutive transitions where $q_i \in Q$, $c_i \in \mathsf{CIO}$, and $q_i \xrightarrow{g_{i+1}} q_{i+1}$ for all $i \geq 0$ and some g_{i+1} such that $c_{i+1} \in \mathsf{CIO}(g_{i+1})$. As we focus on infinite behaviors we define a *path* of $\mathcal{A}$ to be an infinite execution and write $\mathsf{Paths}(q)$ to denote the set of all paths starting

in q. Let $\pi = q_0 \xrightarrow{c_1} q_1 \xrightarrow{c_2} \ldots$ be a path and $0 \leq n$. Then, $\pi{\downarrow}n$ denotes the prefix of path π with length n and $\pi{\uparrow}n$ the suffix starting at the n-th state. $\pi{\downarrow}n \stackrel{\text{def}}{=} q_0 \xrightarrow{c_1} \ldots \xrightarrow{c_n} q_n$ and $\pi{\uparrow}n \stackrel{\text{def}}{=} q_n \xrightarrow{c_{n+1}} q_{n+1} \xrightarrow{c_{n+2}} q_{n+2} \xrightarrow{c_{n+3}} \ldots$.

The notion of an *I/O-stream* for CA corresponds to action sequences in LTSs. The I/O-stream $\mathsf{ios}(\eta)$ of a finite execution η is the finite word over CIO that is obtained by taking the projection to the labels of the transitions. Formally, if $\eta = q_0 \xrightarrow{c_1} \ldots \xrightarrow{c_n} q_n$ is a finite execution then $\mathsf{ios}(\eta) \stackrel{\text{def}}{=} c_1 \ldots c_n \in \mathsf{CIO}^*$.

3 Specifying and Verifying Components and Connectors

CA yield a general framework for the behavior of a component, a connector or a composite system and serve as starting point for model checking. The model-checking problem asks whether a given property holds for the automaton. In our framework and the tool Vereofy, the properties can be specified by temporal formulas with classical modalities to formalize safety or liveness conditions, but also constraints on the observable data flow (I/O-streams). Vereofy supports model checking against linear-time, branching-time or alternating-time properties. In this section, we will provide a brief overview of the syntax and semantics of the logics for linear and branching-time properties used in the case study. As noted above, we will only consider infinite behavior. For a more detailed description, we refer to [4,15,16]. Throughout this section, we fix a finite set AP of atomic propositions for the states and a finite set of port names $\mathcal{N}$.

3.1 Linear-Time Properties

Vereofy supports model checking of linear-time properties specified using the logic LTL [22], augmented with an operator that allows to refer to the concurrent I/O-operations using I/O-constraints. It is appropriate to specify complex temporal conditions on paths and their I/O streams, such as Boolean combinations of reachability, repeated reachability or persistence conditions.

Syntax of LTL. The abstract syntax of LTL formulas over AP and $\mathcal{N}$ is defined by the following grammar.

$$\varphi \;::=\; \text{true} \;\mid\; a \;\mid\; \neg\varphi \;\mid\; \varphi_1 \wedge \varphi_2 \;\mid\; \langle\!\langle g \rangle\!\rangle \varphi \;\mid\; \varphi_1 \,\mathtt{U}\, \varphi_2$$

where $a \in \mathsf{AP}$ is an atomic proposition and $g \in \mathsf{IOC}$ is an I/O-constraint over $\mathcal{N}$.

Semantics of LTL. Let $\pi = q_0 \xrightarrow{c_1} q_1 \xrightarrow{c_2} \ldots$ be a path in a CA, with $q_i \in Q$ and $c_i \in \mathsf{CIO}$. Let φ be LTL formula over AP and $\mathcal{N}$. The satisfaction relation $\pi \models \varphi$ is defined as follows:

$$\begin{array}{lll}
\pi \models \text{true} & & \\
\pi \models a & \text{iff} & a \in L(q_0) \\
\pi \models \neg\varphi & \text{iff} & \pi \not\models \varphi \\
\pi \models \varphi_1 \wedge \varphi_2 & \text{iff} & \pi \models \varphi_1 \text{ and } \pi \models \varphi_2 \\
\pi \models \langle\!\langle g \rangle\!\rangle \varphi & \text{iff} & c_1 \in \mathsf{CIO}(g) \text{ and } \pi{\uparrow}1 \models \varphi \\
\pi \models \varphi_1 \,\mathtt{U}\, \varphi_2 & \text{iff} & \text{there exists } n \geq 0 \text{ such that } \pi{\uparrow}n \models \varphi_2 \text{ and} \\
 & & \pi{\uparrow}i \models \varphi_1 \text{ for all } 0 \leq i < n
\end{array}$$

Recall that $\pi\!\uparrow\! i$ is the suffix of π starting at the i-th state. As usual, we can derive the standard propositional operators ($\vee$, $\rightarrow$, $\leftrightarrow$, etc.). The $\langle\!\langle g \rangle\!\rangle \varphi$ operator is satisfied if the next I/O-operation satisfies the I/O-constraint g and the path suffix starting in the next state satisfies φ. For the common use of this operator with $\varphi = \text{true}$, we use the shorthand $\langle\!\langle g \rangle\!\rangle \stackrel{\text{def}}{=} \langle\!\langle g \rangle\!\rangle \,\text{true}$. The standard next step operator can then be derived by $\mathtt{X}\,\varphi \stackrel{\text{def}}{=} \langle\!\langle tt \rangle\!\rangle \varphi$. We can derive the standard LTL operators "eventually $\Diamond$" with $\Diamond\varphi \stackrel{\text{def}}{=} \text{true}\,\mathtt{U}\,\varphi$ and "always $\Box$", with $\Box\varphi \stackrel{\text{def}}{=} \neg\Diamond\neg\varphi$. As example formulas for the CA of Fig. 2, consider

$$\varphi_1 = \Box\,(\langle\!\langle \text{in} \wedge d_{\text{in}} = 1 \rangle\!\rangle \rightarrow X\,\text{“full}_1\text{”}) \qquad \varphi_2 = \Box\Diamond\,\text{“empty”}$$

Formula φ_1 asserts that after a 1 has been received, the buffer is full and contains data value 1, while φ_2 asserts that the "empty" state is always reached again. Given a CA $\mathcal{A}$, the model-checking problem asks whether all paths in $\mathcal{A}$ starting in an initial state satisfy the formula φ:

$$\mathcal{A} \models \varphi \iff^{\text{def}} \pi \models \varphi \text{ for all } \pi \in \mathsf{Paths}(q_0) \text{ and all } q_0 \in Q_0$$

To model check an LTL formula against a CA, Vereofy uses the standard automata-theoretic approach to LTL model checking [23], i.e., the negated property is translated to a non-deterministic Büchi automaton. The product of the CA and the Büchi automaton is then analyzed using cycle-check algorithms to determine the absence of paths in the CA that violate the formula. As Vereofy uses a symbolic representation of the CA, the cycle-check algorithms are also performed on the symbolic representation [11].

3.2 Branching-Time Properties

Branching-time stream logic BTSL [15] allows reasoning about the branching structure of the states by means of branching-time temporal formulas stating, e.g., the existence of a path where a certain path property holds (as in CTL [9]). The special path modality $\langle\!\langle \alpha \rangle\!\rangle$ and its dual $[\![\alpha]\!]$ allow to reason about the data streams observable at the I/O-ports of components and the network by means of a regular stream expression. The atoms for an regular stream expression α will be the I/O-constraint as they were used by our linear-time logic introduced in the previous section.

Stream expressions. To impose conditions on the data flow at the ports of an automaton, we will use a symbolic representation for sets of I/O-streams by means of regular I/O-stream expressions, briefly called stream expressions. The abstract syntax of stream expressions over $\mathcal{N}$ is given by the following grammar:

$$\alpha \quad ::= \quad g \;\mid\; \alpha^* \;\mid\; \alpha_1;\alpha_2 \;\mid\; \alpha_1 \cup \alpha_2$$

where g ranges over all I/O-constraints over $\mathcal{N}$. The formal definition of the regular languages $\mathsf{IOS}(\alpha) \subseteq \mathsf{CIO}^*$ is defined by structural induction. $\mathsf{IOS}(g)$ is the set consisting of the I/O-streams of length 1 given by g, i.e., $\mathsf{IOS}(g) = \mathsf{CIO}(g)$. Union ($\cup$), Kleene star ($*$) and concatenation ($;$) have their standard meaning.

Syntax of BTSL. The node-set $\mathcal{N}$ and set AP of atomic propositions will serve as signature for BTSL-formulas. State-formulas (denoted by capital Greek letters

Φ, Ψ) and path-formulas (denoted by small Greek letters φ, ψ) of BTSL are built by the following grammar:

$$\Phi ::= \text{true} \mid a \mid \Phi_1 \wedge \Phi_2 \mid \neg\Phi \mid \exists\varphi \mid \forall\varphi$$
$$\varphi ::= \langle\!\langle\alpha\rangle\!\rangle\Phi \mid \Phi_1 \,\mathtt{U}\, \Phi_2$$

where $a \in \mathsf{AP}$ and α is a stream expression. The operator $\exists$ corresponds to an existential quantification while the $\forall$ corresponds to a universal quantification over all paths. The dual path modality $[\![\alpha]\!]$ can be derived by

$$\exists[\![\alpha]\!]\Phi \stackrel{\text{def}}{=} \neg\forall\langle\!\langle\alpha\rangle\!\rangle\neg\Phi \quad \text{and} \quad \forall[\![\alpha]\!]\Phi \stackrel{\text{def}}{=} \neg\exists\langle\!\langle\alpha\rangle\!\rangle\neg\Phi.$$

The standard CTL operators for "next step" and "eventually" are obtained by $\mathtt{X}\,\Phi \stackrel{\text{def}}{=} \langle\!\langle tt\rangle\!\rangle\Phi$ and $\Diamond\Phi \stackrel{\text{def}}{=} (\text{true}\,\mathtt{U}\,\Phi)$. The definition of the always operator in BTSL is as follows: $\exists\Box\Phi \stackrel{\text{def}}{=} \neg\forall(\text{true}\,\mathtt{U}\,\neg\Phi)$ and $\forall\Box\Phi \stackrel{\text{def}}{=} \neg\exists(\text{true}\,\mathtt{U}\,\neg\Phi)$. Other Boolean connectives, like disjunction or implication are obtained in the obvious way.

Semantics of BTSL. Let $\mathcal{A}$ be a CA and $\pi = q_0 \xrightarrow{c_1} q_1 \xrightarrow{c_2} \ldots$ be a path in $\mathcal{A}$. The satisfaction relation $\models$ for BTSL state formulas is defined by structural induction as shown below:

$$\begin{array}{lll}
q \models \text{true} & & \\
q \models a & \text{iff} & a \in L(q) \\
q \models \Phi_1 \wedge \Phi_2 & \text{iff} & q \models \Phi_1 \text{ and } q \models \Phi_2 \\
q \models \neg\Phi & \text{iff} & q \not\models \Phi \\
q \models \exists\varphi & \text{iff} & \text{there is a path } \pi \in \mathsf{Paths}(q) \text{ such that } \pi \models \varphi \\
q \models \forall\varphi & \text{iff} & \pi \models \varphi \text{ for all paths } \pi \in \mathsf{Paths}(q)
\end{array}$$

The satisfaction relation $\models$ for BTSL path-formulas and the path π in $\mathcal{A}$ is defined as follows:

$$\begin{array}{lll}
\pi \models \langle\!\langle\alpha\rangle\!\rangle\Phi & \text{iff} & \text{there exists } n \in \mathbb{N} \text{ such that } q_n \models \Phi \\
 & & \text{and } \mathsf{ios}(\pi\downarrow n) \in \mathsf{IOS}(\alpha) \\
\pi \models \Phi_1 \,\mathtt{U}\, \Phi_2 & \text{iff} & \text{there exists } n \in \mathbb{N} \text{ such that } q_n \models \Phi_2 \\
 & & \text{and } q_i \models \Phi_1 \text{ for all } 0 \leq i < n
\end{array}$$

A BTSL state formula Φ is said to hold for a CA $\mathcal{A}$, written $\mathcal{A} \models \Phi$, if $q_0 \models \phi$ for all initial states q_0 of $\mathcal{A}$. To illustrate some BTSL example formulas we reconsider the CA from Fig. 2.

$$\Phi_1 = \exists\langle\!\langle tt^*; \{\text{in}\}\rangle\!\rangle \text{ "buffer is full"}$$
$$\Phi_2 = \forall[\![(tt^*; \{\text{in}\})^+]\!] \text{ "buffer is full"}$$
$$\Phi_3 = \exists[\![(tt^*; \{\text{in}\})^+]\!] \text{ "buffer is empty"}$$

The first formula asserts the existence of an execution where the last action is a successful receipt of a data value, i.e., the error port is not active, and where the resulting state corresponds to a full buffer. Clearly, $\mathcal{A} \models \Phi_1$. All paths in $\mathcal{A}$ enjoy the property that whenever there is a successful receipt the buffer will be full in the subsequent state, and thus $\mathcal{A} \models \Phi_2$. The third formula asserts the existence of a path where the buffer is empty whenever the last step was a successful receipt. As this property holds for the path where no successful receipt occurs, i.e., always $\{\text{in}, \text{error}\}$, we have $\mathcal{A} \models \Phi_3$.

The model-checking problem for BTSL asks whether, for a given CA $\mathcal{A}$ and BTSL state formula Φ_0, all initial states q_0 of $\mathcal{A}$ satisfy Φ_0. The main procedure for BTSL model checking follows the standard approach for CTL-like branching-time logics [9,10] and recursively calculates the satisfaction sets $\mathsf{Sat}(\Psi) \stackrel{\text{def}}{=} \{q \in Q : q \models \Psi\}$ for all subformulas Ψ of Φ_0. The treatment of the propositional operators is obvious and the satisfaction sets $\mathsf{Sat}(\exists(\Phi_1 \,\mathtt{U}\, \Phi_2))$ and $\mathsf{Sat}(\forall(\Phi_1 \,\mathtt{U}\, \Phi_2))$ can be computed by means of the standard procedures for least fixed points of monotonic operators. To compute the satisfaction sets of $\exists\langle\!\langle\alpha\rangle\!\rangle\Phi$ and $\forall\langle\!\langle\alpha\rangle\!\rangle\Phi$, we follow an automata-theoretic approach which resembles the standard automata-based LTL model-checking procedure and relies on a representation of α by means of a finite automaton $\mathcal{Z}$ and model checking BTSL state formulas of the form $\exists\Diamond\Phi$ and $\forall\Diamond\Phi$, respectively, in the product of $\mathcal{A}$ and $\mathcal{Z}$.

4 Case Study: A Biomedical Sensor Network

We illustrate the Vereofy approach of modeling and model checking by means of a biomedical sensor network consisting of multiple, autonomous sensors. Each sensor is equipped with a radio unit, which is used to form a mobile ad-hoc network. Using this network, the acquired sensor data is transmitted to a designated node, where the data is evaluated and monitored. Such a sensor network provides a very complex scenario with a vast amount of aspects, e.g., quality of service concerns, energy consumption due to the battery powered nature of the sensor nodes, resilience to failures, etc. Our focus in this case study lies on the level of the routing protocol used to establish routes in a distributed way and react to changes in the network topology. As such, we abstract away higher-level aspects like the actual data generated by the sensors, as well as lower-level aspects like details of the physical medium used for communication. E.g., we assume that there is a mechanism on the lower communication layer that guarantees that a message is either correctly received by a neighboring node or an error notification is presented to the sensor node. We also abstract from the real-time aspects of the routing protocol. In the sequel, we provide an overview of our model and present some results of our analysis.

The routing protocol relies on a recative AODV-like (Ad-hoc On-demand Distance Vector [21]) routing policy where routes are determined when needed. We give a short introduction to the aspects of the routing protocol relevant at the level of abstraction used in the model.

The routing protocol relies on three types of messages, which neighboring nodes in the network can send each other: Routing requests (RREQ) to determine whether one of the neighbors has a route to a specific destination node, routing replies (RREP) to provide answers to requests and routing error messages (RERR) to signal that a route has become invalid, e.g., due to movement out of range of a neighbor that is considered the next hop of a route. Each node maintains a local routing table, including information about the next hop for each destination, as well as the number of hops to the destination. In addition, each node maintains an (always increasing) sequence number that is used in the

routing protocol to determine freshness of information. When a node in the network needs to discover a route to some destination, it broadcasts a route request to its neighbors. Upon receipt, each node checks whether it has a route to that destination or is the destination itself. In both cases, it responds with a unicast route reply message, which is subsequently forwarded to the originator of the request. When it does not have a route to the destination, it broadcasts a route request to its neighbors. Upon receipt of a route reply, the sensor nodes check whether they need to update their routing tables, by taking into account the length of the proposed route as well as the freshness as determined by the sequence number of the destination included in the message. When a node detects a link failure to a neighbor along a route, it invalidates the route in the routing table and notifies its neighbors via an error message that it should no longer be considered as a hop along a route to that destination. The other nodes can then determine whether they need to notify their neighbors in turn.

4.1 The Model

In our model of a biomedical sensor network, which is inspired by the a real world application in a hospital, multiple sensor nodes attached to patients are connected by a wireless ad-hoc network. There is one designated node that acts as the *sink node*. It is the destination of all the data packets and never produces data of its own.

Data domain. The nodes communicate by sending messages to their neighbors. Vereofy supports various data types for the messages sent through the network, including finite integer ranges, enumerations and structured data types. In the model, the messages are encoded as a struct (see Fig. 3), yielding the data domain Data for the CA. The first field determines the type of the message, either one of the routing messages or a message carrying a data payload. Each message has two fields for the identifier of the current sender and the identifier of the neighbor node for whom the message is intended, with a special value signifying broadcast to all available neighbor nodes. In addition, the message carries the ID of the target destination node and of the original originator of the message. A hop counter provides the value of hops the message has traveled from the originator. If this number exceeds an upper limit the message is dropped to prevent forwarding loops. The routing messages additionally use sequence numbers to infer knowledge about the freshness of the routing information. We use a parameter constant to specify the number of bits used for the representation of sequence numbers. Incrementing and comparing sequence numbers is carried out using the sequence number arithmetic as specified in the AODV standard.

Sensor nodes. The sensor nodes are modeled as CARML modules, with each node being assigned a globally unique identifier. Each sensor node has three ports ("receive", "send" and "failure") for receiving and sending messages. The "failure" port is used by the medium connecting the nodes to signal failure during sending, i.e., if the "failure" port is simultaneously active with the "send" port, the neighbor the message was directed to cannot receive, either due to a network topology change, power loss or other error conditions. Each sensor

```
TYPE message_type_t  = enum {RREQ,RREP,RERR,DATA};
TYPE address_t       = int(0,nodes);
TYPE id_t            = int(0,nodes-1);

TYPE Data = struct {
 message_type_t      msg_type;

 // the address of the next hop and current sender
 // to = nodes signifies broadcast to all neighbors
 address_t           to;
 id_t                from;

 // the ultimate destination and the original sender
 id_t                dest_id;
 id_t                orig_id;

 hop_counter_t       hop_count;

 // sequence number information
 seq_no_t            orig_seq_no;
 seq_no_t            dest_seq_no;
 Bool                dest_seq_no_unknown;

 // the payload for DATA messages
 data_type_t         the_data;
};
```

Fig. 3. Data domain of the sensor network model

node maintains a local routing table, with entries for the other nodes, storing the last known sequence number, hop distance, next neighbor on the route, etc. The sensor nodes maintain their routing table by update procedures which are triggered whenever a message is received. The update procedures are modeled by sub-modules of the CARML module for the sensor node. The model contains three different types of updates. The *neighbor update* updates the routing information for the sender of a message, while the *originator update* updates the routing information of the originator of the message. The third update is the *update on error* which is executed on link failures.

Additionally to the routing table, the sensor node has buffers for temporarily storing routing and data messages. The sensor part of the sensor node generates new data to be forwarded to the sink, while the radio part of the sensor node receives, sends, processes and forwards messages, both routing and data messages. The main behavioral complexity arises out of the correct processing of the routing messages and correct updating of the routing table. Fig. 4 provides a scheme of the major control locations used for message handling.

The support of structured data types (struct, arrays) for the local state variables provided by CARML, as well as the possibility to specify complex guard conditions on the states and messages allows the convenient modeling of the behavior of a sensor node. The support of parametrized templates for modules allows the flexible instantiation of the different sensor node instances with unique IDs and adaption for network scenarios of different sizes. In addition, support for

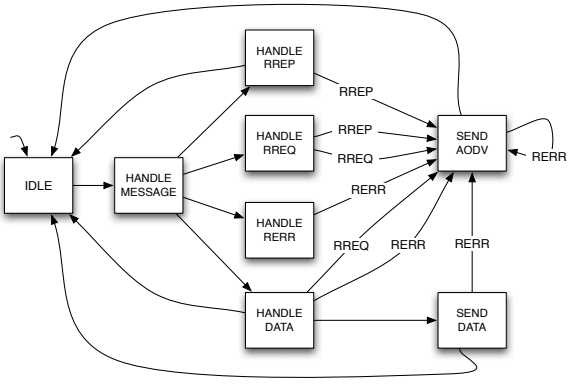


Fig. 4. Schema of the main control locations for a sensor nodes

conditional compilation in Vereofy facilitates the generation of model variants, e.g., to generate a variant of the sensor node abstracting from error handling.

Omnicast medium. The medium represents the topology in which the sensor nodes are arranged. It is responsible for transferring messages to the correct neighbor node in the unicast case and to all neighbors in the broadcast case. Each sensor node is connected to the medium via its send, receive and failure port. The correct routing and failure detection is achieved by a combination of filter channels, i.e., channels that use a filter condition on the message data, synchronously passing a message if the value satisfies the condition and blocking the message when the condition is not satisfied. Fig. 5 shows a medium for five sensor nodes, detailing the arrangement of the filter channels attached to the send port of the sensor node with ID 1 with nodes 2 and 4 as neighbors. Unicast messages to one of these nodes are passed and broadcast messages are replicated to the receive ports of both neighbors. Messages sent by sensor node 1 to the other, unreachable nodes are sent back to the failure port of sensor node 1 to signal an unsuccessful transmission.

The connections for the other nodes are realized in a similar way. As the medium is realized as a component connector generated from an RSL script, it is very easy to modify and generate variants, e.g., by introducing FIFO buffers in front of the receive ports to provide buffered communication or by introducing lossy channels to model lossy communication, all without changing the modules realizing the sensor nodes.

4.2 Analysis of the Model

In this section we show selected `LTL` and `BTSL` formulas to illustrate the expressiveness of our logics. The main focus in the validation of our model will be to analyze the maintenance procedure for the routing information held by an individual sensor node. Additionally we present formulas addressing properties

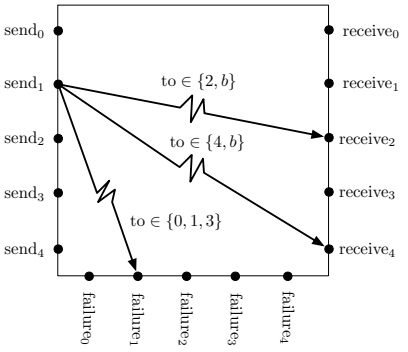


Fig. 5. Omnicast medium for five sensor nodes

of the composite system and summarize the experimental results of the model checking for the selected properties.

Verification of a single sensor node. The first part of the verification inspects the behavior of a single sensor node in isolation, i.e., a complete nondeterministic environment. In the sequel, let ownID be the ID of the inspected sensor node and 0 be the ID of the sink node. The heartbeat of our routing protocol are the three update procedures as they were described briefly in the previous section. We use different `LTL` formulas of the form $\Box(\Phi_1 \rightarrow \Diamond\Phi_2)$ and verified that all updates (1) are called when expected, (2) do terminate, and (3) lead to a route entry if they contain fresher information. We present here formulas for the *neighbor update*. E.g., the formula below holds if each RREQ and RREP message from sensor node with ID i will eventually lead to a route entry for the node with ID i.

$$\Box\,(\langle\!\langle d_{\text{receive}} \in D_i \rangle\!\rangle \rightarrow \Diamond \text{ “have route to node } i\text{”}) \text{ for all } i \in \text{ID}, i \neq \text{ownID}$$

Here, D_i is the set of RREQ or RREP messages from neighbor node with ID i.

The second focus of our verification is the interaction of a sensor node when certain messages arrive. For this we address the different cases of the protocol and provide `BTSL` and `LTL` formulas to verify the responsiveness of a sensor node. E.g., we are interested in the fact that RREQs are being either ignored, forwarded or answered with an appropriate RREP message. Analogous checks can be done for RREP and RERR messages. This kind of properties are specified in terms of `LTL` formulas of the form

$$\Box\,(\langle\!\langle d_{\text{receive}} \in D \rangle\!\rangle \rightarrow \Diamond \langle\!\langle d_{\text{send}} \in D' \rangle\!\rangle)$$

where D and D' are the sets of messages encoding the received message and the response of the sensor node to be sent. The sets D and D' may impose conditions on the current configuration of the inspected sensor node. E.g., D could be the the set of RREQ messages for a node *with known routing information*, then D' could be the set of appropriate RREP messages. Similar properties can be specified using `BTSL` formulas of the form

$$\neg\exists\langle\!\langle tt^*; (d_{\text{receive}} \in D); (\neg\text{send} \vee d_{\text{send}} \notin D')^*\rangle\!\rangle \text{“nothing to be done”}$$

where “nothing to be done” is an atomic proposition for the states where the message buffers are empty and the CA is in the IDLE configuration. Such a formula states that there is no path where a message in D has been received but not answered with a message in D' attaining in a state labeled with “nothing to be done”.

Other interesting properties were specified with the help of our logics. E.g., we were able to show that sequence numbers are non-decreasing and the protocol contains no deadlock, i.e., there is always a chance to recover from any situation and return back to the IDLE state with empty message buffers.

$$\forall\Box\,(\exists\langle\!\langle\alpha\rangle\!\rangle \text{“nothing to be done”})$$

The regular stream expression α may impose further conditions on how to return to the IDLE state, e.g., a simple condition can be $\alpha = (\neg\text{receive})^*$ stating that along the path back to IDLE no further input from the environment is required.

Verification of the composite system. For the verification of the composite system we whether sensors can establish existing routes. For this purpose we specified `BTSL` and `LTL` formulas stating the existence of a path where messages from the connected nodes can eventually reach the sink.

$$\exists\langle\!\langle tt^*; d_{\text{receive}[0]} \in D_i\rangle\!\rangle \text{true for all } i \in \text{ID}$$

Here, D_i is the set of payload messages from the node with ID i. To verify such properties for all paths, additional fairness assumptions have to be made, which are also expressible in our logic framework. For static topologies one can look at additional properties, e.g., that from some moment in time only payload messages will be sent or that the handling of RERR messages becomes superfluous.

Due to the high data dependency of the behavior of the sensor nodes the analysis of the inspected routing protocol is a challenging task. With an increasing number of sensor nodes, more BDD-variables are needed for encoding of the addresses inside of messages, the message buffers, and the routing table of each node. The table below exposes the complexity of the model by presenting the number of BDD-variables (bits) that are needed in the context of a network of k sensor nodes for the encoding of a) a single message, b) the state variables of a single sensor node, and c) the state variables of the composite system.

Bits for the encoding	$k = 2$	$k = 3$	$k = 4$	$k = 5$
Message	15	18	20	23
Sensor node states	58	74	91	111
System states	116	222	364	555

The performance of our model checker crucially depends on a compact BDD representation of the system. For such data dependent protocols it is not obvious how to obtain good variable orderings. Nevertheless, we were able to compose and verify network structures with up to three sensor nodes with a state space of about 10^{23} reachable states when applying abstraction techniques for the model

specification as well as techniques and heuristics for good variable ordering. With the help of Vereofy we verified (and falsified) various properties and found different kinds of modeling errors, like wrongly addressed messages, deadlocks, missing robustness features, incorrectness of assumptions for simplifications, etc. The counterexamples and witnesses created by Vereofy can then be analyzed to fix the model. The ability to examine witnesses satisfying specific stream expressions further aid in exploration and debugging.

The model checking has been performed on an AMD Athlon 64 X2 Dual Core Processor 5000+ (2.6 GHz) with 8GB of RAM, running Linux. For the analysis of a single sensor node we were able to build the symbolic representation of the CA for sensor nodes with up to seven routing table entries without applying any abstraction or simplification to the specification of a sensor node. For the verification we used the modules of sensor nodes with up to three or four routing table entries. The model checking took less than three minutes for most of the properties above. The most complex ones can be checked within ten minutes.

For the composite system, network structures with only a few nodes can be built without applying abstractions of the model. For the verification we created simpler variants of the model, with a fixed linear topology, and less data dependencies. E.g., we removed the handling of error messages as we were able to show that they never appear in the static model case with a fixed topology. Additionally, we pre-computed variable orderings to benefit from preferably compact BDD representation of the system behavior. The pre-computation itself consumed a few hours, with the system composition afterwards taking a few minutes. Using the pre-computed variable orderings the composition and verification of the composite system can be performed in less than 40 minutes for each of the above properties.

5 Conclusion

We presented a framework for specifying and analyzing components and connectors in an exogenous framework. The tool-kit Vereofy with its input languages CARML and RSL has successfully been used to model a non-trivial case study of a sensor network. In future work, advanced abstraction techniques, compositional methods and better heuristics for BDD variable orderings will be studied.

References

1. Alur, R., de Alfaro, L., Grosu, R., Henzinger, T.A., Kang, M., Kirsch, C.M., Majumdar, R., Mang, F.Y.C., Wang, B.-Y.: Jmocha: A model checking tool that exploits design structure. In: Proceedings of the 23rd International Conference on Software Engineering (ICSE), pp. 835–836. IEEE Computer Society, Los Alamitos (2001)
2. Arbab, F.: Reo: A Channel-Based Coordination Model for Component Composition. Mathematical Structures in Computer Science 14(3), 329–366 (2004)
3. Baier, C., Blechmann, T., Klein, J., Klüppelholz, S.: A Uniform Framework for Modeling and Verifying Components and Connectors. In: Field, J., Vasconcelos, V.T. (eds.) COORDINATION 2009. LNCS, vol. 5521, pp. 247–267. Springer, Heidelberg (2009)

4. Baier, C., Blechmann, T., Klein, J., Klüppelholz, S.: Formal Verification for Components and Connectors. In: de Boer, F.S., Bonsangue, M.M., Madelaine, E. (eds.) FMCO 2008. LNCS, vol. 5751, pp. 82–101. Springer, Heidelberg (2009)
5. Baier, C., Sirjani, M., Arbab, F., Rutten, J.: Modeling Component Connectors in Reo by Constraint Automata. Science of Computer Programming 61, 75–113 (2006)
6. Capizzi, S., Solmi, R., Zavattaro, G.: From endogenous to exogenous coordination using aspect-oriented programming. In: De Nicola, R., Ferrari, G.-L., Meredith, G. (eds.) COORDINATION 2004. LNCS, vol. 2949, pp. 105–118. Springer, Heidelberg (2004)
7. Chiyangwa, S., Kwiatkowska, M.: A timing analysis of AODV. In: Steffen, M., Zavattaro, G. (eds.) FMOODS 2005. LNCS, vol. 3535, pp. 306–322. Springer, Heidelberg (2005)
8. Cimatti, A., Clarke, E., Giunchiglia, F., Roveri, M.: NUSMV: A new symbolic model checker. International Journal on Software Tools for Technology Transfer 2(4), 410–425 (2000)
9. Clarke, E., Emerson, E., Sistla, A.: Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. ACM Transactions on Programm. Languages and Systems 8(2), 244–263 (1986)
10. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (1999)
11. Emerson, E., Lei, C.: Efficient Model Checking in Fragments of the Propositional μ–Calculus. In: Proc. of LICS, pp. 267–278. IEEE Computer Society Press, Los Alamitos (1986)
12. Gößler, G., Sifakis, J.: Component-based construction of deadlock-free systems: Extended abstract. In: Pandya, P.K., Radhakrishnan, J. (eds.) FSTTCS 2003. LNCS, vol. 2914, pp. 420–433. Springer, Heidelberg (2003)
13. Guillen-Scholten, J., Arbab, F., de Boer, F., Bonsangue, M.: MoCha-pi: an exogenous coordination calculus based on mobile channels. In: Proceedings of the 2005 ACM Symposium on Applied Computing (SAC), pp. 436–442. ACM, New York (2005)
14. Holzmann, G.J.: The model checker SPIN. IEEE Transactions on Software Engineering 23, 279–295 (1997)
15. Klüppelholz, S., Baier, C.: Symbolic model checking for channel-based component connectors. Science of Computer Programming 74(9), 688–701 (2009)
16. Klüppelholz, S., Baier, C.: Alternating-time stream logic for multi-agent systems. Science of Computer Programming 75(6), 398–425 (2010)
17. Liu, X., Wang, J.: Formal Verification of Ad hoc On-demand Distance Vector (AODV) Protocol using Cadence SMV, Report, Univ. of British Columbia (2004)
18. Majster-Cederbaum, M., Minnameier, C.: Everything is PSPACE-complete in interaction systems. In: Fitzgerald, J.S., Haxthausen, A.E., Yenigun, H. (eds.) ICTAC 2008. LNCS, vol. 5160, pp. 216–227. Springer, Heidelberg (2008)
19. Musuvathi, M., Park, D., Chou, A., Engler, D., Dill, D.: CMC: A Pragmatic Approach to Model Checking Real Code. In: OSDI 2002 (2002)
20. Obradovic, D.: Formal Analysis of Routing Protocols. PhD thesis, University of Pennsylvania (2001)
21. Perkins, C., Belding-Royer, E., Das, S.: Ad hoc On-Demand Distance Vector (AODV) Routing. RFC 3561, IETF (July 2003)
22. Pnueli, A.: The Temporal Logic of Programs. In: Proc. of 18th FOCS, pp. 46–57. IEEE Computer Society Press, Los Alamitos (1977)
23. Vardi, M., Wolper, P.: An Automata-Theoretic Approach to Automatic Program Verification. In: LICS, pp. 332–345. IEEE Computer Society Press, Los Alamitos (1986)

A Case Study in Model-Based Adaptation of Web Services

Javier Cámara[1], José Antonio Martín[2], Gwen Salaün[3], Carlos Canal[2], and Ernesto Pimentel[2]

[1] INRIA Rhône-Alpes, France
Javier.Camara-Moreno@inria.fr
[2] Department of Computer Science, University of Málaga, Spain
{jamartin,canal,ernesto}@lcc.uma.es
[3] Grenoble INP-INRIA-LIG, France
Gwen.Salaun@inria.fr

Abstract. Developing systems through the composition of reusable software services is not straightforward in most situations since different kinds of mismatch may occur among their public interfaces. Service adaptation plays a key role in the development of such systems by solving, as automatically as possible, mismatch cases at the different interoperability levels among interfaces by synthesizing a mediating adaptor between services. In this paper, we show the application of model-based adaptation techniques for the construction of service-based systems on a case study. We describe each step of the adaptation process, starting with the automatic extraction of behavioural models from existing interface descriptions, until the final adaptor implementation is generated for the target platform.

1 Introduction

The widespread adoption of Service-Oriented Architectures in the last few years has fostered the need to develop complex systems in a timely and cost-effective manner by assembling reusable software services. These can be considered as blocks of functionality which are often developed using different technologies and platforms. On the one hand, SOA enables developers to build applications almost entirely from existing services which have already been tested, resulting in a speed-up of the development process without compromising quality. On the other hand, the potential heterogeneity of the different services often results in interoperability issues at different levels which have to be solved by system architects on an ad-hoc basis.

In order to ensure interoperability, service interfaces must provide a comprehensive description of the way in which they have to be accessed by service consumers. Composition of services is seldom achieved seamlessly because mismatch may occur at the different interoperability levels (*i.e.,* signature, interaction protocol/behaviour, quality of service, and functional). *Software adaptation* [6,17] is a recent discipline which aims at generating, as automatically as possible, mediating services called *adaptors*, used to solve mismatches among services in a non-intrusive way.

T. Margaria and B. Steffen (Eds.): ISoLA 2010, Part II, LNCS 6416, pp. 112–126, 2010.

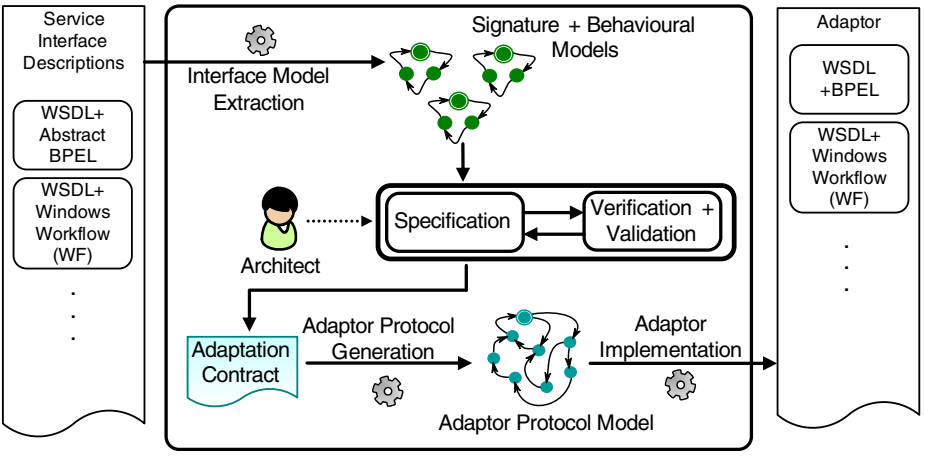


Fig. 1. Generative adaptation process

So far, most adaptation approaches have assumed interface descriptions that include signatures (operation names and types) and behaviours (interaction protocols). Describing protocols in service interfaces is essential because erroneous executions or deadlock situations may occur if the designer does not consider them while composing services.

In this paper, we show the application of model-based adaptation techniques [5,12] (see Figure 1) for Web services, focusing on a case study that we use to illustrate the different steps in the approach. The process starts with the automatic extraction of behavioural service models from existing interface descriptions. These descriptions include a WSDL specification of the different operations made available at the service interface, as well as a specification of the behaviour of the service which can be given in languages such as Abstract BPEL, or Windows Workflows. Next, the designer can build an adaptation contract, which is an abstract specification of how mismatch cases among the different service interfaces can be solved. This is not a trivial task, therefore we propose a graphical representation and an interactive environment to guide the designer through the process. Once the adaptation contract is built, its design can be validated and verified using techniques which enable the visual simulation of the execution of the system step-by-step, finding out as well which parts of the system lead to erroneous behaviour (deadlocks, infinite loops, violation of safety properties, etc.). In such a way, the designer can check if the behaviour of the system complies with his/her intentions. Once the designer is satisfied with the design, an adaptor protocol model can be generated and implemented into an actual adaptor which can be deployed in the target platform.

The rest of this paper is structured as follows: first, we present in Section 2 a description of the case study that we use to illustrate the different steps of the development process in the remaining sections. Section 3 presents an overview of the adaptation process for our case study, starting with the extraction of behavioural models from existing interface descriptions of the services that we intend to reuse in the system given in WSDL and Abstract BPEL (Section 3.1). Section 3.2 illustrates the contract

specification process for our case study. Sections 3.3 and 3.4 describe the adaptor protocol generation and its implementation into an actual adaptor using BPEL as target language, respectively. Section 4 concludes the paper.

2 Case Study: Online Medical Management System

We present a case study in the context of a health care organization, which describes the development of a management system which is required to handle online patient appointments with general practitioners, as well as with specialist doctors in the organization. In particular, the system must be able to create appointments for valid system users who are provided with a username and an access password. After logging in to the system, the user must be able to request an appointment for a given date either with a general practitioner, or with a specialist doctor. After checking doctor availability, the system will return a ticket identifier to the user that corresponds to the provided appointment. If the system does not find a time slot for the user request, the user should be allowed to perform additional requests for further appointments.

In order to build this new system, we aim at reusing two existing sub-systems whose functionality is exposed through different services:

- Service ServerDoc handles appointments with general practitioners.
- Service ServerEsp handles appointments with specialist doctors.

Furthermore, we also reuse a client that implements an example of user requirements. It is worth observing that this client enables the user to perform requests both to general practitioners and specialist doctors in any arbitrary order, whereas a new policy within the organization establishes that users should not be allowed to schedule appointments with specialist doctors without a prior appointment with a general practitioner. Hence, this is an important requirement that the system resulting from our service composition must meet.

3 Overview of the Adaptation Process

3.1 Interface Model Extraction

We assume that service interfaces are specified using both a signature and a protocol. Signatures correspond to operation names associated with arguments and return types relative to the messages and data being exchanged when the operation is called. Protocols are represented by means of Symbolic Transition Systems (STSs), which are Labelled Transition Systems (LTSs) extended with value passing [15]. This formal model has been chosen because it is simple, graphical, and provides a good level of abstraction to tackle verification, composition, or adaptation issues [9,10,16].

At the user level, developers can specify service interfaces (signatures and protocols) using respectively WSDL, and Abstract BPEL (ABPEL) or WF workflows (AWF) [7].

In order to build the interface models of the services and the client to be reused in the system, we parse their WSDL descriptions and generate the corresponding signatures. Moreover, we can generate the behavioural part of the model (STSs) from service interfaces specified using ABPEL or AWF. To ease the addition of other possible notations to describe service interfaces, we use as an intermediate step in this parsing process an abstract Web services class (AWS). Thus, one can add as a front-end another description language with its parser to AWS, and take advantage of the existing parser from AWS to our model (see Figure 2).

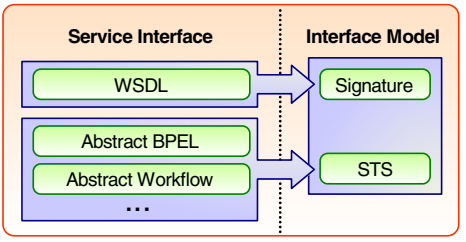


Fig. 2. Interface model extraction

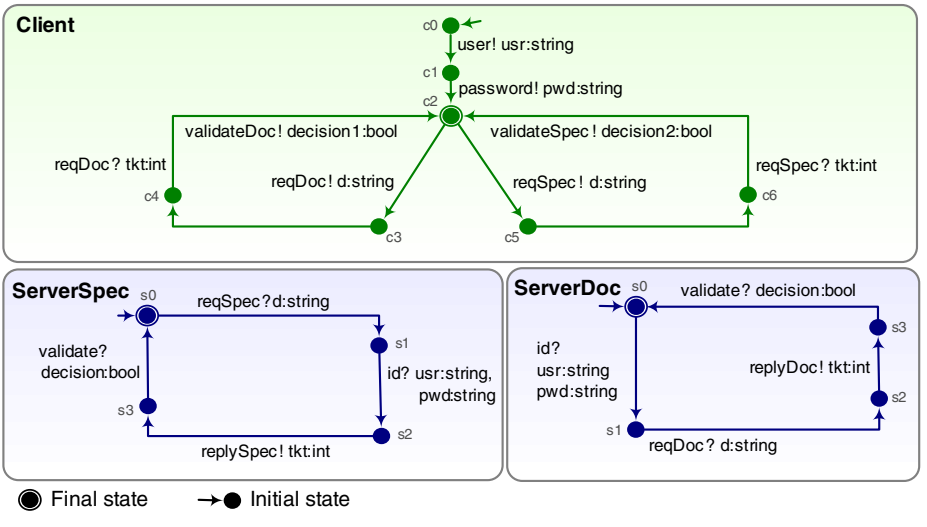


Fig. 3. Behavioural models for the different services and the client

Example. The STSs depicted in Figure 3 are obtained after the application of the parsing process to each of the elements of the (running) example.

- The Client can first log on to a server by sending respectively his/her user name (user!) and password (password!). Then, depending on his/her preferences, the client can stop at this point, or ask for an appointment either with a general practitioner (reqDoc!) or a specialist doctor (reqSpec!), and then receive an appointment identifier. Finally, the client can accept or reject the appointment obtained if it is not convenient for him/her (validateDoc/validateSpec!).
- Service ServerDoc first receives the client user name and password (id?). Next, this service receives a request for an appointment with a general practitioner

(reqDoc?) and replies (replyDoc!). Finally, the service waits for an acknowledgement from the user either accepting or rejecting the provided appointment (validate?).
– Service ServerSpec first receives a request for an appointment with a specialist doctor (reqSpec?), followed by the client user name and password (id?). After checking doctor availability for the given date, an appointment identifier is returned (replySpec!) to the client. As it happened in the case of the ServerDoc service, ServerSpec finishes its interaction waiting for an acknowledgement from the user either accepting or rejecting the provided appointment (validate?).

The composition of the different services in our example is subject to different mismatch situations among their interfaces:

– **Name mismatch** occurs if a service expects a particular message, while its counterpart sends one with a different name (*e.g.*, service ServerDoc sends replyDoc!, whereas the client is expecting reqDoc?).
– **N to M correspondence** appears if a message on a particular interface corresponds to several ones in its counterpart's interface (or similarly, a message has no correspondence at all). In Figure 3 it can be observed that while the client intends to log in to a service sending user! and password! subsequently, service ServerDoc expects only message id? for authentication.
– **Incompatible order of messages.** The relative order of operation invocations among the different protocols involved is not compatible. We may observe this in our example when the client first sends its authentication information and then requests an appointment with a specialist doctor, whereas the ServerSpec service expects these messages in the inverse order.
– **Argument mismatch** may occur when the number and/or type of arguments either being sent or received do not match between the operations on the different interfaces. This can be observed in ServerDoc, when id? expects both a username (usr) and a password (pwd). The first data term corresponds to user! on the client interface, whereas the second belongs to password!.

3.2 Adaptation Contract Specification

Once the interface models have been extracted from the WSDL and ABPEL descriptions, we can use them to produce the adaptation contract for our system. In particular, we use *vectors* and a *vector LTS* (VLTS) as adaptation contract specification language [14,6,12]. A vector contains a set of events (message, direction, list of parameters). Each event is executed by one service, and the overall result corresponds to one or several interactions between the involved services and the adaptor. Vectors express correspondences between messages, like bindings between ports, or connectors in architectural descriptions. In particular, we consider a binary communication model, therefore our vectors are always reduced to one event (when a service evolves independently) or two (when services communicate indirectly through the adaptor). Furthermore, variables are used as placeholders in message parameters. The same variable names appearing in different labels (possibly in different vectors) relate sent and received arguments in the messages.

In addition to vectors, the contract notation includes a Labelled Transition System (LTS) with vectors on transitions (that we call vector LTS or VLTS). This element is used as a guide in the application order of interactions specified by vectors. VLTSs go beyond port and parameter bindings, and express more advanced adaptation properties (such as imposing a sequence of vectors or a choice between some of them). If the application order of vectors does not matter, the VLTS contains a single state and all transitions looping on it.

$$
\begin{aligned}
\{v_{user} &= \langle c : \mathsf{user!U} \rangle, \\
v_{pwd} &= \langle c : \mathsf{password!P} \rangle, \\
v_{vidsp} &= \langle ss : \mathsf{id?U, P} \rangle, \\
v_{viddoc} &= \langle sd : \mathsf{id?U, P} \rangle, \\
v_{reqsp1} &= \langle c : \mathsf{reqSpec!DATE}; ss : \mathsf{reqSpec?DATE} \rangle, \\
v_{reqsp2} &= \langle c : \mathsf{reqSpec?RES1}; ss : \mathsf{replySpec!RES1} \rangle, \\
v_{reqdoc1} &= \langle c : \mathsf{reqDoc!DATE}; sd : \mathsf{reqDoc?DATE} \rangle, \\
v_{reqdoc2} &= \langle c : \mathsf{reqDoc?RES2}; sd : \mathsf{replyDoc!RES2} \rangle, \\
v_{valid1} &= \langle sd : \mathsf{validate?B1}; c : \mathsf{validateDoc!B1} \rangle, \\
v_{valid2} &= \langle ss : \mathsf{validate?B2}; c : \mathsf{validateSpec!B2} \rangle \ \}
\end{aligned}
$$

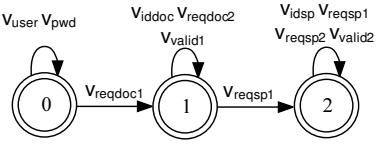


Fig. 4. Adaptation contract for our example: vectors (left) and VLTS (right)

Example. Going back to our on-line medical management system described in Section 2, let us recall that we intend to compose our services into a working system where the client can request an appointment with a general practitioner, or also request an appointment with a specialist doctor, provided that there is a previous appointment with a general practitioner (*i.e.*, the client cannot directly schedule an appointment with a specialist).

Figure 4 displays the set of vectors used to solve mismatch among our interfaces. In order to understand how vectors relate messages on the different interfaces, let us focus on v_{reqsp2}, for instance. Here we may observe that the message reqSpec? on the client interface is related with the replySpec! message on the ServerSpec interface the appointment ticket identifier argument on both messages is related by placeholder RES1 (please refer to Figure 7 for more details about how placeholders relate message arguments). However, expressing correspondences between messages is not always as straightforward as in the previous example. We may now focus on the initial part of the composition, where we want to connect the general practitioner server (ServerDoc) with the client, and make authentication work correctly. For this, we need three vectors, respectively v_{user}, v_{pwd} and v_{viddoc}, in which we solve existing mismatches by relating different message names (id, user and password). Here, we first specify the independent evolution of the client through user! and password! (this is expressed by vectors v_{user} and v_{pwd}, which only contain one message). Next, we also specify the independent evolution of ServerDoc through v_{viddoc}. Exchanged data parameters among the three involved messages in the vectors are connected using placeholders U and P.

Regarding the specification of additional constraints on the composition, we can observe on the right-hand side of Figure 4 that the VLTS for the contract constrains the

interaction of the Client, ServerDoc, and ServerSpec interfaces by imposing the request for an appointment with a general practitioner ($v_{reqdoc1}$) always before the request of an appointment with a specialist doctor (v_{reqsp1}). This is achieved by excluding v_{reqsp1} from the possible transitions in state 0, and including the transition $(0, v_{reqdoc1}, 1)$. □

In order to make the specification as simple and user-friendly as possible, we employ interactive specification techniques to support the architect through this process. To this purpose, we use a notation to graphically make explicit bindings between ports using an interactive environment that enables graphical contract construction and verification called ACIDE. The graphical notation for a service interface includes a representation of its behavioural model (STS) and a collection of ports. Each label on the STS corresponds to a *port* in the graphical description of the interface. Ports include a *data port* for each parameter contained in the parameter list of the label. Figure 5 summarizes ports and bindings used in our notation.

Correspondences between the different service interfaces are represented as *port bindings* and *data port bindings* (solid and dashed connector lines, respectively). Starting from the graphical representation of the interfaces, the architect can build a contract between them by successively connecting ports and data ports. This results in the creation of bindings which specify how the interactions should be carried out between the services. It is also possible to add a T-shaped *port cap* on a port in order to indicate that it does not have to be connected anywhere.

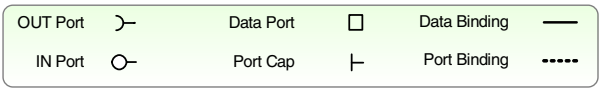


Fig. 5. Graphical notation: ports and bindings

The VLTS imposing an order on the application of the bindings is built implicitly in ACIDE as new bindings are created by the designer. Initially, the VLTS has a single state and no transitions. Each time a new connection is made, the VLTS is extended in a different way, depending on the current VLTS extension mode selected by the user:

- **Abstract mode.** No order on the application of the bindings is imposed. Creating a binding in this mode results in the creation of a transition looping on the current state in the VLTS.
- **Sequential mode.** Bindings created in this mode must be executed one after the other. This results in the extension of the VLTS with a fresh state and a transition from the current state to the new one. Once this transition is added, the newly created state becomes the current VLTS state.
- **Branching mode.** Bindings created in this mode are mutually exclusive. The VLTS is extended in this case with a fresh state and a transition to it from the current state. Unlike in sequential mode, the current state is not updated.

Using this implicit method it is possible to build a VLTS for most contracts. However, the user is also able to directly manipulate the VLTS from within the graphical environment in order to adjust it to particular situations such as a binding

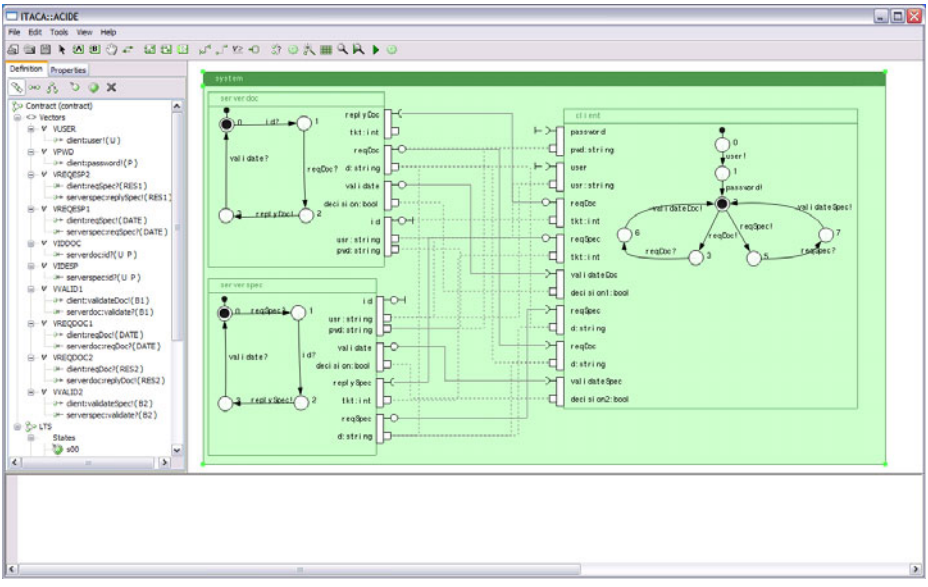


Fig. 6. Contract specification for the Online Medical System in ACIDE

representing an interaction which has to be carried out more than once in different parts of the specification.

During the specification of the contract, we can also make use of a set of validation and verification techniques to check that an adaptation contract makes the involved services work correctly. These techniques are intended to help the designer in understanding potential problematic behaviours of the system which are not obvious (even to the trained eye) just by observing service interaction protocols and adaptation contracts. These problems may include potential deadlocks, as well as unintended interactions that are not explicitly addressed at the contract level, which is only an abstract specification of the adaptation and does not take into account every interaction scenario among services. These techniques are completely automated, and include four kinds of checks: (i) static checks on the contract *wrt.* STS service interfaces involved, (ii) simulation of the system execution, (iii) trace-checking to find potential deadlocking executions and infinite loops, and (iv) verification of temporal logic formulas.

Example. Figure 6 shows a screenshot of the graphical representation of our final adaptation contract specification for the medical management system in ACIDE. If we focus on the graphical representation of the ServerDoc, it can be observed that it contains a port for the reception of the reqDoc? request with a data port attached representing the date, and another port for the emission of the replyDoc response with a data port attached representing the ticket identifier issued for the given date. Moreover, it can be observed that the interactions expressed by vectors in our contract are represented by port bindings in the graphical environment.

3.3 Generation of the Adaptor Protocol

Being given a set of service interfaces and an adaptation contract, an adaptor protocol can be generated using automatic techniques as those presented in [12]. An adaptor is a third-party component that is in charge of coordinating the services involved in the system with respect to the set of constraints defined in the contract. Consequently, all the services communicate through the adaptor, which is able to compensate mismatches by making required connections as specified in the contract.

From adaptor protocols, either a central adaptor can be implemented, or several service wrappers can be generated to distribute the adaptation and preserve parallelism in the system's execution. In the former case, the implementation of executable adaptors from adaptor protocols can be achieved for instance using Pi4SOA technologies [1], or techniques presented in [12] and [7] for BPEL and Windows Workflow Foundation, respectively. In the latter case, each wrapper constrains the functionality of its service to make it respect the adaptation contract specification [15].

Adaptor and wrapper protocols are automatically generated in two steps: (i) system's constraints are encoded into the LOTOS [11] process algebra, and (ii) adaptor and wrapper protocols are computed from this encoding using on-the-fly exploration and reduction techniques. These techniques are platform-independent, therefore while exploring the state space, all the behaviours (interleaving) respecting the adaptation constraints are kept in the adaptor model. The reader interested in more details may refer to [12,15].

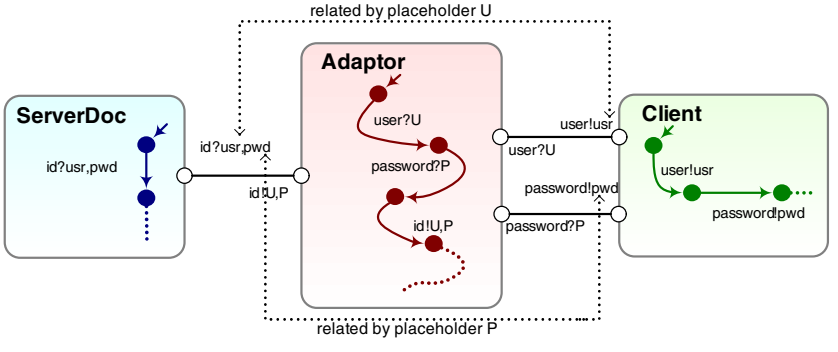


Fig. 7. Example of adaptation for authentication mismatches

Example. Figure 7 shows a small portion of the adaptor protocol generated from the three vectors $v_{user} = \langle c : \mathsf{user!U} \rangle$, $v_{pwd} = \langle c : \mathsf{password!P} \rangle$ and $v_{viddoc} = \langle sd : \mathsf{id?U, P} \rangle$ given in Figure 4. This makes service ServerDoc and the Client interact correctly. We emphasize that the adaptor synchronizes with the services using the same name of messages but the reversed directions, *e.g.*, communication between id? in ServerDoc and id! in the adaptor. Furthermore, when a vector includes more than one communication action, the adaptor always starts the set of interactions formalized in the vector with the receptions (which correspond to emissions on service interfaces), and next handles the emissions.

Figure 8 displays the adaptor protocol generated using the adaptation contract shown in Figure 4. This adaptor contains 51 states and 73 transitions, and therefore has reasonable size and complexity. Interaction starts by receiving the user?, password? and reqDoc? messages sent by the Client. Next, the adaptor starts interacting with ServerDoc by first sending authentication information (id!) and then the request posted previously by the client (reqDoc!). Once the client and the doctor have ended their interaction, the adaptor can send a request to interact with the specialist (reqSpec?), this is the case in state 25 for example. Note that in the bottom part of the adaptor protocol, corresponding to the interaction with the specialist, the adaptor can treat several specialist requests (*e.g.*, reqSpec? in state 31). Notice also that the adaptor can terminate at different points of its execution (transitions labelled with FINAL). The adaptor protocol corrects all the mismatch cases presented in Section 3.1, for instance we can see in state 33 that the adaptor can submit first the request to the specialist (reqSpec!) and then the authentication information in state 38 (id!) solving the reordering problem existing between the client and the specialist service.

If we consider the adaptation contract with vectors only (the corresponding VLTS consists of a single state with all vector transitions looping on it), the adaptor protocol consists of 243 states and 438 transitions. This quite high number of states and transitions is due to the release of constraints specified in the original VLTS which imposes sequentiality on the system (interactions first with the doctor and in a second step with the specialist), thus reducing interleaving.

3.4 Implementation

Our internal model (STS) can express some additional behaviours (interleaving) that cannot be implemented into executable languages (*e.g.,* BPEL). To make platform-independent adaptor protocols implementable *wrt.* a specific platform we proceed in two steps: (i) filtering the interleaving cases that cannot be implemented, and (ii) encoding the filtered model into the corresponding implementation language.

Filtering. Techniques presented in this paper to generate adaptor protocols are platform-independent, therefore the adaptor model may contain parts which cannot be implemented in a given language (*e.g.*, BPEL). This filtering step aims at removing in the adaptor protocol these non-implementable parts. These parts represent the interleaving of parallel operations and they are not necessary for the functionality of the system. As an example, if there are several emission transitions going out from the same state, this leads to non-determinism which cannot be implemented in BPEL. One of the filtering rules states that in such a case, only one emission is kept. In this paper, we reuse filtering rules presented in [12]. We show in Figure 9 the filtered adaptor protocol. One can see that the protocol contains first a sequence corresponding to the interaction with ServerDoc, and next a loop corresponding to the interaction with ServerSpec. The client must start interacting with ServerDoc, and in state 11, (s)he can choose either to interact again with ServerSpec, or to stop. This protocol can be implemented in BPEL as we will see in the remainder of this section.

BPEL implementation. The adaptor protocol is implemented using a state machine pattern. The main body of the BPEL process corresponds to a global *while* activity

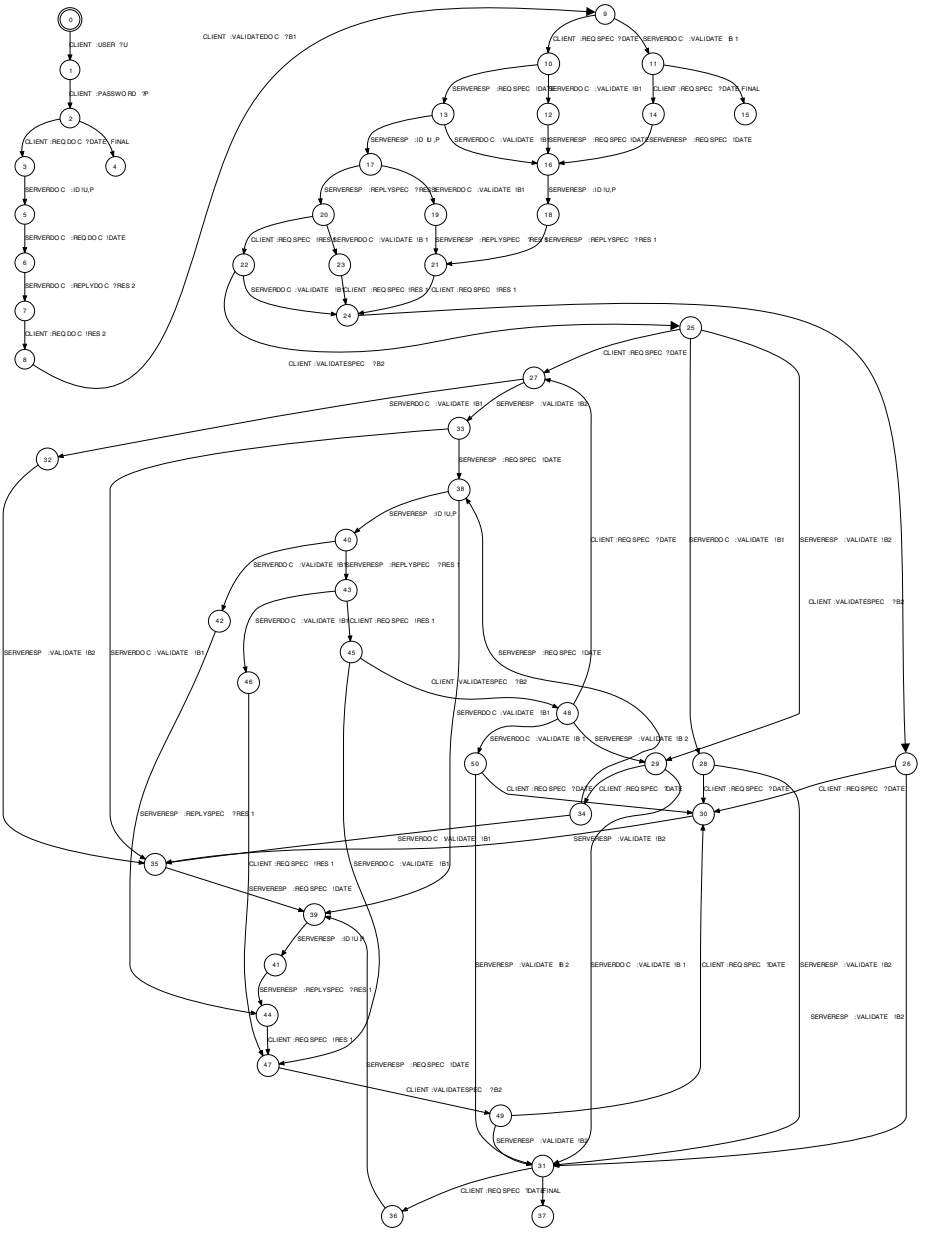


Fig. 8. Adaptor protocol generated for the Online Medical System example

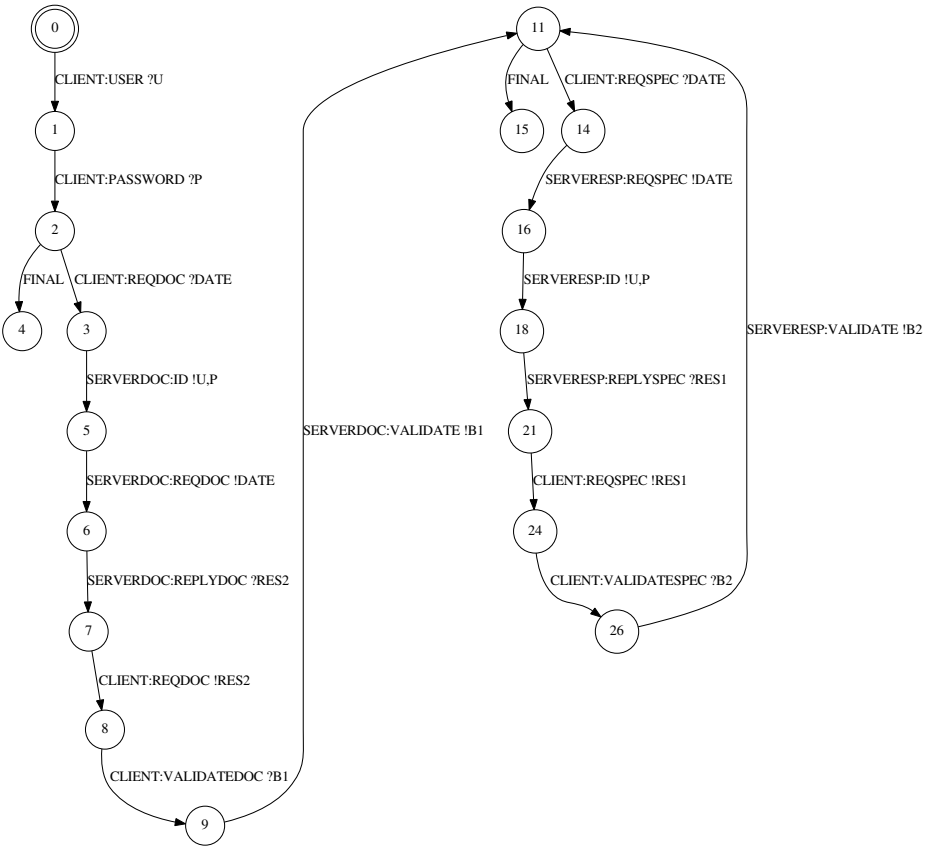


Fig. 9. Filtered adaptor protocol obtained for the Online Medical System example

with *if* statements used inside it to encode adaptor states and *pick* statements to choose which branch to follow depending on the received messages. Variables are used to store data passing through the adaptor and the current state of the protocol. Timeouts (*onAlarm* activities) are used in these *pick* activities to model FINAL transitions. Every state in the adaptor behaviour with several incoming or outgoing transitions is encoded as a new branch of the *if* activity in the implementation. That branch might contain an internal *pick* activity with sequences of communication activities alternated with assignments to update the variables and change the current state of the execution. Adaptors whose protocol is a global loop beginning with a final state (such as services ServerDoc and ServerSpec) are modelled as a sequence, therefore every iteration of the global loop will be a new instantiation of the adaptor. Let us note that the implementation of the adaptor is constrained by the specifics of the implementation of the services and the actual BPEL engine. These might avoid the proper implementation and execution

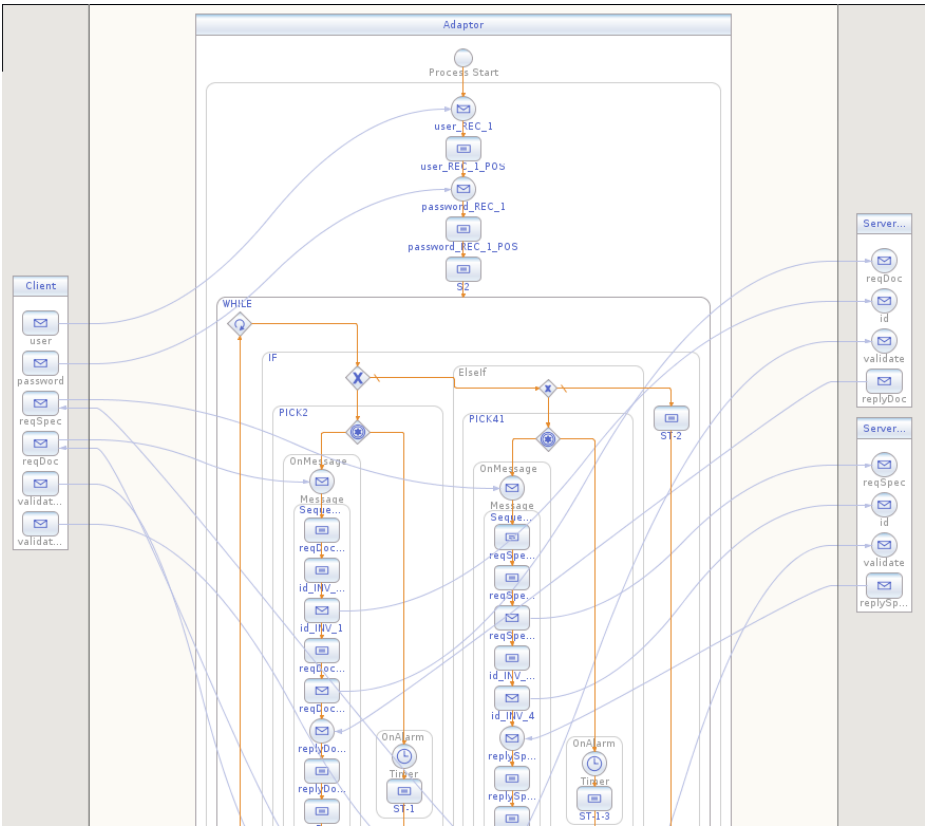


Fig. 10. Implementation of the adaptor in BPEL

of the adaptor. For instance, BPEL allows bi-directional and blocking *invoke* activities and their corresponding *receive* activities, in some scenarios, these could block the adaptor and avoid its proper execution. In addition, current BPEL engines do not fully support BPEL 2.0, for instance, Glassfish 2.1.x does not execute properly BPEL processes with two different *receive* activities of the same operation and partner link. This restricts even further the implementation of the adaptor.

Example. Figure 9 shows the behaviour of the filtered adaptor corresponding to the running example, whereas Figure 10 displays a graphical model of part of its implementation in BPEL. The client interface and the intarfaces of the two servers are located on the left-hand side and right-hand side of the BPEL implementation, respectively. This adaptor presents an initial log-in sequence followed by the *while* activity with a nested *if* activity, as previously mentioned. In the first iteration of the loop, the current state of the adaptor is 2 (see Figure 9), therefore the execution continues through the *if* and *pick* activities on the left-hand side of the loop. We have an *onAlarm* which finishes

the session because the client might not perform any request at all. Otherwise, once a request for a general practitioner is processed, the current state of the adaptor is set to be 11 and we iterate once more. In the second iteration, we have an analogous structure for specialists requests in the right-hand side of the loop. In this case, however, the adaptor can process several of such requests (or none) and, when there are not anymore requests, the *onAlarm* activity triggers the end of the session.

4 Concluding Remarks

Building systems by adapting a set of reusable software services whose functionality is accessed through their behavioural interfaces is an error-prone task which can be supported by assisting developers with the automatic procedures and tools supplied by model-based software adaptation. In particular, existing works dedicated to model-based behavioural adaptation are often classified in two families. A first class of existing works can be referred to as *restrictive approaches* [3,4,13], and favour the full automation of the process, trying to solve interoperability issues by pruning the behaviours that may lead to mismatch, thus restricting the functionality of the services involved. These techniques are limited since they are not able to fix subtle incompatibilities between service protocols by remembering and reordering messages and their parameters when necessary. A second class of solution which can be referred to as *generative approaches* [2,6,8] avoid the arbitrary restriction of service behaviour, and supports the specification of advanced adaptation scenarios. Generative approaches build adaptors automatically from an abstract specification of how the different mismatch situations can be solved, often referred to as *adaptation contract*.

In this paper, we have shown the application of model-based adaptation techniques for Web services, using a case study on the development of an online medical management system to illustrate all the steps of the process. The approach used gathers desirable features from existing model-based behavioural adaptation approaches in a single process. All the steps of the process presented in this paper are fully tool-supported by a toolbox called ITACA [5], which enables the specification and verification of adaptation contracts, automates the generation of adaptor protocols, and relates our abstract models with implementation languages.

With respect to the results that we obtained from the application of the approach to this and other case studies, we were able to assess that there is a remarkable reduction both in the amount of effort that the developer has to put into building the adaptor, as well as in the number of errors present in the final result. Since the test cases used so far for our experiments were of a small-medium size and complexity, we think that the difficulty of specifying contracts for bigger systems involving dozens of services would be not manageable by the developers without tool-supported, model-based adaptation techniques. This puts forward the importance of providing such support for the development of service-based systems.

Acknowledgements. This work has been partially supported by the project TIN2008-05932 funded by the Spanish Ministry of Innovation and Science (MICINN), and project P06-TIC-02250 funded by the *Junta de Andalucía*.

References

1. Pi4SOA Project, http://www.pi4soa.org
2. Brogi, A., Bracciali, A., Canal, C.: A formal approach to component adaptation. The Journal of Systems and Software 74, 45–54 (2005)
3. Autili, M., Inverardi, P., Navarra, A., Tivoli, M.: SYNTHESIS: A Tool for Automatically Assembling Correct and Distributed Component-based Systems. In: Proc. of ICSE 2007, pp. 784–787. IEEE Computer Society, Los Alamitos (2007)
4. Brogi, A., Popescu, R.: Automated generation of BPEL adapters. In: Dan, A., Lamersdorf, W. (eds.) ICSOC 2006. LNCS, vol. 4294, pp. 27–39. Springer, Heidelberg (2006)
5. Cámara, J., Martín, J.A., Salaün, G., Cubo, J., Ouederni, M., Canal, C., Pimentel, E.: ITACA: An Integrated Toolbox for the Automatic Composition and Adaptation of Web Services. In: Proc. of ICSE 2009, pp. 627–630. IEEE Computer Society, Los Alamitos (2009)
6. Canal, C., Poizat, P., Salaün, G.: Model-based adaptation of behavioural mismatching components. IEEE Transactions on Software Engineering 4(34), 546–563 (2008)
7. Cubo, J., Salaün, G., Canal, C., Pimentel, E., Poizat, P.: A Model-Based Approach to the Verification and Adaptation of WF/.NET Components. In: Proc. of FACS 2007. ENTCS, vol. 215, pp. 39–55. Elsevier, Amsterdam (2007)
8. Dumas, M., Spork, M., Wang, K.: Adapt or Perish: Algebra and Visual Notation for Service Interface Adaptation. In: Dustdar, S., Fiadeiro, J.L., Sheth, A.P. (eds.) BPM 2006. LNCS, vol. 4102, pp. 65–80. Springer, Heidelberg (2006)
9. Foster, H., Uchitel, S., Kramer, J.: LTSA-WS: A Tool for Model-based Verification of Web Service Compositions and Choreography. In: Proc. of ICSE 2006, pp. 771–774. ACM Press, New York (2006)
10. Fu, X., Bultan, T., Su, J.: Analysis of Interacting BPEL Web Services. In: Proc. of WWW 2004, pp. 621–630. ACM Press, New York (2004)
11. ISO/IEC: LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807, ISO (1989)
12. Mateescu, R., Poizat, P., Salaün, G.: Adaptation of Service Protocols using Process Algebra and On-the-Fly Reduction Techniques. In: Bouguettaya, A., Krueger, I., Margaria, T. (eds.) ICSOC 2008. LNCS, vol. 5364, pp. 84–99. Springer, Heidelberg (2008)
13. Nezhad, H.R.M., Benatallah, B., Martens, A., Curbera, F., Casati, F.: Semi-automated adaptation of service interactions. In: Proc. of WWW 2007, pp. 993–1002. ACM, New York (2007)
14. Poizat, P., Salaün, G.: Adaptation of Open Component-based Systems. In: Bonsangue, M.M., Johnsen, E.B. (eds.) FMOODS 2007. LNCS, vol. 4468, pp. 141–156. Springer, Heidelberg (2007)
15. Salaün, G.: Generation of Service Wrapper Protocols from Choreography Specifications. In: Proc. of SEFM 2008, pp. 313–322. IEEE Computer Society, Los Alamitos (2008)
16. Salaün, G., Bordeaux, L., Schaerf, M.: Describing and Reasoning on Web Services using Process Algebra. International Journal of Business Process Integration and Management 1(2), 116–128 (2006)
17. Yellin, D.M., Strom, R.E.: Protocol specifications and component adaptors. ACM Transactions on Programming Languages and Systems 2(19), 292–333 (1997)

Quantitative Verification in Practice

Boudewijn R. Haverkort[1,2], Joost-Pieter Katoen[2,3], and Kim G. Larsen[4]

[1] Embedded Systems Institute, Eindhoven, The Netherlands
[2] University of Twente, Formal Methods and Tools, The Netherlands
[3] RWTH Aachen University, Software Modeling and Verification Group, Germany
[4] Center for Embedded Software Systems, Aalborg, Denmark

Abstract. Soon after the birth of model checking, the first theoretical achievements have been reported on the automated verification of quantitative system aspects such as discrete probabilities and continuous time. These theories have been extended in various dimensions, such as continuous probabilities, cost constraints, discounting, hybrid phenomena, and combinations thereof. Due to unremitting improvements of underlying algorithms and data structures, together with the availability of more advanced computing engines, these techniques are nowadays applicable to realistic designs. Powerful software tools allow these techniques to be applied by non-specialists, and efforts have been made to embed these techniques into industrial system design processes. Quantitative verification has a broad application potential — successful applications in embedded system design, hardware, security, safety-critical software, schedulability analysis, and systems biology exemplify this. It is fair to say, that over the years this application area grows rapidly and there is no sign that this will not continue. This session reports on applying state-of-the-art quantitative verification techniques and tools to a variety of industrial case studies.

T. Margaria and B. Steffen (Eds.): ISoLA 2010, Part II, LNCS 6416, p. 127, 2010.

Ten Years of Performance Evaluation for Concurrent Systems Using CADP

Nicolas Coste[1], Hubert Garavel[2], Holger Hermanns[2,3], Frédéric Lang[2], Radu Mateescu[2], and Wendelin Serwe[2]

[1] STMicroelectronics, 12, rue Jules Horowitz, BP 217, 38019 Grenoble, France
[2] Inria Grenoble – Rhône-Alpes, Inovallée, 655, av. de l'Europe, Montbonnot, 38334 Saint Ismier, France
{Hubert.Garavel,Frederic.Lang,Radu.Mateescu,Wendelin.Serwe}@inria.fr
[3] Saarland University, Dept. of Computer Science, 66123 Saarbrücken, Germany
hermanns@cs.uni-saarland.de

Abstract. This article comprehensively surveys the work accomplished during the past decade on an approach to analyze concurrent systems qualitatively and quantitatively, by combining functional verification and performance evaluation. This approach lays its foundations on semantic models, such as Imc (*Interactive Markov Chain*) and Ipc (*Interactive Probabilistic Chain*), at the crossroads of concurrency theory and mathematical statistics. To support the approach, a number of software tools have been devised and integrated within the Cadp (*Construction and Analysis of Distributed Processes*) toolbox. These tools provide various functionalities, ranging from state space generation (Cæsar and Exp.Open), state space minimization (Bcg_Min and Determinator), numerical analysis (Bcg_Steady and Bcg_Transient), to simulation (Cunctator). Several applications of increasing complexity have been successfully handled using these tools, namely the Hubble telescope lifetime prediction, performance comparison of mutual exclusion protocols, the Scsi-2 bus arbitration protocol, the Send/Receive and Barrier primitives of Mpi (*Message Passing Interface*) implemented on a cache-coherent multiprocessor architecture, and the xStream multiprocessor data-flow architecture for embedded multimedia streaming applications.

1 Introduction

The design of models suited for performance and reliability analysis is challenging due to complexity and size of the modeled systems, in particular for those with a high degree of irregularity. Traditional performance models like Markov chains and queueing networks are not easy to apply for large-sized systems, mainly because they lack hierarchical composition and abstraction means.

Therefore, various specification formalisms have been proposed, which enable systems to be modeled in a compositional, hierarchical manner. A prominent example of such specification formalisms is the class of process algebras, which provide abstraction mechanisms to treat system components as black boxes, making their internal implementation details invisible. Among the many process

T. Margaria and B. Steffen (Eds.): ISoLA 2010, Part II, LNCS 6416, pp. 128–142, 2010.

algebras proposed in the literature, LOTOS [1] has received much attention, due to its technical merits and its status of ISO/IEC International Standard. CADP (*Construction and Analysis of Distributed Processes*) is a widespread tool set for the design and verification of complex systems. CADP supports, among others, the process algebra LOTOS for specification, and offers various tools for simulation and formal verification, including equivalence checkers (bisimulations) and model checkers (temporal logics and modal μ-calculus). About a decade ago, CADP has been extended with performance evaluation capabilities, based on the IMC (*Interactive Markov Chain*) theory [2,3]. More recently, the IMC theory has been transposed into a discrete-time setting, leading to the IPC (*Interactive Probabilistic Chain*) theory [4]. These theories combine well with the approach behind CADP by integrating both, on the one hand, Markov chains and, on the other hand, classical process algebra and the underlying standard notion of LTS (*Labeled Transition System*). Over the years, the performance evaluation branch of CADP has gained maturity, new tools have been added, and many applications have been carried out with the toolbox. This paper provides a survey of the principal modeling and analysis ingredients, and applications of performance evaluation with CADP.

2 The Interactive Markov Chain Model

An IMC (*Interactive Markov Chain*) [3] is a state-transition graph with a denumerable state space, action-labeled transitions, as well as stochastic transitions (also called Markovian transitions). The latter are labeled with rates of exponential distributions. Actions are ranged over by a and b; the particular action τ models internal, i.e., unobservable activity, whereas all other actions model observable activities.

Definition 1 (Interactive Markov Chain). *An* IMC *is a tuple* $\mathcal{I} = (S, \mathcal{A}, \longrightarrow, \Longrightarrow, s_0)$ *where:*

- S *is a nonempty set of states with* initial state $s_0 \in S$,
- $\mathcal{A}$ *is a set of actions,*
- $\longrightarrow \subseteq S \times \mathcal{A} \times S$ *is a set of* interactive *transitions, and*
- $\Longrightarrow \subseteq S \times \mathbb{R}_{>0} \times S$ *is a set of* stochastic *transitions.*

An IMC is a natural extension of a standard LTS (*Labeled Transition System*), as well as of a CTMC (*Continuous-Time Markov Chain*): a standard LTS is an IMC with $\Longrightarrow = \emptyset$, while a CTMC is an IMC with $\longrightarrow = \emptyset$.

Behavioral interpretation. Roughly speaking, the interpretation of a stochastic transition $s \overset{\lambda}{\Longrightarrow} s'$ is that the IMC can switch from state s to s' within d time units with probability $1 - e^{-\lambda \cdot d}$. The positive real value λ thus uniquely identifies a negative exponential distribution. For state s, let $\mathbf{R}(s, s') = \sum\{\lambda \mid s \overset{\lambda}{\Longrightarrow} s'\}$ be the *rate* to move from s to state s'. If $\mathbf{R}(s, s') > 0$ for more than one state s', a competition between the transitions of s exists, known as the *race condition.*

The probability to move from such state s to a particular state s' within d time units, i.e., the stochastic transition $s \Longrightarrow s'$ wins the race, is given by:

$$\frac{\mathbf{R}(s,s')}{E(s)} \cdot \left(1 - e^{-E(s)\cdot d}\right),$$

where $E(s) = \sum_{s' \in S} \mathbf{R}(s,s')$ denotes the *exit rate* of state s. Intuitively, it states that after a delay of at most d time units (second term), the IMC moves probabilistically to a direct successor state s' with discrete branching probability $\mathbf{P}(s,s') = \mathbf{R}(s,s')/E(s)$.

An *internal* interactive transition is a τ-labeled interactive transition, also called τ-transition for short in the sequel, which plays a special role in an IMC. As a τ-transition is not subject to any interaction, it cannot be delayed. Thus, τ-transitions can be assumed to take place immediately. Now consider a state s with both a τ-transition and a stochastic transition. At the precise instant, when the IMC moves to s, the τ-transition can be taken immediately, but the probability that the stochastic transition executes immediately is zero. This justifies that τ-transitions take precedence over stochastic transitions, a property called the *maximal progress assumption*.

Definition 2 (IMC parallel composition). *Let* $\mathcal{I}_1 = (S_1, \mathcal{A}_1, \longrightarrow_1, \Longrightarrow_1, s_{0,1})$ *and* $\mathcal{I}_2 = (S_2, \mathcal{A}_2, \longrightarrow_2, \Longrightarrow_2, s_{0,2})$ *be* IMCs. *For a set of actions* A *such that* $\tau \notin A$, *the* parallel composition *of* $\mathcal{I}_1$ *and* $\mathcal{I}_2$ *wrt.* A *is defined by:*

$$\mathcal{I}_1 \,||_A\, \mathcal{I}_2 = (S_1 \times S_2, \mathcal{A}_1 \cup \mathcal{A}_2, \longrightarrow, \Longrightarrow, (s_{0,1}, s_{0,2}))$$

where $\longrightarrow$ *and* $\Longrightarrow$ *are defined as the smallest relations satisfying:*

1. $s_1 \xrightarrow{a}_1 s_1'$ *and* $s_2 \xrightarrow{a}_2 s_2'$ *and* $a \in A$ *implies* $(s_1, s_2) \xrightarrow{a} (s_1', s_2')$
2. $s_1 \xrightarrow{a}_1 s_1'$ *and* $a \notin A$ *implies* $(s_1, s_2) \xrightarrow{a} (s_1', s_2)$ *for any* $s_2 \in S_2$
3. $s_2 \xrightarrow{a}_2 s_2'$ *and* $a \notin A$ *implies* $(s_1, s_2) \xrightarrow{a} (s_1, s_2')$ *for any* $s_1 \in S_1$
4. $s_1 \overset{\lambda}{\Longrightarrow}_1 s_1'$ *implies* $(s_1, s_2) \overset{\lambda}{\Longrightarrow} (s_1', s_2)$ *for any* $s_2 \in S_2$
5. $s_2 \overset{\lambda}{\Longrightarrow}_2 s_2'$ *implies* $(s_1, s_2) \overset{\lambda}{\Longrightarrow} (s_1, s_2')$ *for any* $s_1 \in S_1$.

The first three constraints define a LOTOS-like parallel composition [1]: actions in A need to be performed by both IMCs simultaneously (first constraint), whereas actions not in A are performed autonomously (second and third constraint). According to the last two constraints, an IMC can delay independently. This differs from timed models such as timed automata, in which individual processes typically need to synchronize on the advance of time. Independent delaying is justified, because whenever two stochastic transitions with rates λ and μ are competing to be executed, then the remaining delay of the μ-transition after the λ-transition has been taken is exponentially distributed with rate μ, due to the memoryless property of exponential distributions.

To compare IMCs, notions of strong and branching bisimulation are used, which extend the notions defined on standard LTS in a conservative fashion, and also extend Markov chain lumpability [3]. Both relations are congruences for

parallel composition and other process algebraic operators. Therefore an IMC in a parallel composition can be replaced by an equivalent, but possibly smaller one, while preserving semantics.

Constraint-oriented specification of performance aspects. To evaluate the performance of a system using the IMC approach, one can insert delays — i.e., a probability distribution approximated arbitrarily closely by a terminating CTMC — into the standard LTS of the system. This insertion can be achieved following the constraint-oriented specification style [5], originally developed to support the early phases of system design. Put in a nutshell, constraints are viewed as separate processes (in our case, IMCs), and parallel composition is used to combine these constraints much in the same vein as logical conjunction.

To illustrate the constraint-oriented specification style applied to an IMC, consider an IMC $\mathcal{I}$ and let a and b be two successive actions in $\mathcal{I}$. To insert a delay approximated by a terminating CTMC Δ with a single final state, construct an IMC $\mathcal{I}_\Delta$ whose initial state contains a single outgoing transition labeled a to the initial state of Δ, and whose final state can only be reached from the final state of Δ by a transition labeled b. The resulting system is then obtained as $\mathcal{I} \,||_{\{a,b\}}\, \mathcal{I}_\Delta$.

3 The Interactive Probabilistic Chain Model

An IPC (*Interactive Probabilistic Chain*) [4] can be seen as a transposition of the IMC approach to a discrete time setting. Thus, in this section, we focus on the differences between the two models. An IPC is essentially a state-transition graph with a denumerable state space, action labeled transitions, and probabilistic transitions. As for an IMC, actions are ranged over by a and b, and the internal action is denoted τ.

Definition 3 (Interactive Probabilistic Chain). *An* IPC *is a tuple* $\mathcal{D} = (S, \mathcal{A}, \rightarrow, \rightsquigarrow, s_0)$ *where:*

- S *is a nonempty set of states with* initial state $s_0 \in S$,
- $\mathcal{A}$ *is a set of actions,*
- $\rightarrow \subseteq S \times \mathcal{A} \times S$ *is a set of* interactive *transitions, and*
- $\rightsquigarrow \subseteq S \times \,]0..1] \times S$ *is a set of* probabilistic *transitions, satisfying for each state* $s \in S$ *that the sum of probabilities of outgoing probabilistic transitions is equal to one, i.e.,* $(\forall s \in S)\ \sum_{s' \in S} \sum \{p \mid s \overset{p}{\rightsquigarrow} s'\} = 1$

Notice that the constraint on probabilistic transitions implies that each state has at least one outgoing probabilistic transition, which may be a self-loop with probability one, i.e., a transition of the form $s \overset{1}{\rightsquigarrow} s$.

An IPC is a natural extension of a standard LTS as well as of a DTMC (*Discrete-Time Markov Chain*): a standard LTS is an IPC with $\rightsquigarrow = \emptyset$, while a DTMC is an IPC with $\rightarrow = \emptyset$.

Behavioral interpretation. Executing a probabilistic transition takes exactly one time step. Any choice between probabilistic transitions is solved according

to the probability distribution. Self-loops with probability one express the *arbitrary waiting* property [6]: an IPC may be blocked waiting for a synchronization that is arbitrarily long (even infinitely) while still letting time advance. As for an IMC, internal interactive transitions take precedence over probabilistic transitions following the maximal progress assumption.

The parallel composition of the IPC model differs from the composition of the IMC model, because the memoryless property of exponential distributions does not apply to the IPC model.

Definition 4 (IPC parallel composition). *Let* $\mathcal{D}_1 = (S_1, \mathcal{A}_1, \rightarrow_1, \rightsquigarrow_1, s_{0,1})$ *and* $\mathcal{D}_2 = (S_2, \mathcal{A}_2, \rightarrow_2, \rightsquigarrow_2, s_{0,2})$ *be* IPCs. *For a set of actions* A *such that* $\tau \notin A$, *the* parallel composition *of* $\mathcal{D}_1$ *and* $\mathcal{D}_2$ *wrt.* A *is defined by:*

$$\mathcal{D}_1 \,||_A\, \mathcal{D}_2 = (S_1 \times S_2, \mathcal{A}_1 \cup \mathcal{A}_2, \rightarrow, \rightsquigarrow, (s_{0,1}, s_{0,2}))$$

where $\rightarrow$ *and* $\rightsquigarrow$ *are defined as the smallest relations satisfying:*

1. $s_1 \xrightarrow{a}_1 s_1'$ *and* $s_2 \xrightarrow{a}_2 s_2'$ *and* $a \in A$ *implies* $(s_1, s_2) \xrightarrow{a} (s_1', s_2')$
2. $s_1 \xrightarrow{a}_1 s_1'$ *and* $a \notin A$ *implies* $(s_1, s_2) \xrightarrow{a} (s_1', s_2)$ *for any* $s_2 \in S_2$
3. $s_2 \xrightarrow{a}_2 s_2'$ *and* $a \notin A$ *implies* $(s_1, s_2) \xrightarrow{a} (s_1, s_2')$ *for any* $s_1 \in S_1$
4. $s_1 \overset{p_1}{\rightsquigarrow}_1 s_1'$ *and* $s_2 \overset{p_2}{\rightsquigarrow}_2 s_2'$ *implies* $(s_1, s_2) \overset{p_1 p_2}{\rightsquigarrow} (s_1', s_2')$.

Again, the first three constraints define a LOTOS-like parallel composition [1]: actions in A need to be performed by both IPCs simultaneously (first constraint), whereas actions not in A are performed autonomously (second and third constraint). Contrary to the IMC model, the last constraint forces IPCs to synchronize on the advance of time, similar to discrete-timed models. Notice that the probability of the synchronized transition is precisely the product of the probabilities to take each transition separately.

Strong and branching bisimulations for the IPC model are defined similar to the corresponding bisimulations for the IMC model. As for the IMC model, strong and branching probabilistic bisimulations are congruences wrt. parallel composition and other operators.

4 CADP Tools for Extended Markovian Models

A key benefit of the IMC/IPC approach is its compatibility with most existing process calculi, without requiring syntactic and semantic extensions to handle stochastic/probabilistic features. Consequently, it is possible to reuse or extend tools already available in the CADP toolbox, rather than developing a whole set of new tools.

In this section, we present the various CADP tools handling extended Markovian models, i.e., state-transition models that combine features from standard LTS and discrete-time and continuous-time Markov chains. A number of these tools (namely BCG_STEADY, BCG_TRANSIENT, and DETERMINATOR [7]) have been developed specifically to support performance evaluation. Other tools already existed but were already compatible or have been extended to be compatible with the proposed approach. Contrary to most CADP tools that operate

on standard LTSs, these tools operate on extended Markovian models, encoded as probabilistic/stochastic extensions of LTSs. Precisely, an extended Markovian model is an LTS, where all transition labels must be one of the following, a representing either an observable action or the internal action τ:

- a rate "λ", called a *stochastic transition*, or
- a pair "$a; \lambda$" of an action and a rate, called a *labeled stochastic transition*, or
- a probability "p" with $p \in [0, 1]$, called a *probabilistic transition*, or
- a pair "$a; p$" of an action and a probability, called a *labeled probabilistic transition*, or
- an action "a", called an *interactive transition* (also called *ordinary transition* in the CADP documentation).

Note that extended Markovian models are sufficiently general to include IMC, IPC, and various other probabilistic[1] and stochastic models[2]. In CADP, extended Markovian models are represented explicitly in the BCG format, or implicitly using the OPEN/CÆSAR environment [8].

4.1 State Space Generation Using CÆSAR.ADT and CÆSAR

CÆSAR.ADT [9] and CÆSAR [10,11] are two complementary LOTOS to C compilers, the former for the data part, the latter for the behavior part of LOTOS. The C code generated by these compilers is then used by other CADP tools for various purposes: simulation, random execution, on-the-fly verification, test generation, etc. Additionally, CÆSAR can generate the LTS corresponding to a LOTOS specification, if of finite size. This LTS is encoded in the BCG format and can be verified using bisimulations and/or model checking of μ-calculus or temporal logic formulas.

A LOTOS specification, whose functional correctness has been already verified, can be enriched with stochastic information as follows: the user must insert in the LOTOS specification, at each place where a Markov delay or a probabilistic transition should occur, a new LOTOS gate λ_i.

After all gates λ_i have been inserted in the LOTOS specification, CÆSAR and CÆSAR.ADT are invoked as usual to generate the corresponding LTS. This LTS is then turned into an extended Markovian model (still encoded in the BCG format) by replacing each transition labeled with λ_i by a stochastic or probabilistic transition of known numerical value. This can be achieved using the BCG_LABELS tool of CADP, which performs hiding and/or renaming on the labels attached to the transitions of a BCG file, according to a set of regular expression and substitution patterns specified by the user.

[1] Discrete Time Markov Chains, Discrete Time Markov Reward Models, Alternating Probabilistic LTS, Discrete Time Markov Decision Processes, Generative Probabilistic LTS, Reactive Probabilistic LTS, Stratified probabilistic LTS.

[2] Continuous Time Markov Chains, Continuous Time Markov Reward Models, Continuous Time Markov Decision Processes, Timed Processes for Performance Models, Performance Evaluation Process Algebra Models, Extended Markovian Process Algebra Models.

4.2 Compositional Verification Using EXP.OPEN

Exp.Open [12] is a compositional verification tool for on-the-fly exploration of a graph corresponding to a *network of communicating automata* (represented as a set of Bcg files). These automata are composed together in parallel using either algebraic operators (as in the Ccs, Csp, Lotos, and μCrl process algebras), "*graphical*" operators (as in E-Lotos and Lotos NT), or synchronization vectors (as in the Mec and Fc2 tools). Additional operators are available to hide and/or rename labels (using regular expressions), to cut certain transitions, and to give priority of certain transitions over others.

To address state explosion, Exp.Open is equipped with partial order reduction techniques that preserve either deadlocks, weak traces, or branching bisimulation. For an extended Markovian model, Exp.Open uses the maximal progress property to cut all stochastic transitions in choice with τ-transitions (see Section 2).

Branching bisimulation (and both its stochastic and probabilistic variants) as well as trace equivalence, weak trace equivalence, safety equivalence, observational equivalence, and strong bisimulation are congruences for all Exp.Open operators except priorities. This is a key property for compositional verification, which extends the congruence property mentioned in Section 2 to the more general network of communicating automata model.

4.3 Bisimulation Reduction Using BCG_MIN

The Bcg_Min tool enables graph minimization modulo strong bisimulation or branching bisimulation, extended to the probabilistic and stochastic cases. Thus, it can be used for functional verification and performance evaluation. Bcg_Min accepts as input three kinds of extended Markovian models, all encoded in the Bcg graph format:

- either a *standard* Lts, containing only interactive transitions,
- or a *probabilistic* model, containing only interactive, probabilistic, or labeled probabilistic transitions (e.g., an Ipc),
- or a *stochastic* model, containing only interactive, stochastic, or labeled stochastic transitions (e.g., an Imc).

A new version 2.0 of Bcg_Min implementing a *signature-based partition refinement* algorithm [13] generalized to stochastic and probabilistic bisimulations has been released in 2010. This new version brings spectacular performance improvements with respect to the previous version. In particular, the reduction modulo stochastic and probabilistic bisimulations of a test base consisting of 1335 probabilistic and stochastic models was 540 times faster, and up to 8500 times faster for one particular model.

4.4 Nondeterminism Elimination Using DETERMINATOR

The Determinator tool [7] eliminates stochastic nondeterminism in extended Markovian models on the fly. It takes as input an extended Markovian model $\mathcal{M}$

(encoded in the BCG graph format) containing probabilistic and/or stochastic transitions and attempts at translating $\mathcal{M}$ to a CTMC (*Continuous Time Markov Chain*), i.e., an LTS (encoded in the BCG graph format) that contains (labeled) stochastic transitions only. The aim is therefore to eliminate the interactive and probabilistic transitions, while keeping the stochastic information present in $\mathcal{M}$.

Because the translation is impossible in the general case, DETERMINATOR only handles models verifying a sufficient condition (*well-specified check*) [14], which guarantees that the resulting CTMC is unique, independently of the way nondeterministic choices are resolved. This enables to eliminate the nondeterminism without modifying the stochastic behavior of the system.

The translation algorithm implemented in DETERMINATOR is a variant of the one presented in [14]. It works on the fly using the functionalities of the OPEN/CÆSAR environment [8].

Although the BCG_MIN tool also enables, in some way, to eliminate nondeterminism (by doing more general reductions based on the concept of *lumpability*), it differs from DETERMINATOR: BCG_MIN does not handle the case of LTS containing both probabilistic and stochastic transitions and it does not eliminate interactive transitions.

4.5 Numerical Analysis Using BCG_STEADY and BCG_TRANSIENT

The BCG_STEADY and BCG_TRANSIENT tools take as input a CTMC (*Continuous Time Markov Chain*), a DTMC (*Discrete Time Markov Chain*), or even any extended Markovian model without interactive transitions meeting the restrictions detailed on the respective manual pages[3]. The input model is represented internally as a (sparse) matrix indexed by states that is used by numerical algorithms for performance evaluation:

- BCG_STEADY computes, for each state s, the probability to be in s on the long run, i.e., in the equilibrium or "steady state". These probabilities are computed iteratively using a Gauss-Seidel algorithm [15].
- BCG_TRANSIENT computes, for each state s and for each time instant t in a discrete set provided by the user, the probability to be in s at instant t. This computation uses the uniformization algorithm [16,15] and the Fox-Glynn [17] method to approximate Poisson probabilities.

Based on the computed probabilities, BCG_STEADY and BCG_TRANSIENT can also compute the corresponding transition throughputs, i.e., the average number of transition executions per time unit. These measures can provide important high-level information to assess the system performance, reliability or productivity, such as operation latencies (see Section 6).

BCG_STEADY and BCG_TRANSIENT generate output in the standard CSV (*Comma Separated Values*) format used by mainstream data processing tools, including EXCEL and GNUPLOT.

[3] Available at `http://vasy.inria.fr/cadp/man/bcg_steady.html` and `http://vasy.inria.fr/cadp/man/bcg_transient.html`

The BCG_MIN (see Section 4.3) and DETERMINATOR (see Section 4.4) tools might be required to determinize an extended Markovian model to be given as input to BCG_STEADY or BCG_TRANSIENT.

4.6 On-the-Fly Steady-State Simulation Using CUNCTATOR

CUNCTATOR is an on-the-fly steady-state simulator for stochastic models. It takes an extended Markovian model (represented using the OPEN/CÆSAR environment) as input, applies any user-defined hiding and renaming operations, and explores a random execution sequence on the fly. Exploration is aborted whenever an observable interactive transition or a probabilistic transition is encountered. During the exploration, the tool sums up the virtual time elapsed in the states, determined according to the rates of their outgoing stochastic transitions. The simulation terminates when either the virtual time, or the length of the simulation sequence (number of transitions) reaches a maximum value specified by the user. Upon termination, the throughputs of the labeled stochastic transitions of interest are displayed, together with additional information (number of τ-transitions encountered, presence of nondeterminism, etc.). The context reached at the end of a simulation can be saved in order to restart subsequent simulations from this context. This mechanism is useful for implementing convergence criteria (e.g., based on confidence intervals) by allowing to perform in linear time a series of increasingly long simulations, each one being the prefix of the subsequent ones.

When a nondeterministic state (with at least two outgoing τ-transitions) is reached, CUNCTATOR explores one of the outgoing τ-transitions. The choice of this transition can be made currently according to three scheduling policies: the first τ-transition encountered, the last one, or a randomly chosen one. When a simulation has encountered nondeterministic states, the user has the possibility of launching other simulations using different scheduling policies in order to obtain more insight about the stochastic behavior of the model.

CUNCTATOR stores in memory only the last state of the simulation sequence, thus consuming only a small amount of memory, independent from the length of the simulation sequence and from the size of the CTMC. Compared to BCG_STEADY, which computes exact throughputs in a CTMC represented as a BCG file, CUNCTATOR consumes less memory but may require a longer execution time in order to achieve the same accuracy.

5 Additional Tools for Interactive Probabilistic Chains

To support the IPC model, we took advantage of the open architecture of CADP and prototyped additional tools (4,900 lines of C code) that are not yet integrated into CADP. Together with CADP, these tools support the compositional construction of an IPC (following a constraint oriented style) and the computation of latency distributions.

Parallel composition of IPCs. Because the parallel composition of LOTOS and those supported by EXP.OPEN are incompatible with the parallel composition of the IPC model, a different parallel composition is required.

The IPC_COMPOSE tool takes as input a network of communicating IPCs (represented as a set of BCG files, composed in parallel using LOTOS parallel compositions) and produces the corresponding IPC.

Constraint oriented delay insertion. Ideally, one would like to use IPC_COMPOSE to insert delays into an LTS, following an approach similar to the constraint-oriented style supported by CADP for the IMC model. Unfortunately, this approach may lead to non-determinism in the case a choice between two actions depends whether a delay has elapsed or not, because in the IPC model, delays in two concurrent processes may expire at exactly the same time instant. Notice that this cannot happen in an IMC model, because the probability for two exponential distributions to expire at the same time is zero.

Thus, we developed the IPC_INSERT tool, which takes as input an LTS (represented as a BCG file) and a probabilistic distribution for a single delay (also represented as a BCG file) and produces the corresponding IPC.

In practice, the IPC of a complete system is obtained by first generating the LTS of each sequential interacting subcomponent, then inserting delays into these components (using IPC_INSERT), and finally computing their parallel composition (using IPC_COMPOSE).

Computation of latency distributions. The IPC_DISTRIBUTION tool takes as input a deterministic IPC and two actions a and b and computes the long-run average probability distribution of the latency between a and b [4].

6 Applications

In this Section, we report about five case studies that have been tackled using the proposed methodology and its associated tools.

6.1 The Hubble Telescope Lifetime

The first case study with the performance evaluation tools provided by CADP was the Hubble space telescope example described in [18].

A 50-line LOTOS specification was developed for this example; it consists of seven concurrent processes: one controller process and one process for each of the six Hubble stabilizing units (i.e., gyroscopes that may fail as time elapses). This LOTOS specification is parameterized by three constants λ, μ, and ν representing the average lifetime of a gyroscope, the time needed to stop all Hubble equipments and the time needed to replace all gyroscopes.

Using the CÆSAR and CÆSAR.ADT compilers, an LTS (877 states, 3341 transitions) was generated; this LTS was then turned into an IMC by replacing by their actual values the λ, μ, and ν parameters present in the transition labels. Then, the BCG_MIN tool was used to minimize this IMC modulo stochastic branching

minimization, leading to a CTMC (9 states, 12 transitions) that is small enough to be verified visually. Finally, the BCG_TRANSIENT tool was used to compute failure probabilities at various time instants, thus giving an estimation of the Hubble telescope lifetime.

6.2 Mutual Exclusion Protocols

Recently, several mutual exclusion protocols for shared memory computers have been analyzed using CADP [19]. These protocols are an essential building block of concurrent systems, required whenever a shared resource has to be protected against concurrent non-atomic accesses.

For a system with two processes communicating through up to seven shared variables, 24 mutual exclusion protocols have been described in LOTOS NT and translated automatically into LOTOS. The LTS of each protocol was transformed into an IMC (from 89 states and 130 transitions up to 31,222 states and 43,196 transitions), which was then reduced using BCG_MIN. Finally, the throughput of the accesses to the shared resource was computed using BCG_STEADY.

Besides comparing the performance of the various protocols, this study gave insight about the performance impact of changing the rates for accessing the shared resource. The results corroborate functional properties, in particular asymmetric behavior, i.e., overtaking of one process by the other.

6.3 The SCSI-2 Bus Arbitration Protocol

Another case study [20] with the performance evaluation tools of CADP was a storage system developed by Bull in the early 90's. This system consisted of a disk controller and (at most) seven disks connected by a SCSI-2 (*Small Computer System Interface*) bus. During the testing phase, Bull engineers discovered potential starvation problems for disks having a smaller SCSI number than the disk controller.

The storage system was formally described in LOTOS, and it was found that the multiway rendezvous of LOTOS was most appropriate to model the SCSI-2 bus arbitration protocol concisely. This LOTOS description was submitted to model checking verification using CADP enabling to reproduce the starvation problem automatically.

Then, the LOTOS description was turned into a performance model by inserting at various places two stochastic delays λ (load stress imposed on the disk controller) and μ (average time for a disk to service a transfer request) and by adding an auxiliary LOTOS process modeling a phase-type distribution (Erlang law with parameter ν) between two SCSI-2 bus arbitration periods.

The corresponding LTS was generated using CÆSAR.ADT and CÆSAR, and then minimized using BCG_MIN modulo branching bisimulation after hiding and/or renaming actions unrelated to performance. Due to the use of compositional state space generation techniques, state space explosion does not occur (the largest automaton produced has 56,169 states and 154,752 transitions only).

Then, the λ, μ, and ν parameters were replaced with a series of numerical constants; for each instantiation, the IMC obtained was minimized using BCG_MIN modulo stochastic branching bisimulation, yielding a CTMC in which nondeterminism had vanished; finally, the BCG_STEADY tool was applied to each such CTMC to compute the equilibrium (steady-state) probabilities for each state, as well as throughputs for relevant actions, enabling a precise study of unfairness in the SCSI-2 system under heavy load.

6.4 The MPI Send/Receive and Barrier Primitives

In the context of the MULTIVAL project together with Bull, we studied an implementation of MPI (*Message Passing Interface*) to be run on FAME2 (*Flexible Architecture for Multiple Environments*), a CC-NUMA multiprocessor architecture developed at Bull for teraflop mainframes and petaflop computing.

In a first study [21] we focused on the *ping-pong* MPI benchmark, with the goal of estimating the latency of *send/receive* primitives on FAME2 machines. Several configurations of the benchmark were specified in LOTOS, by considering three interconnection topologies, two implementations of the *send/receive* primitives SR_1 and SR_2 (based on linked lists with locks and on lock-free buffers) and two cache coherency protocols A and B (in which a variable written by a process becomes either owned by that process, or shared between that process and the previous owner). The performance analysis was carried out by extending the LOTOS specification with exponential distributions and applying the BCG_MIN, DETERMINATOR, and BCG_STEADY tools of CADP to compute the *send/receive* latency. The computed latencies were close (down to 9% of difference) to the experimental measures, even for the relatively simple model considered. This analysis also enabled to estimate the number of cache misses corresponding to each instruction, showing that the most efficient configuration is given by the SR_2 *send/receive* implementation and the cache coherency protocol A.

We applied the same approach to study five protocols implementing the *barrier* primitive of MPI (*centralized*, *combining*, *tournament*, *dissemination*, and *tree-based*). Using EXP.OPEN, the final Markov chain was generated compositionally for the centralized barrier with six processes and the tree-based barrier with four processes, and computed the latencies of barrier traversals using BCG_STEADY. The remaining protocols, which have prohibitively large state spaces, were analyzed by simulation using CUNCTATOR with the confidence interval criterion for convergence (automated using the save/restore mechanism), for configurations containing up to four processes. The throughputs obtained by simulation were close (less than 5%) to those computed by BCG_STEADY.

6.5 The xSTream Data-Flow Architecture

In the context of the MULTIVAL project together with STMicroelectronics, we studied XSTREAM, a multiprocessor data-flow architecture for high performance embedded multimedia streaming applications. In this architecture, computation nodes (e.g., filters) communicate using XSTREAM queues connected by a NOC

(*Network on Chip*). An xSTream queue generalizes a bounded Fifo queue in two ways: it provides additional primitives (such as *peek* to consult items in the middle of the queue, which is not possible with the standard *push*/*pop* primitives of Fifo queues), and a *backlog* (extra memory) to allow the increase of the queue size when the queue overflows.

Our performance evaluation study [4] aimed at predicting throughput and latency of communication between xSTream queues. For us, a key challenge is to combine probabilistic/stochastic information (e.g., the rates at which xSTream applications push and pop elements in and out of the queues) with precise timing information (e.g., memory access time). We studied the performance impact of the flow-control protocol, which ensures that every message emitted into the Noc can be received, i.e., leave the Noc. By enriching a functional Lotos model with probabilistic delays, we compositionally constructed an Ipc of a system of two data streams sharing the Noc (3205 states and 4630 transitions; before hiding and minimization, the Ipc contained 539,302 states and 1,412,168 transitions, and the largest intermediate Ipc contained 46,940,161 states and 198,490,980 transitions). The performance measures obtained with the prototype tools presented in Section 5 justified the relevance of the flow-control protocol. On the one hand, without the flow-control protocol, increasing the latency of one stream also increases the latency of the other stream, because the slow stream might fill the buffers in the Noc (functional verification even showed the possibility of a deadlock for particular kinds of applications). On the other hand, with the flow-control protocol, increasing the latency of one stream even reduces the latency of the other stream.

7 Conclusion and Future Work

This paper has given a survey of foundations, methodology, tool components, and applications of the Cadp approach to compositional performance evaluation. The Ipc and the Imc models have very close conceptual roots, and one can view an Ipc as a clock-ticked version of an Imc and, vice versa, one can view an Imc as the continuous time limit of an Ipc, with clock intervals tending to zero. A recent proposal [22] introduces a model that integrates both worlds in one, and develops the basic compositional theory, along the lines of Imc for this model. It is interesting to see how the available tool support can be extended to this setting.

Recent applications of Cadp in large industrial projects are very promising, but are also fostering the development of new and improved analysis support. This opens two challenging directions. First of all, the Cunctator tool opens an analysis avenue, based on discrete-event simulation, that does not suffer from the state space explosion, is straightforward to parallelize, and can support distributions that are not restricted by the Markov property. We are exploring this avenue in relation to discrete-event simulation activities revolving around the Modest language and tool [23].

The analysis performed with CADP is an instance of the general theme of combining performance evaluation and model checking [24]. An interesting research direction concerns recent advances in model checking a general IMC. So far, IMC analysis with CADP is limited to cases where the branching bisimulation quotient — obtained with BCG_MIN — is free of nondeterminism. With the advances reported in [25] this restriction is — at least in principle — obsolete, but an implementation of this technique inside CADP is still to be done.

References

1. ISO/IEC: LOTOS — a formal description technique based on the temporal ordering of observational behaviour. International Standard 8807, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, Genève (September 1989)
2. Hermanns, H., Katoen, J.P.: Automated compositional Markov chain generation for a plain-old telephone system. Science of Computer Programming 36(1), 97–127 (2000)
3. Hermanns, H.: Interactive Markov Chains. LNCS, vol. 2428. Springer, Heidelberg (2002)
4. Coste, N., Hermanns, H., Lantreibecq, E., Serwe, W.: Towards performance prediction of compositional models in industrial gals designs. In: Bouajjani, A., Maler, O. (eds.) Computer Aided Verification. LNCS, vol. 5643, pp. 204–218. Springer, Heidelberg (2009)
5. Vissers, C.A., Scollo, G., van Sinderen, M., Brinksma, E.: Specification styles in distributed systems design and verification. Theoretical Computer Science 89(1), 179–206 (1991)
6. Hansson, H.A.: Time and Probability in Formal Design of Distributed Systems. Elsevier Science Inc., Amsterdam (1994)
7. Hermanns, H., Joubert, C.: A set of performance and dependability analysis components for CADP. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 425–430. Springer, Heidelberg (2003)
8. Garavel, H.: Open/Cæsar: An open software architecture for verification, simulation, and testing. In: Steffen, B. (ed.) TACAS 1998. LNCS, vol. 1384, pp. 68–84. Springer, Heidelberg (1998)
9. Garavel, H.: Compilation of LOTOS abstract data types. In: Vuong, S.T. (ed.) Proceedings of the 2nd International Conference on Formal Description Techniques FORTE 1989, Vancouver B.C., Canada, pp. 147–162. North Holland, Amsterdam (December 1989)
10. Garavel, H., Sifakis, J.: Compilation and verification of LOTOS specifications. In: Logrippo, L., Probert, R.L., Ural, H. (eds.) Proceedings of the 10th International Symposium on Protocol Specification, Testing and Verification, Ottawa, Canada. IFIP, pp. 379–394. North Holland, Amsterdam (June 1990)
11. Garavel, H., Serwe, W.: State space reduction for process algebra specifications. Theoretical Computer Science 351(2), 131–145 (2006)
12. Lang, F.: Exp.open 2.0: A flexible tool integrating partial order, compositional, and on-the-fly verification methods. In: Romijn, J.M.T., Smith, G.P., van de Pol, J. (eds.) IFM 2005. LNCS, vol. 3771, pp. 70–88. Springer, Heidelberg (2005)
13. Blom, S., Orzan, S.: Distributed state space minimization. Springer International Journal on Software Tools for Technology Transfer (STTT) 7(3), 280–291 (2005)

14. Deavours, D.D., Sanders, W.H.: An efficient well-specified check. In: Proceedings of the 8th International Workshop on Petri Nets and Performance Models PNPM 1999, Zaragoza, Spain, pp. 124–133. IEEE Press, Los Alamitos (1999)
15. Stewart, W.J.: Introduction to the Numerical Solution of Markov Chains. Princeton University Press, Princeton (1994)
16. Jensen, A.: Markov chains as an aid in the study of markov processes. Skand. Aktuarietidskrift 3, 87–91 (1953)
17. Fox, B.L., Glynn, P.W.: Computing Poisson probabilities. Communications of the ACM 31(4), 440–445 (1987)
18. Hermanns, H.: Construction and verification of performance and reliability models. Bulletin of the EATCS 74, 135–154 (2001)
19. Mateescu, R., Serwe, W.: A study of shared-memory mutual exclusion protocols using CADP. In: Kowalewski, S., Roveri, M. (eds.) FMICS 2010. LNCS, vol. 6371, pp. 180–197. Springer, Heidelberg (2010)
20. Garavel, H., Hermanns, H.: On combining functional verification and performance evaluation using CADP. In: Eriksson, L.-H., Lindsay, P.A. (eds.) FME 2002. LNCS, vol. 2391, pp. 410–429. Springer, Heidelberg (2002)
21. Chehaibar, G., Zidouni, M., Mateescu, R.: Modeling multiprocessor cache protocol impact on mpi performance. In: Proceedings of the 2009 IEEE International Workshop on Quantitative Evaluation of Large-Scale Systems and Technologies QuEST 2009, Bradford, UK. IEEE Computer Society Press, Los Alamitos (2009)
22. Eisentraut, C., Hermanns, H., Zhang, L.: On probabilistic automata in continuous time. In: Proceedings of the 25th Annual IEEE Symposium on Logic in Computer Science, LICS 2010. IEEE Computer Society, Los Alamitos (2010)
23. Bohnenkamp, H.C., D'Argenio, P.R., Hermanns, H., Katoen, J.P.: MoDeST: A compositional modeling formalism for hard and softly timed systems. IEEE Transactions on Software Engineering 32(10), 812–830 (2006)
24. Baier, C., Haverkort, B., Hermanns, H., Katoen, J.P.: Performance evaluation and model checking join forces. Communications of the ACM 53(9), 76–85 (2010)
25. Zhang, L., Neuhäußer, M.R.: Model checking interactive markov chains. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 53–68. Springer, Heidelberg (2010)

Towards Dynamic Adaptation of Probabilistic Systems

S. Andova[1], L.P.J. Groenewegen[2], and E.P. de Vink[1]

[1] Formal Methods, TU Eindhoven, The Netherlands
[2] FaST Group, LIACS, Leiden University, The Netherlands

Abstract. Dynamic system adaptation is modeled in the coordination language Paradigm as coordination of collaborating components. A special component McPal allows for addition of new behavior, of new constraints and of new control in view of a new collaboration. McPal gradually adapts the system dynamics. It is shown that the approach also applies to the probabilistic setting. For a client-server example, where McPal adds, step-by-step, probabilistic behavior to deterministic components, precise modeling of changing system dynamics is achieved. This modeling of the transient behavior, spanning the complete migration range from as-is collaboration to to-be collaboration, serves as a stepping stone to quantitative analysis of the system during adaptation.

1 Introduction

Many systems today are affected while running by changes in their operational environment, while they cannot be shutdown to be updated and restarted again. Instead, dynamic adaptive systems must be able to manage adaptation steps on-the-fly to accommodate a new plan. Dynamic adaptive systems usually consist of interactive components, architecturally organized. However, system adaptation requires proper coordination. A carefully chosen coordination should guarantee that, during the adaptation, the system functionality is neither interrupted nor disturbed, and non-functional quantities, though possibly changing, should not exceed allowed bounds.

The coordination language Paradigm has been shown suitable to model dynamic adaptation [1,2] without the need of quiescence, i.e. no component has to be isolated from the system before being changed. In Paradigm, a system architecture is organized along specific collaboration dimensions, called partitions. A partition of a component specifies various phases the component goes through when a protocol is executed. Phases are temporarily valid constraints on ongoing component behaviour. In the protocol, at a higher layer in the architecture, the component participates via its role, an abstract representation of the phases. A protocol coordinates the phase transfers for the components involved. Progress within a phase is completely local to the component. In fact, the use of phase transfer instead of state transfer, where phases allow components to pursue their local dynamics, is the key concept of Paradigm. This makes it possible to model architectural changes and, at the same time, to model behavioural changes per

T. Margaria and B. Steffen (Eds.): ISoLA 2010, Part II, LNCS 6416, pp. 143–159, 2010.

component and per collaboration. In [4] an encoding of Paradigm models for the mCRL2 model checker is presented. The connection is exploited in [2]: (*i*) to verify that the system under adaptation indeed migrates from original to new behaviour; (*ii*) to perform a qualitative analysis of the adaptation itself. Such a formal analysis is relevant for detecting conflicts and revealing inconsistencies, in particular in case of multiple, simultaneous adaptation, guided by different change managers.

In this paper we extend the approach of modeling system adaptation with Paradigm and subsequent model checking of transitional properties and invariants by considering Markov decision processes (MDPs) instead of state-transition diagrams (STDs). We revisit our example of a critical section problem with four clients and one server. Thus, following Paradigm's methodology, the source and target behaviours are modeled as collaborative MDPs, which are STDs in the degenerate case. An adaptation strategy is given as well. Starting from deterministic round-robin servicing we aim to evolve to a client-to-serve selection based on a probability distribution. By guiding components phase by phase, probabilistic behavior is added gradually to deterministic components. They smoothly migrate from the source model to the target model. As mCRL2 does not allow probabilities, we now encode the whole adaptation trajectory in Prism [12]. The Prism specification consists of several modules, one for each client component and one for the server component. Within each module, both the detailed behaviour of the component is captured as well as the more global phase constraints and transfers. In fact, dynamic constraints, essential in Paradigm for the coordination of collaboration, can straightforwardly be specified with Prism via guards. So-called consistency rules that enforce multi-party synchronization in Paradigm at the level of phases, are distributively encoded by reactive commands sharing a label. Here, the specification language of the model checker fits hand in glove with the component interaction mechanism of Paradigm. We were able to generate a complete model of the dynamic adaptation, on which further qualitative and quantitative analysis on the transitional behaviour of the system is conducted. For instance, it is possible to guarantee mutual exclusion during adaptation. As for the quantitative properties it is, for instance, possible to compute the expected time needed for the system to migrate and to calculate the maximal waiting time for service during migration.

Related work. Most of the existing approaches, [8,6,9,16] to mention a few of them, focus on adaptive software architectures, where functionalities, considered as black boxes, are connected via ports. Following [13], new behavior is introduced by replacing an existing component by a new version. However, a component can only be removed if it is quiet and all affected components are passive. Thus, the actual adaptation is mainly achieved by reorganizing the architecture. [1] and [15] are the first efforts to analyze system adaptation with model checking.

Though recognized as an important issue and challenge [14], formal analysis of transitional behaviour of dynamic adaptive systems has triggered attention only recently. The approach of [18,19] is closest to ours: Cheng et al. manage to

model and formally analyse behavioural adaptation through weakening the need for quiescence. Their formal modeling uses Petri Nets and automata. In this way, functional properties expressed in an LTL-based logic, can be formally verified. A drawback is that different adaptation trajectories cannot be combined in a single model; also, adaptation is not being coordinated. In none of the approaches mentioned quantitative analysis is addressed.

To the best of our knowledge, none of the existing approaches supports modeling and property analysis of dynamic graceful adaptation without quiescence, neither qualitative nor quantitative. However, for simple value adaptation, as for instance in [17], service reconfiguration is addressed using quality description parameters to determine potential target configurations. The supporting verification framework includes a model checker, e.g., to verify reachability of configurations. The AADL modeling language of [7] supports dynamic reconfiguration of component connections. The language is rather expressive, allowing to specify timed, stochastic as well as hybrid systems. It is supported by a verification environment, including MRMC for model checking quantitative properties.

Organization of the paper. Section 2 introduces Paradigm by example, discussing the central notions for the deterministic version of the client-server system. A probabilistic service policy is modeled in Section 3, the migration from round-robin to probabilistic service is covered in Section 4. The encoding in Prism and further analysis of the adaptation are discussed in Section 5. The last section wraps up with conclusions and future work.

2 As-Is Situation: Deterministic Round Robin Service

This section presents a first variant of a Client-Server system: one Server component and four Client components, with merely deterministic behaviour. The five components collaborate on the basis of a round robin scheme. We shall refer to this variant as *as-is*.

Coordination language Paradigm can specify coordination solutions for foreseen as-is collaborations [3,5,4], for originally unforeseen to-be collaborations as well as for migration, i.e. ongoing but smoothly changing collaboration during adaptation from as-is to to-be [2,11]. To explain how, we first look briefly at Paradigm's coordination specification through the example of the Client-Server system. Second and briefly too, we introduce Paradigm's special component McPal, not influencing the system at all (as yet), but present in view of later system migration. The references give more technical background.

Paradigm has five basic notions: STD, phase, (connecting) trap, role and consistency rule. (Definitions are in [5,3], semantics in [5].) Figure 1 visualizes four of the notions for the Clients of the as-is system. Component behaviour is specified by STDs, state-transition diagrams. Figure 1a gives the STD for each Client component, in UML style. It says, Client_i starts in state Out and has cyclic behaviour, forever visiting its five states by subsequently taking actions enter, choose, explain, thank, leave. The idea is, by being in state Busy, a Client should exclusively occupy the one Server. In a nut-shell, this requirement lies

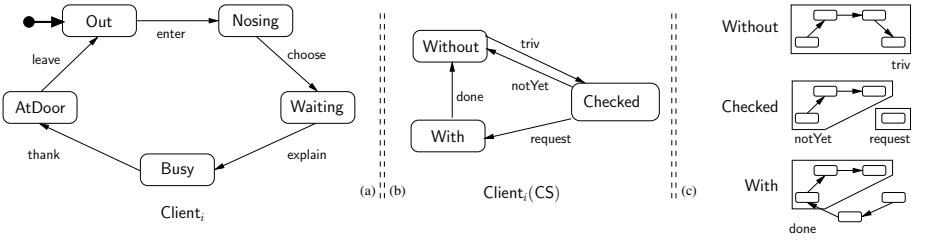


Fig. 1. (a) Client$_i$ STDs, (b) their CS role dynamics by (c) phase/trap constraints

at the basis of the critical section collaboration (CS) of the as-is system. The CS collaboration should provide suitable, simultaneous constraints on the Client behaviours, such that (i) never two or more Clients are in Busy at the same time; (ii) after arrival in Waiting, permission to visit Busy will be given sooner or later, as by being in Waiting a Client is asking for the permission.

Within Paradigm, a component participating in a collaboration does not contribute to the collaboration via its STD behaviour directly. Instead, the component contributes via a so-called *role*, being a different STD for the component, exhibited at a more global level. Figure 1b specifies role Client$_i$(CS) that Client$_i$ contributes to the CS collaboration. States of role Client$_i$(CS) are so-called *phases* of the Client$_i$ STD (Figure 1c): temporarily valid behavioural *constraints imposed* on the STD Client$_i$. Any current role state (phase) has as semantics: it keeps the behaviour of the STD it is a role of, *constrained to that phase*. Figure 1b and, correspondingly, Figure 1c mention three phases: Without, Checked, With. Here, Without prohibits a Client to be in Busy; With permits a Client to enter and to leave Busy once; Checked is an interrupted form of Without, to see whether a Client asks permission for entering Busy. In Figure 1c each phase is additionally decorated with one or more polygons, each polygon grouping states of that phase into a set. Polygons visualize so-called *traps*: a trap, as set of states, once entered, cannot be left as long as the phase remains the current role state. A trap serves as a guard for a phase transfer. Therefore, traps label transitions in a role STD, cf. Figure 1b. If all states in a trap serve as starting states of the next phase, the trap is called *connecting from* the one phase to the next.

Thus, role Client$_i$(CS) behaviour, Figure 1b, expresses possible sequences of phase transfers: the phase transfer from Without to Checked is generally possible, as trap triv is always connecting from Without to Checked; the step from Checked to With is only possible if connecting trap request has been entered, which means, if Client$_i$ asks the permission in state Waiting; otherwise, via connecting trap notYet, the phase transfer is from Checked back to Without; the phase transfer from With to Without is only possible after connecting trap done has been entered.

The STD of the Server is visualized in Figure 2a. The idea of Server is, (i) being in state Checking$_i$ means, Client$_i$ behaviour is kept within phase Checked while the other Clients are being kept within phase Without; (ii) being in state Helping$_i$ means, Client$_i$ behaviour is kept within phase With while the other Clients are

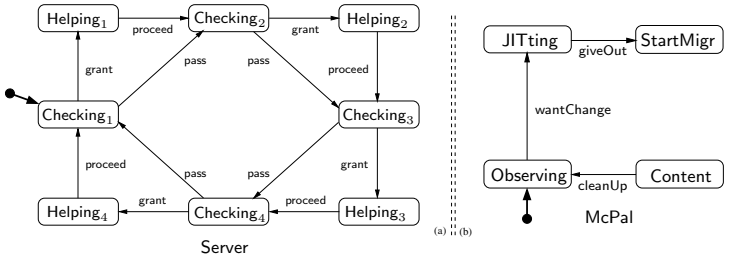


Fig. 2. STDs (a) Server and (b) McPal; the latter in Hibernating form only

being kept within phase Without. Note Server's round robin strategy in addressing the next Client_{i+1}, after having checked and possibly even helped Client_i.

In view of possible adaptation, the additional STD McPal, acting as an adaptation change conductor, is in place in its so-called hibernating form, visualized in Figure 2b. The idea is, as long as adaptation is not triggered, McPal is as yet interfering neither with Clients and Server nor with their collaboration. In particular, from Figure 2b we see McPal, starting in Observing, can go as far as StartMigr. But without interfering with itself, it can neither reach state Content nor return to state Observing. So the next idea is, once McPal has reached StartMigr it still has not started whatever adaptation of the as-is system, but by then it just-in-time has prepared such later migration through taking its last step giveOut, thus updating the specification of the original Paradigm model; to that aim the model specification is stored in McPal's local variable Crs. This is *reflectivity* of Paradigm models: a model contains its own specification and it can update it.

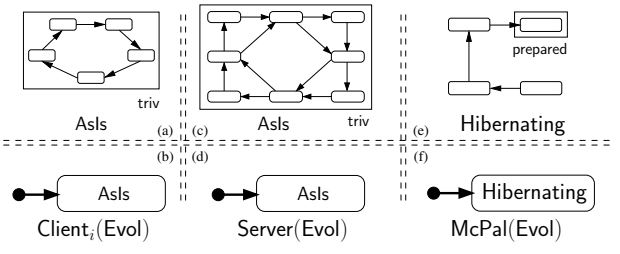


Fig. 3. Evol phases and roles: Client_i, Server, McPal

Moreover, in view of whatever possible change McPal might wish to exert on the STDs Server, Clients and McPal itself, each such STD has an Evol role that does not restrict their dynamics at all, as yet, see Figure 3. Note that each Evol role comprises exactly one state (and no steps).

To formulate a coordination solution for a collaboration in terms of constraints specified earlier, single steps from different roles, are synchronized into one protocol step. A protocol step can be coupled with one detailed step of a so-called conductor component. Also, variables local to the conductor can be updated. It is through a *consistency rule*, Paradigm specifies a protocol step: (i) at the

left-hand side of an asterisk $*$ the one conductor step is given, if relevant; (*ii*) the right-hand side lists the role steps being synchronized; (*iii*) optionally, a change clause can be given for updating variables, in particular the variable Crs containing the full model specification including the consistency rules, cf. [11]. A consistency rule with a conductor step is called an *orchestration* step, a consistency rule without it is called a *choreography* step.

The consistency rules for the orchestration of the $\mathsf{Client}_i(\mathsf{CS})$ roles, conducted by Server, are given by the first three rules below. Rule (1) says, if STD Server is in $\mathsf{Checking}_i$ and if role $\mathsf{Client}_i(\mathsf{CS})$ is in Checked and trap request of Checked has been entered, then Server can take step grant, thereby enforcing $\mathsf{Client}_i(\mathsf{CS})$ to take step request synchronously. Similarly, rule (2) synchronously couples Server's step proceed from $\mathsf{Checking}_i$ to $\mathsf{Checking}_{i+1}$ with $\mathsf{Client}_i(\mathsf{CS})$'s role step notYet from Checked to Without as well as with $\mathsf{Client}_{i+1}(\mathsf{CS})$'s role step triv from Without to Checked. Etc.

$$\mathsf{Server}\colon \mathsf{Checking}_i \xrightarrow{grant} \mathsf{Helping}_i \ * \ \mathsf{Client}_i(\mathsf{CS})\colon \mathsf{Checked} \xrightarrow{request} \mathsf{With} \tag{1}$$

$$\mathsf{Server}\colon \mathsf{Checking}_i \xrightarrow{pass} \mathsf{Checking}_{i+1} \ * \quad \mathsf{Client}_i(\mathsf{CS})\colon \mathsf{Checked} \xrightarrow{notYet} \mathsf{Without}, \ \mathsf{Client}_{i+1}(\mathsf{CS})\colon \mathsf{Without} \xrightarrow{triv} \mathsf{Checked} \tag{2}$$

$$\mathsf{Server}\colon \mathsf{Helping}_i \xrightarrow{proceed} \mathsf{Checking}_{i+1} \ * \quad \mathsf{Client}_i(\mathsf{CS})\colon \mathsf{With} \xrightarrow{done} \mathsf{Without}, \ \mathsf{Client}_{i+1}(\mathsf{CS})\colon \mathsf{Without} \xrightarrow{triv} \mathsf{Checked} \tag{3}$$

$$\mathsf{McPal}\colon \mathsf{JITting} \xrightarrow{giveOut} \mathsf{StartMigr} \ * \ \mathsf{McPal}\colon [\,\mathsf{Crs} := \mathsf{Crs} + \mathsf{Crs}_{\mathsf{migr}} + \mathsf{Crs}_{\mathsf{toBe}}\,] \tag{4}$$

$$\mathsf{McPal}\colon \mathsf{Content} \xrightarrow{cleanUp} \mathsf{Observing} \ * \ \mathsf{McPal}\colon [\,\mathsf{Crs} := \mathsf{Crs}_{\mathsf{toBe}}\,] \tag{5}$$

Please note, McPal is not involved in any coordination at all, rules (4) and (5), as no role steps are coupled with its non-role steps. But it prepares such migration coordination when going to state StartMigr (4): by extending the specification of the as-is coordination with the coordination for the adaptation as well as for the to-be situation. The specification's extension is being compensated (later) by McPal's step cleanUp when returning to Observing (5): by reducing the coordination specification to the to-be situation only.

3 To-Be Situation: Stationary Probabilistic Service

In this section we indicate, by example only, how Paradigm models can be endowed with probabilistic transitions. Thus, our STDs become Markov decision processes. For the probabilistic example we again consider a collaboration between four Client STDs and a Server STD. Moreover, in view of the adaption we want to discuss later, these probabilistic STDs will figure as to-be versions of the original non-probabilistic STD versions discussed in Section 2.

The STD of the Client_i in its new to-be form, see Figure 4a, has a new action goOn, replacing action leave. As one can see, the probabilistic transition labeled with action goOn points to two target states, to state Out, with probability q_1 and to state Nosing with probability q_2. Note $\mathsf{q}_1, \mathsf{q}_2 \geq 0$. As there are no other

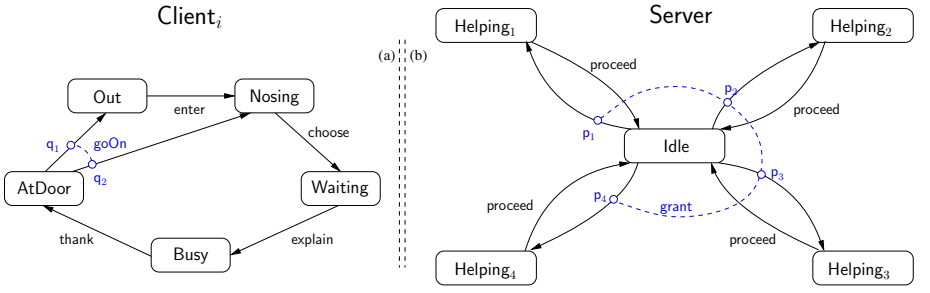


Fig. 4. STDs (a) Client$_i$ to-be, (b) Server to-be

target states, we have $q_1 + q_2 = 1$. Graphically, the two arrows, leaving the same state and referred to by the same action, are *shackled* by a (blue) dashed line.

The to-be form of process Server has changed rather more drastically, see Figure 4b. In the new state Idle there is one action grant available with four probabilistic outcomes: the four arrows are from Idle to the four states Helping$_i$ and have probabilities $p_1, p_2, p_3, p_4 \geq 0$, respectively, $p_1 + p_2 + p_3 + p_4 = 1$. The idea of Server is, in Idle it serves no Client at all, and in Helping$_i$ it serves Client$_i$ exclusively. As the probability to go to Helping$_i$ is always the same, although possible different fo each i, this kind of service strategy is called a stationary probabilistic service.

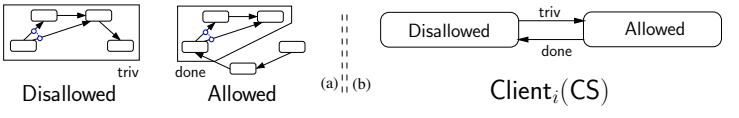


Fig. 5. Client$_i$ to-be: (a) phases and traps, for (b) role Client$_i$(CS)

The to-be role Client$_i$(CS) as given in Figure 5 differs from the as-is role in Figure 1 in two points. (*i*) The new versions of phases Without and With are called Disallowed and Allowed respectively, as in the to-be situation they have to contain action goOn instead of leave. (*ii*) For phase Checked there is no to-be version, as granting service to any Client$_i$ is done independently from Client$_i$ asking for it. As a consequence, the service turn should terminate immediately if Client$_i$ doesn't need it right then. And indeed it does, as trap done of phase Allowed is entered exactly when phase Allowed gets imposed: process Client$_i$ is in one of the states of trap done *already* because it did *not yet* request for the service turn. Please note, this is mimicked in the Server behaviour by the probabilistic transition grant leading from Idle immediately to state Helping$_i$.

The consistency rules for the orchestration of the to-be Client$_i$(CS) roles, conducted by the to-be Server, are as follows.

$$\begin{array}{l} \mathsf{p}_1 \cdot [\,\mathsf{Server}\colon \mathsf{Idle} \xrightarrow{grant} \mathsf{Helping}_1 \ * \ \mathsf{Client}_1(\mathsf{CS})\colon \mathsf{Disallowed} \xrightarrow{triv} \mathsf{Allowed}\,] \oplus \\ \mathsf{p}_2 \cdot [\,\mathsf{Server}\colon \mathsf{Idle} \xrightarrow{grant} \mathsf{Helping}_2 \ * \ \mathsf{Client}_2(\mathsf{CS})\colon \mathsf{Disallowed} \xrightarrow{triv} \mathsf{Allowed}\,] \oplus \\ \mathsf{p}_3 \cdot [\,\mathsf{Server}\colon \mathsf{Idle} \xrightarrow{grant} \mathsf{Helping}_3 \ * \ \mathsf{Client}_3(\mathsf{CS})\colon \mathsf{Disallowed} \xrightarrow{triv} \mathsf{Allowed}\,] \oplus \\ \mathsf{p}_4 \cdot [\,\mathsf{Server}\colon \mathsf{Idle} \xrightarrow{grant} \mathsf{Helping}_4 \ * \ \mathsf{Client}_4(\mathsf{CS})\colon \mathsf{Disallowed} \xrightarrow{triv} \mathsf{Allowed}\,] \end{array} \tag{6}$$

$$\mathsf{Server}\colon \mathsf{Helping}_i \xrightarrow{proceed} \mathsf{Idle} \ * \ \mathsf{Client}_i(\mathsf{CS})\colon \mathsf{Allowed} \xrightarrow{done} \mathsf{Disallowed} \tag{7}$$

$$\mathsf{McPal}\colon \mathsf{JITting} \xrightarrow{giveOut} \mathsf{StartMigr} \ * \ \mathsf{McPal}\colon [\,\mathsf{Crs} := \mathsf{Crs} + \mathsf{Crs}_{\mathsf{migr}} + \mathsf{Crs}_{\mathsf{toBe}}\,] \tag{8}$$

$$\mathsf{McPal}\colon \mathsf{Content} \xrightarrow{cleanUp} \mathsf{Observing} \ * \ \mathsf{McPal}\colon [\,\mathsf{Crs} := \mathsf{Crs}_{\mathsf{toBe}}\,] \tag{9}$$

Please note, for the coordination, we have coupled Server action grant in state Idle, via its probabilistic outcomes in terms of four possible detailed steps, to steps in four different roles (6). But each of these role steps is deterministically coupled to one specific detailed Server step. In this manner, the conductor throws the dice and the four participants, each one at the level of its CS role, deterministically obey to the probabilistic outcome. Moreover note, here too McPal is not involved in any coordination at all, as it only gets involved when the ongoing orchestration is to be adapted (8), (9).

4 From Deterministic to Probabilistic Service

Based on the as-is and to-be versions of process Client_i in Sections 2 and 3 we observe the following: (*i*) Phase ToBe from Figure 6a exactly specifies the constraint on Client_i needed for its Evol role during the to-be situation, as it does not prohibit any detailed step that should be able to occur; (*ii*) Figure 6b then specifies a feasible migration in terms of role $\mathsf{Client}_i(\mathsf{Evol})$, in one go from AsIs to ToBe. Here we do not need any migration phase in between.

In preparation of role Server(Evol), Figure 7 gives the detailed steps of Server during as-is, during to-be as well as during adaptation from as-is to to-be. Based on the foregoing Sections 2 and 3, it is easy to recognize in Figure 7, the steps from both the as-is and the to-be situations. But the other steps, apparently present in view of the intermediate migration trajectories, are not so clear.

In particular, the four probabilities $\mathsf{p}_1, \mathsf{p}_2, \mathsf{p}_3, \mathsf{p}_4$ are those from Section 3. But probability p_r, labeling the transition from Idle to $\mathsf{Checking}_3$, serves two purposes and, accordingly, has two values: (*i*) $\mathsf{p}_r = \mathsf{p}_{134} = \mathsf{p}_1 + \mathsf{p}_3 + \mathsf{p}_4$, if linked to only the transition from Idle to $\mathsf{Helping}_2$ (2-legs-shackle); (*ii*) $\mathsf{p}_r = \mathsf{p}_{34} = \mathsf{p}_3 + \mathsf{p}_4$, if linked to only the two transitions from Idle to $\mathsf{Helping}_2$ and to $\mathsf{Helping}_1$ (3-legs-shackle). Thus there are three shackles, linking either two transitions leaving state Idle, or

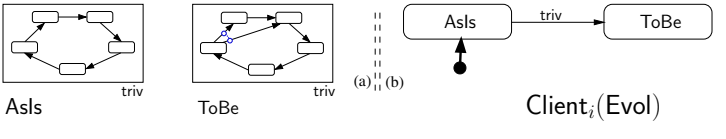


Fig. 6. Client_i during migration: (a) phases and traps, for (b) role $\mathsf{Client}_i(\mathsf{Evol})$

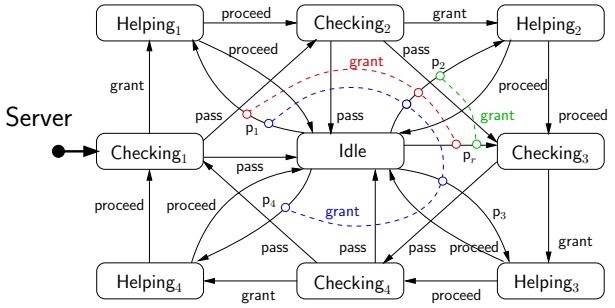


Fig. 7. STD Server during migration

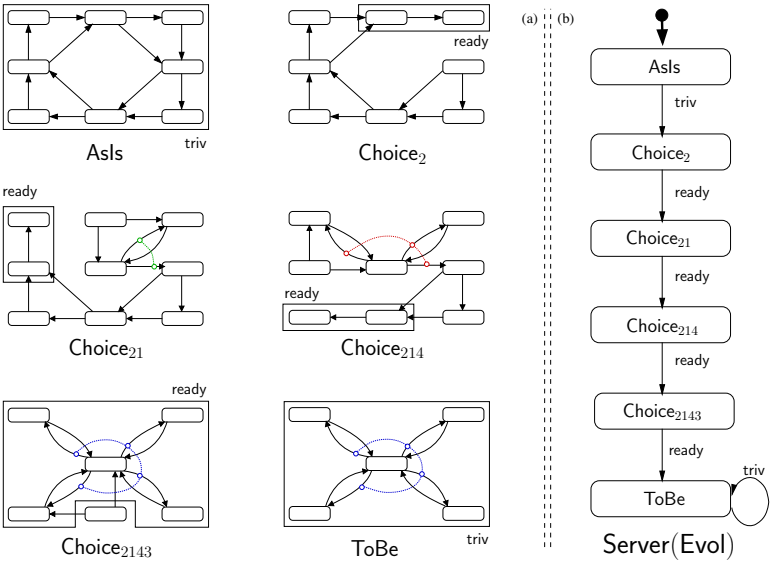


Fig. 8. Server during migration: (a) phases and traps, for (b) role Server(Evol)

three or four such transitions –the last one is the 4-legs-shackle from Section 3. To enlarge the entanglement even more, each shackle has the same label grant, thus referring to three different actions of that name, available in state Idle for making a transition from it; two of these must be there in view of the adaptation, as they neither belong to the as-is nor to the to-be situation.

Figure 8 repairs the unclarity, by giving the historical overview of the detailed dynamics of the Server through the six phases of partition Evol: phase AsIs visualizing the original, deterministic service provision, phase ToBe visualizing the target, probabilistic service provision and the actual migration phases Choice_2, Choice_{21}, Choice_{214} and Choice_{2143} visualizing in that order, how to get rather gradually from as-is to to-be service provision.

In more detail one can see the following: (i) In Choice_2 the deterministic round robin approach can only go as far as addressing Client_2. (ii) Upon addressing Client_2, it is decided for the last time in round robin fashion, whether it gets the turn. From then on, once Server within phase Choice_{21} returns to state Idle, process Client_2 has probability p_2 to get the service turn right then. But the other three Clients, with probability p_{134}, will be served instead in the usual round robin fashion. So within phase Choice_{21} we have action grant labeling the 2-legs-shackle. (iii) Next, as soon as Client_1 is addressed, it is decided for the last time in round robin fashion whether it gets the turn. From then on, once Server within phase Choice_{214} returns to state Idle, process Client_1 has probability p_1 to get the service turn right then. But the other two Clients, with probability p_{34}, will be served instead in the usual round robin fashion. So within phase Choice_{214} we have action grant labeling the 3-legs-shackle. (iv) Then, as soon as Client_4 is addressed, it is decided for the last time in round robin fashion whether it gets the turn. From then on, once Server within phase Choice_{2143} returns to state Idle, process Client_4 has probability p_4 to get the service turn right then. And from then on too, the last Client_3 will be served with probability p_3. So within phase Choice_{2143} we have action grant labeling the 4-legs-shackle, as needed in the to-be situation. (v) Finally, as soon as Client_3 is addressed, the to-be situation is indeed considered as reached and the migration coordination will stop accordingly.

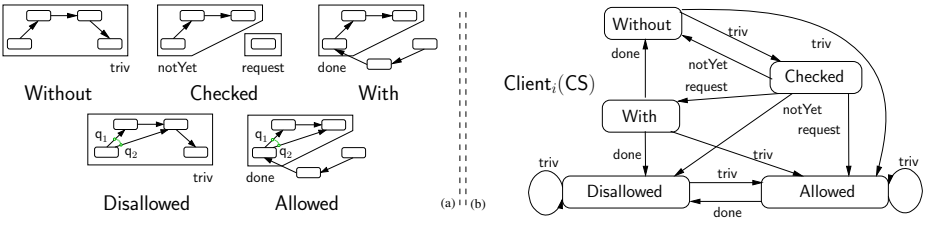


Fig. 9. $\mathsf{Client}_i(\mathsf{CS})$ during migration: (a) phases and traps, for (b) role $\mathsf{Client}_i(\mathsf{CS})$

The adaptation of CS role of the Clients from the as-is dynamics in Figure 1b to the to-be dynamics in Figure 5 is visualized in Figure 9.

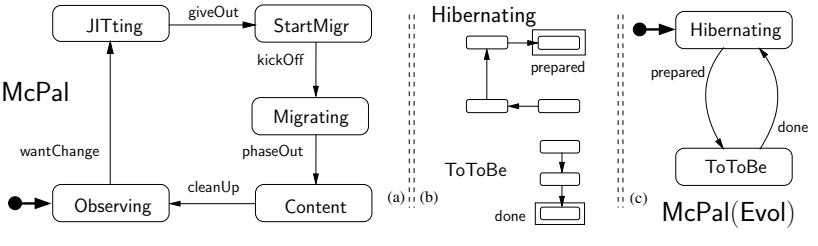


Fig. 10. Full McPal: (a) STD, (b) phases and traps, (c) role McPal(Evol)

The full McPal STD during adaptation from as-is to to-be situation is drawn in Figure 10, together with its Evol role and phases and traps for it. Please note, McPal in its phase ToToBe has only one state Migrating in between its states StartMigr and Content. The step to Migrating is meant to start the migration by conducting synchronous phase transfers in the Evol roles of Server as well as of all Clients. By doing so, McPal delegates the coordination of the remaining steps of the Server(Evol) role to either the role itself (choreography) or to Server (orchestration). Hence McPal has to wait in Migrating until Server actually achieves the coordination task just delegated to it. Finally, the step from Migrating coincides with McPal observing that Server has finished the task delegated.

The consistency rules specifying how to coordinate the adaptation along the lines visually clarified above, are given below. They appear in four groups. Please note, rules (10)-(31), together with the corresponding STDs constitute the value of McPal's local variable $\mathsf{Crs}_{\mathsf{migr}}$. Likewise, the value of McPal's local variable $\mathsf{Crs}_{\mathsf{toBe}}$ contains the rules (6)-(9) from Section 3, while Crs contains the rules (1)-(5) and corresponding specifications from Section 2 as its initial value.

The first group of rules given here are for McPal. In particular they cover the two choreography steps from phase Hibernating to ToToBe and from ToToBe back to Hibernating. Moreover, they cover the coordination conducted by McPal when within phase ToToBe. Please note, the delegation of adaptation tasks from McPal to Server is captured in the rule (11).

$$* \ \mathsf{McPal(Evol)}\colon \mathsf{Hibernating} \xrightarrow{prepared} \mathsf{ToToBe} \tag{10}$$

$$\begin{aligned}\mathsf{McPal}\colon \mathsf{StartMigr} \xrightarrow{kickOff} \mathsf{Migrating} \ * \ \mathsf{Server(Evol)}\colon \mathsf{AsIs} \xrightarrow{triv} \mathsf{Choice}_2, \\ \mathsf{Client}_1\mathsf{(Evol)}\colon \mathsf{AsIs} \xrightarrow{triv} \mathsf{ToBe}, \ \mathsf{Client}_2\mathsf{(Evol)}\colon \mathsf{AsIs} \xrightarrow{triv} \mathsf{ToBe}, \\ \mathsf{Client}_3\mathsf{(Evol)}\colon \mathsf{AsIs} \xrightarrow{triv} \mathsf{ToBe}, \ \mathsf{Client}_4\mathsf{(Evol)}\colon \mathsf{AsIs} \xrightarrow{triv} \mathsf{ToBe}\end{aligned} \tag{11}$$

$$\mathsf{McPal}\colon \mathsf{Migrating} \xrightarrow{phaseOut} \mathsf{Content} \ * \ \mathsf{Server(Evol)}\colon \mathsf{ToBe} \xrightarrow{triv} \mathsf{ToBe} \tag{12}$$

$$* \ \mathsf{McPal(Evol)}\colon \mathsf{ToToBe} \xrightarrow{migrDone} \mathsf{Hibernating} \tag{13}$$

All further rules are for Server, which guides the adaptation changes. As there are quite many of them, we split them into similar groups, corresponding to the remaining role steps of Server(Evol). First those guiding the transfer from Choice_2 to Choice_{21}.

$$* \ \mathsf{Server(Evol)}\colon \mathsf{Choice}_2 \xrightarrow{ready} \mathsf{Choice}_{21}, \ \mathsf{Client}_2\mathsf{(CS)}\colon \mathsf{Checked} \xrightarrow{notYet} \mathsf{Disallowed} \tag{14}$$

$$* \ \mathsf{Server(Evol)}\colon \mathsf{Choice}_2 \xrightarrow{ready} \mathsf{Choice}_{21}, \ \mathsf{Client}_2\mathsf{(CS)}\colon \mathsf{With} \xrightarrow{triv} \mathsf{Allowed} \tag{15}$$

$$* \ \mathsf{Server(Evol)}\colon \mathsf{Choice}_2 \xrightarrow{ready} \mathsf{Choice}_{21}, \ \mathsf{Client}_2\mathsf{(CS)}\colon \mathsf{Checked} \xrightarrow{request} \mathsf{Allowed} \tag{16}$$

$$\mathsf{Server}\colon \mathsf{Checking}_2 \xrightarrow{pass} \mathsf{Idle} \ * \ \mathsf{Client}_2\mathsf{(CS)}\colon \mathsf{Disallowed} \xrightarrow{triv} \mathsf{Disallowed} \tag{17}$$

$$\mathsf{Server}\colon \mathsf{Checking}_2 \xrightarrow{grant} \mathsf{Helping}_2 \ * \ \mathsf{Client}_2\mathsf{(CS)}\colon \mathsf{Allowed} \xrightarrow{triv} \mathsf{Allowed} \tag{18}$$

$$\begin{aligned}\mathsf{p}_2 \cdot [\,\mathsf{Server}\colon \mathsf{Idle} \xrightarrow{grant} \mathsf{Helping}_2 \ * \ \mathsf{Client}_2\mathsf{(CS)}\colon \mathsf{Disallowed} \xrightarrow{triv} \mathsf{Allowed}\,] \ \oplus \\ \mathsf{p}_{134} \cdot [\,\mathsf{Server}\colon \mathsf{Idle} \xrightarrow{grant} \mathsf{Checking}_3 \ * \ \mathsf{Client}_3\mathsf{(CS)}\colon \mathsf{Without} \xrightarrow{triv} \mathsf{Checked}\,]\end{aligned} \tag{19}$$

The first three choreography steps address the actual phase transfer from Choice_2 to Choice_{21} once trap ready of Choice_2 has been entered. The choreography is moreover coupled to relevant CS role steps of Client_2: phase transfers from

Checked or With to Disallowed or Allowed, thus only now enabling the to-be probabilistic behaviour to Client_2 exclusively. The last three orchestration steps cover the deviating part of the CS protocol under migration during phase Choice_{21} only.

$$* \ \mathsf{Server(Evol)}\colon \mathsf{Choice}_{21} \xrightarrow{ready} \mathsf{Choice}_{214},\ \mathsf{Client}_1(\mathsf{CS})\colon \mathsf{Checked} \xrightarrow{notYet} \mathsf{Disallowed} \tag{20}$$

$$* \ \mathsf{Server(Evol)}\colon \mathsf{Choice}_{21} \xrightarrow{ready} \mathsf{Choice}_{214},\ \mathsf{Client}_1(\mathsf{CS})\colon \mathsf{With} \xrightarrow{triv} \mathsf{Allowed} \tag{21}$$

$$* \ \mathsf{Server(Evol)}\colon \mathsf{Choice}_{21} \xrightarrow{ready} \mathsf{Choice}_{214},\ \mathsf{Client}_1(\mathsf{CS})\colon \mathsf{Checked} \xrightarrow{request} \mathsf{Allowed} \tag{22}$$

$$\mathsf{Server}\colon \mathsf{Checking}_1 \xrightarrow{pass} \mathsf{Idle} \ * \ \mathsf{Client}_1(\mathsf{CS})\colon \mathsf{Disallowed} \xrightarrow{triv} \mathsf{Disallowed} \tag{23}$$

$$\mathsf{Server}\colon \mathsf{Checking}_1 \xrightarrow{grant} \mathsf{Helping}_1 \ * \ \mathsf{Client}_1(\mathsf{CS})\colon \mathsf{Allowed} \xrightarrow{triv} \mathsf{Allowed} \tag{24}$$

$$\begin{aligned} &\mathsf{p}_1 \cdot [\, \mathsf{Server}\colon \mathsf{Idle} \xrightarrow{grant} \mathsf{Helping}_1 \ * \ \mathsf{Client}_1(\mathsf{CS})\colon \mathsf{Disallowed} \xrightarrow{triv} \mathsf{Allowed}\,] \oplus \\ &\mathsf{p}_2 \cdot [\, \mathsf{Server}\colon \mathsf{Idle} \xrightarrow{grant} \mathsf{Helping}_2 \ * \ \mathsf{Client}_2(\mathsf{CS})\colon \mathsf{Disallowed} \xrightarrow{triv} \mathsf{Allowed}\,] \oplus \\ &\mathsf{p}_{34} \cdot [\, \mathsf{Server}\colon \mathsf{Idle} \xrightarrow{grant} \mathsf{Checking}_3 \ * \ \mathsf{Client}_3(\mathsf{CS})\colon \mathsf{Without} \xrightarrow{triv} \mathsf{Checked}\,] \end{aligned} \tag{25}$$

Finally, the consistency rules for transfer from Choice_{214} to Choice_{2143}.

$$* \ \mathsf{Server(Evol)}\colon \mathsf{Choice}_{214} \xrightarrow{ready} \mathsf{Choice}_{2143},\ \mathsf{Client}_4(\mathsf{CS})\colon \mathsf{Checked} \xrightarrow{notYet} \mathsf{Disallowed} \tag{26}$$

$$* \ \mathsf{Server(Evol)}\colon \mathsf{Choice}_{214} \xrightarrow{ready} \mathsf{Choice}_{2143},\ \mathsf{Client}_4(\mathsf{CS})\colon \mathsf{With} \xrightarrow{triv} \mathsf{Allowed} \tag{27}$$

$$* \ \mathsf{Server(Evol)}\colon \mathsf{Choice}_{214} \xrightarrow{ready} \mathsf{Choice}_{2143},\ \mathsf{Client}_4(\mathsf{CS})\colon \mathsf{Checked} \xrightarrow{request} \mathsf{Allowed} \tag{28}$$

$$\mathsf{Server}\colon \mathsf{Checking}_4 \xrightarrow{pass} \mathsf{Idle} \ * \ \mathsf{Client}_4(\mathsf{CS})\colon \mathsf{Disallowed} \xrightarrow{triv} \mathsf{Disallowed} \tag{29}$$

$$\mathsf{Server}\colon \mathsf{Checking}_4 \xrightarrow{grant} \mathsf{Helping}_4 \ * \ \mathsf{Client}_4(\mathsf{CS})\colon \mathsf{Allowed} \xrightarrow{triv} \mathsf{Allowed} \tag{30}$$

$$\begin{aligned} &\mathsf{p}_1 \cdot [\, \mathsf{Server}\colon \mathsf{Idle} \xrightarrow{grant} \mathsf{Helping}_1 \ * \ \mathsf{Client}_1(\mathsf{CS})\colon \mathsf{Disallowed} \xrightarrow{triv} \mathsf{Allowed}\,] \oplus \\ &\mathsf{p}_2 \cdot [\, \mathsf{Server}\colon \mathsf{Idle} \xrightarrow{grant} \mathsf{Helping}_2 \ * \ \mathsf{Client}_2(\mathsf{CS})\colon \mathsf{Disallowed} \xrightarrow{triv} \mathsf{Allowed}\,] \oplus \\ &\mathsf{p}_4 \cdot [\, \mathsf{Server}\colon \mathsf{Idle} \xrightarrow{grant} \mathsf{Helping}_4 \ * \ \mathsf{Client}_4(\mathsf{CS})\colon \mathsf{Disallowed} \xrightarrow{triv} \mathsf{Allowed}\,] \oplus \\ &\mathsf{p}_3 \cdot [\, \mathsf{Server}\colon \mathsf{Idle} \xrightarrow{grant} \mathsf{Helping}_3 \ * \ \mathsf{Client}_3(\mathsf{CS})\colon \mathsf{Without} \xrightarrow{triv} \mathsf{Allowed}, \\ &\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{Server(Evol)}\colon \mathsf{Choice}_{2143} \xrightarrow{ready} \mathsf{ToBe}\,] \end{aligned} \tag{31}$$

Again, highly similar to the previous two groups of rules, we have three choreography steps addressing the actual phase transfer from Choice_{214} to Choice_{2143} as well as three orchestration steps covering the deviating part of the CS protocol under migration. But here, by taking the last orchestration step, addressing Client_3 for the first time during phase Choice_{2143}, the migration coordination is finished by additionally conduction the phase transfer from Choice_{2143} to ToBe. This then enables McPal to take over the migration coordination, actually by instigating the phase transfer back to Hibernating and the later removal of consistency rules and other model fragments obsolete by then.

5 Adaptation Analysis with Prism

We analyze the Paradigm models and their dynamic adaptation with the probabilistic model checker Prism [12]. As it turns out, the Paradigm models involved can conveniently be translated into the Prism modeling language. In particular, each component of the system, Clients, Server and McPal, are interpreted as a

separate module. Thus, the detailed behaviour of a component together with its two roles, for the CS collaboration and for the Evol adaptation collaboration, are brought together in one module. This way, the temporary behavioural constraints on the detailed STD imposed by a current phase of the global STDs, can be imposed using a guard on detailed transitions.

A fragment of the Prism specification of Client_1 is shown below. The current state of the detailed STD is stored in local variable s_1. Variables S_1 and E_1 hold, respectively, the current phase of the CS partition and the Evol partition. The Prism fragment specifies the detailed transition of Client_1 as constrained by phase Without of Client(CS), combined with the constraints imposed by phases AsIs and ToBe of the Client(Evol) partition.

$$[\texttt{enter}_1] \quad S_1{=}\mathsf{Without}\,\&\,(E_1{=}\mathsf{AsIs}\,|\,E_1{=}\mathsf{ToBe})\,\&\,s_1{=}\mathsf{Out} \rightarrow s_1'{=}\mathsf{Nosing};$$
$$[\texttt{choose}_1] \quad S_1{=}\mathsf{Without}\,\&\,(E_1{=}\mathsf{AsIs}\,|\,E_1{=}\mathsf{ToBe})\,\&\,s_1{=}\mathsf{Nosing} \rightarrow s_1'{=}\mathsf{Waiting};$$
$$[\texttt{leave}_1] \quad S_1{=}\mathsf{Without}\,\&\,E_1{=}\mathsf{AsIs}\,\&\,s_1{=}\mathsf{AtDoor} \rightarrow s_1'{=}\mathsf{Out};$$
$$[\texttt{leave}_1] \quad S_1{=}\mathsf{Without}\,\&\,E_1{=}\mathsf{ToBe}\,\&\,s_1{=}\mathsf{AtDoor} \rightarrow \mathsf{q}_1 : (s_1'{=}\mathsf{Out}) + \mathsf{q}_2 : s_1'{=}\mathsf{Nosing};$$

For the protocol steps within a collaboration, e.g. between the server and its clients, a unique action label identifies a protocol step. The same action label is shared among all components involved. Synchronization on the shared label, hence fulfillment of all relevant guards, leads to execution of the corresponding consistency rule: a detailed transition of the conductor, phase changes for the participants involved. In this case, the guard indicates that the corresponding trap within a current phase has been entered (at the level of detailed dynamics of the component). In Paradigm, for a phase transfer to be enabled, information about their current states from both the detailed as well as the global STDs, needs to be provided. The information is extracted from the local variables in the Prism module, conjunctively combined as a guard for the global transition. For instance, the unique name of the consistency rule 2 for $i = 1$ is $cr2_{12}$. The consistency rule in Prism is specified by three separate commands, all having the same action label:

In module Client_1: $[cr2_{12}]\ S_1{=}\mathsf{With}\ \&\ (s_1{=}\mathsf{AtDoor}\,|\,s_1{=}\mathsf{Out}\,|\,s_1{=}\mathsf{Nosing}) \rightarrow S_1'{=}\mathsf{Without};$

In module Client_2: $[cr2_{12}]\ S_2{=}\mathsf{Without}\ \&$
$(s_2{=}\mathsf{AtDoor}\,|\,s_2{=}\mathsf{Out}\,|\,s_2{=}\mathsf{Nosing}\,|\,s_2 = \mathsf{Waiting}) \rightarrow S_2'{=}\mathsf{Checked};$

In module Server: $[cr2_{12}]\ (ES{=}\mathsf{AsIs}\,|\,ES{=}\mathsf{Choice}_2)\ \&\ r{=}\mathsf{Helping}_1 \rightarrow r'{=}\mathsf{Checking}_2;$

The condition $(s_2{=}\mathsf{AtDoor}\,|\,s_2{=}\mathsf{Out}\,|\,s_2{=}\mathsf{Nosing}\,|\,s_2{=}\mathsf{Waiting})$, for instance, specifies that the current local state of the detailed STD of Client_2 belongs to trap triv of the CS phase Without.

Probabilistic consistency rules are translated into Prism in a slightly different manner. A probabilistic rule is applied in two consecutive stages. During the first stage, the conductor of the rule, in this case the Server, selects the next step to be executed according to the underlying probability distribution. This step is not synchronized with any other participant, in particular the Clients. Once the next step is selected, i.e. a Client to serve is chosen, the Server executes the second part of the rule: it accomplishes the step by conducting, this time synchronized, changing its local state and assigning the phase changes to the participants involved.

We have verified a number of qualitative and quantitative properties of the adapting system.[1]

At any moment during system migration, in any phase, including the source phase AsIs and the target phase ToBe, at most one client will be given service. Let `clients_in_cs` count the number of clients being currently served, i.e. having With or Allowed as global state. Then mutual exclusion can easily be expressed as `clients_in_cs <= 1`.

During any phase, if a client is requesting service, eventually a client will get service. With `one_trying` denoting that a client is in state Waiting, and `one_has_service` denoting that a client is served, this liveness property is expressed as `"one_trying" => P>=1 [ F "one_has_service" ]`.

At any time, in any phase, if a client is requesting a service, then eventually this client will get served. More concretely for Client_1, similar for other clients, this is expressed as: `s1=Waiting => P>=1 [ F (S1=With|S1=Allowed) ]`. Note that implicitly this property also expresses that the functionality to provide service is never interrupted during adaptation, no matter what dynamics of the server or what trajectory towards target behaviour is taken.

Assuming that the adaptation is triggered, the system will adapt to the target ToBe dynamics: `P>=1 [ F ES=6 & E1=2 & E2=2 & E3=2 & E4=2]`. Here, `ES=6` and `Ei=2`, $i = 1, .., 4$, refer to the completely adapted phases of the server and clients. Moreover, every system component, once adapted to the final stage, does not execute old AsIs behaviour anymore. E.g., for Client_1 we have that `"ToBeOfClient1" => P>=1 [ G !"AsIsOfClient1" ]`.

Various quantitative properties can be checked against the model as well. The results discussed below use a reward structure appropriately defined on the model, assigning a reward of 1 each time the server addresses a client, either for checking or for helping.

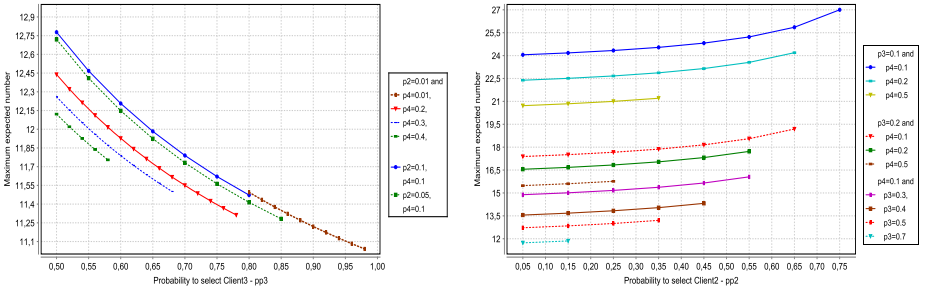


Fig. 11. Maximum expected number of clients addressed by the server during adaptation: fixed p_2 and p_4 (left) and fixed p_3 and p_4 (right)

The expected number of clients addressed by the server during system adaptation is expressed as: `Rmax=? [ F (ES=6) {ES=2} ]`. We compute that for

[1] Complete Prism specifications of the collaborative processes in adaptation, as described in Section 4, can be found at `http://www.win.tue.nl/~andova`

probabilities $p_1 = p_2 = p_3 = p_4 = 0.25$ this expectation equals 15.3. The experiments show that this actually depends on values of probabilities $p_1, \ldots, p_4$. The left graph in Figure 11 shows that the system adapts in less steps as the probability p_3, of selecting Client_3 increases. The right graph in Figure 11 shows that value of p_2, the probability to select Client_2, hardly influences the speed of the system adaptation, but again it is influenced by the value of p_3. From the moment on Client_1 requests service, at any time during the adaptation, the expected number of clients the server addresses before it addresses Client_1 equals 2. We find that the worst case expected number, computed as `Rmax=? [F (S1=With|S1=Allowed){s1=Waiting & ES=evol_phase & r=server_state}]`, gives better insight into the system behaviour during adaptation. The `evol_phase` to be analyzed can be selected, as well as the current local `server_state` of the server, at the moment Client_1 is requesting service. Experiments show that the (worst) expected waiting time for Client_1 decreases as probability p_1 increases, but not for all Evol phases. As expected, for the first two phases, in which Client_1 is not yet selected probabilistically but in round-robin fashion, this measure has a constant value. Figure 12 shows the results of the experiments for probabilities p_2 and p_3 set to 0.25. As observed, waiting time depends on the current state of the Server at the moment Client_1 requests service. In the graph, r shows the worst case for all particular Evol phases.

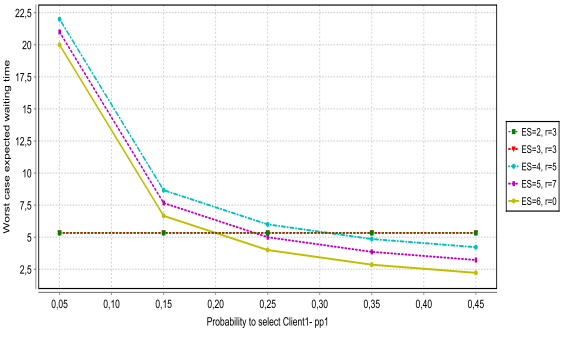


Fig. 12. Worst case waiting time for service for Client_1, for $p_2 = 0.25$ and $p_3 = 0.25$

6 Conclusions

We have addressed the issue of formal modeling and analysis, both qualitative and quantitative, of dynamic system adaptation without quiescence. We have shown that Paradigm is well suited to model dynamic adaptation of systems that exhibits probabilistic behaviour. The approach is illustrated for a client-server example. In the source situation, the clients have strictly deterministic dynamics and are served in round-robin fashion. In the target system, clients have probabilistic behaviour, and the server probabilistically selects which client to serve. In the Paradigm model of the adaptation, components smoothly change their behaviour, gradually replacing old deterministic by probabilistic behaviour.

The system components migrate from one phase to another, without having their activity disrupted at all.

In addition, a translation to Prism is presented. Each component is represented as a separate module in Prism, synchronizing with other components via shared labels, just as specified by Paradigm's consistency rules. Dynamic constraints typical for Paradigm, whether a phase transfer can take place and whether a local step is allowed, in Prism are specified as command guards, rather straightforwardly. The translation enables the verification of the adaptation model. Transitional properties, both qualitative and quantitative, of the system during the adaptation, are established using the Prism model checker.

As future work we consider the general translation of probabilistic Paradigm into Prism, taking the example presented in this paper as a starting point. Furthermore, we will conduct more case studies of dynamic adaptation of larger systems, involving more intricate probabilities that are expected to mix well with our architectural approach. In particular, we will compare and connect with the Cactus protocol framework [10], which seems the only other work providing smooth adaptation of distributed systems.

References

1. Allen, R., Douence, R., Garlan, D.: Specifying and analyzing dynamic software architectures. In: Astesiano, E. (ed.) ETAPS 1998 and FASE 1998. LNCS, vol. 1382, pp. 21–37. Springer, Heidelberg (1998)
2. Andova, S., Groenewegen, L.P.J., Stafleu, J., de Vink, E.P.: Formalizing adaptation on-the-fly. In: Proc. FOCLASA 2009. ENTCS, vol. 255, pp. 23–44 (2009)
3. Andova, S., Groenewegen, L.P.J., Verschuren, J.H.S., de Vink, E.P.: Architecting security with Paradigm. In: de Lemos, R. (ed.) Architecting Dependable Systems VI. LNCS, vol. 5835, pp. 255–283. Springer, Heidelberg (2009)
4. Andova, S., Groenewegen, L.P.J., de Vink, E.P.: Dynamic consistency in process algebra: From Paradigm to ACP. In: Proc. FOCLASA 2008. ENTCS, vol. 229, pp. 3–20 (2009)
5. Andova, S., Groenewegen, L.P.J., de Vink, E.P.: Dynamic consistency in process algebra: From Paradigm to ACP. Science of Computer Programming, 45 (2010), doi:10.1016/j.scico.2010.04.011
6. Bencomo, N., Sawyer, P., Blair, G.S., Grace, P.: Dynamically adaptive systems are product lines too. In: Proc. DSPL 2008, Limerick, pp. 23–32 (2008)
7. Bozzano, M., et al.: Safety, dependability, and performance analysis of extended AADL models. The Computer Journal (2010), doi:10.1093/com
8. Bradbury, J.S., et al.: A survey of self-management in dynamic software architecture specifications. In: Proc. WOSS 2004, pp. 28–33. ACM, New York (2004)
9. Cetina, C., Fons, J., Pelechano, V.: Applying software product lines to build autonomic pervasive systems. In: Proc. SPLC 2008, pp. 117–126. IEEE, Los Alamitos (2008)
10. Chen, W., Hiltunen, M.A., Schlichting, R.D.: Constructing adaptive software in distributed systems. In: Proc. ICDCS 2001, pp. 635–643. IEEE, Los Alamitos (2001)
11. Groenewegen, L.P.J., de Vink, E.P.: Evolution on-the-fly with Paradigm. In: Ciancarini, P., Wiklicky, H. (eds.) COORDINATION 2006. LNCS, vol. 4038, pp. 97–112. Springer, Heidelberg (2006)

12. Hinton, A., Kwiatkowska, M.Z., Norman, G., Parker, D.: PRISM: A tool for automatic verification of probabilistic systems. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 441–444. Springer, Heidelberg (2006)
13. Kramer, J., Magee, J.: The evolving philosophers problem: dynamic change management. IEEE Transactions on Software Engineering 16, 1293–1306 (1990)
14. Kramer, J., Magee, J.: Self-managed systems: an architectural challenge. In: Proc. FOSE 2007, pp. 259–268. IEEE, Los Alamitos (2007)
15. Magee, J., Kramer, J.: Dynamic structure in software architectures. SIGSOFT Software Engineering Notes 21, 3–14 (1996)
16. Morin, B., et al.: An aspect-oriented and model-driven approach for managing dynamic variability. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 782–796. Springer, Heidelberg (2008)
17. Schneider, K., Schuele, T., Trapp, M.: Verifying the adaptation behavior of embedded systems. In: Proc. SEAMS 2006, pp. 16–22. ACM, New York (2006)
18. Zhang, J., Cheng, B.H.C.: Model-based development of dynamically adaptive software. In: Proc. ICSE 2006, pp. 371–380. ACM, New York (2006)
19. Zhang, J., Goldsby, H.J., Cheng, B.H.C.: Modular verification of dynamically adaptive systems. In: Proc. AOSD 2009, pp. 161–172. ACM, New York (2009)

UPPAAL in Practice: Quantitative Verification of a RapidIO Network*

Jiansheng Xing[1,2], Bart D. Theelen[2], Rom Langerak[1], Jaco van de Pol[1], Jan Tretmans[2], and J.P.M. Voeten[2,3]

[1] University of Twente, Faculty of EEMCS
P.O. Box 217, Formal Methods and Tools
7500 AE Enschede, The Netherlands
{xingj,r.langerak,j.c.vandepol}@cs.utwente.nl
[2] Embedded Systems Institute
P.O. Box 513
5600 MB Eindhoven, The Netherlands
{bart.theelen,jan.tretmans}@esi.nl
[3] Eindhoven University of Technology, Faculty of Electrical Engineering
Information and Communication Systems group
5600 MB Eindhoven, The Netherlands
j.p.m.voeten@tue.nl

Abstract. Packet switched networks are widely used for interconnecting distributed computing platforms. RapidIO (Rapid Input/Output) is an industry standard for packet switched networks to interconnect multiple processor boards. Key performance metrics for these platforms include average-case and worst-case packet transfer latencies. We focus on verifying such quantitative properties for a RapidIO based multiprocessor platform that executes a motion control application. A performance model is available in the Parallel Object-Oriented Specification Language (POOSL) that allows for simulation based estimation results. It is however required to determine the exact worst-case latency as the application is time-critical. A model checking approach has been proposed in our previous work which transforms the POOSL model into an UPPAAL model. However, such an approach only works for a fairly small system. We extend the transformation approach with various heuristics to reduce the underlying state space, thereby providing an effective approximation approach that scales to industrial problems of a reasonable complexity.

Keywords: UPPAAL; POOSL; transformation; quantitative verification; heuristic.

1 Introduction

A packet switched network is a digital communication network that groups all transmitted data into suitably-sized blocks, called packets. The network over

* This work has been supported by the EU FP7 under grant number ICT-214755: Quasimodo.

T. Margaria and B. Steffen (Eds.): ISoLA 2010, Part II, LNCS 6416, pp. 160–174, 2010.

which packets are transmitted is a shared network which routes each packet independently from others and allocates transmission resources as needed. The principal goals of packet switching are to optimize utilization of available link capacity, minimize response times and increase the robustness of communication. When traversing network adapters, switches and other network nodes, packets are buffered, resulting in variable delay and throughput, depending on the traffic load in the network.

RapidIO is an industry standard [9] for packet-switched networks to connect chips on a circuit board, and also circuit boards to each other using a backplane. It has found widespread adoption in the following applications: wireless base station, video, medical imaging, etc. Key performance metrics for these applications include average-case and worst-case packet transfer latencies.

In this paper, we consider a motion control system that includes a packet switched network conforming to the RapidIO standard. The motion control system is characterized by feedback/feedforward control strategies and periodic execution. It is constrained by strict timing requirements that all packets must arrive at their destinations before the period ends. The considered motion control algorithms are distributed over a multi-processor platform. Various processors in this platform are inter-connected by a RapidIO network. The main challenge for system designers is how to map the motion control algorithms on the multi-processor platform such that the timing constraints are met. Packet transfer latencies as worst-case latencies and average-case latencies are essential criteria for finding a feasible mapping.

A simulation based approach is available that relies on the Parallel Object-Oriented Specification Language (POOSL) for investigating the performance of the motion control system. First, a POOSL model is constructed for a given mapping and then end-to-end packet transfer latencies are analyzed. These steps are repeated for alternative mappings, until a feasible mapping has been found, whose end-to-end latencies satisfy the timing constraints. POOSL analysis gives both average-case and worst-case latencies. The obtained latencies are estimation results since the POOSL analysis is based on simulation. However, as the motion control application is time-critical, worst-case latencies are strict timing constraints. Exact worst-case latencies are therefore demanded.

In this paper, we focus on verifying worst-case packet transfer latencies for a realistic motion control system. In earlier work [11], we have shown that transforming a POOSL model into an UPPAAL model is feasible. Based on the obtained UPPAAL model, we also showed that quantitative verification for worst-case latencies is only feasible for a fairly small system. We extend such an approach to enable analyzing an industrial-sized problem in this paper. First the POOSL model of a realistic motion control system is illustrated and transformed into an UPPAAL model according to the transformation patterns in [11]. Second scalability experiments show that the UPPAAL model is not capable for handling realistic (high volume) traffics. Then we propose some heuristics for the UPPAAL approach. Experiments show that the UPPAAL approach with heuristics can find worse scenarios (worse latencies) than the POOSL approach

and thus is an effective approximation method which complements the POOSL approach for performance analysis.

The rest of the paper is organized as follows. Section 2 presents the POOSL model for a realistic motion control system. The transformation from the POOSL model into an UPPAAL model is discussed in Section 3. We present the scalability of the UPPAAL model and then propose some heuristics for worst-case latency analysis in Section 4. Finally, conclusions are drawn in Section 5.

2 POOSL Model of a Realistic Motion Control System

POOSL was originally defined in [7] as an object-oriented extension of CCS [6]. Meanwhile, POOSL has been extended with time in [4] and probabilities in [3] to become a very expressive formal modeling language accompanied with simulation, analysis and synthesis techniques that scale to large industrial design problems [10].

Three types of objects are supported in POOSL: data, process, and cluster. Data models the passive components of a system representing information that is generated, communicated, processed and consumed by active components. The elementary active components are modeled by processes while groups of active components are modeled by clusters in a hierarchical fashion.

Figure 1 depicts the top-level POOSL model for a realistic motion control system. The left part is for channel (refers to a specific type of packet with specified size, source, and destination) generation. The right part represents the underlying RapidIO network of the motion control system. These two parts are connected by message *si*. The left part is implemented as a process class *RIOChannelTrafficGenerator*. More details of this process class are shown in figure 2.

The right part of figure 1 is implemented as a cluster class *RIOResource*. We note that message *si* is multiplexed into 5 messages *slot1,...,slot5* which connect *RIOChannelTrafficGenerator* with different subparts of cluster *RIOResource*. More details of *RIOResource* are shown in figure 3. *RIOResource* is mainly constituted by 5 subparts *CB1,CB2,CB3,CB4*, and *CB5*. Each subpart is an instance of the cluster class *RIOCarrierBlade*.

RIOCarrierBlade represents a pair of packet switches and their connected endpoints. Each switch connects with 4 endpoints directly. The details of *RIOCarrierBlade* are shown in figure 4. *B1_EP1,B2_EP1*,..., etc., each represents an endpoint which is implemented as a process class *RIOEndPoint*. More details of this process class are shown in figure 5.

In figure 4, *SW1* and *SW2* represent two packet switches. A packet switch is implemented as a process class *RIOSwitch*. More details of this process class are shown in figure 6.

The system works as follows: first the channel (each channel refers to a specific kind of packet, which will be divided into several equal-sized packets during generation, see next section for details) generation part (left part of figure 1) generates channels (actually packets); the generated packets are actually put into

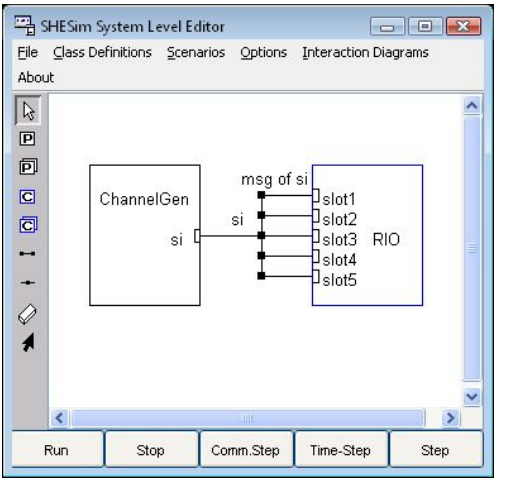


Fig. 1. POOSL model for a realistic motion control system

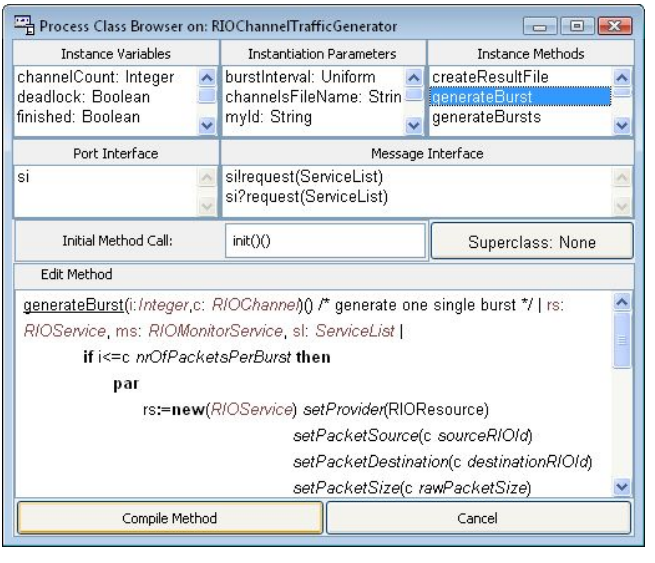


Fig. 2. Process Class RIOChannelTrafficGenerator

corresponding endpoints. Each endpoint represents a processor that is actually sending and/or receiving packets. The generated packets then are sent from the endpoints and routed among the whole RapidIO network until they are received by their corresponding target endpoints. We notice that the whole underlying RapidIO network includes 5 subparts; each subpart contains 2 packet switches and 8 endpoints.

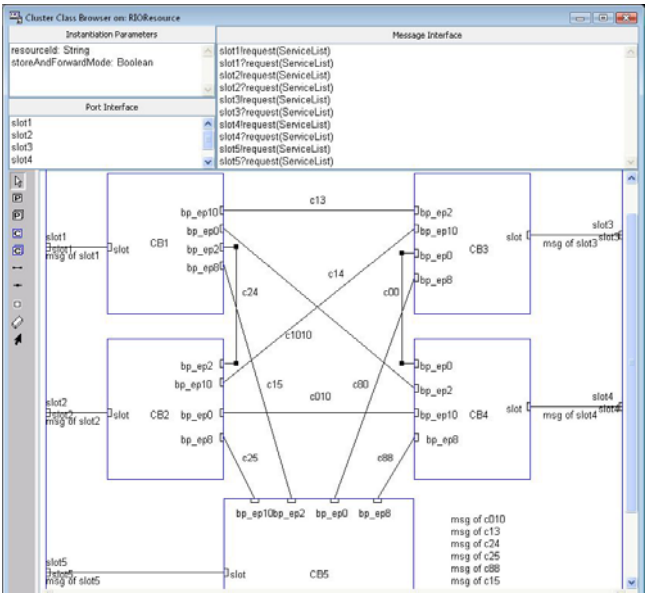


Fig. 3. Cluster Class RIOResource

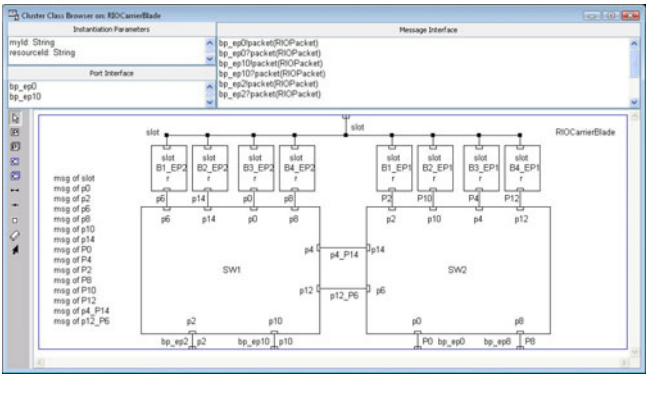


Fig. 4. Cluster Class RIOCarrierBlade

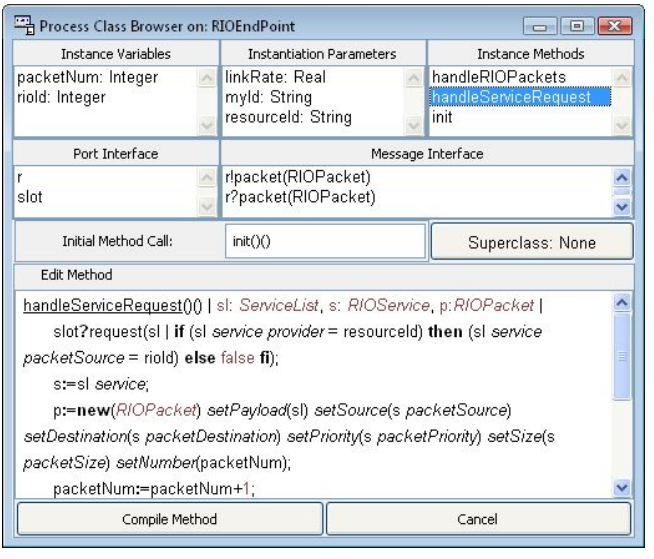


Fig. 5. Process Class RIOEndPoint

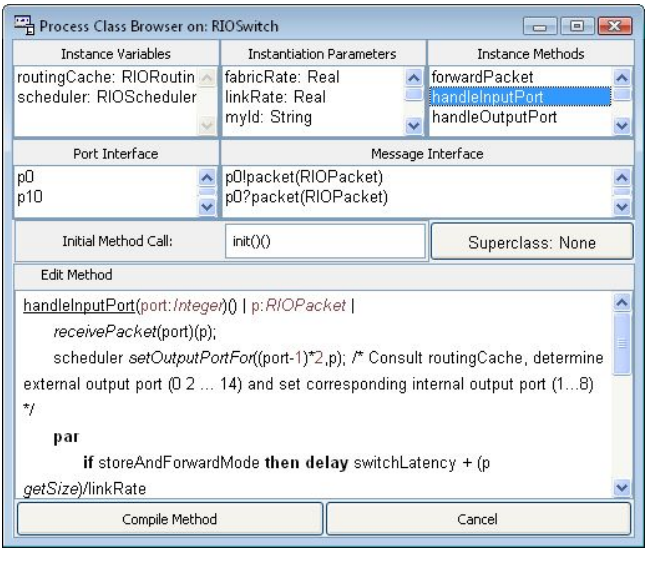


Fig. 6. Process Class RIOSwitch

3 Transformation from POOSL to UPPAAL

UPPAAL is a tool for modeling, validation and verification of real-time systems. It is based on the theory of timed automata (TA) [1] and its modeling language offers additional features such as bounded integer variables and urgency [2]. The query language of UPPAAL, used to specify properties to be checked, is a subset of CTL (computation tree logic) [8,5].

A timed automaton is a finite-state machine extended with clock variables. It uses a dense-time model where a clock variable evaluates to a real number. All the clocks progress synchronously. A system is modeled as a network of several such timed automata in parallel. The state of a system is defined by the locations of all automata, the clock constraints, and the values of the discrete variables. Every automaton may fire an edge (sometimes called a transition) separately or synchronize with another automaton, leading to a new state. We refer the reader to [2] for a more thorough description of the timed automata used in UPPAAL.

Several general transformation patterns from POOSL to UPPAAL have been characterized in [11]. In this section, we will mainly follow such patterns for the transformation. However, we note that more abstractions and techniques (such as timed automation template) have been used to simplify the transformation as well as the model.

3.1 Data Part Transformation

The POOSL model in figure 1 includes various data classes. *RIOPacket*, *RIOChannel*, *RIOQueue*, *RoutingCache* and *RIOScheduler* are the most important ones. We only explain the transformation for *RIOChannel* as it is a newly introduced data class. The transformation for other data classes has been introduced in [11] and is omitted here.

RIOChannel. *RIOChannel* is a data structure that refers to a specific kind of packet which has a specified size, source, and destination. Realistic traffic scenarios can be easily expressed by grouping different kinds of channels. This data class is transformed into an UPPAAL struct *channel*. Data methods for accessing its member variables come for free in the UPPAAL model. Other data methods such as *nrOfPacketsPerBurst* (a channel is divided into several predefined equal-sized packets, *nrOfPacketsPerBurst* refers to the number of such packets) are also easily transformed into the UPPAAL model.

3.2 Process Part Transformation

Three main improvements have been made for the process part transformation. First, timed automaton templates are used to abstract typical activities such that similar activities can be modeled by instantiating a timed automaton template with different parameters. Second, transfer activity is abstracted into a non-time-consuming activity, which greatly simplifies the model. Third, the endings of concurrent timed automata are synchronized as they consume the same time (all packets have the same size), which also simplifies the model.

Fig. 7. Channel Traffic Generation

From the POOSL model point of view, each switch contains at most 24 concurrent activities: 8 input activities (one for each input port that is actually used), 8 transfer and 8 output activities (one for each output port that is actually used). Each endpoint includes a sending and a receiving activity, depending on whether the endpoint is actually used to send/receive packets into/from the network. Besides, a traffic generation activity, a traffic sending activity and a traffic monitor activity exist in the process class *RIOChannelTrafficGenerator*. As there are 10 switches and 40 endpoints in the POOSL model, the maximum number of concurrent activities is 323 (24*10+40*2+3) for the POOSL model.

However, we will follow a new point of view to characterize such activities in the UPPAAL model. Three types of concurrent activities have been characterized:

1. Packet transfer from endpoint to switch which combines the sending activity of an endpoint and the input activity of a switch; 2. Packet transfer from switch to switch which combines the output activity of a switch and the input activity of a switch; 3. Packet transfer from switch to endpoint which combines the output activity of a switch and the receiving activity of an endpoint. Each endpoint has its own traffic sending activity. Traffic generation activity and transfer activity within a switch are abstracted into non-time-consuming activities. Traffic monitor activity is abstracted into the worst-case latency verification problem of the obtained UPPAAL model.

Channel Traffic Generation Activity. The POOSL model splits generation of channel traffic from actually sending them. Method *generateBurst* (POOSL code is shown at the bottom part of figure 2) includes most of the details for such activity. First, the POOSL model reads channel traffic information from a file *channels.txt* and creates sending queues for each involved endpoint. Then channel traffic will be generated in a uniformly random order according to the channel traffic information and stored in corresponding sending queues. Later, the sending queues will be used by corresponding endpoints for actually sending packets.

Such activity is transformed into an UPPAAL function *handleRIOChannel* where only a specific ordering (refers to the sequence of channels in sending queues) is considered (the ordering is embedded in the UPPAAL code). Sending queues are implemented as arrays, additional functions are also provided such that a FIFO accessing policy is enforced. The timed automaton which includes this activity is shown in figure 7.

If more orderings for channel traffic generation are considered, traffic generation for each channel is implemented as a timed automaton template shown in figure 8. When the UPPAAL model starts, all such timed automata will fire the edge from the initial location and execute the update *channelGen(i)* to generate

Fig. 8. Channel Generation

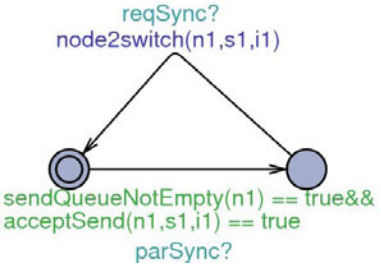


Fig. 9. Packet Transfer from Endpoint to Switch

the traffic for the channel indexed by *i*. The nondeterminism among these timed automata models the uniformly random generation of channel traffic in POOSL model.

Packet Transfer from Endpoint to Switch. We illustrate transforming the sending of packets by an endpoint and the input handling of packets received by a switch, which connects the output of an endpoint to an input port of a switch with a message. For example, message *P2* connects the output of endpoint *B1_EP1* to the input port *p2* of switch *SW2* as in figure 4. The sending of packets for a *RIOEndPoint* is specified by method *handleServiceRequest* shown in figure 5. The input handler for a *RIOSwitch* is specified by method *HandleInputPort()* shown in figure 6.

Such activity is transformed into a timed automaton template shown in figure 9. Three parameters are provided to specify the endpoint, the switch and its input port. When broadcast synchronization signal *parSync* arrives, and the condition *sendQueueNotEmpty(n1) == true && acceptSend(n1,s1,i1) == true* is satisfied, the timed automaton will move to the second location. The condition checks if endpoint *n1* is not empty and the input queue *i1* of switch *s1* can accept the current packet of endpoint *n1*. The timed automaton will stay in the second location for the time it takes to transmit the packet over the link. In our approach, the time consumed here is abstracted into the synchronization timed automaton (see figure 10) as all concurrent activities consume the same time (all packets actually transferred are equal-sized as mentioned earlier). The timed automaton will move the current packet of endpoint *n1* to the input queue *i1* of switch *s1* by function *node2switch(n1,s1,i1)* when it receives the signal *reqSync* and then return to its initial location.

The other two kinds of concurrent activities can be transformed in a similar way. We omit the details due to space limitation.

Transfer activity. The transfer activity refers to the actual scheduling of packets from input queues to a specific output port (namely output arbitration). This behavior is executed for each output port of a *RIOSwitch* and is specified by method *scheduleForOutput* in the POOSL model.

According to the configuration of the POOSL model, the transfer activity starts when the head of a packet has arrived in the input queue. Then the head is immediately forwarded to the output queue before the packet has completely arrived in the output queue. The reason is: the switch fabric rate is always larger than the link rate and therefore the packet is forwarded in "cut-through "mode. As transfer activity seamlessly connects an input queue to an output queue as if time is not consumed for this activity, we abstract it into a non-time-consuming activity which is implemented as an update as shown in figure 10.

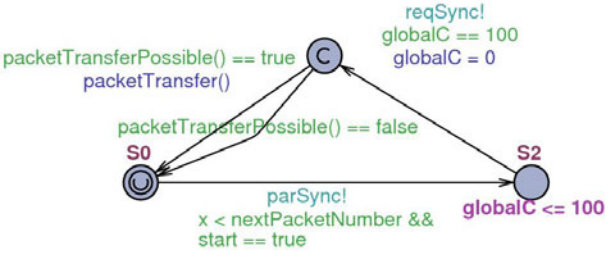


Fig. 10. Synchronization

Synchronization. To enforce that only selected timed automata are fired concurrently and are synchronized with other timed automata, we use the following timed automaton shown in figure 10. When the condition $x < nextPacketNumber$ && $start == true$ is satisfied, a broadcast synchronization signal *parSync* will be raised. Then the timed automaton will move to the second location. x denotes the number of received packets and *nextPacketNumber* denotes the number of generated packets. The condition checks if there still exist packets that have not been received and all channel traffic generations have finished. The timed automaton stays at the second location for a constant length of time (as discussed above). Then it raises a *reqSync* signal to end all concurrent activities and then return to its initial location. It will handle the transfer activity *packetTransfer* if the condition $packetTransferPossible() == true$ is satisfied during the return. Or else, it will return to the initial location directly.

Worst-case Latency Verification. The traffic monitor activity is transformed into the worst-case latency verification problem in the UPPAAL model. As we only consider the scenarios that all channels are generated in a burst, the worst-case latency is defined as the time when the whole burst has been handled (all packets have been received). We constructed the timed automaton in figure 11 to verify the worst-case latency. When condition $x == nextPacketNumber$ && $x > 0$ is satisfied, the timed automaton will move to location *S1*. A self-loop is

added here to distinguish the deadlock from end of operation. The condition checks if all packets have been received. The *endSync* signal is defined as an urgent channel to enforce that the transition will be taken immediately when the condition is satisfied. The global clock *globalT* perfectly describes the worst-case latency for the burst. Assume W is the worst-case latency, we just need to check if the property $A[]\ globalT \leq W$ is satisfied. If this property is not satisfied, we can increase W step by step until this property is satisfied. In other words, the smallest upper bound for the worst-case latency can be found iteratively.

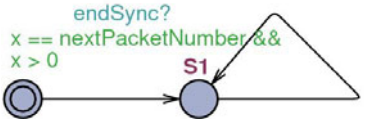


Fig. 11. Worst-case Latency Monitor

System Declaration. Based on the building blocks obtained above, we can instantiate timed automata templates into concrete timed automata according to the system configuration. In the UPPAAL model, a concrete timed automaton will be instantiated for each concurrent activity. Each endpoint is assigned an id according to the file *RIOIds.txt*, whereas the id for a switch is actually implemented as an index of an array (the total 10 switches are grouped into an array). For example, *n14s1i5 = Node2Switch(14,1,5)* denotes the instantiation of a concurrent activity referring to the packet transfer from endpoint 14 to switch 1 at input port 5. In total about 80 concurrent activities are instantiated in the UPPAAL model. Compared with more than 300 concurrent activities in the POOSL model, the UPPAAL approach is considerably simplified.

4 Heuristics

In this section, we first present the scalability of the UPPAAL model. Then we propose some heuristics to reduce the underlying state space such that realistic application scenarios can be analyzed.

4.1 Scalability of the UPPAAL Model

Different orderings for channel traffic generation result in different sending orders for the packets in endpoints which are also referred to as traffic scenarios. The difficulty of the state space exploration for the RapidIO network worst-case latency analysis comes from this combinatorial problem. Experiments have been run to show the scalability of the UPPAAL model and the results are listed in table 1. The second row refers to number of packets included in the corresponding channels. The channel information is selected from the file *channels.txt*. If n channel generation timed automata are declared in the model and the worst-case

Table 1. Scalability of the UPPAAL model

Number of Channels	10	20	30	40
Number of Packets	(129)	(300)	(372)	(459)
Supported orderings	6!	4!	4!	3!
Whole orderings	10!	20!	30!	40!

latency can be verified, we say that the UPPAAL model support $n!$ orderings (verification time is not constrained here, JVM memory is set to the maximum value (1500M) supported in our windows vista system).

From table 1, the following conclusions can be drawn: 1. With the increase of the number of channels (as well as packets), state space explosion occurs. 2. The UPPAAL model can only support investigating a small part of the state space for high volume traffics. Further, the heavier the traffic, the smaller part of the state space it investigates. Table 1 actually shows why the verification for worst-case latency is so difficult.

4.2 Heuristics

From the above analysis for the scalability of the UPPAAL model, we see that exact verification is not possible for high volume traffics. We then turn to heuristic methods and anticipate that worse packet transfer latencies can be found (than the POOSL approach). In this section, we will introduce some heuristics for state space reduction of the UPPAAL model such that high volume traffic scenarios can be analyzed.

Before we go into the details of the heuristics, we assume that: 1. The routing information and the output arbitration are fair. 2. Channel traffic is uniformly distributed in the endpoints and switches involved. 3. Traffic is never overloaded such that only a few collisions occur.

Heuristics H1. Each endpoint has its own sending queue. The sending of an endpoint is independent from others unless collisions occur. As collisions are few compared with the whole traffic load according to the assumption, the sending of each endpoint can be seen as independent most of the time. The orderings among different endpoints are not relevant for the worst-case latency and will not be considered. We can thus focus on the orderings within each endpoint. Besides, when two or more channels within an endpoint have the same type, the orderings among these channels are also irrelevant and will not be considered (as they are symmetric). We combine these two heuristics and refer it as H1. Table 2 shows the result obtained by applying heuristic H1. Compared with table 1, we can see that: the state space obtained by using the heuristic H1 is still too large to be analyzed.

Heuristic H2. Among all the activities operated in the RapidIO network, collisions only occur when two or more input queues compete for one output

Table 2. Heuristic H1

Number of Channels	10	20	30	40
Number of Packets	(129)	(300)	(372)	(459)
Reduced orderings with H1	5!	5!5!4!2!	8!6!5!5!2!2!	9!6!6!5!5!3!2!

port (queue) within a switch. Based on the above assumptions, the following observations hold. 1. The longest sending queue (endpoint) is most probably to be the last one finished. 2. The more collisions within a sending queue, the worse scenario it is. A heuristic easily follows: generating more collisions for the longest sending queue. However, it would be too complicated to implement as the detailed routing information must be investigated.

We then come up with another idea: generating more collisions for all endpoints (the whole system). It is obvious that: the more switches a packet goes through, the more likely collisions occur for this packet. We can thus put such packets in the front end of all endpoints to generate more collisions. The number of switches a channel goes through is reflected in the latency listed in table 3 (packet size is assumed to be 100 bytes). These latencies are obtained by simulation (both POOSL and UPPAAL simulator can do) for each channel (packet) type. It is obvious that: the larger the latency, the more switches it goes through. In table 3, the first row and first column both denote the id of endpoints. For example, the element 5 in the second row and fifth column means the latency from source endpoint 2 to destination endpoint is 5. Due to the space limitation, only part of the possible packet types is listed (there are 40 endpoints in the system).

We have developed a java application program to sort all channels in each endpoint such that channels are put in sending queues in descending order (the larger latency of the channel type is, the more forward this channel is in the sending queue of the endpoint) according to the table. As we have mentioned earlier, only a few orderings can be considered. We thus only investigate several different orderings for front channels in the largest sending queue.

Experiment Results. Experiments have been run with the use of the above heuristics H1 and H2 in the UPPAAL model. POOSL results are also provided

Table 3. Latency for Channel (Packet) Types)

	2	3	4	5	6	7	8	9
2	0	2	2	5	5	5	4	4
3	2	0	2	5	5	5	4	4
4	2	3	0	5	5	5	4	4
5	5	5	5	0	2	2	3	3
6	5	5	5	2	0	2	3	3
7	5	5	5	2	2	0	3	3
8	4	4	4	3	3	3	0	2
9	4	4	4	3	3	3	3	0

Table 4. Experiment Results: UPPAAL Heuristic vs. POOSL

Number of Channels	20	40	60	80	100	120	140	160
POOSL Result	128	227	298	350	504	501	504	515
UPPAAL Heuristic Result	129	220	322	371	538	539	539	522

for comparison. The first row in table 4 refers to the number of channels. The results are the worst-case packet transfer latency obtained by two approaches.

Both heuristics H1 and H2 are applied in the experiment. From table 4, we can see that: for low volume traffic scenarios, the UPPAAL approach using heuristics cannot guarantee to find worse latencies. This phenomenon comes from the fact that: POOSL's simulation engine is very effective and can explore a large part of the state space for low volume traffic scenarios. Whereas, even for low volume traffic scenarios UPPAAL can only explore a small part of the state space (see table 1). However, for high volume traffic scenarios, with the progress of state space explosion, the superiority of the high-speed engine for POOSL disappears. The UPPAAL approach using heuristics can always find worse scenarios than with the POOSL approach. The UPPAAL approach using heuristics is an efficient approach to complement the POOSL approach for finding approximate worst-case latencies.

5 Conclusions and Future Work

The exact worst-case packet transfer latency is an important metric for motion control applications that run on multiple processors interconnected by a RapidIO network. We have proposed a model checking approach using UPPAAL for this problem [11]. However, such an approach only applies to small scale models. In this paper, we extend such an approach and apply it to a realistic application scenario. First, we transform the POOSL model of a realistic motion control system into an UPPAAL model. Then we show that the application of the UPPAAL approach for exact worst-case packet transfer latency verification is limited to low volume traffics. We propose to use heuristics for high volume traffics. Although only approximate results can be obtained, the heuristics is still valuable as worse scenarios can be found than POOSL approach. Experiments show that the UPPAAL approach with heuristics is effective to complement the POOSL approach for finding approximate worst-case latencies.

In this paper, we only apply the heuristics to the UPPAAL approach. It is somewhat unfair for the comparison of the two approaches. We will apply the heuristics to POOSL approach and compare their performance in a fairly fashion. Besides, the RapidIO network (includes 323 concurrent activities) discussed in our paper is only a small part of the POOSL model of the total system, which includes an estimated 2500 processes that all include between 1 and about 10 concurrent activities. Further abstraction techniques are still needed to scale

up to such real industrial sized problems. Future work also includes transforming POOSL into UPPAAL at a more semantical level by means of additional transformation patterns[11].

References

1. Alur, R., Dill, D.: Automata for modeling real-time systems. In: Paterson, M. (ed.) ICALP 1990. LNCS, vol. 443, pp. 322–335. Springer, Heidelberg (1990)
2. Behrmann, G., David, R., Larsen, K.G.: A tutorial on UPPAAL. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 200–236. Springer, Heidelberg (2004)
3. van Bokhoven, L.: Constructive Tool Design for Formal Languages: From Semantics to Executing Models. Ph.D. thesis, Eindhoven University of Technology (2002)
4. Geilen, M.: Formal Techniques for Verification of Complex Real-Time Systems. Ph.D. thesis, Eindhoven University of Technology (2002)
5. Logothetis, G., Schneider, K.: Symbolic model checking of real-time systems. In: International Syposium on Temporal Representation and Reasoning, p. 214 (2001)
6. Milner, R.: Communication and Concurrency. Prentice-Hall, Englewood Cliffs (1989)
7. van der Putten, P., Voeten, J.: Specification of Reactive Hardware/Software Systems. Ph.D. thesis, Eindhoven University of Technology (1997)
8. Queille, J.P., Sifakis, J.: Specification and verification of concurrent systems in Cesar. In: Dezani-Ciancaglini, M., Montanari, U. (eds.) Programming 1982. LNCS, vol. 137, pp. 337–351. Springer, Heidelberg (1982)
9. Shippen, G.: A technical overview of RapidIO. (November 2007), http://www.eetasia.com/ART_8800487921_499491_NP_7644b706.HTM
10. Theelen, B.D., Florescu, O., Geilen, M., Huang, J., van der Putten, P., Voeten, J.: Software/hardware engineering with the Parallel Object-Oriented Specification Language. In: MEMOCODE 2007: Proceedings of the 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign, pp. 139–148. IEEE Computer Society, Washington (2007)
11. Xing, J., Theelen, B.D., Langerak, R., van de Pol, J., Tretmans, J., Voeten, J.: From POOSL to UPPAAL: Transformation and quantitative analysis. In: Proceedings of the International Conference on Application of Concurrency to System Design, pp. 47–56. IEEE Computer Society, Los Alamitos (2010)

Schedulability Analysis Using Uppaal: Herschel-Planck Case Study

Marius Mikučionis[1], Kim Guldstrand Larsen[1], Jacob Illum Rasmussen[1], Brian Nielsen[1], Arne Skou[1], Steen Ulrik Palm[2], Jan Storbank Pedersen[2], and Poul Hougaard[2]

[1] Aalborg University, 9220 Aalborg Øst, Denmark
{marius,kgl,illum,bnielsen,ask}@cs.aau.dk
[2] Terma A/S, 2730 Herlev, Denmark
{sup,jnp,poh}@terma.com

Abstract. We propose a modeling framework for performing schedulability analysis by using Uppaal real-time model-checker [2]. The framework is inspired by a case study where schedulability analysis of a satellite system is performed. The framework assumes a single CPU hardware where a fixed priority preemptive scheduler is used in a combination with two resource sharing protocols and in addition voluntary task suspension is considered. The contributions include the modeling framework, its application on an industrial case study and a comparison of results with classical response time analysis.

Keywords: schedulability analysis, timed automata, stop-watch automata, model-checking, verification.

1 Introduction

The goal of schedulability analysis is to check whether all tasks finish before their deadline. Traditional approaches like [5] provide generic frameworks which assume worst case scenario where worst case execution time and blocking times are estimated and then worst case response times are calculated and compared w.r.t. deadlines. Often, such conservative scenarios are never realized and thus negative results from such analysis are often too pessimistic. The idea of our method is to base the schedulability analysis on a system model with more details, taking into account specifics of individual tasks. In particular this will allow a safe but far less pessimistic schedulability analysis to be settled using real-time model checking. Moreover, the model-based approach provides a self-contained visual representation of the system with formal, non-ambiguous interpretation, simulation and other possibilities for verification and validation.

Our model-based approach is motivated by and carried out on example applications in a case study of Herschel-Planck satellite system. Compared with classical response time analysis our model-based approach is found to uniformly provide less pessimistic response time estimates and allow to conclude schedulability of all tasks, in contrast to negative results obtained from the classical approach.

T. Margaria and B. Steffen (Eds.): ISoLA 2010, Part II, LNCS 6416, pp. 175–190, 2010.

Related Work. During the last years, timed automata modelling and analysis of multitasking applications running under real-time operating systems has received substantial research effort. Here the goals are multiple: to obtain less pessimistic worst-case response time analysis compared with classical methods for single-processor systems or to relax the constraints be of period task arrival times of classical scheduling theory to task arrival patterns that can be described using timed automata.

In [13] it is shown how a multitasking application running under a real-time operating system compliant with an OSEK/VDX standard can be modelled by timed automata. Use of this methodology is demonstrated on an automated gearbox case study and the worst-case response times obtained obtained from model-checking is compared with those provided by classical schedulability analysis showing that the model-checking approach provides less pessimistic results due to a more detailed model and exhaustive state-space exploration.

Times tool [1] can be used to analyse single processor systems, however it supports only highest locker protocol (simplified priority ceiling protocol) [8]. An approach of [4] provides Java code transformation into Uppaal [2] timed-automata for schedulability analysis. Similarly, we use the model-checking framework provided by Uppaal, where the modelling language is extended with stop-watches and C-like code structures allows to express preemption, suspension and mixed resource sharing using two different protocols, in a more intuitive way without a need for more complex model transformations or workarounds.

A framework from [7] provides a generic set of tasks and resources that can be instantiated with concrete parameters (specific offsets, release times, deadlines, dependencies, periods) and processor resources together with their scheduling policies (i.e., preemption vs. non-preemption, earliest deadline first, fixed priority, first-in-first-out). The instantiated system can then be analysed for schedulability in a precise manner as all the concrete information is taken into account. This means that the framework will be able to verify schedulability of some systems that would otherwise be declared "non-schedulable" by other methods. Although the framework is general to cover multi-processor systems it does not tackle passive resource sharing protocols like priority ceiling or inheritance.

The remainder of the paper is organised as follows: in the next Section 2 we provide a brief overview of the Herschel-Planck satellite mission, its software setup and the response time analysis already carried out by Terma. Section 3 describes our model-based methodology for solving the schedulability problem, Section 4 presents the results of our method and compares it to traditional response time analysis. Finally, Section 5 discusses conclusions and future work.

2 The Herschel-Planck Mission

The Herschel-Planck mission consists of two satellites: Herschel and Planck. The satellites have different scientific objectives and thus the sensor and actuator configurations differ, but both satellites share the same computational architecture.

The architecture consists of a single processor, real-time operating system (RTEMS), basic software layer (BSW) and application software (ASW).

Terma A/S has performed an extended worst case response time analysis described in [5] by analysing [11] and [12] resulting in [10] (we provide the necessary details of those documents in this paper). The goal of the study is to show that ASW tasks and BSW tasks are schedulable on a single processor with no deadline violations. The framework uses preemptive fixed priority scheduler and a mixture of priority ceiling and priority inheritance protocols for resource sharing and extended deadlines (beyond period). In addition, some tasks need to interact with external hardware and effectively suspend their execution for a specified time. Due to suspension, this single-processor system has some elements of multi-processor systems since parts of activities are executed elsewhere and the classical worst case response analysis (applicable to single-processor systems) is pushed into extreme. One of the results of [10] is that one task may miss its deadline on Herschel (and thus the system is not schedulable) but this violation has never been observed in neither stress testing nor deployment.

The system is required to be schedulable in four instances: Herschel in nominal and event modes and Planck in nominal and event modes. The processor consumption should be less than 45% for Herschel and less than 50% for Planck.

In response time analysis, in order to prove schedulability it is enough to calculate worst case response times (WCRT) for every task and compare it with its deadline: if for every task the WCRT does not exceed the correspond deadline the system is schedulable.

Figure 1 shows the work-flow performed by Terma A/S: the deadline requirements are obtained from ASW and BSW documentation, worst case execution times (WCET) of BSW are obtained from BSW documentation [12] and ASW timings are obtained from time measurements. The tasks are carefully picked and timings aggregated (documented in [10]) and processed by the proprietary Terma tool SCHEDULE which performs worst case response time analysis as described in [5] and the outcome is processor utilisation and worst case response times (WCRT) for each task. The system is schedulable when for every task i $WCRT_i$ is less than its $Deadline_i$, i.e. the task always finishes before its deadline. We use the index i as a global task identifier.

We provide only the formulas used in response time analysis and refer to [5] for details on how this analysis works. The important property of the analysis is that it always takes conservative estimates of blocking times even though the actual blocking may not appear, thus resulting in too pessimistic response times. $WCRT_i$ is calculated by recursive formula for task computation windows $w(q)$ which may overlap with another task release (due to deadline extending past period), where q is the number of window starting with 0, $hp(i)$ is a set of tasks with higher priority than i. Then the longest window is taken as WCRT:

$$w_i^{n+1}(q) = Blocking_i + (q+1)\,WCET_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n(q)}{Period_j} \right\rceil WCET_j$$

$$Response_i(q) = w_i^n(q) - qPeriod_i$$

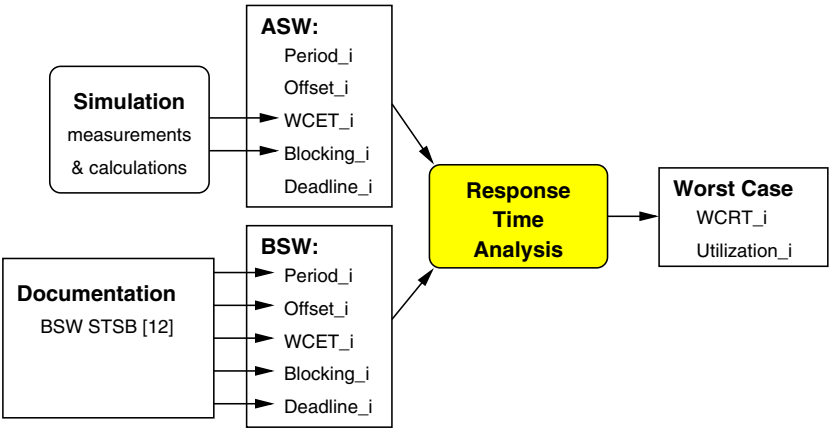


Fig. 1. Work-flow of schedulability analysis [10]

$$WCRT_i = \max_q Response_i(q)$$

$Blocking_i$ denotes the blocking time when task i is waiting for a shared resource being occupied by a lower priority task. Blocking times calculation is specific to the resource sharing protocol used. BSW tasks use priority inheritance protocol and thus their blocking times are calculated using the following equation:

$$Blocking_i = \sum_{r=1}^{R} usage(r,i)\, WCET_{CriticalSection}(r)$$

ASW tasks use priority ceiling protocol and therefore their blocking times are:

$$Blocking_i = \max_{r=1}^{R} usage(r,i)\, WCET_{CriticalSection}(r)$$

The matrix $usage(r,i)$ captures how resource r is used by the tasks: $usage(r,i) = 1$ if r is used by at least one task with a priority less than task i and at least one task with a priority greater than or equal to task i, otherwise $usage(r,i) = 0$.

Some BSW tasks are periodic and some sporadic, but we simplify the model by considering all BSW tasks as periodic. ASW tasks are started by periodic task `MainCycle`. In order to obtain more precise results, [10] splits the analysis into 0-20ms and 20-250ms windows, distinguishes two operating modes and analyse six cases in total separately. However, one case shows up as not schedulable anyway and application of our framework improves this result by showing that all tasks finish within their deadlines.

Resource Usage by Tasks. Some tasks require shared resources and those are protected by semaphore locking to ensure exclusive usage. Sometimes tasks use resources repeatedly (locking and unlocking several times). When the resource semaphore is locked, a task may suspend its execution by calling hardware services and waiting for the hardware to finish thus temporarily releasing the

processor for other tasks. The processor may be released multiple times during one semaphore lock. In response time analysis, the processor utilisation is computed by dividing a sum of worst case execution times by duration of analysed time window.

3 Model-Based Schedulability Methodology

This section explains the principles and concepts used throughout the modelling framework and then describes the modelling templates in detail.

The main idea is to translate schedulability analysis problem into a reachability problem for timed automata and use the real-time model-checker Uppaal to find worst case blocking and response times, processor utilisation and to check whether all the deadlines are met. In our modelling framework clocks and stopwatches control task release patterns, track task execution progress, check response time against deadline bound and thus all the computations are performed by model-checker according to the model, in contrast to carefully customised specific formula.

The framework consists of the following process models: fixed priority preemptive CPU scheduler, a number of task models and one process for ensuring global invariants. We provide several templates for task models: for periodic tasks and for tasks with dependencies, all of which are parameterised with concrete program control flow and may be customised to a particular resource sharing protocol. Our approach takes the same task descriptions as [10] and produces results which are more optimistic and provides the proof that all the tasks will actually finish before the deadline.

We use stopwatches to track task progress and stop the task progress during preemption. In Uppaal, the stopwatch support is implemented through a concept of derivative over clock, where the derivative can be either 1 (valuation progresses with a rate of 1 as regular clocks) or 0 (valuation is not allowed to progress – the clock is stopped). Syntactically stop-watch expressions appear in invariant expressions in a form of $x' == c$, where x is declared of type `clock` and c is an integer expression which evaluates to either 0 or 1. The reachability analysis of stopwatch automata is implemented as an over-approximation in Uppaal, but the approximation still suffices for safety properties like checking if a deadline can ever be violated.

The following outlines the main modelling ingredients:

- One template for the CPU scheduler.
- One template for "idle" task to keep track of CPU usage times.
- One template for all BSW tasks, where resources are locked based on priority inheritance protocol.
- One template for MainCycle ASW task, which is released periodically, starts other ASW tasks and locks resources based on priority ceiling protocol.
- One template for all other ASW tasks, which is released by synchronisations, and locks resources based on priority ceiling protocol.

- Task specialisation is performed during process instantiation by providing individual list of operations encoded into *flow* array of structures.
- Each task (either ASW or BSW) uses the following clocks and data variables:
  - Task and its clocks are parameterised by identifier id.
  - Execution time is modelled by a stopwatch $job[id]$ which is reset when the task is started and stopped by a global invariant when the task is not being run on the processor. A worst case execution time (WCET) guard ensures that task cannot finish before WCET elapses. To ensure progress, the clock $job[id]$ is constrained by an invariant of $WCET$ so that the task releases the processor as soon as it has finished computing.
  - A local clock x controls when the task is released and is reset upon task is released. The task then moves to an error state if x is greater than its deadline.
  - A local clock sub controls progress and execution of individual operations.
  - A local integer ic is an operation counter.
  - Worst case response time for task id is modelled by a stopwatch $WCRT[id]$ which is reset when the task is started and is allowed to progress only when the task is ready (global invariant $WCRT[id]' == ready[id]$ ensures that). In addition $WCRT[id]$ is reset when the task is finished in order to allow model checker to apply active clock reduction to speed up analysis as the value of this clock is no longer used. The worst case response time is estimated as maximum value of $WCRT[id]$.
  - An `error` location is reachable and *error* variable is set to *true* if there is a possibility to finish after deadline.

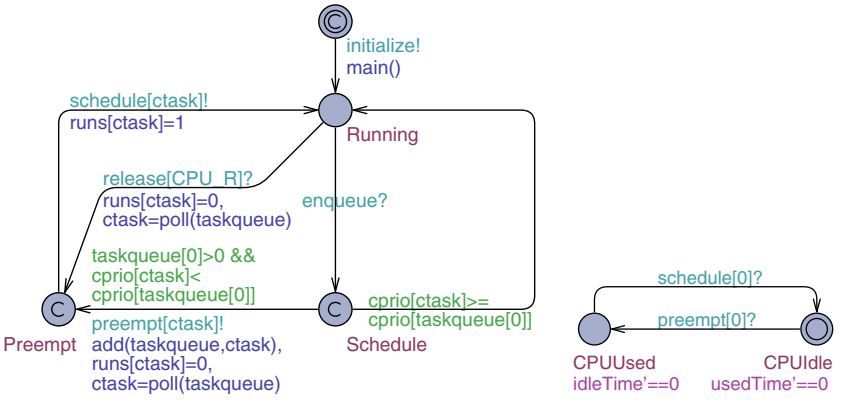
Further we explain the most important model templates, while the complete model is available for download at `http://www.cs.aau.dk/~marius/Terma/`.

3.1 Processor Scheduler

Figure 2a shows the model of CPU scheduler. In the beginning Scheduler initialises the system (computes the current task priorities by computing default priority based on `id` and starts the tasks with zero offset) and in location `Running` waits for tasks to become ready or current task to release the CPU resource. When some task becomes ready, it adds itself to the `taskqueue` and signals on `enqueue` channel, thus moving Scheduler to location `Schedule`. From location `Schedule`, the Scheduler compares the priority of a current task `cprio[ctask]` with highest priority in the queue `cprio[taskqueue[0]]` and either returns to `Running` (nothing to reschedule) or preempts the current task `ctask`, puts it into `taskqueue` and schedules the highest priority task from `taskqueue`.

The processor is released by a signal `release[CPU_R]`, in which case the Scheduler pulls the highest priority task from `taskqueue` and optionally notifies it with broadcast synchronisation on channel `schedule` (the sending is performed always in non-blocking way as receivers may ignore broadcast synchronisations).

The `taskqueue` always contains at least one ready task: IdleTask. Figure 2b shows how IdleTask reacts to Scheduler events and computes the CPU usage time with stopwatch `usedTime`, the total CPU load is then calculated as $\frac{\text{usedTime}}{\text{globalTime}}$.



(a) Template for CPU scheduler. (b) Idle task model.

Fig. 2. Models for CPU scheduler and the simplest task

3.2 Tasks Templates

Figure 3 shows the parameters which describe each periodic task: period duration showing how often the task is started, offset showing how far into the cycle the task is started (released), deadline is measured from the instance when task is started and worst case execution time within deadline.

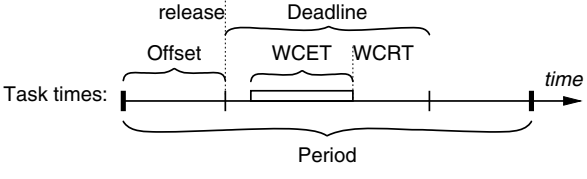


Fig. 3. Periodic task execution parameters

Figure 4 shows a template used by MainCycle which is started periodically. At first MainCycle waits for `Offset` time to elapse and moves to location `Idle` by setting the clock x to `Period`. Then the process is forced to leave `Idle` location immediately, to signal other ASW tasks, add itself to the ready task queue and arrive to location `WaitForCPU`. When MainCycle receives notification from scheduler it moves to location `GotCPU` and starts processing commands from the *flow* array. There are four types of commands:

1. LOCK is executed from location `tryLock` where the process attempts to acquire the resource. It blocks if the resource is not available and retries by adding itself to the processor queue again when resource is released. It continues to location `Next` by locking the resource if the resource is available.

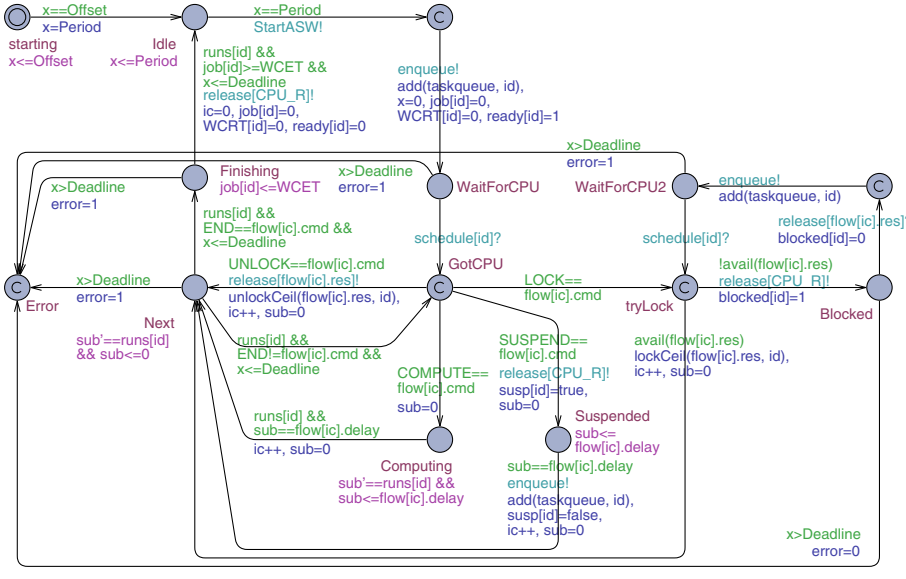


Fig. 4. MainCycle task: periodically starts ASW functions

2. UNLOCK simply releases the resource and moves on to location `Next`. The implementation of locking and unlocking is shown in Listing 1.2.
3. SUSPEND releases the processor for specified amount of time, adds itself to the queue and moves to location `Next`. The task progress clock $job[id]$ is not increasing but the response measurement clock $WCRT[id]$ is.
4. COMPUTE makes the task stay in location `Computing` for at least the specified duration of pure running time, i.e. the clock sub is stopped whenever the task is preempted and $runs[id]$ is set to 0. Once the required amount of CPU time is consumed, the process moves on to location `Next`.

From location `Next`, the process is forced by $runs[id]$ invariant to continue with the next operation: if it is not the END and it is running, then it moves back to `GotCPU` to process next operation, and it moves to `Finishing` if it's the END. In `Finishing` location the process consumed CPU for the remaining time so that complete WCET is exhausted and then it moves back to `Idle`. From locations `Next` and `Finishing` the outgoing edges are constrained to check whether the deadline has been reached since the last task release (when x was set to 0), and edges force the process into `Error` location if $x > Deadline$.

The *flow* for MainCycle is very simple: it computes for its WCET while keeping a lock on Sgm_R. A more sophisticated example of flow is in Listing 1.1. We do not know the exact times the resources are locked and the points in time are chosen arbitrarily, thus it may not necessarily lead to worst-case blocking timed for higher priority tasks.

Table 1. The description of PrimaryF task from [10] inputs

```
Primary Functions
- Data processing          20577/2521
                           Icb_R(LNS: 2, LCS: 1200, LC: 1600, MaxLC: 800)
- Guidance                 3440/0
- Attitude determination   3751/1777
                           Sgm_R(LNS: 5, LCS: 121, LC: 1218, MaxLC: 236)
- PerformExtraChecks       42/0
- SCM controller           3479/2096
                           PmReq_R(LNS: 4, LCS: 1650, LC: 3300, MaxLC: 3300)
- Command RWL              2752/85
```

The template for BSW tasks is almost the same as MainCycle, except that 1) BSW tasks do not have to start other ASW tasks and thus from `Idle` they go directly to `WaitForCPU` with enqueueing edge, 2) instead of ceiling protocol (*lockCeil* and *unlockCeil*) it uses inheritance (*lockInh* and *unlockInh*) and 3) it boosts the owners priority by calling $boostPrio(flow[ic].res, id)$ on the edge from `tryLock` to `Blocked`. BSW tasks have their own local clock x, while MainCycle shares its x with other ASW tasks.

Other ASW tasks are started by MainCycle, thus instead of broadcast shout synchronisation on `StartASW` channel they have receive synchronisation on `StartASW`. Also, they share the same clock x with MainCycle, because response time is measured from the same $20ms$ offset (as in [10], so that the results are comparable).

Table 1 shows the description of PrimaryF from [10] as an example that we used to create *flow* structure. This particular description consists of six activities.

Each activity is described by two numbers (CPU time / BSW service time, BSW service time is included in CPU time, thus is not used in our model), followed by resource usage pattern if any. The resource usage is described by the following parameters:

LNS – total number of times the CPU has been released while the resource was locked (task suspension count).
LCS – total time the CPU has been released while the resource was locked (task suspension duration).
LC – total time the resource has been locked.
MaxLC – the longest time the resource has been locked.

From this description we use only LCS and LC, where we assume that LC-LCS is the CPU busy time while the resource is locked. Listing 1.1 shows an example of detailed control flow structure for PrimaryF task, where the numbers mean the time duration and comments relate each step to an item in Table 1. Listing 1.2 shows functions for priority inheritance and priority ceiling protocols, which use *owner* and *cprio* to track current resource owner and task priority.

Listing 1.1. Operation flow for PrimaryF task

```
const ASWFlow_t PF_f = { // Primary Functions:
    { LOCK,   Icb_R, 0 },          // 0) ----- Data processing
    { COMPUTE, CPU_R, 1600-1200 }, // 1) computing with Icb_R
    { SUSPEND, CPU_R, 1200 },      // 2)  suspended with Icb_R
    { UNLOCK, Icb_R, 0 },          // 3)
    { COMPUTE, CPU_R, 20577-(1600-1200) }, // 4) computing without Icb_R
    { COMPUTE, CPU_R, 3440 },      // 5) ----- Guidance
    { LOCK,   Sgm_R, 0 },          // 6) ----- Attitude determination
    { COMPUTE, CPU_R, 1218-121 }, // 7) computing with Sgm_R
    { SUSPEND, CPU_R, 121 },       // 8)  suspended with Sgm_R
    { UNLOCK, Sgm_R, 0 },          // 9)
    { COMPUTE, CPU_R, 3751-(1218-121) }, //10) computing without Sgm_R
    { COMPUTE, CPU_R, 42 },        //11) ----- Perform extra checks
    { LOCK,   PmReq_R,0 },         //12) ----- SCM controller
    { COMPUTE, CPU_R, 3300-1650 }, //13) computing with PmReq_R
    { SUSPEND, CPU_R, 1650 },      //14)  suspended with PmReq_R
    { UNLOCK, PmReq_R, 0 },        //15)
    { COMPUTE, CPU_R, 3479-(3300-1650) }, //16) computing without PmReq_R
    { COMPUTE, CPU_R, 2752 },      //17) ----- Command RWL
    FIN, FIN, FIN, FIN, FIN, FIN, FIN, FIN, FIN, FIN, FIN // fill the array
};
```

Listing 1.2. Resource locking based on two different protocols

```
/** Check if the resource is available: */
bool avail(resid_t res) { return (owner[res]==0); }
/** Lock the resource based on priority ceiling protocol: */
void lockCeil(resid_t res, taskid_t task) {
    owner[res] = task; // mark resource occupied by task
    cprio[task] = ceiling[res]; // assume the priority of resource
}
/** Unlock the resource based on priority ceiling protocol: */
void unlockCeil(resid_t res, taskid_t task){
    owner[res] = 0; // mark resource as released
    cprio[task] = def_prio(task); // return to default priority
}
/** Boost the priority of resource owner based on priority inheritance protocol: */
void boostPrio(resid_t res, taskid_t task) {
    if (cprio[owner[res]] <= def_prio(task)) {
        cprio[owner[res]] = def_prio(task)+1;
        sort(taskqueue);
    }
}
/** Lock the resource based on priority inheritance protocol: */
void lockInh(resid_t res, taskid_t task) {
    owner[res] = task; // mark resource occupied by task
}
/** Unlock the resource based on priority inheritance protocol: */
void unlockInh(resid_t res, taskid_t task) {
    owner[res] = 0; // mark resource as released
    cprio[task] = def_prio(task); // return to default priority
}
```

3.3 System Model Instantiation

Listing 1.3 shows how tasks are instantiated. Listing 1.4 shows system declaration using priorities which help enforce the specified priorities in verification. The resulting model is deterministic, thus the expected state space shape is narrow (single sequence of steps) but potentially very deep.

Initial experiments showed that in fact the state space is so deep that UPPAAL exhausts the memory in a few minutes by storing most of the state space just to check for loops and ensure the verification termination. To address this issue

a sweep-line method [6] is used. The basic idea behind sweep-line method is to store only those passed states which have the greatest progress measure and purge the rest, thus effectively releasing and reusing most of the memory.

Listing 1.3. Task instantiation

```
1  //                taskid, Offset, Period, flow,    WCET, Deadline
2  RTEMS_RTC  =     BSW(1,   0, 10000, WCET_f, 13,     1000);
3  AswSync_SyncPulselsr=BSW(2, 0,250000, WCET_f, 70,    1000);
4  Hk_SamplerIsr =   BSW(3,62500,125000, WCET_f, 70,    1000);
5  SwCyc_CycStartIsr= BSW(4,   0,250000, WCET_f, 20,    1000);
6  SwCyc_CycEndIsr=  BSW(5,200000,250000, WCET_f, 100,  1000);
7  Rt1553_Isr      =  BSW(6,   0, 15625, WCET_f, 70,    1000);
8  Bc1553_Isr      =  BSW(7,   0, 20000, WCET_f, 70,    1000);
9  Spw_Isr         =  BSW(8,   0, 39000, WCET_f, 70,    2000);
10 Obdh_Isr        =  BSW(9,   0,250000, WCET_f, 70,    2000);
11 RtSdb_P_1       =  BSW(10,  0, 15625, WCET_f, 150,  15625);
12 RtSdb_P_2       =  BSW(11,  0,125000, WCET_f, 400,  15625);
13 RtSdb_P_3       =  BSW(12,  0,250000, WCET_f, 170,  15625);
14 // #13 is reserved for ASW resource priority ceiling
15 FdirEvents      =ASWspor(14,20000,250000, WCET_f, 5000, 230220);
16 NominalEvents_1=ASWspor(15,20000,250000, WCET_f, 720, 230220);
17 mainCycle =  MainCycle(16,20000,250000, 400,  230220, ASWclock);
18 HkSampler_P_2 =  BSW(17,62500,125000, WCET_f, 500, 62500);
19 HkSampler_P_1 =  BSW(18,62500,250000, WCET_f, 6000, 62500);
20 Acb_P           =  BSW(19,200000,250000,WCET_f, 6000, 50540);
21 IoCyc_P         =  BSW(20,200000,250000,WCET_f, 3000, 50540);
22                 //ASW: id, start, finish,    flow,   WCET, Deadline
23 primaryF =         ASW(21,StartASW,Done, PF_f, 34050, 59600, ASWclock);
24 rCSControlF=       ASW(22,StartASW,Done, RCS_f, 4070, 239600, ASWclock);
25 Obt_P           =  BSW(23,  0,1000000,Obt_f,  1100, 100000);
26 Hk_P            =  BSW(24,  0,250000, WCET_f, 2750, 250000);
27 StsMon_P        =  BSW(25,62500,250000,StsMon_f,3300, 125000);
28 TmGen_P         =  BSW(26,  0,250000, WCET_f, 4860, 250000);
29 Sgm_P           =  BSW(27,  0,250000, Sgm_f,  4020, 250000);
30 TcRouter_P      =  BSW(28,  0,250000, WCET_f, 500, 250000);
31 Cmd_P           =  BSW(29,  0,250000, Cmd_f, 14000, 250000);
32 NominalEvents_2=  BSW(30,20000,250000, WCET_f, 1780, 230220);
33 secondF_1    =      ASW(31,StartASW, Done, SF1_f, 20960, 189600, ASWclock);
34 secondF_2    =      ASW(32,StartASW, Done, SF2_f, 39690, 230220, ASWclock);
35 Bkgnd_P        =  BSW(33,  0,250000, WCET_f, 200, 250000);
```

Periodic models—like the ones for schedulability—do not have inherent progress measure, so we propose to create an artificial one based on how many cycles the system has executed so far, and then reset it to zero once it reaches a pre-specified limit. In fact, such drops in progress measure are tolerated by generalised sweep-line method [9], which is also implemented in UPPAAL. The progress measure is based on the variable *cycle* defined on line 8 in Listing 1.4, incremented by guarded loops in Global process (Fig. 5) after each 250ms and is reset together with all the global clocks to zero once the CYCLELIMIT is reached. The process Global also takes care of global invariants on $job[i]$ and $WCRT[i]$ stopwatches of each task i.

Listing 1.4. System declaration using UPPAAL priorities

```
1 system Scheduler, Bkgnd_P < secondF_2 < secondF_1 < NominalEvents_2 < Cmd_P <
2     TcRouter_P < Sgm_P < TmGen_P < StsMon_P < Hk_P < Obt_P < rCSControlF <
3     primaryF < IoCyc_P < Acb_P < HkSampler_P_1 < HkSampler_P_2 <
4     mainCycle < NominalEvents_1 < FdirEvents < RtSdb_P_3 < RtSdb_P_2 < RtSdb_P_1
          < Obdh_Isr <
5     Spw_Isr < Bc1553_Isr < Rt1553_Isr < SwCyc_CycEndIsr < SwCyc_CycStartIsr <
6     Hk_SamplerIsr < AswSync_SyncPulselsr <
7     RTEMS_RTC, IdleTask, Global;
8 progress { cycle; }
```

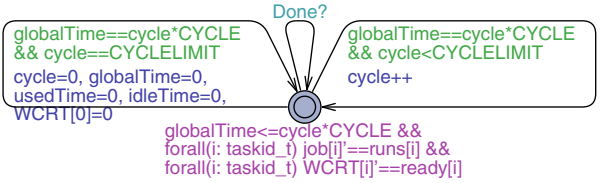


Fig. 5. Global process enforce invariants on stopwatches and cyclic progress

Further, we explore some values of CYCLELIMIT in order to minimise the verification resources. We postulate that a good heuristics is to explore at least to a hyper-period (least common multiple of periods) of all the periodic processes before reseting the cycle counter. There are the following different periods in the system: 125000, 15625, 20000, 39000, 250000 and 1000000μs, therefore a potential hyper-period is 39000000μs, or 156 cycles of 250ms each, but it can be much larger due to non-trivial resource sharing and task interaction.

3.4 Verification Queries

The following is a list of queries used to check schedulability properties:

- Check if the system is schedulable (the error state is not reachable): `E<> error`
- Check if any task can be blocked at all: `E<> exists(i:taskid_t) blocked[i]`
- Find the total worst CPU usage: `sup: usedTime, idleTime`
- Find the worst case response times: `sup: WCRT[0], WCRT[1], ... WCRT[33]`
- Find worst case blocking times, where $B[i]$ is a stopwatch growing when task i is blocked: `sup: B[0], B[1], B[2], ... B[33]`

A `sup`-query explores the entire reachable state space and computes the maximum (supremum) value of an argument expression, which is useful for computing several bounds at once. However, in such queries, UPPAAL treats the specified clocks as active, therefore the exploration can be significantly slower when the clock list is large. Therefore, we create a separate model to estimate blocking times instead WCRT by purging expressions with $WCRT[id]$ and adding $B[id]$ reset statements on edges from `tryLock` to `Blocked` in order to save half of expensive stopwatches.

4 Results

The results of our model-based framework consist of three parts: visualisation of a schedule in Gantt chart, worst case response times estimates and CPU utilisation estimation with verification benchmarking based on cycle limit.

Visualisation. A Gantt chart can be used to visualise a trace of the system, thus providing a rich picture for inspection. For example, in the generated chart, it can be seen that Cmd_P is blocked more than 5 times during the first cycle, while blocking times for PrimaryF and StsMon_P are significantly long only starting

from the second cycle. Listing 1.5 shows a chart declaration accepted by UPPAAL TIGA [3] which assigns colours for each line (T for task lines, R for resource lines) based on the state (ready, running, blocked or suspended; locked and used, locked and preempted or locked and suspended respectively).

Listing 1.5. Specification for Gantt chart

```
gantt {
  T(i: taskid_t):
    (ready[i] && !runs[i]) -> 1,// green: ready
    (ready[i] && runs[i]) -> 2, // blue:  running
    (blocked[i]) -> 0,          // red:   blocked
    susp[i] -> 9;               // cyan:  suspended
  R(i: resid_t):
    (owner[i]>0 && runs[owner[i]]) -> 2, // blue: locked and actively used
    (owner[i]>0 && !runs[owner[i]] && !susp[owner[i]]) -> 1, // green: locked, preempted
    (owner[i]>0 && susp[owner[i]]) -> 9; // cyan: locked and suspended
}
```

Verification and CPU Load Estimates. UPPAAL takes about 2min (112s) to verify that the system is schedulable and about 3 times as much to find WCRT on a Linux laptop PC with Intel Core 2 Duo 2.2GHz processor.

In [10] CPU utilisation for 20-250ms window is estimated as 62.4%, and our estimate for entire worst case cycle is 63.65% which is slightly larger, possibly due to the fact that it also includes the consumption during 0-20ms window.

Table 2 shows UPPAAL verification resources used for estimating WCRT and CPU utilisation for various cycle limits. The instances where cycle limit is a divisor or a multiple of a hyper-period (156) are in bold. Notice that for such cycle limits the verification resources are orders of magnitude lower, and there is nearly perfect linear correlation between cycle limit and resource usage in both sub-sequences when evaluated separately (both coefficients are ≥ 0.993).

Table 2. Verification resources and CPU utilisation estimates

cycle	Uppaal resources			Herschel CPU utilization				
limit	CPU, s	Mem, KB	States, #	Idle, μs	Used, μs	Global, μs	Sum, μs	Used, %
1	465.2	60288	173456	91225	160015	250000	251240	0.640060
2	470.1	59536	174234	182380	318790	500000	501170	0.637580
3	461.0	58656	175228	273535	477705	750000	751240	0.636940
4	474.5	58792	176266	363590	636480	1000000	1000070	0.636480
6	474.6	58796	178432	545900	955270	1500000	1501170	0.636847
8	912.3	58856	352365	727110	1272960	2000000	2000070	0.636480
13	507.7	58796	186091	1181855	2069385	3250000	3251240	0.636734
16	1759.0	58728	704551	1454220	2545850	4000000	4000070	0.636463
26	541.9	58112	200364	2363640	4137530	6500000	6501170	0.636543
32	3484.0	75520	1408943	2908370	5091700	8000000	8000070	0.636463
39	583.5	74568	214657	3545425	6205745	9750000	9751170	0.636487
64	7030.0	91776	2817704	5816740	10183330	16000000	16000070	0.636458
78	652.2	74768	257582	7089680	12411420	19500000	19501100	0.636483
128	14149.4	141448	5635227	11633480	20366590	32000000	32000070	0.636456
156	789.4	91204	343402	14178260	24821740	39000000	39000000	0.636455
256	23219.4	224440	11270279	23266890	40733180	64000000	64000070	0.636456
312	1824.6	124892	686788	28356520	49643480	78000000	78000000	0.636455
512	49202.2	390428	22540388	46533780	81466290	128000000	128000070	0.636455
624	3734.7	207728	1373560	56713040	99286960	156000000	156000000	0.636455

Table 3. Specification, blocking and worst case response times of individual tasks

ID	Task	Specification			Blocking times			WCRT		
		Period	WCET	Deadline	Terma	UPPAAL	Diff	Terma	UPPAAL	Diff
1	RTEMS_RTC	10.000	0.013	1.000	0.035	0	0.035	0.050	0.013	0.037
2	AswSync_SyncPulseIsr	250.000	0.070	1.000	0.035	0	0.035	0.120	0.083	0.037
3	Hk_SamplerIsr	125.000	0.070	1.000	0.035	0	0.035	0.120	0.070	0.050
4	SwCyc_CycStartIsr	250.000	0.200	1.000	0.035	0	0.035	0.320	0.103	0.217
5	SwCyc_CycEndIsr	250.000	0.100	1.000	0.035	0	0.035	0.220	0.113	0.107
6	Rt1553_Isr	15.625	0.070	1.000	0.035	0	0.035	0.290	0.173	0.117
7	Bc1553_Isr	20.000	0.070	1.000	0.035	0	0.035	0.360	0.243	0.117
8	Spw_Isr	39.000	0.070	2.000	0.035	0	0.035	0.430	0.313	0.117
9	Obdh_Isr	250.000	0.070	2.000	0.035	0	0.035	0.500	0.383	0.117
10	RtSdb_P_1	15.625	0.150	15.625	3.650	0	3.650	4.330	0.533	3.797
11	RtSdb_P_2	125.000	0.400	15.625	3.650	0	3.650	4.870	0.933	3.937
12	RtSdb_P_3	250.000	0.170	15.625	3.650	0	3.650	5.110	1.103	4.007
14	**FdirEvents**	250.000	5.000	230.220	0.720	0	0.720	7.180	5.153	2.027
15	**NominalEvents_1**	250.000	0.720	230.220	0.720	0	0.720	7.900	5.873	2.027
16	**MainCycle**	250.000	0.400	230.220	0.720	0	0.720	8.370	6.273	2.097
17	HkSampler_P_2	125.000	0.500	62.500	3.650	0	3.650	11.960	5.380	6.580
18	HkSampler_P_1	250.000	6.000	62.500	3.650	0	3.650	18.460	11.615	6.845
19	Acb_P	250.000	6.000	50.000	3.650	0	3.650	24.680	6.473	18.207
20	IoCyc_P	250.000	3.000	50.000	3.650	0	3.650	27.820	9.473	18.347
21	**PrimaryF**	250.000	34.050	**59.600**	5.770	0.966	4.804	**65.470**	**54.115**	11.355
22	**RCSControlF**	250.000	4.070	239.600	12.120	0	12.120	76.040	53.994	22.046
23	Obt_P	1000.000	1.100	100.000	9.630	0	9.630	74.720	2.503	72.217
24	Hk_P	250.000	2.750	250.000	1.035	0	1.035	6.800	4.953	1.847
25	StsMon_P	250.000	3.300	125.000	16.070	0.822	15.248	85.050	17.863	67.187
26	TmGen_P	250.000	4.860	250.000	4.260	0	4.260	77.650	9.813	67.837
27	Sgm_P	250.000	4.020	250.000	1.040	0	1.040	18.680	14.796	3.884
28	TcRouter_P	250.000	0.500	250.000	1.035	0	1.035	19.310	11.896	7.414
29	Cmd_P	250.000	14.000	250.000	26.110	1.262	24.848	114.920	94.346	20.574
30	**NominalEvents_2**	250.000	1.780	230.220	12.480	0	12.480	102.760	65.177	37.583
31	**SecondaryF_1**	250.000	20.960	189.600	27.650	0	27.650	141.550	110.666	30.884
32	**SecondaryF_2**	250.000	39.690	230.220	48.450	0	48.450	204.050	154.556	49.494
33	Bkgnd_P	250.000	0.200	250.000	0.000	0	0.000	154.090	15.046	139.044

Herschel CPU utilisation estimate does not improve much, therefore we conclude that individual cycles are very similar. The sum of idle and used times is slightly larger than global supremum meaning that some cycles are only slightly more stressed than others.

Worst Case Response Times. Table 3 shows the response timed from UPPAAL analysis in comparison to response time analysis by Terma. For most of BSW tasks (1-12,17-18) resource patterns are not available and thus UPPAAL could not determine their blocking times. Blocking times by Terma also include the suspension times related to locking of resources. We note that in all cases the WCRT estimates provided by UPPAAL are smaller (hence less pessimistic) than those originally obtained [10]. In particular, we note that the task PrimaryF (task 21) is found to be schedulable using model-checking in contrast to the original negative result obtained by Terma.

5 Discussion

We have shown how the UPPAAL model-checker can be applied for schedulability analysis of a system with single CPU, fixed priorities preemptive scheduler, mixture of periodic tasks and tasks with dependencies, and mixed resource sharing

protocols. Worst case response times (WCRT), blocking times and CPU utilisation are estimated by model-checker according to the system model structure. Modelling patterns use stopwatches in a simple and intuitive way. A breakthrough in verification scalability for large systems (more than 30 tasks) is achieved by employing sweep-line method. Even better control over verification resources can be achieved by carefully designing progress measure.

The task templates are demonstrated to be generic through many instantiations with arbitrary computation sequences and specialised for particular resource sharing. The framework is modular and extensible to accommodate a different scheduler and control flow can be expanded with additional instructions if some task algorithm is even more complicated. In addition, Uppaal toolkit allows easy visualisation of the schedule in Gantt chart and the system behaviour can be examined in both symbolic and concrete simulators.

The case study results include a self-contained non-ambiguous model which formalises many assumptions described in [10] in human language. The verification results demonstrate that the timing estimates correlate with figures from the response time analysis [10]. The worst case response time of PrimaryF is indeed very close to deadline, but overall all estimates by Uppaal are lower (more optimistic) and they all ($WCRT_{21}$ in particular) are below deadlines, whereas the response time analysis found that PrimaryF may not finish before deadline and does not provide any more insight on how the deadline is violated or whether such behaviour is realizable.

References

1. Amnell, T., Fersman, E., Mokrushin, L., Pettersson, P., Yi, W.: TIMES – a tool for modelling and implementation of embedded systems. In: Katoen, J.-P., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, pp. 460–464. Springer, Heidelberg (2002)
2. Behrmann, G., David, A., Larsen, K.: A tutorial on Uppaal. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 200–236. Springer, Heidelberg (2004)
3. Behrmann, G., Cougnard, A., David, A., Fleury, E., Larsen, K.G., Lime, D.: Uppaal-tiga: Time for playing games! In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 121–125. Springer, Heidelberg (2007)
4. Bøgholm, T., Kragh-Hansen, H., Olsen, P., Thomsen, B., Larsen, K.G.: Model-based schedulability analysis of safety critical hard real-time java programs. In: Bollella, G., Locke, C.D. (eds.) JTRES. ACM International Conference Proceeding Series, vol. 343, pp. 106–114. ACM, New York (2008)
5. Burns, A.: Preemptive priority based scheduling: An appropriate engineering approach. In: Principles of Real-Time Systems, pp. 225–248. Prentice-Hall, Englewood Cliffs (1994)
6. Christensen, S., Kristensen, L., Mailund, T.: A Sweep-Line method for state space exploration. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 450–464. Springer, Heidelberg (2001), http://dx.doi.org/10.1007/3-540-45319-9_31
7. David, A., Illum, J., Larsen, K.G., Skou, A.: Model-Based Framework for Schedulability Analysis Using UPPAAL 4.1. In: Model-Based Design for Embedded Systems, pp. 93–119. CRC Press, Boca Raton (2010)

8. Fersman, E.: A generic approach to schedulability analysis of real-time systems. Acta Universitatis Upsaliensis (2003)
9. Kristensen, L., Mailund, T.: A generalised Sweep-Line method for safety properties. In: Eriksson, L.-H., Lindsay, P.A. (eds.) FME 2002. LNCS, vol. 2391, pp. 215–229. Springer, Heidelberg (2002), http://dx.doi.org/10.1007/3-540-45614-7_31
10. Palm, S.: Herschel-Planck ACC ASW: sizing, timing and schedulability analysis. Tech. rep., Terma A/S (2006)
11. Terma A/S: Herschel-Planck ACMS ACC ASW requirements specification. Tech. rep., Terma A/S (Issue 4/0)
12. Terma A/S: Software timing and sizing budgets. Tech. rep., Terma A/S (Issue 9)
13. Waszniowski, L., Hanzálek, Z.: Formal verification of multitasking applications based on timed automata model. Real-Time Systems 38(1), 39–65 (2008)

Model-Checking Temporal Properties of Real-Time HTL Programs

André Carvalho[2], Joel Carvalho[1],
Jorge Sousa Pinto[2], and Simão Melo de Sousa[1]

[1] Departamento de Informática, Universidade da Beira Interior, Portugal,
and LIACC, Universidade do Porto, Portugal
[2] Departamento de Informática / CCTC
Universidade do Minho, Braga, Portugal

Abstract. This paper describes a tool-supported method for the formal verification of timed properties of HTL programs, supported by the automated translation tool HTL2XTA, which extracts from a HTL program (i) an Uppaal model and (ii) a set of properties that state the compliance of the model with certain automatically inferred temporal constraints. These can be manually extended with other temporal properties provided by the user. The paper introduces the details of the proposed mechanisms as well as the results of our experimental validation.

1 Introduction

New requirements arise from the continuous evolution of computer systems. Processing power alone is not sufficient to satisfy all the industrial requirements. For instance in the context of critical systems, the safety and reliability aspects are fundamental [14]: it is not sufficient to merely provide the technical means for a set of tasks to be executed; it is also required that the system (as a whole) can correctly execute all of the tasks in due time. The focus of this paper is precisely on the reliability of safety-critical systems. Such systems are usually real-time systems [11] that add to traditional reliability requirements the intrinsic need to ensure that tasks are executed within a well-established time scope. For such systems, missing these timing requirements corresponds to a system failure.

Our study considers the Hierarchical Timing Language (HTL) [5,6,10] as a basis for real-time system development, and addresses the issue of the (automated) formal verification of timing requirements. Since HTL is a coordination language [4] for which schedulability analysis is decidable, our focus here is on the verification of complementary timing properties. The verification framework we propose relies on model checking based on timed automata and timed temporal logic. The contribution of this paper is a detailed description of the methodology and its underlying tool-supported verification mechanism.

Our tool takes as input a HTL program and extracts from it an Uppaal model and a set of proof obligations that correspond to certain expected timed temporal properties. The resulting model can be used to run a timed simulation of the

T. Margaria and B. Steffen (Eds.): ISoLA 2010, Part II, LNCS 6416, pp. 191–205, 2010.

program execution, and the properties can be checked using the proof facilities provided by the Uppaal tool. With the help of these mechanisms, the development team can audit the program against the expected temporal behaviour.

Motivation and Related Work. The HTL language is derived from Giotto [9]. Giotto-based languages share the important feature that they allow one to statically determine the schedulability of programs. Although academic, these languages have a number of interesting properties that cannot be found in languages currently used in industry, including efficient reuse of code; theoretical ease of adaptation of a program to several platforms; hierarquical construction of programs; and the use of functional features of languages without limitations.

HTL introduces several improvements with respect to Giotto, but the HTL platform still lacks verification mechanisms to complement schedulability analysis, in order to allow the language to compete with other tools more widely used in industry. Bearing in mind this aspect, we propose to complement the verification of temporal HTL with model checking [3]. While the static analysis performed by the HTL compiler enforces the schedulability (seen as a safety property) of the set of tasks in a program, a model checker allows the system designer to perform a temporal analysis of the tasks' behaviour from the specified timing requirements – an aspect that is ignored by the HTL tools.

The verification methodology proposed in this paper is inspired by [13], but uses a different abstraction based on the *logical execution time* of each task. Unlike [13], a key point of our tool chain is that the verification is fully automatic. [12] proposes the use of Uppaal with a related goal: the verification of a Ravenscar-compliant scheduler for Ada applications.

HTL. The Hierarchical Timing Language [6,5,10,7] is a coordination language [4] for real-time critical systems with periodic tasks, which allows for the static verification of the schedulability of the implemented tasks. The aim of coordination languages is the combination and manipulation of programs written in heterogeneous programming languages. A system may be implemented by providing a set of tasks written in possibly different programming languages, together with a HTL layer, and additionally specifying how the tasks interact. This favours a clear separation, in the system design, between the functional layer and the concurrent and temporal aspects. The HTL toolchain provides code generators that translate the HTL layer into executable code of the target execution platform.

A fundamental aspect of HTL is the *Logical Execution Time* (LET), that provides an abstraction for the physical execution of tasks. The LET of a task considers a time scope in which the task can be executed regardless of how the operating system assigns resources to this task. The LET of a periodic task implementing a *read data; process; write data* cycle begins in the instant when the last variable is read and ends when the first variable is written.

For illustration purposes, we give in Listing 1.1 an excerpt of a HTL program (based on the 3TS_Simulink case study, see Section 5). A HTL program is composed by a number of main commands which allow programmers to describe the desired behaviour of almost any program. These commands are *communicator*,

```
1  module IO start readWrite{
2    task t_read
3         input() state()
4         output(c_double p_h1, c_double p_h2,c_bool p_V1, c_bool p_V2)
5         function f_read;
6    (...)
7
8    mode readWrite period 500{
9         invoke t_read
10             input()
11             output((h1,3), (h2,3), (v1,1), (v2,1));
12        (...)
13   }
14 }
```

Listing 1.1. 3TS Simulink code snippet

module, *task*, *port*, *mode*, *invoke* and *switch*. Briefly, a communicator is a typed variable which can be accessed any time during the execution; modules have to be declared after communicators and their bodies are composed by ports, tasks and modes. At least one (initial) mode must be declared. The task command, as the name indicates, is used to declare tasks, taking as arguments possible input/output ports and a Worst Case Execution Time (WCET) estimation. Similarly to a communicator, a port is a typed variable accessed during program execution, but in this case declared inside a module. The set of modes declared inside a module defines the module's behaviour. Through the modes declaration it is possible to know which tasks will be executed, and at which moment. The invocations are responsible for dictating when the tasks should be executed, and define the LET of each task. Finally, the switch command, which takes as input a condition and a mode identifier, is used to change the current execution mode.

HTL favours a layered approach to the development of programs. Tasks can be organized in refinements that allow programmers to provide details gradually, and also allow for a more finely grained task structure. A concrete task refines an abstract task if it has the same frequency as the abstract task and it is able to provide a time behaviour that is at least as good as the behaviour of the abstract task. The notion of refinement correctness is then expressed in terms of *time safety*. The refined task must be time-indistinguishable from the abstract task; a concrete HTL program is schedulable if it contains only time-safe refinements of the tasks of a schedulable abstract HTL program.

Uppaal. The Uppaal tool is a modelling application developed at the universities of **Upp**sala and **Aal**borg, based on networks of timed automata [2]. The tool offers simulation and verification functionality based on model checking of formulas of a subset of the TCTL logic [1]. Uppaal is particularly suitable for modeling and analysing the timed behaviour of a set of tasks; properties like *two given tasks t_1, t_2 do not reach the states A and B simultaneously* are typical of the kind of analyses that can be performed with Uppaal.

Since the model checking engine is independent from the GUI, both visual and textual representations of timed automata can be used for the verification tasks.

This is particularly interesting when Uppaal is used in cooperation with other tools. Timing requirements (target properties to be checked) can be specified using the editing facilities of the GUI, or separately in a file. This last approach is used by the toolchain introduced in this paper.

2 The HTL2XTA Toolchain

The purpose of the verification methodology proposed in this paper is to extend the verification capabilities provided by the HTL platform. Given a HTL program and the schedulability analysis provided by the regular HTL toolchain [7], the methodology consists in the following two steps:

1. From a HTL program, the HTL2XTA translator produces two files: one (.xta) contains a model of the program (timed automata); the other (.q) a set of automatically inferred properties (timed temporal logic formulas). The translation algorithm has a recursive structure and requires only two depth-first traversals of the AST: the first one produces the model and the second one infers the properties.
2. Both these files are fed to the Uppaal model checker; the GUI or the model checker engine (*verifyta*) can be used to check if the properties are satisfied.

We remark that the automatically generated properties correspond to relatively simple timing requirements; formulas for more complex requirements, such as "task X must not execute at the same time as task Y", or "if task X executes, then after T time units task Y must also execute" are not automatically generated, but can of course be manually incorporated in the .q file after the first step above. Writing the appropriate TCTL formulas must of course take into consideration the requirements and the generated model. We now turn to an exploration of the involved translation mechanisms, which will be detailed in the next two sections.

Model Translation. With the classic state space explosion limitation of model checking [3] in mind, and given the central role of the models in the verification process, it was decided to avoid translation schemes that would result in the construction of very complex models. Therefore, and given that the HTL platform already performs a scheduling analysis, the translation abstracts away from the physical execution of tasks, unlike, say, the approach described in [13]. As such, we consider that the notion of LET is sufficient to allow the remaining interesting timing properties to be checked. A network of timed automata is then obtained from a HTL program as follows:

– Each task is modeled as a single automaton with its own LET, calculated from the concrete ports and the communicators given in the task's declaration. The lower bound of the LET corresponds to the instant in which the last variable reading is performed, and the upper bound to the instant in which the first variable writing is performed.

– For each module in the HTL program a timed automaton is created. Note that each mode in a module represents the execution of a set of tasks, and that, at any moment, each module can only be in one operation mode, thus there is no need to have more than one automaton for each module. Whenever the (module) automaton performs an execution cycle, it will synchronize with the automata representing the tasks invoked in the specified mode. The level of abstraction adopted completely ignores the type of communicator as well as the initialization driver.

Since HTL is a hierarchical coordination language, a very relevant aspect is the number of refinements in the program (directly related to program hierarchies), which can naturally increase the complexity of the model substantially. By default, the translation process reflects faithfully the refinement present in the HTL programs. However in some cases this could make the model exploration impracticable, and for this reason the translator allows for the construction of models to take the desired level of refinement as input.

Inference of Properties. Listing 1.2 shows examples of automatically produced timing properties. The automatically inferred properties are all related with some HTL feature, like the modes' periods, the LET of each task, the tasks invoked in each mode, and the program refinement. To allow for traceability, each property is annotated with a textual description of the feature to check, a reference to the position of the respective feature in the HTL file, and the expected verification result. The inferred properties can and should be manually complemented with information extracted from the established temporal requirements. The automata corresponding to a given module and tasks, as well as the states corresponding to task invocations and LETs, are identified by clearly defined labels, which facilitates writing properties manually.

```
1  /* Deadlock Free -> true */
2  A[] not deadlock
3
4  /* P1 mode readWrite period 500 @ Line 19 -> true */
5  A[] sP_3TS_IO.readWrite imply ((not sP_3TS_IO.t>500) && (not sP_3TS_IO.t
       <0))
6
7  /* P2 mode readWrite period 500 @ Line 19 -> true */
8  sP_3TS_IO.readWrite --> (sP_3TS_IO.Ready && (sP_3TS_IO.t==0  ||
       sP_3TS_IO.t==500))
9
10 /* P1 Let of t_write = [400;500] @ Line 21 -> true */
11 A[] (IO_readWrite_t_write.Let imply (not IO_readWrite_t_write.tt<400 &&
       not IO_readWrite_t_write.tt>500))
```

Listing 1.2. Example of annotated properties

3 Model Translation

Some aspects of HTL are purely ignored by the translation process, either because they do not bring any relevant information, or because the abstraction level

of the model is not sufficient to cope with it. The translation process is syntax-oriented and based on the abstract syntax tree (AST) of the HTL language, which was built using a HTL grammar. It supports all of the HTL language, however there is information that is not analysed or translated by the tool.

Let us consider the definition of the function T that takes as input a HTL program and returns a network of timed automata (NTA). Naturally, this function is defined recursively over the structure of the AST. An auxiliary function A is used for task invocation analysis, that takes as argument a HTL program and returns relevant information to build the NTA.

Translation of Mode_Switch. Consider the abstract representation of a switch instruction as the tuple (n, s, p), where n is the name of the mode for which the change of execution is pretended, s the name of the function (in the functional code) that evaluates whether the change should take place, and p the declaration position in the HTL file. Let *Prog* denote the set of all programs, then we have $\forall switch \in Prog, T_{switch}(n, s, p) = \emptyset$. Note that the non-determinism of Uppaal will be important to guarantee that the modes are alternated during the execution. The translation itself is not affected by any mode switches.

Translation of Types and Initialization Drivers. Let ct be a type and ci the declaration of the initialization driver. We have $\forall dt \in Prog, T_{dt}(ct, ci) = \emptyset$. Neither the type nor the initial value (initialization driver) of a declaration have any impact on the application of the translation process. This information does not contribute to the temporal analysis.

Translation of Task Declarations. Consider the abstract representation of a task as the tuple (n, ip, s, op, f, w, p), where n is the name of the task, ip the list of *input ports*, s the list of internal states, op the list of *output ports*, f the name of the function which implements the task, w the task's WCET, and finally p the task declaration position in the HTL file. We have $\forall task \in Prog, T_{task}(n, ip, s, op, f, w, p) = \emptyset$. Analogously to the previous situations, task declarations do not have any impact on the translation.

Translation of Communicator Declarations. Consider the abstract representation of a communicator as the tuple (n, dt, pd, p), where n is the communicator's name, dt the communicator's type with ct, ci as initialization driver, pd the communicator's period and p the communicator's declaration position in the HTL file, then $\forall communicator \in Prog, T_{communicator}(n, dt, pd, p) = \emptyset$.

Once more the translator ignores the declaration. In order to evaluate the LET (see below) the following clause is defined for the auxiliary function A: $\forall communicator \in Prog,\ A_{communicator}(com, dt, pd, p) = pd$, even if the communicator com does not have a direct representation in the model given the abstraction level adopted.

Translation of Ports Declaration. Let the abstract representation of a port be the tuple (n, dt, p), where n is the ports's name, dt the port's type with ct, ci the initialization driver and p the port's declaration position in the HTL file, then $\forall port \in Prog, T_{port}(n, dt, p) = \emptyset$. The port's declaration is ignored in

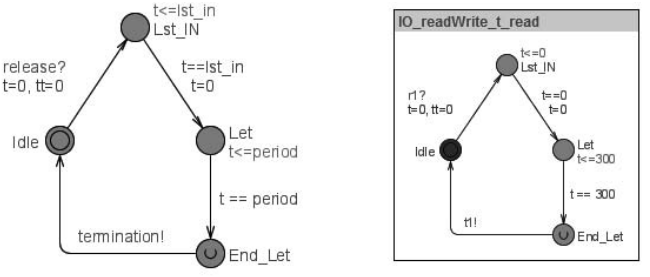


Fig. 1. *taskTA* automata on the left and instantiation on the right

the translation, and moreover no task invocation analysis is performed. In task invocations ports are just names.

LET Transposition

This translation is based on an implementation of the concept of LET, based on the timed automata *taskTA*, *taskTA_S*, *taskTA_R* and *taskTA_SR*. These four automata result from the use of concrete ports in the task invocations: $taskTA$ represents the task invocations where only communicators are used, $taskTA\_S$ (S for *send*) those where a single concrete port is used as output, $taskTA\_R$ (R for *receive*) those where a single concrete port is used as input, and finally $taskTA\_SR$ where two concrete ports are used, as input and as output.

In the following, a task invocation will be seen in abstract terms as the tuple (n, ip, op, s, pos) where n is the invoked task's name, ip is the input port's (variables) mapping, op the output port's (variable) mapping, s the name of the task's parent, and finally pos is the task's declaration position in the HTL file.

TaskTA. Let *Port* be the set of all concrete ports, cp be one concrete port, and $taskTA(r, t, p, li)$ be a timed automaton where r is a *release* urgent synchronization, t is a *termination* urgent synchronization, p the task's LET period and li the exact moment where the last variable is read. Then we have

$$\forall cp \in Port, \forall invoke \in Prog, cp \notin ip, cp \notin op, \Rightarrow \\ T_{invoke}(n, ip, op, s, pos) = taskTA(r, t, p, li)$$

Each task invocation in which no concrete ports are used either in the input or in the output variables, gives rise to an automaton *taskTA* (see Figure 1). The urgent synchronization channels r and t are calculated in the system declaration. For each task instantiation the channel r has an unique name, produced by an enumeration $r_1, r_2, r_3, \ldots$ Similarly, the channel t has an unique name for each set of mode automata, produced by an enumeration $t_1, t_2, t_3, \ldots$

The instant at which the last input variable li is read is calculated as a product of the maximum value of each instance of input communicator and the period (in the case of non-existence of input variable, this instant is considered to be

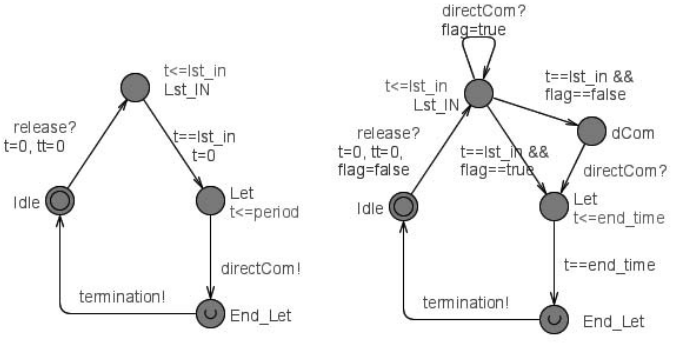


Fig. 2. $taskTA\_S$ automaton on the left and $taskTA\_R$ on the right

zero). The LET's period p is the subtraction between the instant where the first output port is written (in case of non-existence, the value is the respective mode's period) and li.

TaskTA_S. Let $taskTA\_S(r, t, dc, p, li)$ be a timed automaton where r is the *release* urgent synchronization, t a *termination* urgent synchronization, dc the urgent synchronization of a direct communication (*directCom*), p the task's LET period and li the instant where the last input variable is read, then

$$\forall cp \in Port, \forall invoke \in Prog, cp \notin ip, cp \in op, \Rightarrow \\ T_{invoke}(n, ip, op, s, pos) = taskTA\_S(r, t, dc, p, li)$$

For each task invocation, the existence of a concrete port in the set of output variables and non-existence in the set of input variables originates the instantiation of a $taskTA\_S$ automaton (Figure 2, left). This automaton is very similar to $taskTA$ – the difference is just the inclusion of direct communication.

TaskTA_R. Let $taskTA\_R(r, t, dc, p, li)$ be a timed automaton with r, t, dc, p, and li the same as in the previous case, then

$$\forall cp \in Port, \forall invoke \in Prog, cp \in ip, cp \notin op, \Rightarrow \\ T_{invoke}(n, ip, op, s, pos) = taskTA\_R(r, t, dc, p, li)$$

For each task invocation, the existence of a concrete port in the set of input variables and non-existence in the set of output variables originates the instantiation of $taskTA\_R$ automaton (Figure 2, right). This automaton is slightly more complex than $taskTA\_S$ since it considers two alternative paths for the initialization of the task's LET. The first one encodes the direct communication done before the reading of the last communicator (with no impact on the LET's start) and the later encodes awaiting of the port reading after all communicators have been read (the LET's start becomes dynamic). In this last case, the start of the LET depends on a direct communication with another task in the same mode.

Modules and Modes. Consider a module abstracted as a tuple (n, h, mi, bm, pos), where n is the module's name, h a list of hosts, mi the initial mode, bmu the module's body and pos the module's declaration position in the HTL file.

Let $moduleTA(ref, rl, tl)$ be a timed automaton with ref the refinement's urgent synchronization channel (if it exists), rl the set of all release urgent synchronization channels coming from the invocations of module tasks, and finally tl the set of all termination urgent synchronization channels coming from the invocations of module tasks, then

$$\forall module \in Prog, T_{module}(n, h, mi, bm, pos) = moduleTA(ref, rl, tl)$$

For each module a timed automaton is dynamically created. Unlike tasks automata, where the instantiation of the different default automata is done by just matching the input parameters, here a single automaton is attributed to each module, instantiated by passing as parameters the synchronization channels used by the module's task invocations.

Consider now a mode abstracted as the tuple $(n, p, refP, bmo, pos)$, where n is the mode's name, p is the period, $refP$ is the refinement program for that mode (if it exists), bmo the mode's body, and pos the mode's declaration position in the HTL file. Let $subModule(e, t)$ be a subset of the timed automaton's $moduleTA$ declaration where e is the set of states (with invariants) and t the set of transitions (with guards, updates and synchronizations), then we have

$$\begin{aligned}\forall mode \in module, \exists subModule(e, t) \in moduleTA,\\ T_{mode}(n, p, refP, bmo, pos) = subModule(e, t)\end{aligned}$$

4 Inference of Properties

This section presents the definition of a function P which accepts a HTL program and returns the specification of properties to verify. Naturally, this function is again defined recursively over the AST structure of the HTL language.

Absence of Block. Let $Prog$ be the set of all programs and df be the absence of blocking property description, then we have $P_{Prog} = df$. The application of this method to any program always produces the same absence of blocking property $(A \,\square\ not\ deadlock)$.

Modes Period. Let the tuple $(n, p, refP, bmo, pos)$ be the abstraction of a mode, where n is the mode's name, p the period, $refP$ the refinement program for that mode (if it exists), bmo the mode's body and pos the mode's declaration position in the HTL file. In the following vm denotes the property specifications of a mode's period. We have

$$\forall mode \in Prog, P_{mode}(n, p, refP, bmo, pos) = vm(p1, p2)$$

We also have, with *moduleTA* a module automaton and *NTA* a set of timed automata,

$$\begin{aligned}&\forall mode \in Prog, \exists moduleTA \in NTA,\\ &\quad p1 = A\,\square\, moduleTA.n \Rightarrow ((\neg\ moduleTA.t > p) \wedge\ (\neg\ moduleTA.t < 0)),\\ &\quad p2 = moduleTA.n \Rightarrow (moduleTA.Ready\\ &\qquad \wedge\ (moduleTA.t == 0\ \vee\ moduleTA.t == p)\end{aligned}$$

The first property $p1$ states that whenever the control state is the mode state, the module's (automaton) local clock is lower than the mode's period, and not negative. The second property $p2$ on the other hand states that whenever the mode's state is reached, the state *Ready* is also reached, which implies that the local clock is either zero or exactly equal to the period's value. The combination of both properties allows the restriction of a mode's period to the interval $[0, p]$, and guarantees that the period's maximum value is reached.

Task Invocations. Let (n, ip, op, s, pos) be a task invocation, where n is the task's name, ip the input port's (variables) mapping, op the output port's (variable) mapping, s the name of the parent task and finally pos the task's declaration position in the HTL file. In the following vi denotes the specification of properties in a mode's task invocation. We have

$$\forall invoke \in Prog, P_{invoke}(n, ip, op, s, pos) = vi(p1, p2)$$

Let $taskTA_i$ be the automaton of task i, *taskTA* the set of task automata, $taskState_i$ the task i invocation's state, *modeState* the mode's state where the invocation is done, *moduleTA* a module automaton and *NTA* a set of timed automata, then

$$\begin{aligned}&\forall i, \exists moduleTA \in NTA, \exists taskTA_i \in TaskTA,\\ &\quad p1 = A\,\square\,(moduleTA.taskStaste_i \Rightarrow (\neg\ taskTA_i.Idle))\\ &\qquad \wedge\ (moduleTA.Ready \Rightarrow taskTA_i.Idle),\\ &\quad p2 = A\,\square\,(taskTA_i.Let\ \wedge\ taskTA.tt! = 0) \Rightarrow moduleTA.modeState\end{aligned}$$

The property $p1$ states that for all executions, every time an invocation's state is equal to a control state, that task's automaton cannot be in the *Idle* state. Moreover, when the respective *moduleTA*'s control state is equal to *Ready*, the task's automaton must be in the *Idle* state. The second property specifies that whenever a task's automaton *Let* state is the control state and the local clock *tt* is different from zero, the execution of the module's automaton must be in the state representing the mode in which the tasks are invoked.

Tasks LET. Considering a task invocation *vlet* in a correct mode, its properties are specified as

$$\begin{aligned}&\forall invoke \in Prog, P_{invoke}(n, ip, op, s, pos) = vlet(p1, p2, p3), \forall i, \exists moduleTA \in NTA,\\ &\quad p1 = A\,\square\,(taskTA_i.Let \Rightarrow (\neg\ taskTA_i.tt < 0\ \wedge\ \neg\ taskTA_i.tt > p)),\\ &\quad p2 = A \diamond moduleTA.modeState \Rightarrow (taskTA_i.Lst\_IN\ \wedge\ taskTA_i.tt == 0),\\ &\quad p3 = A \diamond moduleTA.modeState \Rightarrow (taskTA_i.Let\ \wedge\ taskTA_i.tt == p)\end{aligned}$$

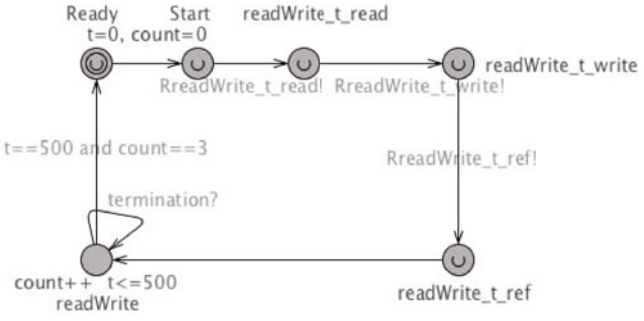


Fig. 3. `P_3TS_IO` automaton (automatic instantiation of `taskTA`)

The LET's validation is done via three distinct properties. Property $p1$ specifies that whenever a task's *Let* is reached, the automaton's local clock *tt* must lie between 0 and the period. Property $p2$ specifies that every time the mode's state is reached, the *Lst_IN* state is also reached, necessarily with the local clock *tt* set to zero. Finally, property $p3$ specifies that every time the mode's state is reached, the *Let* state is inevitably reached, with the clock *tt* set to the maximum value of the task's period.

5 Case Studies

We consider here the main case study used for illustration purposes by the HTL team: the *three-tank system*. A HTL program implements the controller of a physical system that includes three interconnected tanks with two pumps (for tanks 1 and 3), three taps (one for each tank) and two interconnection taps. The controller supervises the taps in order to maintain the liquid at a specific level.

Description of the Problem. The controller is implemented as a program that contains three modules; two of them (T1 and T2) specify the timing for the controllers of tanks T1 and T2, and the third module specifies the timing for the communications (IO) controller. Each controller module contains one mode which invokes one task and which is refined by a program into a P or PI controller. We assume that in addition to height measuring sensors there exist also sensors that detect perturbation in a tank (this determines the switch between P and PI). The IO module contains one mode named `readWrite` and invokes three tasks: `t_read` reads sensor values and updates communicators `h1`, `h2`, `v1` and `v2`; `t_write` reads communicators `u1` and `u2` and sends commands to the pumps; `t_ref` reads target values and updates communicators `h1_ref` and `h2_ref`.

Generated Model (excerpt). The HTL program is translated into a network consisting of nine timed automata, of which four are default automata that are instantiated by each task invocation depending on the modules and ports declared;

Table 1. Results

File	Levels	HTL	Model	Verifications	States
3TS-simulink.htl	0	75	263	62/62	7'566
	1	75	199	30/30	666
3TS.htl	0	90	271	72/72	18'731
	1	90	207	40/40	1'123
3TS-FE2.htl	0	134	336	106/106	280'997
	1	134	208	42/42	1'580
3TS-PhD.htl	0	111	329	98/98	172'531
	1	111	201	34/34	1'096
steer-by-wire.htl	0	873	1043	617/0	N/A
	1	873	690	394/0	N/A
flatten_3TS.htl	0	60	203	31/31	411

the remaining five represent the modules and respective execution modes. Taking as example the IO module, in which three tasks (`t_read`, `t_write` and `t_ref`) and no ports are used, it is translated as three timed automata, with the default `taskTA` automaton instantiated for each task. An example is shown below, extracted from the (.xta) file produced by the tool. Task invocations are represented by signals RreadWrite_t_read, RreadWrite_t_write and RreadWrite_t_ref.

Properties. In abstract terms the automatically inferred properties can be seen as divided in four classes: *Absence of Block*, *Modes Period*, *Task Invocations* and *Tasks' LET*. We give below an excerpt from the (.q) file generated for the 3TS_Simulink program, that shows two classes of properties : *Absence of Block* for the first property shown, and *Modes Period* for the second. The properties are annotated with a descriptive string and the expected verification result.

```
//Deadlock Free -> true
A[] not deadlock

//P1 mode readWrite period 500 @ Line 19 -> true
A[] sP_3TS_IO.readWrite imply ((not sP_3TS_IO.t>500) && (not sP_3TS_IO.
    t<0))
```

In some situations, small modifications in the code can have serious effects in the program and affect the verification of properties. In these cases the solution is to analyse and manually specify properties appropriate to each scenario.

Verification. In this case study the HTL2XTA translator for all levels of refinement (switch -L 0) has generated 62 properties automatically, which were all successfully checked (using *verifyta* version 4.0.10). 291794 states were explored and the maximum number of states consumed by a single property was 7566. Some properties were trivially verified. These numbers contribute to an increased confidence degree on the 3TS_Simulink's HTL specification. In spite of the large number of properties and states, this goal is achieved in reasonable time.

Other Case Studies. Using the current version of the translator it was possible to successfully generate models and properties for several HTL programs from [6, 10], and the HTL website. Table 1 summarizes relevant information about the results, specifically the number of applied levels (0=all, 1=main program), the

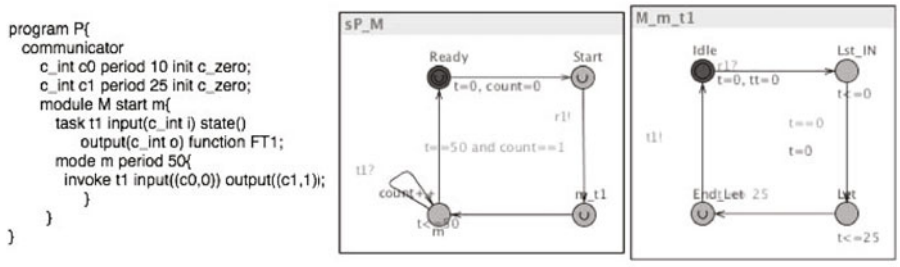


Fig. 4. A misbehaved HTL program and the corresponding Uppaal automata

number of lines in the HTL file, the number of lines in the model's specification file, the number of specified properties versus the number of properties successfully verified, and the number of states explored. The values in the table concern the Uppaal verification, given several different models translated from HTL to XTA.

The proposed toolchain was able to cope with all except one program, the more complex *steer-by-wire* example, for which the verification process does not terminate in reasonable time. Clearly, this is due to the use of all the advanced features of HTL (including a large and complex coordination layer).

6 Towards Correctness

The correctness of the proposed approach has not yet been established; we give here some preliminary remarks. The desired correctness property can be formulated as follows, where p is a HTL program, and MC corresponds to execution of the Model Checker.

If $MC(T(p))$ = Error then there exists an execution that derives to a timed error execution, following the operational semantics of HTL [5].

Although we have not proven such a correctness result, we give here an example to give the reader an intuition of why the approach should in principle be correct.

The example is shown in Figure 4. The period of the last task's (t_1) output is 25 and the first input is 0; the mode's period is 50, so it is trivial to conclude that this system is schedulable. As such, this program is validated by the HTL toolchain. However, this is not satisfactory, since with these values the LET of this task is specified as [0;25]. Due to the period of the communicator c_0 this task must not execute between instants 0 and 9, and the standard HTL toolchain contains no mechanism to specify or prove situations like this.

It is obvious that in such a small example this problem could be easily detected and corrected by simply changing the instant when the c_0 communicator is used

from 0 to 1. But in more complex systems it is hard to obtain any insight about this kind of temporal behaviours.

Considering again the above example, it is straightforward to see that the HTL2XTA translator preserves the bad temporal requirement in the timed automata model. Checking the property $A[]M\_m\_t_1.Let\ imply\ (not(M\_m\_t_1.tt < 10))$, which can be manually inserted in Uppaal and specifies that task t_1 must never occur in an instant inferior to 10, will produce a counter-example.

7 Conclusion and Future Work

The HTL language was created in an academic context, and its transfer to the industrial context remains a challenge. This work is a contribution towards that goal. The tool is available online[1] and runs only on the Linux platform. The HTL2XTA translator was developed in Ocaml, following the traditional compiler design process (but we rely on the HTL compiler for type checking).

We envision two natural improvements of our current methodology. First, the translation methodology has not yet been formally verified (i.e. it has not been proved that the translation preserves the timed semantics of HTL programs). The proof of the theorem sketched in Section 6 is a heavyweight task that must be carefully carried out.

Secondly, the current version of the translator is unable to deal with large-scale HTL programs, and moreover there are still some features of HTL syntax that are not covered by the current version. The translation of the currently covered HTL features can be improved in order to lower the size of the resulting NTA. As future work we plan to analyse these possibilities, and also to extend HTL with *annotations* to introduce supplementary behaviour rules. For instance this may provide insight about the behaviour of programs in the presence of *switch cases*. The impact of such annotations in the model and their influence on the design of the translator will of course have to be carefully considered.

Moreover, in the short term, the toolchain could be improved with a script that provides an automatic analysis of the logfile generated by Uppaal. Such a script could establish conveniently which timing requirements have been checked and which have not, and create a final report based on this information.

Finally, we are interested in transferring our work to the context of the SPARK/Ada language, widely used in the development of safety-critical systems. The *Giotto in Ada* [8] initiative should make this process quite straightforward. We also believe that our translation mechanisms may in principle be applied to other (more exploratory) concurrency models, but this remains an open issue.

Acknowledgment. This work was partially supported by the projects Rescue (PTDC/EIA/65862/2006) and FAVAS (PTDC/EIA-CCO/105034/2008), and by LIACC-UP through the Programa de Financiamento Plurianual, all funded by Fundação para a Ciência e Tecnologia (FCT).

[1] http://sourceforge.net/projects/htl2xta/

References

1. Behrmann, G., David, A., Larsen, K.G.: A tutorial on UPPAAL. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 200–236. Springer, Heidelberg (2004)
2. Bengtsson, J., Yi, W.: Timed automata: Semantics, algorithms and tools (2004)
3. Clarke Jr., E.M., Grumberg, O., Peled, D.A.: Model checking. MIT Press, Cambridge (1999)
4. Gelernter, D., Carriero, N.: Coordination languages and their significance. ACM Commun. 35(2), 97–107 (1992)
5. Ghosal, A.: A Hierarchical Coordination Language for Reliable Real-Time Tasks. PhD thesis, EECS Department, University of California, Berkeley (January 2008)
6. Ghosal, A., Henzinger, T.A., Iercan, D., Kirsch, C., Sangiovanni-Vincentelli, A.L.: Hierarchical timing language. Technical Report UCB/EECS-2006-79, EECS Department, University of California, Berkeley (May 2006)
7. Ghosal, A., Sangiovanni-Vincentelli, A., Kirsch, C.M., Henzinger, T.A., Iercan, D.: A hierarchical coordination language for interacting real-time tasks. In: EMSOFT 2006: Proceedings of the 6th ACM & IEEE International conference on Embedded software, pp. 132–141. ACM, New York (2006)
8. Hagenauer, H., Martinek, N., Pohlmann, W.: Ada meets giotto. In: Llamosí, A., Strohmeier, A. (eds.) Ada-Europe 2004. LNCS, vol. 3063, pp. 237–248. Springer, Heidelberg (2004)
9. Henzinger, T.A., Horowitz, B., Kirsch, C.M.: Giotto: A time-triggered language for embedded programming. In: Henzinger, T.A., Kirsch, C.M. (eds.) EMSOFT 2001. LNCS, vol. 2211, pp. 166–184. Springer, Heidelberg (2001)
10. Iercan, D.: Contribuitions to the Development of Real-Time Programming Techniques and Technologies. PhD thesis, EECS Department, University of California, Berkeley, Set (2008)
11. Levi, S.-T., Agrawala, A.K.: Real-time system design. McGraw-Hill, Inc., New York (1990)
12. Lundqvist, K., Asplund, L.: A ravenscar-compliant run-time kernel for safety-critical systems*. Real-Time Syst. 24(1), 29–54 (2003)
13. Poddar, R.K., Bhaduri, P.: Verification of giotto based embedded control systems. Nordic J. of Computing 13(4), 266–293 (2006)
14. Rushby, J.: Formal methods and their role in the certification of critical systems. Technical report, Safety and Reliability of Software Based Systems (Twelfth Annual CSR Workshop (1995)

Towards an Architecture for Runtime Interoperability

Amel Bennaceur[1], Gordon Blair[2], Franck Chauvel[3], Huang Gang[3], Nikolaos Georgantas[1], Paul Grace[2], Falk Howar[4], Paola Inverardi[5], Valérie Issarny[1], Massimo Paolucci[6], Animesh Pathak[1], Romina Spalazzese[5], Bernhard Steffen[4], and Bertrand Souville[6]

[1] INRIA, CRI Paris-Rocquencourt, France
[2] Lancaster University, UK
[3] School of Electronics Engineering and Computer Science, Peking University, China
[4] Technische Universitat Dortmund, Germany
[5] Universit degli Studi dell'Aquilà, Italy
[6] DOCOMO Euro-Labs, Munich, Germany

Abstract. Interoperability remains a fundamental challenge when connecting heterogeneous systems which encounter and spontaneously communicate with one another in pervasive computing environments. This challenge is exasperated by the highly heterogeneous technologies employed by each of the interacting parties, i.e., in terms of hardware, operating system, middleware protocols, and application protocols. This paper introduces CONNECT, a software framework which aims to resolve this interoperability challenge in a fundamentally different way. CONNECT dynamically discovers information about the running systems, uses learning to build a richer view of a system's behaviour and then uses synthesis techniques to generate a connector to achieve interoperability between heterogeneous systems. Here, we introduce the key elements of CONNECT and describe its application to a distributed marketplace application involving heterogeneous technologies.

1 Introduction

A fundamental requirement of distributed systems is to ensure interoperability between the communicating elements; systems that have been implemented independently of one another must be able to connect, understand and exchange data with one another. This is particularly true in highly dynamic application domains (e.g. mobile and pervasive computing) where systems typically only encounter one another at runtime. Middleware technologies have traditionally resolved many of the interoperability problems arising in these situations, such as operating system and programming language heterogeneity. Where two applications conform to a particular middleware standard, e.g. CORBA [12] and Web Services [3] [5], they are guaranteed to interoperate. However, the next generation of distributed computing applications are characterized by two important properties that force a rethink of how interoperability problems should be tackled:

T. Margaria and B. Steffen (Eds.): ISoLA 2010, Part II, LNCS 6416, pp. 206–220, 2010.

- *Extreme heterogeneity.* Complex pervasive systems are composed of technology dependent islands, i.e. domain specific systems that employ heterogeneous communication and middleware protocols. For example, Grid applications, mobile ad-hoc networks, Enterprise systems, and sensor networks all use their own protocols such that they cannot interoperate with one another.
- *Spontaneous Communication.* Connections between systems are not made until runtime (and are made between systems that were not aware of one another beforehand).

With such characteristics, requiring all applications to be developed upon a common middleware technology, e.g. CORBA or Web Services, is unsuitable in practice. Rather, new approaches are required that allow systems developed upon heterogeneous technologies to interoperate with one another at runtime. In this paper, we present the CONNECT[1] architectural framework that aims to resolve this interoperability challenge in a fundamentally different way. Rather than create a middleware solution that is destined to be yet another legacy platform that adds to the interoperability problem, we propose the novel approach of *generating the required middleware at runtime* i.e. we synthesize the necessary software to connect two end-systems. For example, if a client application developed using SOAP [13] encounters a CORBA server then the framework generates a CONNECTor that resolves the heterogeneity of the i) data exchanged, ii) application behaviour e.g. sequence of operations called, and iii) the lower level middleware and network communication protocols. In this paper we identify the requirements that need to be satisfied to guarantee interoperability, namely interoperability at the discovery, behavioral and data level. We then outline the key elements of the CONNECT framework that underpin a runtime solution to achieving such interoperability, and that are further detailed in the companion papers [14] [19] [2] [1] [10]:

- *Discovering the functionality* of networked systems and applications advertised by legacy discovery protocols e.g. Service Location Protocol (SLP) and Simple Service Discovery Protocol (SSDP). Then, transforming this to a rich intermediary description used to syntactically and semantically match heterogeneous services.
- *Using learning algorithms* to dynamically determine the interaction behaviour of a networked system from its intermediary representation and producing a model of this behaviour in the form of a labelled transition system (LTS) [14].
- *Dynamically synthesising* a software mediator using code generation techniques (from the independent LTS models of each system) that will connect and coordinate the interoperability between heterogeneous end systems [19] [2].

We highlight the potential of this CONNECT framework to achieve interoperability within a case study (a distributed marketplace application) that exhibits

[1] http://connect-forever.eu/

high levels of heterogeneity. Further exploration of maintaining dependability requirements when connecting systems is provided in [10]; while further information regarding the underlying formal theory of the CONNECT framework is found in [1].

The remainder of the paper is structured as follows. In Section 2 we highlight the interoperability challenges and requirements within a distributed marketplace application. We then examine the state of the art in interoperability solutions in Section 3 to highlight their deficiencies compared to the requirements. In Section 4 we present an overview of the CONNECT architecture and its underlying principles. In section 5 we describe how CONNECT resolves interoperability within the marketplace case study, and finally in section 6 we draw conclusions and identify a roadmap for future research in this field.

2 Motivating Scenario: The Distributed Marketplace

Consider a stadium where fans from various countries have gathered together to watch a game. The specific application we focus on in this section is that of a distributed marketplace. Here, **merchants** publicise their wares, and **consumers** can search the market, and order from a merchant. Both merchants and consumers use mobile devices with wireless networks deployed in the stadium. Merchants publish product info which the consumers can browse through. When a consumer requests a product, the merchant gets a notification of the amount ordered and the location of the consumer, to which he can respond with a yes/no. If yes, then when he is close enough to the consumer, both of them get a proximity notification by means of their mobile device (ring/buzz).

Table 1. Potential Implementations of Consumers and Merchants

Country	Discovery	Middleware	Application Data	Currency
Germany	SLP	Tuple Space	GetInfo	EUR
U.K.	SLP	SOAP	GetInfo	GBP
France	SSDP	SOAP	GetInfo	EUR
Italy	SSDP	SOAP	GetLocation+ GetPrice+ GetQuantity	EUR
Spain	SLP	SOAP	GetInfo	EUR

Table 1 highlights how stadiums from different countries implement the application using heterogeneous technology. Importantly, if a client from one country attempts to dynamically interoperate with a merchant in a different country it will fail in each case. We now examine the dimensions of heterogeneity which explain why such interoperation fails:

1. **Heterogeneous discovery protocols** are used by the consumer to locate a merchant, and by the merchant to advertise his services. In Table 1, SLP and SSDP are employed; in situations where the consumer and merchant differ in this aspect, the two will be unable to discover one another and the first step fails.

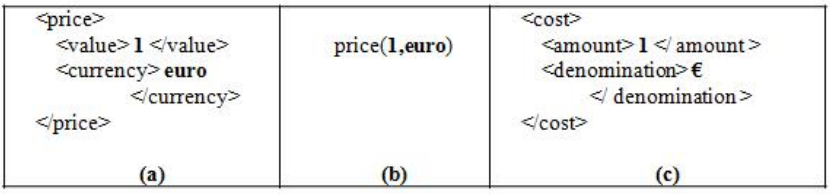


Fig. 1. Representing price in a) XML, b) tuple data, and c) heterogeneous XML

2. Consumers and merchants use **heterogeneous middleware protocols** to implement their functional interactions. In Table 1, a tuple space middleware and the SOAP RPC protocol [13] are used; these are different communication paradigms: the tuple space follows a shared space abstraction to write tuples to and read from, whereas RPC is a synchronous invocation of a remote operation. Hence, the two cannot interoperate directly.
3. **Application level heterogeneity.** Interoperability challenges at the application level arise due to the different ways that application developers implement the functionality. As a specific example, we assume that the merchant implements methods for the consumer to obtain information about his wares in one of two ways this would lead to different sequences of messages between the consumer and merchant: A single `GetInfo()` remote call, or three separate remote calls: `GetLocation()`, `GetPrice()`, and `GetQuantity()`.
4. **Data-representation Heterogeneity.** Implementations may represent data differently. Data representation heterogeneity is typically manifested at two levels. The simplest form of data interoperability is at the syntactic level where two different systems may use very different formats to express the same information. The French system may represent the price of the merchant's product using XML (Figure 1a), while the German tuple space may serialize a Java Object (Figure 1b). Further, even if two systems share a common language to express data, different dialects may still raise interoperability issues. Consider Figure 1c (the Spanish system) against Figure 1a; they (intuitively) carry the same meaning. Any system that recognizes the first structure will also be able to parse the second one, but it will fail to recognize the similarity between them unless it realizes that *price* ≡ *cost*, that *value* ≡ *amount*, that *currency* ≡ *denomination* (where ≡ denotes equivalence). The deeper problem of data heterogeneity is the semantic interoperability whereby all systems should have the same interpretation of data.

Summary of requirements. This scenario illustrates four dimensions where systems may be heterogeneous: i) the discovery protocol, ii) the interaction protocol, iii) application behaviour, and iv) data representation and meaning. A universal interoperability solution must consider all four in order to achieve interoperability.

3 Beyond State of the Art Interoperability Solutions

Achieving interoperability between independently developed systems has been one of the fundamental goals of middleware researchers and developers; and prior efforts have largely concentrated on solutions where conformance to one or other standard is required e.g. as illustrated by the significant standards work produced by the OMG for CORBA middleware [12], and by the W3C for Web Services based middleware [3] [5]. These attempt to make the world conform to a common standard, and this approach has been effective in many areas e.g. routing of network messages in the Internet. To some extent CORBA and Web Services have been successful in connecting systems in Enterprise applications to handle hardware platform, operating system and programming language heterogeneity. However, in the more general sense of achieving universal interoperability and dynamic interoperability between spontaneous communicating systems they have failed. Within the field of distributed software systems, any approach that assumes a common middleware or standard is destined to fail due to the following reasons:

- A one size fits all standard/middleware cannot cope with the extreme heterogeneity of distributed systems e.g. from small scale sensor applications through to large scale Internet applications.
- New distributed systems and application emerge fast, while standards development is a slow, incremental process. Hence, it is likely that new technologies will appear that will make a pre-existing interoperability standard obsolete, c.f. CORBA versus Web Services (neither can talk to the other).
- Legacy platforms remain useful. Indeed, CORBA applications remain widely in use today. However, new standards do not typically embrace this legacy issue; this in turn leads to immediate interoperability problems.

One approach to resolving the heterogeneity of middleware solutions comes in the form of interoperability platforms. ReMMoC [11], Universal Interoperable Core [22] and WSIF [6] are client side middleware which employ similar patterns to increase interoperability with heterogeneous service side protocols. First, the interoperability platform presents an API for developing applications with. Secondly, it provides a substitution mechanism where the implementation of the protocol to be translated to, is deployed locally by the middleware to allow communication directly with the legacy peers (which are simply legacy applications and their middleware). Thirdly, the API calls are translated to the substituted middleware protocol. For the particular use case, where you want a client application to interoperate with everyone else, interoperability platforms are a powerful approach. However, these solutions rely upon a design time choice to develop upon the interoperability platforms. Therefore, they are unsuited to other interoperability cases e.g. when two applications developed upon different legacy middleware want to interoperate spontaneously at runtime.

Software bridges offer another interoperability solution to enable communication between different middleware environments. Clients in one middleware

domain can interoperate with servers in another middleware domain where the bridge acts as a one-to-one mapping between domains; it will take messages from a client in one format and then marshal this to the format of the server middleware; the response is then mapped to the original message format. While a recognised solution to interoperability, bridging is infeasible in the long term as the number of middleware systems grow i.e. due to the effort required to build direct bridges between all of them. Enterprise Service Buses (ESB) can be seen as a special type of software bridge; they specify a service-oriented middleware with a message-oriented abstraction layer atop different messaging protocols (e.g., SOAP, JMS, SMTP). Rather than provide a direct one-to-one mapping between two messaging protocols, a service bus offers an intermediary message bus. Each service (e.g. a legacy database, JMS queue, Web Service etc.) maps its own message onto the bus using a piece of code, to connect and map, deployed on the peer device. The bus then transmits the intermediary messages to the corresponding endpoints that reverse the translation from the intermediary to the local message type. Hence traditional bridges offer 1-1 mapping; ESBs offer an N-1-M mapping. Example ESBs are Artix [23] and IBM Websphere Message Broker [24]. ESBs offer a solution to the problem of middleware heterogeneity; however, it focuses on the messaging abstraction only and the assumption is that all messaging services can be mapped to the intermediary abstraction (which is a general subset of messaging protocols). This decision is enacted at design or deployment time, as the endpoint must deploy code to connect to a particular message bus with an appropriate translator and hence is unsuitable for dynamic interoperation between two legacy platforms.

INDISS [4], uMiddle [18], OSDA [15], PKUAS [9] and SeDiM [8] are examples of transparent interoperability solutions which attempt to ensure legacy solutions unaware of the heterogeneous middleware are still able to interoperate. Here, protocol specific messages, behaviour and data are captured by the interoperability framework and then translated to an intermediary representaion; a subsequent mapper then translates from the intermediary representation to the specific legacy middleware protocol to interoperate with. The use of an intermediary means that one middleware can be mapped to any other by developing these two elements only (i.e. a direct mapping to every other protocol is not required). Another difference to bridging is that the peers are unaware of the translators (and no software is required to connect to them, as opposed to connecting to bridges).

The interoperation solutions proposed above concentrate on the middleware level. They support interoperation by abstract protocols and language specifications. But, by and large they ignore the data dimension. To this extent a number of efforts, which are generically labelled as Semantic Web Services [16] [21] [7], attempt to enrich the Web services description languages with a description of the semantics of the data exchanged. The result of these efforts are a set of languages that describe both the orchestration of the services' operations, in the sense of the possible sequences of messages that the services can exchange as well as the meanings of these messages with respect to some reference ontology. However,

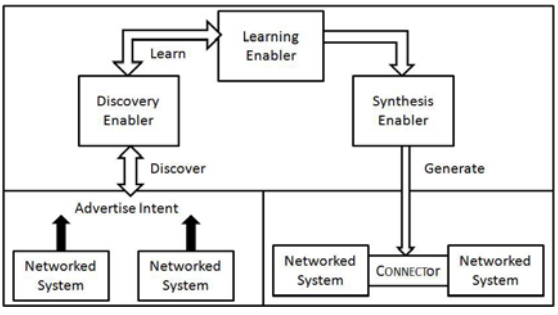


Fig. 2. Actors in the CONNECT architecture

such approaches assume a common middleware standard and do not address all of the heterogeneity problems previously described.

The state of the art investigation shows two important things; first, there is a clear disconnect between the main stream middleware work and the work on application, data, and semantic interoperability; second, none of the current solutions addresses all of the four requirements of dynamic pervasive systems as highlighted in the scenario in refScenario. Hence, these results show that there is significant potential for CONNECT to extend beyond the state of the art in interoperability middleware.

4 The Connect Architectural Framework

The CONNECT architecture provides the underlying principles and software architecture framework to enact the necessary mechanisms to achieve universal interoperability between heterogeneous systems. Figure 2 presents a high-level overview of the following actors involved within the CONNECT architecture and how they interact with one another:

- *Networked systems* are systems that manifest the will to connect to other systems for fulfilling some intent identified by their users and the applications executing upon them.
- *Enablers* are networked entities that incorporate all the intelligence and logic offered by CONNECT for enabling connection between heterogeneous networked systems. In this paper, we focus on how the discovery, learning and synthesis enablers co-ordinate to produce a CONNECTor as shown in Figure 2, while the companion papers discuss the enablers in more detail [14] [19] [2] [10].
- CONNECTors are the synthesized software connectors produced by the action of enablers to connect networked systems.

4.1 Discovery and Learning of Networked Systems

Networked systems use discovery protocols to advertise their will to connect (i.e. their intent); service advertisements are used to describe the services that

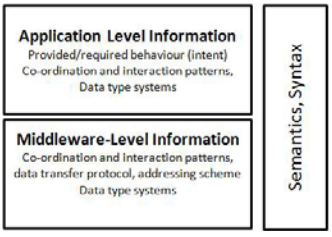


Fig. 3. Networked System Model

a system provides, while service lookup requests document the services that are required. It is the role of the *discovery enabler* to capture this information from the legacy network protocols in use and to create an initial picture of the network systems wishing to connect with one another.

The outputs of this enabler are models of networked system as shown in Figure 3. It is important to note that only a subset of this description is made available by the networked system; the learning enabler utilises an active learning algorithm to learn the co-ordination and interaction patterns of the application [14]. Much of the information about the middleware level is not explicit in the discovery process, but pointers within the discovery descriptions (e.g. this is a SOAP service) can be used to build the model from pre-defined, constant middleware models (e.g. a model of the SOAP protocol). The model builds upon discovery protocol descriptions that convey both syntactic information and semantic information about the externalized networked system. This semantic information is necessary in open environments, where semantics cannot be assumed to be inherently carried in a commonly agreed syntax. Typically, ontologies are used in open environments for providing a common vocabulary on which semantic descriptions of networked systems can be based.

The architecture of the discovery enabler is illustrated in Figure 4. This software framework is deployed upon a third party node within the network and consists of three core elements:

- *Discovery protocol plug-ins.* Discovery protocols e.g. SLP, UPnP, LDAP, Jini, etc. are heterogeneous in terms of their behaviour and message format; further they differ in the data representation used to describe services. To resolve this, individual plug-ins for each protocol receive and send messages in the legacy format; the plug-in also translates the advertisements and requests into a common description format used by the CONNECT networked system model.
- The *Model repository* stores networked system models of all CONNECT ready systems (this is a system which advertises its intent and whose behaviour is learnable). These remains alive for the lifetime of the request-for a system advertising its services this will normally match the length of its lease as presented by the legacy protocol and, for a system's request, this is the length of time before the legacy protocol lookup request times out.

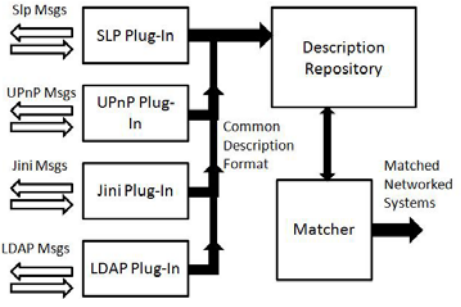


Fig. 4. The Discovery Enabler

– The *Functional Matcher* actively matches potential requests with advertisements i.e. matching the required and provided interface types of a network system. Simple semantic matchers can be plugged into to match descriptions of the same type, or richer semantic matchers can be employed.

Learning of networked systems is performed just after discovery and is necessary due to the fact that the retrieved descriptions of networked systems are incomplete. (as described above) CONNECT learning attempts to infer the complete interaction behaviour and employs methods based on monitoring and model-based testing of the networked systems to elicit their interaction behaviour [14]. Learning attempts to extrapolate from observed behaviour to generic behaviour. The outcome of learning is a complete, as far as possible, instantiated networked system model. The learning enabler is built upon the Learnlib tool[20]; this takes as input the interface descriptions of the networked systems (e.g. in WSDL-S) and then executes a learning algorithm which interacts directly with the service to be learned in order to infer the correct behaviour. The enabler outputs a complete LTS model to represent this behaviour of the network system.

4.2 Synthesis of Connectors

The CONNECTor Synthesis is a two-step process that encompasses the construction of a mediation LTS and its interpretation at runtime. The needed mediation LTS defines the behaviour that will let the networked systems synchronize and interact. It results from the analysis [19] of both the networked systems' behaviours and the ontology, and specifies all the needed message translations from one side to the other. In the following scenario in Section 5 for instance, when receiving a `getInfo` request coming from the customer side, the mediation LTS will properly request the merchant side (e.g. using `getLocation`, `getPrice`, `getQuantity`) and then aggregate and return the data to the customer. The mediation LTS resolves the application-level and data-level interoperability.

The resulting mediation LTS (see Figure 9) remains an abstract specification that does not include enough middleware-level information to be directly executed. Instead, as shown on Figure 5, the mediation LTS is seen as an orchestration of middleware invocations and is dynamically interpreted by an engine,

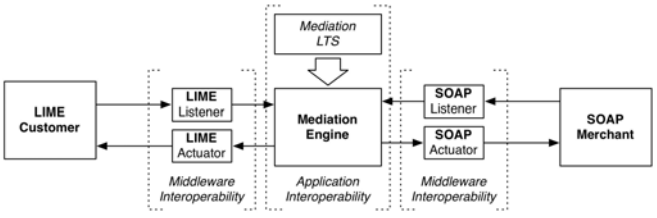


Fig. 5. A Software CONNECTor

which receives, translates and forwards messages from the two sides. In our example, when the mediation engine is notified of a `getInfo` tuple was released by the client, it triggers the emission of three SOAP requests and triggers the generation of one Lime tuple containing the requested information.

As shown in Figure 5, the missing middleware-level knowledge is hard-coded into reusable plug-ins denoted as *Listener* and *Actuator*. According to a given middleware protocol, a listener receives data packets and outputs application messages whereas an actuator composes network messages. In our marketplace example, the proper invocation of the LIME infrastructure and the emission and reception of SOAP messages are handled by those ad-hoc listeners and actuators. The use of such plug-ins finally ensures the middleware-level interoperability. In addition, when a new middleware is released, such plug-ins can be separately generated from the networked system models. By contrast with code-generation, the choice of interpretation eases the monitoring and dependability verification of runtime CONNECTors. Although the CONNECT framework also addresses these two issues, they are not presented here for the sake of conciseness.

5 Connect in Action

To demonstrate the potential of the CONNECT architecture we consider a single case within the distributed marketplace scenario where two heterogeneous end-systems encounter one another. The client consumer employs SLP as the discovery protocol and the Lime tuple space middleware [17] as the interaction protocol (the German system from Table 1). The service merchant employs SSDP as the discovery protocol and SOAP as the interaction protocol (the French system from Table 1). We apply the CONNECT architecture to build a CONNECTor that allows the consumer to interact with the client. In this section we document the outputs of the enablers to illustrate how the architecture co-ordinates to produce a CONNECTor to overcome the interaction and application heterogeneity between the two systems.

The discovery enabler first monitors the running systems, and receives SLP lookup requests that describe the German application's requirements. It also receives the notification messages from the French application in SSDP that advertise the provided interface. The discovery enabler plug-ins transform these messages and produce a WSDL description for both networked systems. A partial

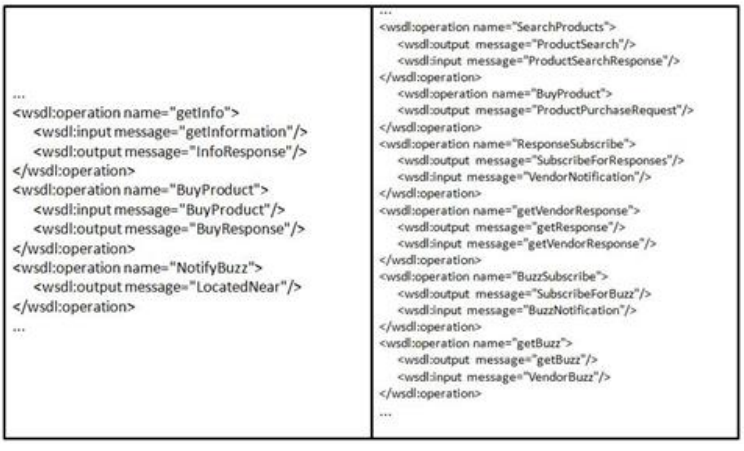


Fig. 6. WSDL of the SOAP merchant (left) and the LIME consumer (right)

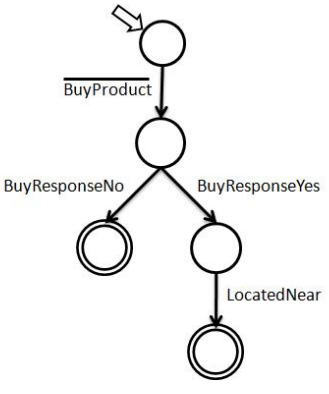


Fig. 7. Behaviour Model of merchant produced by learning enabler

view of these is given in Figure 6, and show the abstract operations provided by the application. In the client consumer application, these operations are bound to the concrete LIME protocol (e.g. the `SearchProducts` operation is bound to an `out` operation followed by a `rd`), and in the Merchant application the operations are bound to SOAP (e.g. the `getInfo` operation is bound to a SOAP RPC request). The WSDL also serves to highlight the heterogeneity of the two interfaces; they offer the same functionality, but do so with different behaviour. The next step in the CONNECT architecture is to learn the behaviours of the two systems.

The learning enabler receives the WSDL documents from the discovery enabler and then interacts with deployed instances of the LIME merchant and the SOAP merchant implementations in order to create the behaviour models for both the consumer and the merchant in this case. The interactions possible in

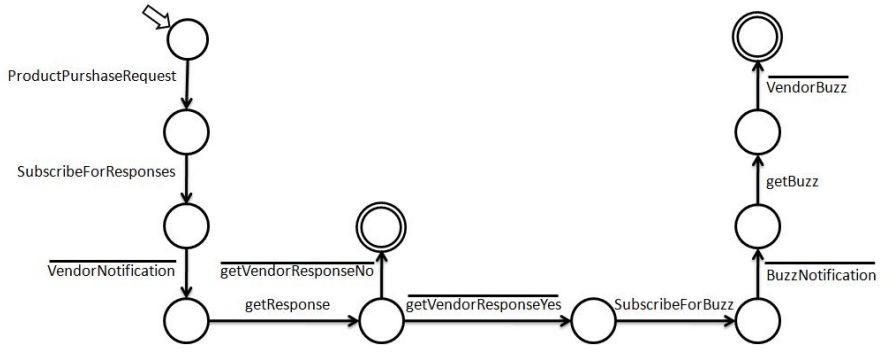


Fig. 8. Behaviour Model of Consumer produced by learning enabler. Messages with a bar are emitted while others are received

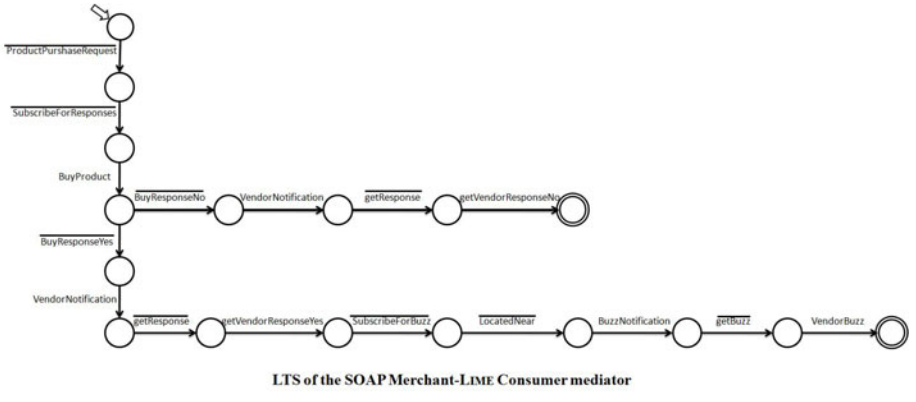


Fig. 9. Model of the textscConnector mediator between LIME and SOAP

these systems are produced as LTS models and are illustrated for the SOAP merchant in Figure 7 and for the LIME Consumer in Figure 8. Here we can see that a merchant receives a `BuyProduct` SOAP message and either responds with a yes or no `BuyResponse` SOAP message. If yes, the merchant moves towards the consumer and when close sends the `LocatedNear` SOAP message. In the consumer case, a `ProductPurchaseRequest` is sent as a Lime `out` message (along with a `SubscribeForResponse` reactsTo message to be informed when there is a response in the tuple space). When the merchant replies, a `VendorNotification` is received by the consumer and they read this reponse from the tuple space using a Lime `rd` message (`GetVendorResponse`). If the response is yes, then the consumer subscribes for the buzz message which is then read when the merchant is near.

The final step in the CONNECT process is to create the CONNECTor that will mediate between the consumer's request and the merchant's response. To complete this the two LTS models are passed to the synthesis enabler. This performs two tasks:

- *Behaviour matching.* An ontology is provided for the domain that states where sequences of operations are equivalent e.g. that the `ProductPurchaseRequest` and the `SubscribeForResponses` in the LIME implemented application are the same as the `BuyProduct` SOAP request. Further information about how the ontology-based behavioural matching is given in the companion paper [2].
- *Model synthesis.* The enabler produces an LTS that will mediate between the two systems; this LTS is shown in Figure 9. Here you can see how the interoperation is co-ordinated; when the LIME requests are received these then produce a `BuyProduct` SOAP request, which eventually leads to a response that is converted into a response that can be read from the tuple space. A more detailed version of the mediator (and in particular how it operates on the more detailed LTS models of this scenario) and its behaviour and outputs is again provided in the companion paper [19].

6 Conclusions and Future Work

In this paper we have shown that in spite of the major research and industrial efforts to solve the problem of interoperability, current solutions demonstrably fail to meet the needs of modern distributed applications especially those that embrace dynamicity and high levels of heterogeneity. An important observations is that there is a significant disconnect between middleware solutions and semantic interoperability solutions, which in turn severely hampers progress in this area. We have introduced the CONNECT architecture as a fundamentally different way to address the interoperability problem; this intrinsically supports middleware and application level interoperability and embraces learning and synthesis. The initial experiment with the architecture provides early evidence of the validity of the proposed approach and we believe that as the architecture matures it will provide further novel and rich contributions to the field of interoperability.

Future work will continue to explore a broader range of issues in the heterogeneity space. Much of this will focus on the important requirements that have been introduced in the companion papers, and their integration into the CONNECT software architecture. These include:

- *Non-functional properties.* That is creating CONNECTors that conform to the non-functional requirements of both interacting parties in the same way they meet the functional requirements currently.
- *Dynamic monitoring* of CONNECTors will be further investigated to ensure that all requirements are maintained over time. In[19] we illustrate a first integration of synthesis and monitoring.
- *Dependability.* Ensuring that the deployed CONNECTors are dependable, trustworthy and secure; this is especially important given the nature of the pervasive computing environments where these solutions will be deployed.

Acknowledgments

This work is done as part of the European FP7 ICT FET Connect project (http://connect-forever.eu/).

References

1. Autili, M., Chilton, C., Inverardi, P., Kwiatkowska, M., Tivoli, M.: Towards a connector algebra. In: ISoLA 2010, Part II. LNCS, vol. 6416, pp. 278–292. Springer, Heidelberg (2010)
2. Bertolonio, A., Inverardi, P., Issarny, V., Sabetta, A., Spalazzese, R.: On-the-fly interoperability though automated mediator synthesis and monitoring. In: ISoLA 2010, Part II. LNCS, vol. 6416, pp. 251–262. Springer, Heidelberg (2010)
3. Booth, D., Haas, H., McCabe, F., Newcomer, E., Champion, M., Ferris, C., Orchard, D.: Web services architecture. In: W3C (February 2004), http://www.w3.org/TR/sawsdl/
4. Bromberg, Y., Issarny, V.: Indiss: Interoperable discovery system for networked services. In: Alonso, G. (ed.) Middleware 2005. LNCS, vol. 3790, pp. 164–183. Springer, Heidelberg (2005)
5. Christensen, E., Curbera, F., Meredith, G., Weerawarana, S.: Web services description language (wsdl) 1.1. In (March 2001), http://www.w3.org/TR/wsdl
6. Duftler, M., Mukhi, N., Slominski, S., Weerawarana, S.: Web services invocation framework (wsif). In: OOPSLA 2001 Workshop on Object Oriented Web Services (2001)
7. Farrell, J., Lausen, H.: Semantic annotations for wsdl and xml schema (August 2007), http://www.w3.org/TR/sawsdl/
8. Flores, C., Blair, G., Grace, P.: An adaptive middleware to overcome service discovery heterogeneity in mobile ad hoc environments. IEEE Distributed Systems Online (2007)
9. Gang, H., Hong, M., Qian-xiang, W., Fu-qing, Y.: A systematic approach to composing heterogeneous components. Chinese Journal of Electronics 12(4), 499–505 (2003)
10. Di Giandomenico, F., Kwiatkowska, M., Martinucci, M., Masci, P., Qu, H.: Dependability analysis and verification for Connected systems. In: Margaria, T., Steffen, B. (eds.) ISoLA 2010, Part II. LNCS, vol. 6416, pp. 263–277. Springer, Heidelberg (2010)
11. Grace, P., Blair, G., Samuel, S.: A reflective framework for discovery and interaction in heterogeneous mobile environments. ACM SIGMOBILE Mobile Computing and Communications Review 9(1), 2–14 (2005)
12. Object Management Group. The common object request broker: Architecture and specification version 2.0 (1995)
13. Gudgin, M., Hadley, M., Mendelsohn, N., Moreau, J., Frystyk Nielsen, H., Karmarkar, A., Lafon, Y.: Soap version 1.2 part 1: Messaging framework (April 2001), http://www.w3.org/TR/soap12-part1
14. Howar, F., Jonsson, B., Merten, M., Steffen, B., Cassel, S.: On handling data in automata learning: Considerations from the Connect perspective. In: Margaria, T., Steffen, B. (eds.) ISoLA 2010, Part II. LNCS, vol. 6416, pp. 221–235. Springer, Heidelberg (2010)

15. Limam, N., Ziembicki, J., Ahmed, R., Iraqi, Y., Li, D., Boutaba, R., Cuervo, F.: Osda: Open service discovery architecture for efficient cross-domain service provisioning. Computer Communications 30(3), 546–563 (2007)
16. Martin, D., Burstein, M., Mcdermott, D., Mcilraith, S., Paolucci, M., Sycara, K., Mcguinness, D., Sirin, E., Srinivasan, N.: Bringing semantics to web services with owl-s. World Wide Web 10(3), 243–277 (2007)
17. Murphy, A., Picco, G., Roman, G.: Lime: A coordination model and middleware supporting mobility of hosts and agents. ACM Transactions on Software Engineering Methodology 15(3), 279–328 (2006)
18. Nakazawa, J., Tokuda, H., Edwards, W., Ramachandran, U.: A bridging framework for universal interoperability in pervasive systems. In: 26th IEEE International Conference on Distributed Computing Systems, ICDCS 2006 (2006)
19. Issarny, V., Inverardi, P., Spalazzese, R.: A theory of mediators for eternal CONNECTors. In: Margaria, T., Steffen, B. (eds.) ISoLA 2010, Part II. LNCS, vol. 6416, pp. 236–250. Springer, Heidelberg (2010)
20. Raffelt, H., Steffen, B.: Learnlib: A library for automata learning and experimentation. In: Baresi, L., Heckel, R. (eds.) FASE 2006. LNCS, vol. 3922, pp. 377–380. Springer, Heidelberg (2006)
21. Roman, D., Keller, U., Lausen, H., de Bruijn, J., Lara, R., Stollberg, M., Polleres, A., Feier, C., Bussler, C., Fensel, D.: Web service modeling ontology. Applied Ontology Journal 1(1), 77–106 (2005)
22. Roman, M., Kon, F., Campbell, R.: Reflective middleware: From your desk to your hand. IEEE Distributed Systems Online 2(5) (August 2001)
23. Artix Enterprise Service Bus Software (2010), http://web.progress.com/en/sonic/artix-esb.html
24. IBM Software WebSphere, http://www-01.ibm.com/software/websphere/

On Handling Data in Automata Learning*

Considerations from the CONNECT Perspective

Falk Howar[2], Bengt Jonsson[1], Maik Merten[2], Bernhard Steffen[2], and Sofia Cassel[1]

[1] Department of Computer Systems, Uppsala University, Sweden
{bengt,sofia.cassel}@it.uu.se
[2] Technical University Dortmund, Chair for Programming Systems, Germany
{falk.howar,maik.merten,steffen}@cs.tu-dortmund.de

Abstract. Most communication with real-life systems involves data values being relevant to the communication context and thus influencing the observable behavior of the communication endpoints. When applying methods from the realm of automata learning, it is necessary to handle such data-occurrences. In this paper, we consider how the techniques of automata learning can be adapted to the problem of learning interaction models in which data parameters are an essential element. Especially, we will focus on how test-drivers for real-word systems can be generated automatically. Our main contribution is an analysis of (1) the requirements on information contained in models produced by the *learning enabler* in the CONNECT project and (2) the resulting preconditions for generating test-drivers automatically.

1 Introduction

Interoperability remains a fundamental challenge when connecting heterogeneous systems which encounter and spontaneously communicate with one another in pervasive computing environments. In this paper, we consider how the techniques of automata learning can be adapted to the problem of learning interaction models in which data parameters are an essential element. Especially, we will discuss how test-drivers for real-word systems can be generated automatically. Our main contribution is an analysis of (1) the requirements on information contained in models produced by the *learning enabler* in the CONNECT project and (2) the resulting preconditions for generating test-drivers automatically.

The CONNECT Integrated Project [15] aims at overcoming the interoperability barrier by synthesizing on the fly the CONNECTors via which networked systems communicate. CONNECTors are implemented through a comprehensive dynamic process [6] based on (i) extracting knowledge from, (ii) learning about and (iii) reasoning about, the interaction behavior of networked systems, together with (iv) synthesizing new interaction behaviors out of the ones exhibited by the systems to be made interoperable, and further (v) generating and deploying corresponding CONNECTor implementations.

* This work is supported by the European FP 7 project CONNECT (IST 231167).

T. Margaria and B. Steffen (Eds.): ISoLA 2010, Part II, LNCS 6416, pp. 221–235, 2010.

One of the challenges in the CONNECT project is to develop techniques for learning models of exploratory interaction with the component, while analyzing the messages exchanged between the component and its environment. The chosen approach in CONNECT is based on existing techniques for learning the temporal ordering between a finite set of interaction primitives. Such techniques have been developed for the problem of regular inference (i.e., automata learning), in which a regular set, represented as a finite automaton, is to be constructed from a set of observations of accepted and unaccepted strings. The most efficient such techniques use the setup of *query learning* (aka. active learning), where information is obtained by posing a sequence of membership query, each asking how the automaton would respond to a particular input string.

Related Work: Previous work on automata learning has assumed the set of interaction primitives to be an unstructured finite set [5,18,22], implying that the alphabet must be made finite, e.g., by suppressing parameters of messages.

The influence of data on control flow is taken into account by model-based test generation tools, such as ConformiQ Qtronic [13]. Recent contributions began transferring automata learning methods to systems that comprise complex parameterized interface alphabets [23] and to systems with (bounded) nondeterminic behavior [3].

The automata we will propose have (some) similar properties to the ones that are constructed in [10], like e.g. both contain information about causal relations between parameters. Only, our approach (based on active learning) will construct successively the smallest representation while the approach in [10] will construct the universe (cf. [17,25]) of possible interaction sequences.

Outline: The remainder of the paper is structured as follows. In the next section, we recapitulate the technique of query learning for unstructured Mealy machine models. Section 3 describes the challenges involved in finding a framework to handle data from the special perspective of the CONNECT project. Section 4 discusses the prototypical application of techniques to overcome these challenges along a case study.

2 Query Learning

2.1 The L^*_{M} Learning Algorithm

Query learning (or *active learning*) attempts to construct a deterministic finite representation, e.g., a Mealy machine, that matches the behavior of a given target system on the basis of observations of the target system and perhaps some further information on its internal structure. Here, we only summarize the basic aspects of our realization L^*_{M} for Mealy machines [18], which is based on Angluin's learning algorithm L^* for finite state acceptors [5]. A more elaborate version of this summary and an extended discussion of the practical aspects of active learning is given in [24].

Definition 1. *A Mealy machine is defined as a tuple* $Sys = \langle Q, q_0, \Sigma, \Omega, \delta, \lambda \rangle$ *where*

- Q *is a finite nonempty set of* states *(be* $n = |Q|$ *the size of Sys),*
- $q_0 \in Q$ *is the* initial state*,*
- Σ *is a finite* input alphabet*,*
- Ω *is a finite* output alphabet*,*
- $\delta : Q \times \Sigma \to Q$ *is the* transition function*, and*
- $\lambda : Q \times \Sigma \to \Omega$ *is the* output function*.*

Intuitively, a Mealy machine evolves through states $q \in Q$, and whenever one applies an input symbol (or action) $a \in \Sigma$, the machine moves to a new state according to $\delta(q,a)$ and produces an output according to $\lambda(q,a)$[1]. We write $q \xrightarrow{i/o} q'$ to denote that on input symbol i the automaton moves from state q to state q' producing output symbol o.

Query learning is also referred to as *active* learning as it constructs Mealy machine models by actively querying the target system. It poses *membership queries* that test whether certain strings (potential runs) are contained in the target system's language (its set of runs), and *equivalence queries* that compare intermediately constructed hypothesis automata for equivalence with the target system. Learning terminates successfully as soon as an equivalence query signals success.

In its basic form, active learning starts with a hypothesis automaton with only one state and refines this automaton on the basis of query results iterating the three main steps shown in Fig. 1: *refining the hypothesis*, *conformance testing*, and *analyzing counterexamples*, until a state-minimal deterministic (hypothesis) automaton consistent with the target system is produced. Key to achieving this result is the Nerode-like dual characterization of states:

- by a set, $S \subset \Sigma^*$, of *access sequences*. This characterization of state is too fine, as different words $s_1, s_2 \in S$ may lead to the same state in the target system. L^*_{M} will construct such a set S, containing access sequences to all states of the target automaton. It will also maintain a second set, SA, which together with S will cover all transitions of the target system (SA will during the course of learning always be $SA = (S \cdot \Sigma) \setminus S$).
- by an ordered set, $D \subset \Sigma^*$, of *distinguishing sequences*. L^*_{M} realizes the characterization of hypothetical states q simply in terms of vectors $row(s) = \langle r_1, \ldots, r_k \rangle$ (with $r_i \in \Omega$), characterizing states by means of subsequent outputs: For $rows(s)$, let $r_i = \lambda(\delta(q_0, s), d_i)$.

The set S will be initialized as $\{\varepsilon\}$, containing only the access sequence to the initial state; SA will accordingly be initialized as Σ, covering all transitions originating in the initial state. The ordered set D will be initialized as Σ, allowing to identify a state by the output that is produced along the transitions originating in this state. The learning procedure then proceeds as shown in Fig. 1 by:

[1] In the remainder of this paper, we will use an extended version of the transition function: $\delta' : Q \times \Sigma^* \to Q$ with $\delta'(q, aw) = \delta'(\delta(q,a), w))$, where $q, q' \in \Sigma$, symbols $a \in \Sigma$, and $w \in \Sigma^*$. The same holds for the output function.

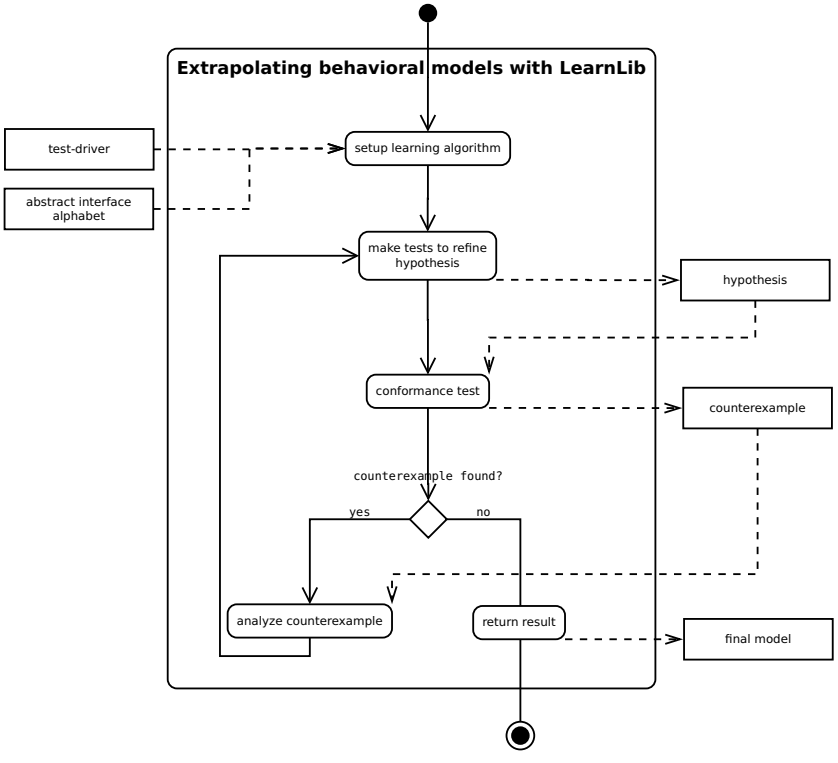


Fig. 1. Structure of Active Learning Algorithms (modeled in XPDD [16])

Refining the Hypothesis: This first step again iterates two phases. The first phase checks whether the constructed automaton is closed under the one-step transitions, i.e., each transition from each state of the hypothesis automaton ends in a well defined state of this very automaton. This is the case if for every $r \in S \cdot \Sigma$ there exists a $s \in S$ with $row(s) = row(r)$. Otherwise, S will be extended by the corresponding r until *closedness* is established (and SA will be extended accordingly). This extension is guaranteed to result in a unique fixpoint, independent of the order in which the rows are processed.

The second phase then checks whether two access sequences $s_1, s_2 \in S$ with the same bit vector, $row(s_1) = row(s_2)$, have also the same outgoing transitions, a necessary precondition for them to represent the same state. This condition, which is called *consistency*, can be formalized as follows:

$$\forall s_1, s_2 \in S \; \forall a \in \Sigma \; . \; row(s_1) = row(s_2) \stackrel{?}{\Rightarrow} row(s_1 \cdot a) = row(s_2 \cdot a).$$

It is easy to see that detected inconsistencies can be removed by elaborating the set D of distinguishing futures in a way that makes some of the difference observed on a distinguishing transition visible before that transition: one simply

needs to prefix the distinguishing future that separates the two target states on the distinguishing transition by the label of this very transition.

Unfortunately, such additions to D may break the previously achieved completeness, which requires to re-iterate the completion procedure. This, in turn, may lead to new violations of consistency. However, successive iteration of these two steps is guaranteed to result in a unique, well-defined, closed, and complete hypothesis automaton whose states are characterized by the bit vectors. In more detail:

- every state $q \in Q$ of the hypothesis automaton is reachable by at least one word $s \in S$, i.e., $row(s)$ corresponds to q,
- there exists a transition $\delta(q,a) = q'$ *iff* there exists $s \in S$ with s reaching q (or with $row(s)$ corresponding to q) and $row(s \cdot a)$ corresponding to q',
- The ouput function can be constructed from the $row()$-vectors as well. As D is initialized as *Sigma*, the values for all $\lambda(q,a)$, where $a \in \Sigma$, are contained in the $row(s)$ vector corresponding to q.

Conformance Testing & Counterexamples: After closedness and consistency have been established, an equivalence query checks whether the language of the hypothesis automaton coincides with the language of the target automaton. If this is true, the learning procedure successfully terminates. Otherwise, the conformance test returns a counterexample, i.e., a word which distinguishes the hypothesis from the target automaton. All prefixes of a counterexample will be added to S (SA will be extended accordingly). This will lead to inconsistency, which in turn will lead to a new distinguishing suffix [18].

The correctness argument for the whole approach follows a straightforward pattern.

1. The state construction guarantees that the number of states of the hypothesis automaton can never exceed the number of states of the smallest (minimal) automaton, behaviorally equivalent to the target system.
2. The treatment of counterexamples guarantees that at least one additional state is added to the hypothesis automaton each round. Thus, due to 1), such treatments can happen only finitely often.
3. The conformance testing (or equivalence query) provides new counter examples as long as the language of the hypothesis automaton does not match the desired result.

Put together, this guarantees that the learning procedure terminates after at most n rounds with a minimal automaton, behaviorally equivalent to the target system, where n is the number of states of this automaton.

2.2 Practical Aspects in Active Learning

Automata learning can be considered a key technology for dealing with *black box systems*, i.e., systems that can be observed, but for which no or little knowledge about the internal structure or even their intent is available. *Active* automata

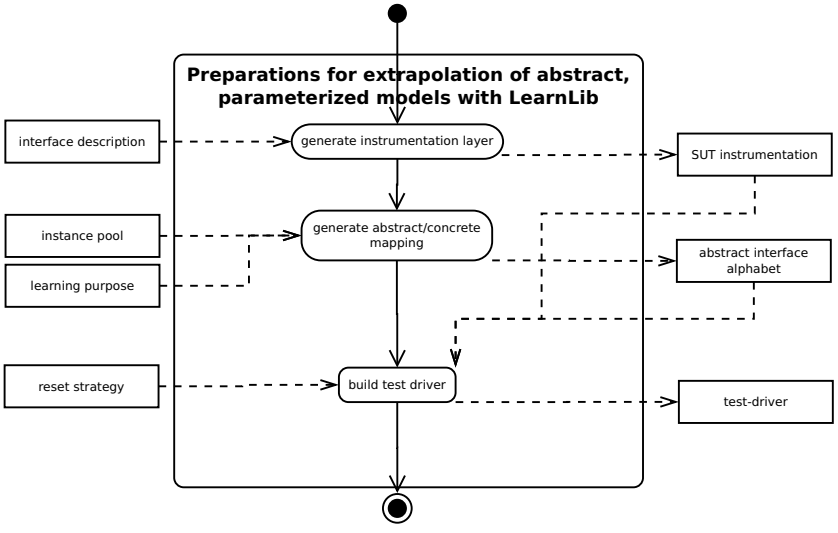


Fig. 2. Test-driver and Alphabet Generation (modeled in XPDD [16])

learning is characterized by its specific means of observation, i.e., its proactive way of posing membership queries and equivalence queries. Thus it requires some effort to realize this query-based interaction for the considered application contexts. The general requirements are shown in Fig. 2.

Reset: Active learning requires membership queries to be independent. Whereas this is no problem for simulated systems, this may be quite problematic in practice. Solutions range here from *reset strategies* via homing sequences [21] or snapshots of the system state to the generation of independent fresh system scenarios. Indeed, in certain situations, executing each membership query with a separate independent user scenario may be the best one can do.

Abstraction: Learning systems which comprise parameterized interface alphabets will usually require abstraction: The learning algorithm will invoke primitives from the input alphabet on the system under test. Usually these primitives are complex actions comprising parameters over infinite domains. The system under test will react by producing complex and parametrized outputs. To work on infinite alphabets, a sensible abstraction will be needed. Usually these abstractions are hand-tailored [2,1,12,19], but there also exist some approaches to automate abstraction and refinement of the input alphabet [7,8,11].

Interfacing Systems: While real systems require certain actual actions to be taken in the course of interaction, a learning algorithm will formulate a test case that is to be executed on the system in an abstract language (as words that are build from an *abstract interface alphabet*). It will also expect the reaction of the system to be encoded into words over an output alphabet. To connect a learning algorithm and a real system, a *test-driver* is needed, which

translates input words into sequences of real actions (e.g., method calls) and real outputs (e.g., return values) into output words.

As shown in Fig. 2, a test-driver can be generated from an *interface description* of the system under test (SUT), an *instance pool*, and a *reset strategy*. Sometimes, by considering a *learning purpose*, uninteresting parts of the SUT can be hidden from the learning algorithm. The interface description will be used to generate code to *instrument the SUT*. The instances that shall be assumed for data values in the communication with the SUT and the purpose allow the formulation of a mapping between abstract interface symbols and concrete actions on the SUT. Finally, the reset strategy will ensure that all tests on the SUT will be executed under the same initial conditions.

The primary focus of this paper are the preconditions for an automatic generation of test-drivers that allow the production of models, which are sensible in the context of CONNECT. We therefore will make quite strong assumptions about already present means of abstraction, knowing that these assumptions have to (and can) be lowered subsequently.

3 Connect Learning Challenges

3.1 Connect Model Requirements

To better understand the requirements connected with learning relevant (in the context of the CONNECT project) behavioral models for networked systems, let us first describe the communication, which we assume to happen between a pair of networked systems that are explicitly designed and developed to interact with each other properly. Figure 3 schematically shows the scenario: Two components communicate via protocol messages (1),(5). The components together realize some protocol. Both components are actual implementations of their specified interfaces. Without giving a formal definition, we can imagine both parties to comprise a control part (2), and a data part (3). The control part can be imagined as a labeled transition system with actual blocks of code labeling the transactions (4). Each code block in the components of Fig. 3 would consist of

- an entry point for one interface method,
- conditions over parameters and local variables,
- assignments and operations on local variables,
- a return statement.

The data parts may be best described as a set of local variables. We will refer to such a set of local variables as a *parameter structure*.

To infer the behavior of one component (e.g., the right one from Fig. 3), the part of the other component has to be taken by a learning algorithm, which will be equipped with the interface alphabet of the component to be learned. Obviously, it will not be possible to present a solution to the problem of generating a model that captures all the behavior defining the component. This task

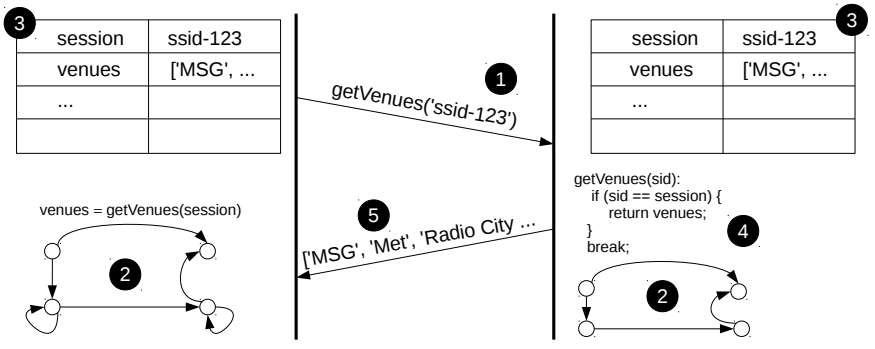


Fig. 3. Communicating Components

would correspond to reverse engineering the component on the basis of its behavioral profile. Fortunately, the models required in the CONNECT project are of a different kind.

While, usually, models produced by active learning are used in model based verification or some other domain that requires complete models of the system under test (e.g., to prove absence of faults), in CONNECT, the inferred models will be used to explain how to interact with the system. This special focus allows us, to apply goal-oriented learning techniques (and the goal here is gathering enough information to interact with a component successfully): From the CONNECT perspective, complete models are rather uninteresting and would even make the synthesis of CONNECTor models more expensive. This may have a positive impact on the costs of producing models. However, to achieve this (capturing how to interact successfully), two kinds of information will have to be captured by the models:

Effects of Primitives: The learned models will only be useful for CONNECTor synthesis within a given semantic context (cf. [14]). Most networked systems have well-defined purposes (or *effects*), e.g., electronic commerce systems. A subset of the offered communication primitives, when certain preconditions are met, will lead to successful conclusion of transactions directly relating to the respective purpose. There may, e.g., in a vending system, be a "purchase" communication primitive, that, when providing additional information like a product-identifier, will conclude the process of buying a desired product. Other communication primitives may not directly lead to a successful conclusion of a business transaction, but may, e.g., be prerequisites for communication primitives serving the immediate purpose of the given system. A vending system may, e.g., have a communication primitive that results in the delivery of a list of products-identifiers, which is a prerequisite for the "purchase" primitive.

It will be necessary to capture in the produced models, at which points in model what effects are achieved (e.g., when a seat is actually booked

in a system). But, these effects will in general not be observable in the communication with a system; the information about effects of primitives rather has to be provided as an additional input to the learning algorithm.

Preconditions of Primitives: *(or Data Causalities)* Many systems of interest for the CONNECT project operate on communication primitives which contain data values relevant to the communication context and having a direct impact on the exposed behavior. One example would be session identifiers or sequence numbers which are negotiated between the communication participants and included in every message. The models will have to make explicit causal relations between data parameters that are used in the communication (e.g, that always the exact session identifier that was returned when opening a new session has to be used in subsequent calls).

Information about causal relations between data parameters is in the scope of active learning. Using sensible interface alphabets will enable inferring data-related behavior.[2] While in classical automata learning these causal relations would implicitly be encoded in the state-space, there exist already methods to make explicit such preconditions in the model [7,8].

Without formalizing, this will lead to models of the following kind. The learned models will have transitions, like the one in Fig. 4, which

- are labeled with statements, the left component in Fig. 3 would execute,
- expose the causal influences of parameter values in form of preconditions, resembling the conditions in the component from the right side of Fig. 3,
- are annotated with corresponding effects.

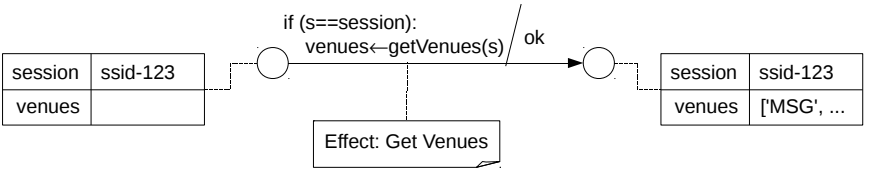


Fig. 4. Transition in Inferred Model

Models comprising preconditions and effects for a given primitive will make it possible to generate sequences of communication primitives that (i) fulfill preconditions successively and (ii) finally conclude a process reaching a special effect. This can be directly used in code synthesis. Also, reachability of the effects can be checked in the learned models: if a given effect cannot be reached this may be an indication that the learning process hasn't yet explored enough parts of the system. On the other hand, if all effects are reachable the learning process may be terminated safely, making sure only relevant parts of the system are being explored.

[2] Active learning classifies states and transitions by the output that is produced by a system. We assume here that as classifying output the information about success or failure of the invocation of a primitive can be used.

3.2 Example

Let us introduce a small example to illustrate the discussed ideas. Imagine a system for booking seats in events. A user of this system has to provide his credentials and can then browse through a list of venues. For each venue a list of available seats can be accessed. Finally, from these lists of seats a single seat can be booked, which will be confirmed in a corresponding receipt.

Table 1. Interface Descriptions

Interface 1	**Interface 2**
-	session ← openSession(user,pwd)
venue[] ← getVenues(user,pwd)	venue[] ← getVenues(session)
seat[] ← getSeats(user,pwd,venue)	seat[] ← getSeats(session,venue)
receipt ← bookSeat(user,pwd,seat)	receipt ← bookSeat(session,venue,seat)

For this system we will define two slightly different interfaces. The signatures of the interfaces' methods are given in Table 1. The differences between the two interfaces are:

1. In *Interface 1* credentials have to provided in all invocations, while in *Interface 2* the credentials are only used once to create a session identifier. This identifier will then be used in subsequent invocations.
2. Assuming that the methods of the interfaces will be called from top to bottom in order to actually book a seat, calling the first method of *Interface 1* corresponds to calling the first two methods of *Interface 2*.
3. While in *Interface 1* to book a seat, only the identifier for this seat has to be provided, in *Interface 2* the identifier of the corresponding venue has to be provided as well.

We now want to connect a front-end of the system, which uses *Interface 1*, to a back-end using *Interface 2*. While the actual construction of a CONNECTor for the two parties is out of the scope of this paper, we will consider here the general requirements on the inferred models of the two networked components that can be derived from this example. Naturally, there will arise a number of other problems when generating a CONNECTor, like transforming data parameters from one format into another, or actually instrumenting the components, etc. However, these problems do not directly lead to requirements on the obtained models.

In the message sequence chart in Figure 5, it is shown how the two implementations can be connected. The first column of the figure shows the synchronization of the two implementations on the shared effects. The second column shows the communication between the component using *Interface 1* and the CONNECTor, while the fourth column shows the communication between the CONNECTor and the second implementation using *Interface 1*. In the third column, the data-flow between the two implementations is visualized. Here, it is interesting to observe that there exist parameters which do not exist in both interfaces (e.g., session).

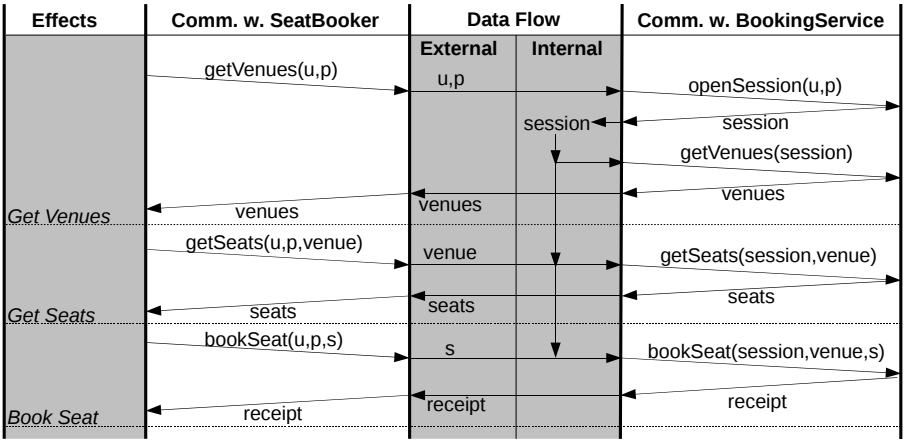


Fig. 5. Message Sequence Chart for Connected Implementations

We refer to those parameters as *control*-parameters as they are only used to control one party and need not to be passed to the other party. The model that is to be constructed by regular inference is now expected to expose exactly this kind of causal relations between the formal parameters of the different primitives (that is has to be same session identifier in all the calls).

4 Experimental Results

We now present experimental results highlighting various aspects of the concepts described above. Basis for the experiments was a Web-service realizing the seat booking system and using the interface from the right half of Table 1. For the interface description WSDL-S [4] was used, which allows

1. annotating primitives with effects,
2. annotating parameters with (abstract) data types.

Technically, this is realized by references into an ontology. A common ontology can then be used to link several heterogeneous systems with the same intention on a semantic level.

We generated Java proxy-classes for the Web-service. Using standard Java APIs, the proxy classes were inspected for methods, which are used as learning alphabet. Also, the WSDL-S document was analyzed for data types and effects. Using the information about data types, a parameter structure was generated: The return values of method invocations were stored in the local parameter structure of the test-driver for usage in future invocations of the target system. The type and name of the output symbol determined the parameter, a returned (output-)value will be stored in. The resulting parameter structure is shown in Table 2. Input-values for method invocations were either

Table 2. Parameter Structure and Effects

Symbol	Output-Parameters	Store as	Effect
openSession(user,pwd)	Session Identifier	session	-
getVenues(session)	List of Venues	venues	GetVenues
getSeats(session,venue)	List of Seats	seats	GetSeats
bookSeat(session,venue,seat)	Receipt	receipt	BookSeat

– predetermined values, e.g., for `user` and `pass` in the `openSession` primitive,
– references to parameters, e.g, for `session` in all primitives, or
– simple expressions over parameters, e.g., `!session` (not session) and `(in)venues` (denoting a venue from a list of venues).

The *abstract interface alphabet* was generated over all possible combinations of data and primitives. The resulting abstract input alphabet is shown in Table 3. The corresponding test-driver (cf. Section 2.2) invokes methods on the WSDL proxy according to the alphabet symbol currently processed.

Table 3. Abstract Input Alphabet

Interface Symbol	Input-Parameters
openSession(u,p)	u='test',p='test'
getVenues(si)	si=session
	si=!session
getSeats(si,v)	si=session, v=(in)venues
	...
	si=!session, v=(!in)venues
bookSeat(si,v,s)	si=session, v=(in)venues, s=(in)seats
	...
	si=!session, v=(!in)venues, s=(!in)seats

All data values used in this examples were sequences of characters. The `!`-function was realized by adding some characters to the corresponding sequence. The `(in)`-function was realized by picking the first element of a list, which would also ensure that two subsequent applications of `(in)` return the same value. The `(!in)`-function was realized by constructing a value, which was not contained in the corresponding list.

To distuingish (i.e., classify) states in active learning, some kind of output is needed from the system under test. As the returned values in this example are stored in the parameter structure and thus are treated symbolically, we used the WSDL 'fault' returns as classifiers: In WSDL, every interface method can return with a normal or with a fault answer. In Java, this fault case corresponds to an exception being thrown. The test-driver would simply catch and evaluate

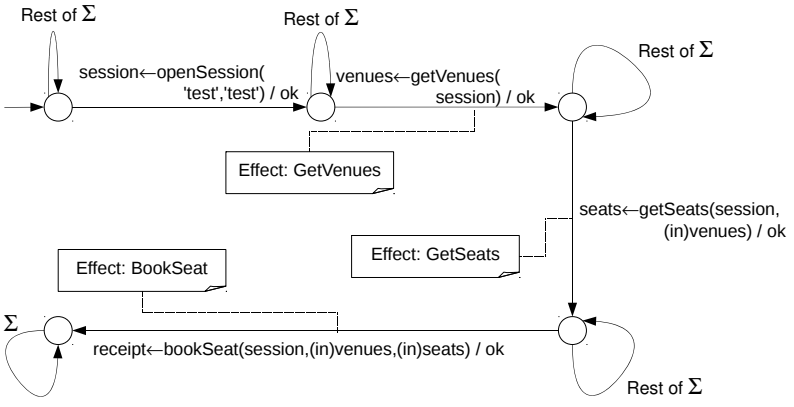


Fig. 6. Behavioral Model of the Seat Booking System

these exceptions. In the example, three effects were annotated to the normal return of the interface methods (cf. Table 2). The behavior of the Web-service was learned using LearnLib [20], the automata learning framework developed at TU Dortmund. The resulting model with 5 states is shown in Figure 6.

In this experiment, the test-driver was put together manually from

- automatically generated instrumentation code,
- manually constructed and analyzed information from the WSDL-S description,
- manually provided fixed values,
- manually implemented functions on data parameters.

To fully automate the generation of adequate test-drivers, the single steps will subsequently have to be automated (where possible) and the information that is required additionally in order to learn such networked systems will have to be provided from the CONNECT framework [6] (e.g., by means of the method proposed in [10] for determining relations between the types of parameters).

5 Conclusion

In this paper we have discussed the requirements on information that is to be contained in models produced by the learning enabler (cf. [9]) in the CONNECT project: causal relations between data parameters and effects of primitives. Basing on these requirements we have given ideas of (1) what kind of information the corresponding models will comprise, and (2) what the preconditions for producing these models are. Namely: semantically rich annotated interface descriptions. Finally, we have presented the (promising) results from a prototypical application of a parameter structure and semi-automatic test-driver generation. Future work will include automatic analysis and reasoning of/on semantically rich interface descriptions in order to generate parameter structures automatically and a more sophisticated way of realizing functions on data values.

References

1. Aarts, F., Blom, J., Bohlin, T., Chen, Y.-F., Howar, F., Jonsson, B., Merten, M., Nagel, R., Sabetta, A., Soleimanifard, S., Steffen, B., Uijen, J., Wilk, T., Windmuller, S.: Establishing basis for learning algorithms (2010)
2. Aarts, F., Jonsson, B., Uijen, J.: Generating Models of Infinite-State Communication Protocols using Regular Inference with Abstraction. Accepted for ICTSS 2010 (2010)
3. Aarts, F., Vaandrager, F.: Learning I/O Automata. Accepted for CONCUR 2010 (2010)
4. Akkiraju, R., Farrell, J., Miller, J., Nagarajan, M., Schmidt, M., Sheth, A., Verma, K.: Web service semantics-WSDL-S. W3C member submission, 7 (2005)
5. Angluin, D.: Learning Regular Sets from Queries and Counterexamples. Information and Computation 75(2), 87–106 (1987)
6. Bennaceur, A., Blair, G.S., Chauvel, F., Georgantas, N., Grace, P., Howar, F., Inverardi, P., Issarny, V., Paolucci, M., Pathak, A., Spalazzese, R., Steffen, B., Souville, B.: Towards an Architecture for Runtime Interoperability. In: Margaria, T., Steffen, B. (eds.) ISoLA 2010, Part II. LNCS, vol. 6416, pp. 206–220. Springer, Heidelberg (2010)
7. Berg, T., Jonsson, B., Raffelt, H.: Regular inference for state machines with parameters. In: Baresi, L., Heckel, R. (eds.) FASE 2006. LNCS, vol. 3922, pp. 107–121. Springer, Heidelberg (2006)
8. Berg, T., Jonsson, B., Raffelt, H.: Regular Inference for State Machines Using Domains with Equality Tests. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE 2008. LNCS, vol. 4961, pp. 317–331. Springer, Heidelberg (2008)
9. Bertolino, A., Blair, G., Chauvel, F., Cortes, C.F., Georgantas, N., Grace, P., Howar, F., Huyn, T., Jonsson, B., Paolucci, M., Pathak, A., Souville, B., Tivoli, M.: Initial CONNECT Architecture. Technical report, 02 (2010)
10. Bertolino, A., Inverardi, P., Pelliccione, P., Tivoli, M.: Automatic synthesis of behavior protocols for composable web-services. In: ESEC/FSE 2009: Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, pp. 141–150. ACM, New York (2009)
11. Grinchtein, O., Jonsson, B., Pettersson, P.: Inference of event-recording automata using timed decision trees. In: Baier, C., Hermanns, H. (eds.) CONCUR 2006. LNCS, vol. 4137, pp. 435–449. Springer, Heidelberg (2006)
12. Hagerer, A., Margaria, T., Niese, O., Steffen, B., Brune, G., Ide, H.D.: Efficient regression testing of CTI-systems: Testing a complex call-center solution. Annual review of communication, Int. Engineering Consortium (IEC) 55, 1033–1040 (2001)
13. Huima, A.: Implementing conformiq qtronic. In: Petrenko, A., Veanes, M., Tretmans, J., Grieskamp, W. (eds.) TestCom/FATES 2007. LNCS, vol. 4581, pp. 1–12. Springer, Heidelberg (2007)
14. Inverardi, P., Issarny, V., Spalazzese, R.: A Theory of Mediators for Eternal Connectors. In: ISoLA 2010, Part II. LNCS, vol. 6416, pp. 236–250. Springer, Heidelberg (2010)
15. Issarny, V., Steffen, B., Jonsson, B., Blair, G.S., Grace, P., Kwiatkowska, M.Z., Calinescu, R., Inverardi, P., Tivoli, M., Bertolino, A., Sabetta, A.: CONNECT Challenges: Towards Emergent Connectors for Eternal Networked Systems. In: ICECCS, pp. 154–161 (2009)

16. Jung, G., Margaria, T., Wagner, C., Bakera, M.: Formalizing a Methodology for Design- and Runtime Self-Healing. In: IEEE International Workshop on Engineering of Autonomic and Autonomous Systems, pp. 106–115 (2010)
17. Lamprecht, A.-L., Naujokat, S., Margaria, T., Steffen, B.: Synthesis-Based Loose Programming. In: QUATIC 2010 - 7th International Conference on the Quality of Information and Communications Technology (accepted, 2010) (in submission)
18. Margaria, T., Niese, O., Raffelt, H., Steffen, B.: Efficient test-based model generation for legacy reactive systems. In: HLDVT 2004: Proceedings of the Ninth IEEE International Workshop on High-Level Design Validation and Test, pp. 95–100. IEEE Computer Society, Los Alamitos (2004)
19. Margaria, T., Raffelt, H., Steffen, B.: Knowledge-based relevance filtering for efficient system-level test-based model generation. Innovations in Systems and Software Engineering 1(2), 147–156 (2005)
20. Raffelt, H., Steffen, B., Berg, T., Margaria, T.: LearnLib: a framework for extrapolating behavioral models. Int. J. Softw. Tools Technol. Transf. 11(5), 393–407 (2009)
21. Rivest, R.L., Schapire, R.E.: Inference of finite automata using homing sequences. Inf. Comput. 103(2), 299–347 (1993)
22. Shahbaz, M., Groz, R.: Inferring Mealy Machines. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 207–222. Springer, Heidelberg (2009)
23. Shahbaz, M., Li, K., Groz, R.: Learning and integration of parameterized components through testing. In: Petrenko, A., Veanes, M., Tretmans, J., Grieskamp, W. (eds.) TestCom/FATES 2007. LNCS, vol. 4581, pp. 319–334. Springer, Heidelberg (2007)
24. Steffen, B., Howar, F., Merten, M., Margaria, T.: Practical Aspects of Active Learning. In: FMICS Handbook. Wiley, Chichester (to appear, 2010)
25. Steffen, B., Margaria, T., Freitag, B.: Module Configuration by Minimal Model Construction (1993)

A Theory of Mediators for Eternal Connectors*

Paola Inverardi[1], Valérie Issarny[2], and Romina Spalazzese[1]

[1] University of L'Aquila, Italy
[2] INRIA, CRI Paris-Rocquencourt, France
http://connect-forever.eu/

Abstract. On the fly synthesis of mediators is a revolutionary approach to the seamless networking of today's and future digital systems that increasingly need be connected. The resulting emergent mediators (or CONNECTors) adapt the interaction protocols run by the connected systems to let them communicate. However, although the mediator concept has been studied and used quite extensively to cope with many heterogeneity dimensions, a remaining key challenge is to support on-the-fly synthesis of mediators. Towards this end, this paper introduces a theory of mediators for the ubiquitous networking environment. The proposed formal model: (i) precisely characterizes the problem of interoperability between networked systems, and (ii) paves the way for automated reasoning about protocol matching (interoperability) and related mediator synthesis.

1 Introduction

Today's ubiquitous networked environments embed networked devices from a multitude of application domains, e.g., home automation, consumer electronics, mobile and personal computing domains to name a few. Middleware then positions itself as a core architectural paradigm to enable the heterogeneous networked systems to actually interact together. Middleware provides upper layer interoperability, bridging the gap between application programs and the lower-level hardware and software infrastructure in order to coordinate how application components are connected and how they interoperate, especially in the networked environment. However, ubiquitous networking has introduced new challenges for middleware. Devices need to dynamically detect services available in the ubiquitous networked environment and adapt their own communication protocols to interoperate with them, since networked applications are implemented on top of diverse middleware. As discussed in companion paper [5], a number of systems have been introduced to provide middleware protocols interoperability. However, these address only interoperability at the middleware-layer. Interoperability between networked software systems further concerns the systems' interfaces and behaviors at the application-layer, which calls for supporting *mediators*.

* The work is partly supported by the CONNECT European Project No 231167.

T. Margaria and B. Steffen (Eds.): ISoLA 2010, Part II, LNCS 6416, pp. 236–250, 2010.

The mediator concept was initially introduced to cope with the integration of heterogeneous data sources [23,22], and as design pattern [4]. However, with the significant development of Web technologies and given abilities to communicate openly for networked systems, many heterogeneity dimensions shall now be mediated. Heterogeneity effectively spans [19]: terminology, representation format and transfer protocols, functionality, and application-layer protocols. The first heterogeneity dimension is addressed by data level mediation, while the second relies on a combination of data level and protocol mediations. Functional mediation then depends on the reasoning about logical relationships between the functional descriptions of networked systems, similar to the notion of behavioral subtyping [14]. Protocol mediation is further concerned with behavioral mismatches that may occur during interactions. Other approaches that share the same formal settings as protocol mediation have been proposed quite some time ago to solve mismatches in the field of supervisory control theory of discrete event systems [12] and, more recently, in the field of software architectures to address communication problems by proposing ad hoc wrappers [18]. However, while the concept of mediator has received a great deal of attention over the last two decades, on-the-fly synthesis of mediators, or *emergent mediation* for short, which is central to seamless interactions in the ubiquitous networked environment, remains a key challenge.

Towards enabling emergent mediation, this paper sets the underlying theory from which protocol matching (interoperability) and mediation may be formally reasoned upon. The work is part of the CONNECT[1] European research project, which investigates the development, from design to prototype implementation, of an overall framework for the seamless networking of heterogeneous networked systems [8,5]. The contribution of this paper is specifically a theory of mediators to precisely characterize:

(i) The interaction protocols that are functionally matching but behaviorally mismatching. Note that we assume given the specification of protocols, either as part of the advertisement of networked systems using some discovery protocol or based on some learning technique like the one discussed in companion paper [7].
(ii) The interoperability notion between protocols based on functional matching. Note that in a first step, we restrict ourselves to interoperability between pairs of protocols and we further do not address data-level heterogeneity, which is being extensively addressed elsewhere [16].
(iii) The behavior of mediators to achieve interoperability under functional matching despite behavioral mismatch.

The paper is organized as follows. We first set the principles of our approach, further describing the terminology we use and giving an illustrative scenario (Section 2). Then, we introduce a formalization for protocols, which paves the way for automated reasoning about protocols functional matching and for the automated synthesis of mediators (Section 3). Finally, we position our contribution

[1] http://connect-forever.eu/

with respect to related work (Section 4) and we conclude with perspectives for future work (Section 5).

2 Eternal Interoperability through Emergent Mediation

The focus of this paper is the *protocol interoperability problem* and our goal is to find an *automated solution* to solve it dynamically. In the following, we give the necessary definitions that set the context of the work.

2.1 Definitions

With the term *protocols*, we refer to *application-layer interaction protocols* (or application-layer observable protocols). That is, a protocol is the behavior of a system in terms of the sequence of messages visible at the interface level, which it exchanges with other systems. We further focus on *compatible* or *functionally matching* application-layer interaction protocols. Functional matching means that protocols can *potentially communicate* by performing *complementary sequences of actions*. "Potentially" means that communication may not be achieved because: (i) the languages of the two protocols are different, although semantically equivalent, (ii) the sequence of actions performed by a protocol is different from the sequence of actions of the other one because of interleaved actions related to third parties communications (i.e., other systems, the environment). In the former case, (i), it is necessary to properly perform a translation of the two languages. In the latter case, (ii), it is necessary to provide an abstraction of the two sequences that results in sequences containing only relevant actions to the communication. Communication is then possible if the two abstracted sequences are complementary, i.e., are the same sequences of actions while having opposite send/receive "type" for all actions.

With *interoperability*, we mean the property referring to the ability of heterogeneous application-layer interaction protocols that *functionally match to coordinate* where the coordination is expressed as synchronization, i.e., two systems succeed in coordinating if they are able to synchronize.

As said before, we want to approach the protocol interoperability problem in an automated fashion. The solution we propose here, is to automatically synthesize *mediators* (also referred to as *mediating connectors* or CONNECTors in our work) that allow the protocols to interoperate by solving their behavioral mismatches. A mediator is then a protocol that is elicited according to the *automated mediation* paradigm.

2.2 Towards Emergent Mediators

The interoperability problem we specifically want to attack concerns automated and on-the-fly mediation between behaviorally mismatching, yet functionally matching application-layer interaction protocols. Starting from two protocols, the first condition we check is if they share some complementary sequence of actions.

Figure 1 depicts the main elements of our methodology:

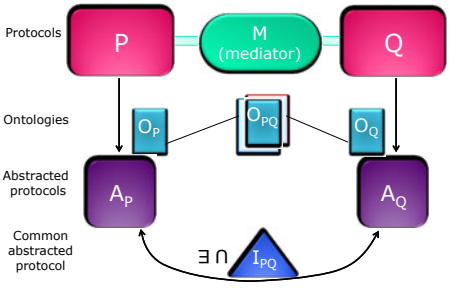


Fig. 1. An overview of our approach

(i) Two application-layer protocols P and Q whose representation is given in terms of *Labeled Transition Systems* (*LTS*s) [11], where the *initial* and *final states* on the LTSs define the *sequences of actions* (traces) that characterize the *coordination policies* of the protocols.
(ii) Two *ontologies* O_P and O_Q describing the meaning of P and Q's actions, respectively.
(iii) Two *ontology mapping functions* defined from O_P and from O_Q to a common ontology. The intersection O_{PQ} on the common ontology identifies the "common language" between P and Q. For simplicity, and without loss of generality, we consider protocols P and Q that have disjoint languages and that are minimal where we recall that every finite LTS has a unique minimal representative LTS.
(iv) Then, starting from P and Q, and based on the ontology mapping, we build two abstractions A_P and A_Q by the relabeling of P and Q respectively, where the actions not belonging to the common language O_{PQ} are hidden by means of silent actions (τs);
(v) After, we check the compatibility of the protocols by looking if there exist complementary traces in the set of traces T_P and T_Q generated by A_P and A_Q respectively. If this is the case, then we are able to synthesize a mediator that makes it possible for the protocols to coordinate.
(vi) Finally, given two protocols P and Q, and an environment E, the mediator M that we synthesize is such that when building the parallel composition $P||Q||E||M$, P and Q are able to coordinate by reaching their final states.

2.3 The Popcorn Scenario

To better illustrate protocol mediation, and to make the theory more concrete, we consider a scenario called Distributed Marketplace or Popcorn scenario that we describe in the following and that is detailed in [1]. Consider an event held within a stadium populated by heterogeneous authorized merchants and consumers. During the event, consumers (respectively merchants) may want to buy (respectively sell) some product by exploiting the applications on their devices. Consider further a consumer application implemented using Lime tuple space

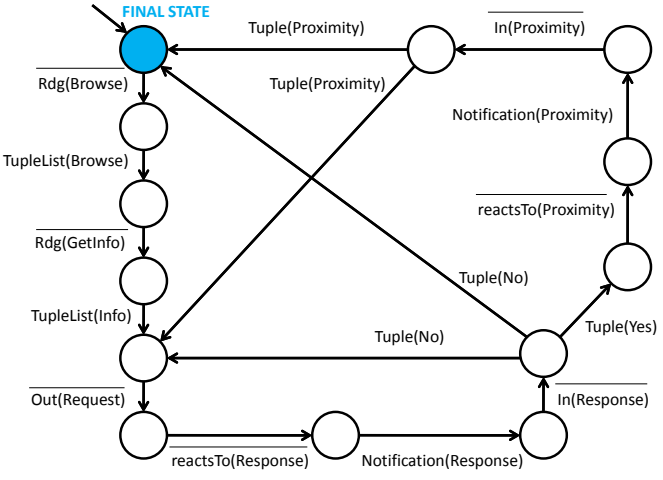


Fig. 2. The LTS of the tuple space consumer

and a merchant application implemented using UPnP. Figures 2 and 3 give their respective behavioral representation in terms of LTS.

Informally, the consumer application (Fig. 2) behaves as follows: first, it browses the tuple space to retrieve the list of all merchants. Once it gets it, it looks for details about the merchants that sell a specific product (popcorn in our case) with a certain price (for example, less than a threshold) within some distance (for example, within a given range). Then, the application writes into the tuple space a request of a given quantity of the product to the chosen merchant and waits for a response. If the request can be satisfied, the consumer receives a positive response and waits for a signal of proximity that the merchant application will send when close enough. Otherwise, the consumer receives a negative response (e.g, because the merchant has no sufficient quantity of product to satisfy the request). In both cases, the consumer can either send a new request or restart from the beginning, i.e., from browsing the tuple space.

The behavior of the merchant application (Fig. 3) can be described as follows: it gets authorization from the event organizers, it receives queries from consumers and sends answers to them advertising its information. Then, the application receives more requests of information from the consumers and answers them providing the required information. Further, it receives requests of ordering of products from consumers and answers a consumer either: positively sending a proximity message when it is physically close to the consumer, or negatively in case it is not able to satisfy the request.

Even though these two applications have some complementary behaviors (protocols are *functionally matching*), they are very different and they are not able to interoperate (because of protocol behavioral *mismatches*). Note that the merchant needs first to be authenticated by a third party. In addition, mediating the

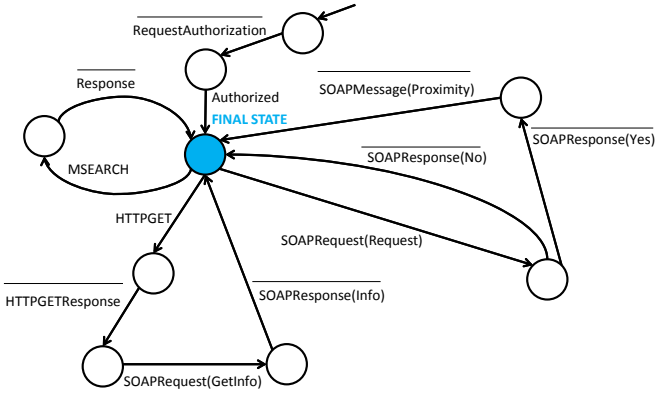


Fig. 3. The LTSs of the UPnP merchant

protocols of the consumer and merchant to achieve interoperability, is far from trivial, especially if one wants to achieve this automatically. This is such an automated support that we aim at in our work, where first results of our approach are presented in [17], which presents a high level description of the theory.

3 A Formalization of Protocols

As discussed in Section 2, the "application-layer interaction protocol" is the behavior of a system in terms of the actions it exchanges with its environment, i.e., other application-layer interaction protocols. We further exploit LTS to characterize such behavior.

3.1 Protocols as LTS

LTSs constitute a widely used model for concurrent computation and are often used as a semantic model for formal behavioral languages such as process algebras. Let *Act* be the set of observable actions (input/output actions), we get the following definition for LTS:

Definition 1 (*LTS*)
A LTS P is a quadruple (S, L, D, s_0) *where:*
- S *is a finite set of states;*
- $L \subseteq Act \bigcup \{\tau\}$ *is a finite set of labels (that denote observable actions) called the alphabet of P.* τ *is the silent action. Labels with an overbar in L denote output actions while the ones without overbar denote input actions. We also use the usual convention that for all* $l \in L, l = \overline{\overline{l}}$.
- $D \subseteq S \times L \times S$ *is a transition relation;*
- $s_0 \in S$ *is the initial state.*

We then denote with $\{L \bigcup \{\tau\}\}^*$ the set containing all words on the alphabet L. We also make use of the usual following notation to denote transitions:

$$s_i \xrightarrow{l} s_j \Leftrightarrow (s_i, l, s_j) \in D$$

We consider an extended version of LTS, where the set of the LTS' *final states* is explicit. An *extended LTS* is then a quintuple (S, L, D, F, s_0) where the quadruple (S, L, D, s_0) is a LTS and $F \subseteq S$. From now on, we use the terms LTS and extended LTS interchangeably, to denote the latter one. The initial state together with the final states, define the boundaries of the protocol's coordination policies. A *coordination policy* is indeed defined as any trace that starts from the initial state and ends into a final state. We get the following formal definition of traces/coordination policy:

Definition 2 (*Trace*)
Let $P = (S, L, D, F, s_0)$.
A trace $t = l_1, l_2, \ldots, l_n \in L^*$ *is such that:*
$\exists(s_0 \xrightarrow{l_1} s_1 \xrightarrow{l_2} s_2 \ \ldots \ s_m \xrightarrow{l_n} s_n)$ *where* $\{s_1, s_2, \ldots, s_m, s_n\} \in S \wedge s_n \in F$.
We also use the usual compact notation $s_0 \overset{t}{\Rightarrow} s_n$ *to denote a trace, where* t *is the concatenation of actions of the trace.*

We adopt the notion of parallel composition *à la* CSP [6]. We recall that the semantics of the parallel composition is that processes P and Q need to synchronize on common actions while can proceed independently when engaged in non common actions.

Definition 3 (*Parallel composition of protocols*)
Let $P = (S_P, L_P, D_P, F_P, s_{0_P})$ *and* $Q = (S_Q, L_Q, D_Q, F_Q, s_{0_Q})$.
The parallel composition between P *and* Q *is defined as:*
the LTS $P||Q = (S_P \times S_Q,\ L_P \cup L_Q,\ D,\ F_P \cup F_Q,\ (s_{0_P}, s_{0_Q}))$ *where the transition relation* D *is defined as follows:*

$$\frac{P \xrightarrow{m} P'}{P||Q \xrightarrow{m} P'||Q} \qquad m \notin L_Q$$

$$\frac{Q \xrightarrow{m} Q'}{P||Q \xrightarrow{m} P||Q'} \qquad m \notin L_P$$

$$\frac{P \xrightarrow{m} P'; Q \xrightarrow{\overline{m}} Q'}{P||Q \xrightarrow{\tau} P'||Q'} \qquad m \in L_P \bigcap L_Q$$

Note that when we build the parallel composition of protocols P and Q with the environment E and the mediator M, the composed protocol is restricted to the languages of P and Q thus forcing them to synchronize.

3.2 Abstract Protocol

Given the definition of enriched LTS associated with two interaction protocols run by networked systems, we want to identify whether such two protocols are functionally matching and, if so, to synthesize the mediator that enables them to interoperate, despite language differences, third parties communications, and behavioral mismatches. With *functional matching* we mean that given two systems with respective interaction protocols P and Q, ontologies O_P and O_Q describing their actions, ontology mapping functions of P and Q, and their common ontology O_{PQ}, there exist at least two complementary traces that allow P and Q to coordinate. That is, sequences of actions of one protocol can synchronize with sequences of actions in the other. This can happen by properly taking into account translation of the languages and communication with third parties. Thus, at a given level of abstraction, we expect to find a common protocol that represents their potential interactions. This leads us to formally analyze such alike protocols to find, if it exists, a suitable mediator that allows the interoperability that otherwise would not be possible.

In order to find the protocols' abstractions, we exploit the information contained in the ontology mapping to suitably relabel the protocols. The relabeling function allows us to substitute (sequences of) actions of the original language into action(s) on the common language thanks to the ontology mapping. After the relabeling operation on the LTSs, we obtain new LTSs labeled only by common actions and τs, that is more abstract than before, e.g., sequences of actions may have been compressed into single actions. In the following, we give the formal definitions concerning the abstract protocol. With respect to the popcorn scenario, Figure 4 summarizes the ontological information of the consumer (first column) and of the merchant (third column). The second column shows the common language mapping where Figures 2 and 3 illustrate the two protocols. Thanks to the ontology mapping, labels of consumer and merchant's protocols (expressed on two different ontologies) are *mapped* onto labels of the common ontology (more abstract). We specialize the usual ontology mapping definition [9,10] by considering pairs of elements made by more than one label. We use the specialized ontology mapping on protocols' ontology where the vocabularies are represented by the languages of the protocols. That is, we consider $P = (S_P, L_P, D_P, F_P, s_{0_P})$, $O_P = (L_P^*, A_P)$ the ontology of P, and $O = (L^*, A)$ another ontology. $maps : L_P^* \rightarrow L^*$ is the ontology mapping function that maps P's ontology into the ontology O.

By applying this ontology mapping, we relabel protocols with words of their common language (ontology) and τs for the thirds parties languages. To identify which is the common language, we first map each protocol's ontology into another one, resulting from ontology mediation, and then by intersection we find their common language.

Figure 5 depicts the abstraction of protocols. We consider two protocols P and Q with respective ontologies O_P and O_Q and ontology mapping functions $maps_P$ and $maps_Q$. We first use the mapping functions to map O_P and O_Q into a target ontology where C_{OP} and C_{OQ} represent the codomain sets of $maps_P$ and $maps_Q$ respectively. The subsets of D_P and D_Q of O_P and O_Q, respectively, represent

Tuple Space Consumer	Common Language Mapping		UPnP+SOAP Merchant
Rdg(Browse). TupleList(Browse)	$\overline{\alpha_1}$ α_2	α_1 $\overline{\alpha_2}$	MSEARCH. Response
Rdg(GetInfo). TupleList(Info)	$\overline{\beta_1}$ β_2	 β_1 $\overline{\beta_2}$	HTTPGET. HTTPGETResponse. SOAPRequest(GetInfo). SOAPResponse(Info)
Out(Request). reactsTo(Response)	 $\overline{\gamma_1}$	γ_1	SOAPRequest(Request)
Notification(Response). In(Response). Tuple(No)	 δ_1	$\overline{\delta_1}$	SOAPResponse(No)
Notification(Response). In(Response). Tuple(Yes). reactsTo(Proximity). Notification(Proximity). In(Proximity). Tuple(Proximity)	 σ_1 σ_2 	$\overline{\sigma_1}$ $\overline{\sigma_2}$	SOAPResponse(Yes). SOAPMessage(Proximity)

Fig. 4. Ontology mapping between tuple space consumer and UPnP Merchant

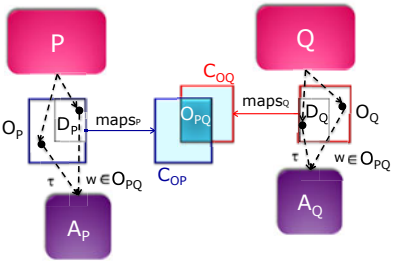
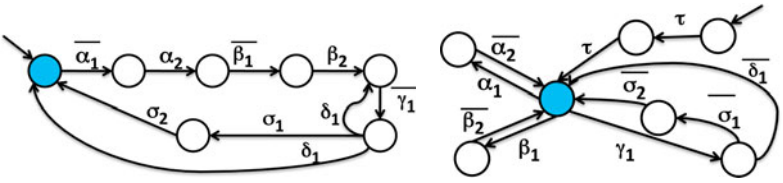


Fig. 5. The abstract protocol building

the portion of the domains of $maps_P$ and $maps_Q$ respectively corresponding to C_{OP} and C_{OQ}. Note that we consider protocols such that for each element of the codomain corresponds only one element of the domain. The common language between P and Q is defined as the intersection O_{PQ} of C_{OP} and C_{OQ}. The relabeled (abstracted) protocols, A_P and A_Q of P and Q respectively, are built as follows: (i) the chunks (states and transitions) of P and Q labeled by words of D_P and D_Q, respectively are substituted by building a single transition labeled with words of O_{PQ} ; (ii) all the other transitions labeled with actions belonging to the thirds parties language, are relabeled with τs . Having P, Q, O_P, O_Q, $maps_P$, and $maps_Q$, the relabeling function is applied after the computation of C_{OP}, C_{OQ}, D_P, D_Q, and O_{PQ}. The relabeling function on P takes as input $P, D_P, O_{PQ}, \{O_P \setminus D_P\}$ and gives as result an abstracted LTS A_P (it applies similarly for Q). More formally, having P, Q, O_P, O_Q, $maps_P$, $maps_Q$, C_{OP}, C_{OQ}, D_P, D_Q, and O_{PQ} the relabeling function is defined as follows:



(a) Consumer (b) Merchant

Fig. 6. Abstracted LTSs of consumer and merchant protocols

Definition 4 (*Relabeling function*)
Through the function relabels : $(P, D_P, O_{PQ}) \rightarrow A_P$, A_P *is built as follow: each chunk of* P *labeled with a trace* t *belonging to* D_P *is substituted by one transition with label* $w \in O_{PQ}$ *and each transition labeled with a word belonging to* $\{O_P \setminus D_P\}$ *is maintained and relabeled with a* τ.

In the popcorn scenario, the only label that is not abstracted in the common language is the authorization that represents a third party coordination. The consumer and merchant's abstracted LTSs are shown in Figure 6. The subsequent step is to check whether the two abstracted protocols share a *complementary coordination policy*, i.e., whether the abstracted protocols may indeed synchronize, which we check over protocol traces as discussed next.

3.3 Towards Automated Matching and Mediator Synthesis

The formalization described so far is needed to: (1) characterize the protocols and (2) abstract them into protocols on the same alphabet. As illustrated previously, to establish whether two protocols P and Q can interoperate we have to check the existence of portions of their respective abstracted protocols (A_P and A_Q) that can interoperate. That is, A_P and A_Q have to share complementary policies. To establish this, we use the *functional matching relation* between A_P and A_Q. This relation succeeds if A_P and A_Q have complementary traces. More formally:

Definition 5 (*Functional matching*)
Let $P = (S_P, L_P, D_P, F_P, s_{0_P})$ *and* $Q = (S_Q, L_Q, D_Q, F_Q, s_{0_Q})$.
Let A_P *and* A_Q *be the abstracted protocols of* P *and* Q, *respectively.*
Let T_{AP} *and* T_{AQ} *be the set of all the traces of* A_P *and* A_Q, *respectively.*
Let $\mathcal{C}$ *be a coordination policy denoted by final state* $f_\mathcal{C}$
P *and* Q *have a functional matching under ontology mapping* $maps_P$ *and* $maps_Q$ *with respect to policy* $\mathcal{C}$ *iff:* $f_\mathcal{C} \in maps_P(F_P)$, $f_\mathcal{C} \in maps_Q(F_Q)$, *Let* $T_{\mathcal{C}_{AP}} = \{s_{0_P} \stackrel{t}{\Rightarrow} f_\mathcal{C} \in T_{AP}\}$ *and* $T_{\mathcal{C}_{AQ}} = \{s_{0_Q} \stackrel{t}{\Rightarrow} f_\mathcal{C} \in T_{AQ}\}$, *then* $T_{\mathcal{C}_{AP}} =_{s/r} T_{\mathcal{C}_{AQ}}$ *where the equality of sets of traces over send-receive (noted* s/r*) denotes equality modulo complementary send-receive actions.*

The *functional matching relation* defines necessary conditions that must hold in order for a set of networked systems to interoperate through a mediator. In

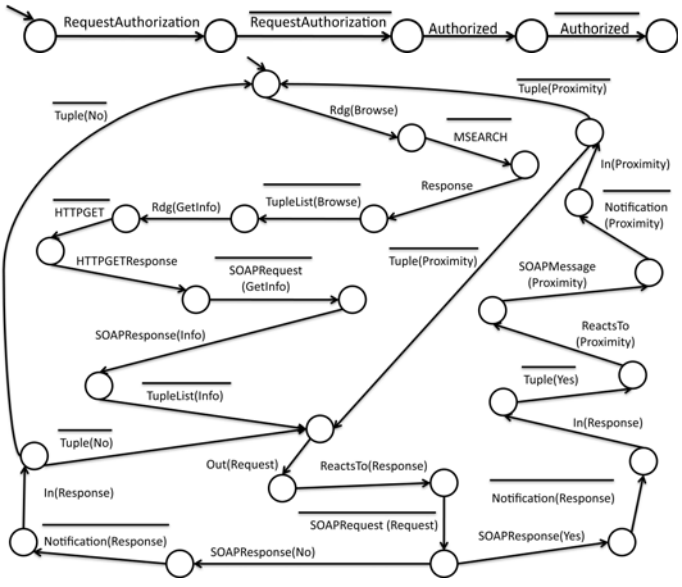


Fig. 7. Mediating connector for the popcorn scenario (M_T and M_C)

our case, till now, the set is made by two networked systems and the matching condition is that they have complementary traces regarding a given coordination policy. Note that these traces are computed on the abstracted protocols and might contain τ actions that represent third parties synchronization.

Then, given given two protocols P and Q that functionally match, we want to synthesize a mediator M such that the parallel composition $P||M||Q$, allows P and Q to evolve to their final states. An action of P and Q can belong either to the *common language* or the *third parties language*, i.e., the environment. We build the mediator in such a way such that it lets P and Q evolve independently for the portion of the behavior to be exchanged with the environment (denoted by τ action in the asbracted protocols) until they reach a "synchronization state" from which they can synchronize on complementary actions. Note that the synchronization cannot be direct since the mediator needs to perform a suitable translation according to the ontology mapping, e.g. $\alpha_1 = \overline{Rdg(Browse)}$ in one protocol and $\alpha_1 = MSEARCH$ in the other.

The mediator is made of two separate components: M_C and M_T. M_C speaks only the common language and M_T speaks only the third parties language. M_C is a LTS built starting from the common language between P and Q whose aim is to solve the protocol-level mismatches occurring among their dual interactions (complementary sequences of actions) by translating and coordinating between them. M_T, if it exists, is built starting from the third parties language of P and Q and represents the environment. The aim of M_T is to let the protocols evolve, from the initial state or from a state where a previous synchronization is ended, to the states where they can synchronize again. Formalization of mediator synthesis

given the specification of functional matching is part of our current work, while we summarize below the principles of our approach using the popcorn scenario.

In our illustration, we assume to have with the behavioral specification of consumer and merchant applications as LTSs (Figures 2 and 3), their coordination policies (thanks to the initial and final states on LTSs), their respective ontologies describing their actions, and the ontology mapping that defines the common language between consumer and merchant, i.e., represents their possible interactions (Figure 4). The first step is to *abstract* the protocols exploiting the ontology mapping. Following the theory, the abstracted protocols for the popcorn scenario are illustrated in Figure 6. The second step is to check whether they share some coordination policies. In this scenario we recall that the merchant is able to simulate the consumer. Then the coordination policies that they share are exactly the consumer's ones. Then, with the application of the theory to the scenario, we obtain the connector of Figure 7. In this case only the merchant have third parties language actions and then the mediator is made by the part that translates and coordinates regarding the common language and the part that simulates the environment.

4 Related Work

A number of solutions to automated protocol mediation have recently emerged, leveraging the rich capabilities of Web services and Semantic Web technologies [21,20,15,24]. They differ with respect to: (a) a priori exposure of the process models associated with the protocols that are executed by networked resources, (b) knowledge assumed about the protocols run by the interacting parties, (c) matching relationship that is enforced. However, most solutions are discussed informally, making it difficult to assess their respective advantages and drawbacks.

What is needed is a new and formal foundation for mediating connectors from which protocol matching and associated mediation may be rigorously defined and assessed. These relationships may be automatically reasoned upon, thus paving the way for on the fly synthesis of mediating connectors. To the best of our knowledge, such an effort has not been addressed in the Web services and Semantic Web area although proposed algorithms for automated mediation manipulates formally grounded process models.

However a work very close to our is [25] that proposes a theory to characterize and solve the interoperability problem of augmented interfaces of applications. The authors formally defines the checks of applications compatibility and the concept of adapters. The latter can be used to bridge the differences discovered while checking the applications that have functional matching but are protocol incompatible. Furthermore they provide a theory for the automated generation of adapters based on interface mapping constraints. The main disadvantages of this work are that the approach is semi-automatic because of the interface mapping. Additionally, applications are assumed to agree on the ordering of messages, thus not solving ordering mismatches.

A recent work [3] addresses the interoperability problem between services and provide experimentation on real Web2.0 social applications. The paper deals with the integration of a new service implementation, to substitute a previous one with the same functionalities. The new implementation does not still guarantee behavioral compatibility despite complying with the same API of the previous one. They hence propose a technique to dynamically detect and fix interoperability problems based on a catalogue of inconsistencies and their respective adapters. This is similar to our proposal to use ontology mapping to discover mismatches and mediator to solve them. Our work differs with respect to theirs because we aim at automatically synthesizing the mediator. Instead, their approach is not fully automatic since although they discover and select mismatches dynamically, the identification of mismatches and of the opportune adapters is made by the engineer.

References [13,2] are related to our work since they identify and classify basic types of mismatches that can possibly occur when compatible but mismatching processes try to interoperate. Moreover, they provide support to the developers by assisting them while identifying protocol mismatches and composing mediators. In [13], the authors also take into consideration more complex mediators obtained by composition of basic ones. The main difference between these two works and ours is the semi-automation issue. Indeed, they require the developer intervention for detecting the mismatches, configuring the mediators, composing basic mediators while, thanks to formal methods, we are able to automatically derive the mediator under some conditions.

5 Conclusion

In this paper, we have formally investigated the interoperability of protocols that are observable at the interface level. Key issue is to solve behavioral mismatches among the protocols although they are functionally matching. We have specifically introduced a theory towards interoperability as a means to: (1) clearly define the problem, (2) show the feasibility of the automated reasoning about protocols, i.e., to check their functional matching and to detect their behavioral mismatches, (3) show the feasibility of the automated synthesis of abstract mediators under certain conditions to dynamically overcome behavioral mismatches of functionally matching protocols. Our theoretical framework is a first step towards the automatic synthesis of actual mediators and we believe that it is very important to devote investigation to this goal. Significant part of our current work is on leveraging practically the proposed theory in particular dealing with automated reasoning about protocol matching and further automated protocol mediation. Our current work is further concerned with the integration with complementary work ongoing within the CONNECT project so as to develop an overall framework enabling the dynamic synthesis of emergent connectors among networked systems. Relevant effort includes the study of: learning techniques to dynamically discover the protocols that are run in the environment, dependability assurance, data-level mediation, as well as algorithms and run-time techniques

towards efficient synthesis. Such effort is presented in the CONNECT companion papers of the Isola'2010 conference, and detail about overall CONNECT results may be found from the project Web site at http://connect-forever.eu/.

References

1. The popcorn scenario dry-run experiment's details, http://www.connect-forever.eu/connect-dry-run/
2. Benatallah, B., Casati, F., Grigori, D., Nezhad, H.R.M., Toumani, F.: Developing adapters for web services integration. In: Pastor, Ó., Falcão e Cunha, J. (eds.) CAiSE 2005. LNCS, vol. 3520, pp. 415–429. Springer, Heidelberg (2005)
3. Denaro, G., Pezzè, M., Tosi, D.: Ensuring interoperable service-oriented systems through engineered self-healing. In: Proceedings of ESEC/FSE 2009. ACM Press, New York (2009)
4. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Resusable Object-Oriented Software. Addison-Wesley, Reading (1995)
5. Grace, P., et al.: Towards an architecture for runtime interoperability. In: Margaria, T., Steffen, B. (eds.) ISoLA 2010, Part II. LNCS, vol. 6416, pp. 206–220. Springer, Heidelberg (2010)
6. Hoare, C.A.R.: Communicating sequential processes. ACM Commun. 26(1), 100–106 (1983)
7. Howar, F., Jonsson, B., Merten, M., Steffen, B., Cassel, S.: On handling data in automata learning: Considerations from the connect perspective. In: Margaria, T., Steffen, B. (eds.) ISoLA 2010, Part II. LNCS, vol. 6416, pp. 221–235. Springer, Heidelberg (2010)
8. Issarny, V., Steffen, B., Jonsson, B., Blair, G., Grace, P., Kwiatkowska, M., Calinescu, R., Inverardi, P., Tivoli, M., Bertolino, A., Sabetta, A.: CONNECT Challenges: Towards Emergent Connectors for Eternal Networked Systems. In: 14th IEEE International Conference on Engineering of Complex Computer Systems, Postdam Germany (2009)
9. Kalfoglou, Y., Schorlemmer, M.: Ontology mapping: the state of the art. Knowl. Eng. Rev. 18(1), 1–31 (2003)
10. Kalfoglou, Y., Schorlemme, M.: Ontology mapping: The state of the art. In: Kalfoglou, Y., Schorlemmer, M., Sheth, A., Staab, S., Uschold, M. (eds.) Semantic Interoperability and Integration, Dagstuhl, Germany. Dagstuhl Seminar Proceedings, vol. 04391, IBFI, Schloss Dagstuhl (2005)
11. Keller, R.M.: Formal verification of parallel programs. ACM Commun. 19(7), 371–384 (1976)
12. Kumar, R., Nelvagal, S., Marcus, S.I.: A discrete event systems approach for protocol conversion. Discrete Event Dynamic Systems 7(3) (1997)
13. Li, X., Fan, Y., Wang, J., Wang, L., Jiang, F.: A pattern-based approach to development of service mediators for protocol mediation. In: Proceedings of WICSA 2008, pp. 137–146. IEEE Computer Society, Los Alamitos (2008)
14. Liskov, B.H., Wing, J.M.: A behavioral notion of subtyping. ACM Trans. Program. Lang. Syst. 16(6) (1994)
15. Motahari Nezhad, H.R., Benatallah, B., Martens, A., Curbera, F., Casati, F.: Semi-automated adaptation of service interactions. In: WWW 2007: Proceedings of the 16th international conference on World Wide Web, pp. 993–1002. ACM, New York (2007)

16. Noy, N.F.: Semantic integration: a survey of ontology-based approaches. SIGMOD Rec. 33(4) (2004)
17. Spalazzese, R., Inverardi, P., Issarny, V.: Towards a formalization of mediating connectors for on the fly interoperability. In: Proceedings of the Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture, WICSA/ECSA 2009 (2009)
18. Spitznagel, B., Garlan, D.: A compositional formalization of connector wrappers. In: ICSE 2003: Proceedings of the 25th International Conference on Software Engineering (2003)
19. Stollberg, M., Cimpian, E., Mocan, A., Fensel, D.: A semantic web mediation architecture. In: Proceedings of the 1st Canadian Semantic Web Working Symposium (CSWWS 2006). Springer, Heidelberg (2006)
20. Vaculín, R., Neruda, R., Sycara, K.P.: An Agent for Asymmetric Process Mediation in Open Environments. In: Kowalczyk, R., Huhns, M.N., Klusch, M., Maamar, Z., Vo, Q.B. (eds.) SOCASE 2008. LNCS, vol. 5006, pp. 104–117. Springer, Heidelberg (2008)
21. Vaculín, R., Sycara, K.: Towards automatic mediation of OWL-S process models. In: IEEE International Conference on Web Services, pp. 1032–1039 (2007)
22. Wiederhold, G.: Mediators in the architecture of future information systems. IEEE Computer 25, 38–49 (1992)
23. Wiederhold, G., Genesereth, M.: The conceptual basis for mediation services. IEEE Expert: Intelligent Systems and Their Applications 12(5), 38–47 (1997)
24. Williams, S.K., Battle, S.A., Cuadrado, J.E.: Protocol mediation for adaptation in semantic web services. In: Sure, Y., Domingue, J. (eds.) ESWC 2006. LNCS, vol. 4011, pp. 635–649. Springer, Heidelberg (2006)
25. Yellin, D.M., Strom, R.E.: Protocol specifications and component adaptors. ACM Trans. Program. Lang. Syst. 19(2), 292–333 (1997)

On-the-Fly Interoperability through Automated Mediator Synthesis and Monitoring*

Antonia Bertolino[1], Paola Inverardi[2], Valérie Issarny[3], Antonino Sabetta[1], and Romina Spalazzese[2]

[1] CNR-ISTI, Pisa, Italy
[2] Università degli Studi dell'Aquila, L'Aquila, Italy
[3] INRIA, CRI Paris-Rocquencourt, France

Abstract. Interoperability is a key and challenging requirement in today's and future systems, which are often characterized by an extreme level of heterogeneity. To build an interoperability solution between the networked systems populating the environment, both their functional and non-functional requirements have to be met.

Because of the continuous evolution of such systems, mechanisms that are fixed a-priori are inadequate to achieve interoperability. In such challenging settings, on-the-fly approaches are best suited.

This paper presents, as an interoperability solution, an approach that integrates an automated technique for the synthesis of mediator protocols with a monitoring mechanism. The former aims to provide interoperability taking care of functional characteristics of the networked systems, whereas the latter makes it possible to assess the non-functional characteristics of the connected system.

1 Introduction

The realization of the Ubiquitous Computing vision [18] is still nowadays challenged by the often extreme level of heterogeneity in the system's underlying infrastructures, which in turns impacts on the ability to seamlessly interoperate.

Interoperability is a primary requirement in such systems, and, in order to achieve it, two aspects have to be considered: *functional interoperability* and *non-functional interoperability*. The first one solely refers to functional properties and aims at allowing the Networked Systems (NSs) to communicate. Instead, non-functional interoperability refers to the assessment and achievement of the non-functional characteristics which qualify the communication (*how* it should be provided). Indeed, while building an interoperability solution, both functional and non-functional properties of the connected system under-construction must be taken into account and ensured.

The fast pace at which technology evolves at all the abstraction layers additionally hampers the interoperability achievement between NSs in the digital environment. Interoperability should be "future-proof", i.e., NSs should be able to interoperate in spite of technological evolution and contextual changes.

* This work is partly supported by the CONNECT European Project No. 231167.

T. Margaria and B. Steffen (Eds.): ISoLA 2010, Part II, LNCS 6416, pp. 251–262, 2010.

To face emerging functional and non-functional requirements in such heterogenous and evolving setting, relying on interoperability mechanisms that are fixed *a-priori*, can be proved to be inadequate, and *on-the-fly* approaches would be best suited.

Overcoming the interoperability barriers in the Ubiquitous Computing systems is at the heart of CONNECT [13]. The CONNECT Integrated Project aims at dropping the interoperability barrier by adopting a revolutionary approach to the seamless and eternal networking of systems, that is, by synthesizing on-the-fly the CONNECTors (or mediators) via which NSs communicate. The synthesis process is based on a formal foundation for CONNECTors, which allows learning, reasoning about, and adapting the interaction behavior of NSs at run-time. Synthesized connectors are concrete emergent system entities that are dependable, unobtrusive, and evolvable, while not compromising the quality of CONNECTed systems.

In this paper, as an excerpt of the CONNECT solution, we present an integrated approach to on-the-fly interoperability that combines automated mediator synthesis and monitoring. The former (based on a theory of mediator synthesis presented in [11]) aims at providing functional interoperability, whereas the latter takes care of non-functional interoperability.

One of the CONNECT underlying principle is to make minimal assumptions on the NSs. In particular, the project considers that NSs information, needed to compute an interoperability solution, is either declaratively provided by them or derived from them exploiting learning techniques. However, for the purpose of this paper, we assume NSs to come with their behavioral description and a set of policies describing their non-functional constraints, which may involve performance, dependability, security and trust requirements. In order to achieve a complete CONNECTion, the functional interoperability is pursued by construction through synthesis, whereas non-functional interoperability is addressed by suitably combining differing analysis, verification and enforcement techniques. An overview of the approaches under development in CONNECT for non-functional interoperability is given in [3]. In particular, we foresee to apply some approaches at synthesis time (see, e.g., the companion CONNECT paper by Di Giandomenico and coauthors [8]), in synergy with mediator construction. However, some operational constraints expressed as policies cannot be assessed statically at synthesis time. This paper focuses on this problem: we overview here the approach through which we combine the mediator synthesis with a runtime checking mechanism, implemented through monitoring.

Several other works in the literature relate to ours, both concerning the automated synthesis of mediators [19,17], and monitoring [12,5], especially in the context of service-oriented systems. The emphasis of this work however is *on the combination* of the two aspects and the approach we follow is mostly independent of specific technological frameworks.

The paper is organized as follows. We give an illustrative scenario and we present our approach at a high level (Section 2). Then, we recall our automated synthesis of mediators and we describe the integration with the monitoring of

mediators (Sections 3 and 4 respectively). Finally, we conclude with perspectives for future work (Section 5).

2 Approach Description

This section outlines our approach by introducing a running example first, and then we explain the principles of our approach applied to that example.

2.1 Running Example

Let us consider a *Photo-Sharing* system in a stadium; the system allows spectators to exchange pictures of the most significant happenings, e.g., goals in the case of football games. The spectators can be *producers* (respectively *consumers*) of pictures and hence they can *upload* (respectively *search, download*) pictures.

Different kind of interaction may be envisioned for the Photo-Sharing, including centralized and peer-to-peer ones. In a centralized implementation, the stadium would offer the Photo-Sharing service and the spectators' smartphones run service clients to upload, search, and download pictures; typical supporting middleware solution would be RPC-based, e.g., using a service-oriented middleware. In a peer-to-peer implementation, the spectators' smartphones would run a peer-to-peer application for photo exchanges (implementing the *upload*, *search* and *download* functionalities), for which a distributed shared memory *à la* tuple space would be the middleware of choice.

Fig. 1. High-level view of the Photo-Sharing NSs

The behavior of the producers consists in *uploading* the photo, and the behavior of the consumer lies in *searching* and then *downloading* photos of interest.

Considering the shared memory implementation, the Photo-Sharing producer writes first the *metadata* and then the *file* associated with the given photo into the shared memory; while the consumer seeks the list of metadata descriptors matching a given *metadata* template into the shared memory and then iteratively reads the files of interest. With respect to the RPC implementation, the producer and consumer exchange photos calling the proper services on the server and receiving the corresponding replies. The producer calls the upload operation with both *metadata* and *file* as parameters. The consumer, instead, calls the search operation with a certain metadata and receives as reply the list of elements matching the request. Then iteratively the consumer calls the download of selected pictures specified thanks to their identifier *ID* and receives as reply the corresponding files.

We consider, as a running example, the case in which an NS (NS_1) running a shared-memory-based photo producer protocol is in a stadium and the stadium is equipped with the shared-memory infrastructure. Another NS (NS_2) running an RPC-based photo consumer protocol accesses the stadium and wants to share pictures. Although apparently simple, this scenario presents substantial challenges to interoperability. In fact, despite the different implementations, the *intents* of the producers and consumers are *compliant* being upload and download pictures respectively. While there is an obvious behavioral (or protocol) mismatch between the RPC-based and the shared-memory implementations from the application down to the middleware layers. This type of mismatch can be addressed by emergent mediators, synthesized on the fly [11]. However, this does not solve the problem entirely because constraints on non-functional properties are still not taken into account. Our proposal to manage these aspects is outlined in the following.

2.2 On-the-Fly Connector Synthesis and Monitoring

The networked systems[1] populating the open environment, e.g. the Photo-Sharing producers and consumers, are characterized by: *intent*, *behavioral description*, *ontological description*, and by *constraints* (expectations) about *non-functional* properties. The above characterization could be either declaratively advertised by the NSs or inferred exploiting learning techniques [10,4].

The NSs constraints or requirements on the non-functional properties characterize "how" the interoperable connection should be provided. These constraints need to be expressed in a format that allows their automated interpretation and processing. In principle, different NSs could use many different languages to represent this information; in this paper we take the simplifying assumption that a mapping from such languages to a common CONNECT reference model exists and thus they are expressed in the same language.

To give an example of a NSs requirement, the photo-sharing consumer may require that the time it takes to get a list of photos that match a query be less than x ms.

We recall that a necessary condition for the networked systems to communicate is to be *compatible*. That is, to make sense for the NSs to communicate, they have to expose *complementary* (*provided*/*required*) intents. From a functional point of view, despite the complementarity of intents, the RPC-based consumer and the shared-memory-based producer cannot interoperate (communicate) directly with each other because their concrete protocols are different. In order to bridge this difference, a suitable *functional mediator* is synthesized on-the-fly [11]. The synthesis process happens at the time when the intention to communicate is manifested by either party. The goal of the synthesis process is to realize the "functional interoperability" by producing a protocol mediator that allows the two NSs to communicate dealing only with functional aspects.

[1] For the sake of simplicity, we explain the mediation process assuming only two NSs; in general, the same principles can be extended to scenarios with more NSs.

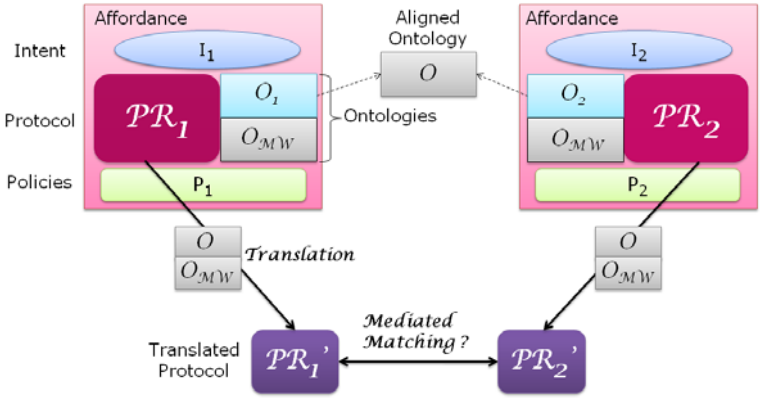


Fig. 2. An overview to mediators synthesis approach

In order to ensure that the functional mediator satisfies the non-functional constraints, we propose to couple the synthesized mediator with a suitable monitoring system whereby the non-functional constraints imposed by each NS can be checked at runtime. In this way, the monitoring is used to make sure that the connected system (i.e., the result of assembling the two NSs with the synthesized mediator) satisfies the expectations, in terms of non-functional characteristics, of both sides of the connection.

Thus, our approach addresses: 1) *functional interoperability* pursued by-construction at synthesis time (i.e.,*a-priori*), and 2) *non-functional interoperability*, that is compliance to non-functional constraints, continuously assessed at execution time (*a-posteriori*), by passive monitoring. Figure 2, whose elements are explained in the following, summarizes the main ingredients of our approach whose theory is presented in the companion paper [11].

We call *affordance* [9] the description of the functionality that is offered/requested by a networked system, i.e., to provide/require pictures in our Photo-Sharing scenario. In other words, an affordance is a high-level action-possibility (or functionality or capability) that characterizes the intended and/or possible interactions between the networked system and its environment.

We specialize this notion of affordance, to characterize our networked systems as mentioned in the beginning of this section.

More precisely, we consider that an affordance includes: (i) an *intent* (I_1, I_2 in Figure 2), that is used to perform a first check about the NSs compatibility in terms of complementarity of intents, i.e., provided/required functionalities; (ii) a *protocol* ($\mathcal{PR}_1$, $\mathcal{PR}_2$ in Figure 2), run by the system to carry on its capability, which is used to perform another check about protocol/behavioral compatibility; (iii) *middleware* and *applications' ontologies* ($\mathcal{O}_{\mathcal{MW}}$ and $\mathcal{O}_1$, $\mathcal{O}_2$ respectively in Figure 2), describing protocol's actions exploited during the protocol translation done before the check of protocol compatibility; (iv) a set of *policies* (P_1, P_2 in Figure 2) that qualify the conditions that are required for the NS to function

correctly. The policies are used to express constraints on the operational conditions under which a connection may take place.

In CONNECT certain types of policies, namely security policies, are not just checked but also enforced (see [7] for details). Other non-functional properties, such as those related to performance and reliability, business rules are assessed by passive observation on the live system and thus we can deal with them.

Our approach to the automated synthesis of mediators is briefly recalled in the next section, while monitoring integrated with the synthesis is the topic of Section 4.

3 Automated Synthesis of Mediators

In this section we introduce the necessary information about our synthesis approach to explain the integration with the monitoring. We summarize the automated synthesis of mediators that builds on the early theory of application-layer mediators presented in [16] and deals with the interoperability of both application- and middleware-layer. Additional details can be found in the companion paper on the theory of mediators [11] and in [1].

Given two NSs *affordances*, first we check (from a functional standpoint) that they have *compliant intents*, i.e. if they amount on the same capability (provided/required respectively).

Having checked the intent compliance, our goal is to synthesize a mediator to solve the *mismatches* occurring between the protocols. We use Labeled Transition Systems (LTSs) [14] to represent the protocols associated with the behavioral description of affordances.

Let Act be the set of observable input/output actions and τ be the silent action (we use the usual convention that the output actions are denoted by an overbar while the input actions have no overbar). An extended LTS, which makes final states explicit, is a quintuple (S, L, D, F, s_0) where: (i) S is a finite set of states, (ii) $L \subseteq Act \bigcup \{\tau\}$ is a finite set of labels called the alphabet of the LTS,

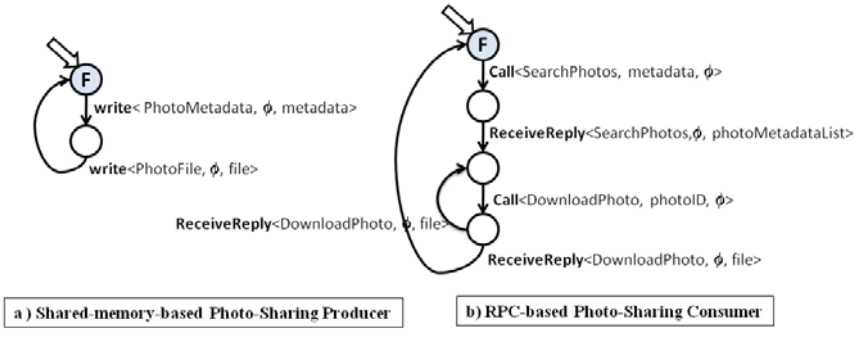


Fig. 3. Heterogeneous protocols in pervasive photo sharing

(iii) $D \subseteq S \times L \times S$ is a transition relation, (iv) $F \in S$ is the set of final states, and (v) $s_0 \in S$ is the initial state.

As an illustration, Figure 3 depicts the LTSs of the affordance protocols associated with the Photo-Sharing scenario that we informally introduced in Section 2.1.

In particular, an action of *Act* is specifically structured as: $MW < AP, IN, OUT >$, where MW denotes the *middleware function* that is called to interact with the peer system through the *application function* AP that is parameterized by *input* and *output parameters*, IN and OUT respectively.

Given the two LTSs, $\mathcal{PR}_1$ and $\mathcal{PR}_2$, characterizing the behaviors of functionally matching affordances, they are translated into LTSs $\mathcal{PR}'_1$ and $\mathcal{PR}'_2$ for the sake of comparison. The translated protocols are defined in a middleware-agnostic way over common application actions following application ontology alignment.

The translation of protocols in particular relies on the alignment of application layer functions into common application-specific ontologies (O in Figure 2) and on the translation of middleware functions from reference middleware ontology ($\mathcal{O}_{\mathcal{MW}}$ in Figure 2) into primitive send/receive actions [1].

Once $\mathcal{PR}'_1$ and $\mathcal{PR}'_2$ have been produced, we check their *compatibility* (also referred to as *mediated matching*) according to the set of traces T_1 and T_2 associated with $\mathcal{PR}'_1$ and $\mathcal{PR}'_2$, respectively. If the two protocols are compatible, then we are able to synthesize a mediator $\mathcal{M}$ that is such that when building the parallel composition $\mathcal{PR}_1||\mathcal{PR}_2||\mathcal{M}$, $\mathcal{PR}_1$ and $\mathcal{PR}_2$ are able to coordinate by reaching their final states.

Figure 4 shows the mediator protocols synthesized by our approach for the Photo-Sharing example.

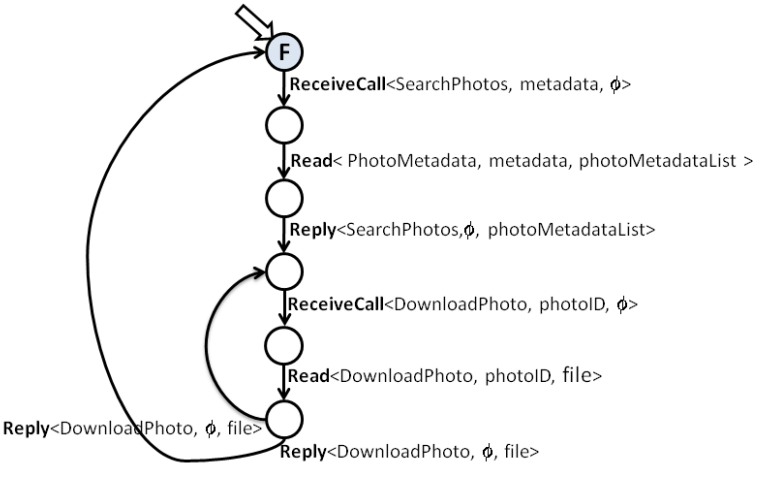


Fig. 4. Mediator protocol in pervasive photo sharing

4 Automated Monitoring of Mediators

The synthesis procedure described in the previous section yields a mediator that is able to bridge behavioral mismatches between NSs. However, affordance descriptions includes policies, which are used to express non-functional *requirements imposed by* the NS, or declarations of non-functional *characteristics*, *guaranteed by* the NS. In the Photo-Sharing example, a client may require that the time to obtain a list of photos that match a query must be less than X time units. This is an example of latency property. Similarly, the client may declare that it will never invoke the search operation more than three times in a minute (i.e., it guarantees it will generate a bounded workload).

In order to ensure that the CONNECTed system as a whole satisfies the requirements imposed by each NS participating in the connection, we adopt a runtime checking approach, supported by a dedicated monitoring infrastructure. This infrastructure (shown in Figure 5) is structured according to a generic, flexible architecture that decouples business-level (or high-level) event specification from the underlying observation and detection mechanisms. From a technical perspective, this decoupling is achieved by delegating to a *probe*, paired with mediator, the task of collecting low-level (i.e., primitive) event occurrences, which happen when a transition on the mediator LTS is taken. Primitive event occurrences are collected from the probes through a message-oriented backbone. The detection of complex events, defined as combinations of primitive events, is done using a Complex Event Recognizer [15]. Finally, complex event occurrences are notified to the interested consumers, again using the message-oriented backbone.

The monitoring manager is responsible for converting non-functional constraints coming from affordance specifications into directives to derive the probe(s), to instruct the Complex Event Recognizer, and to configure the routing

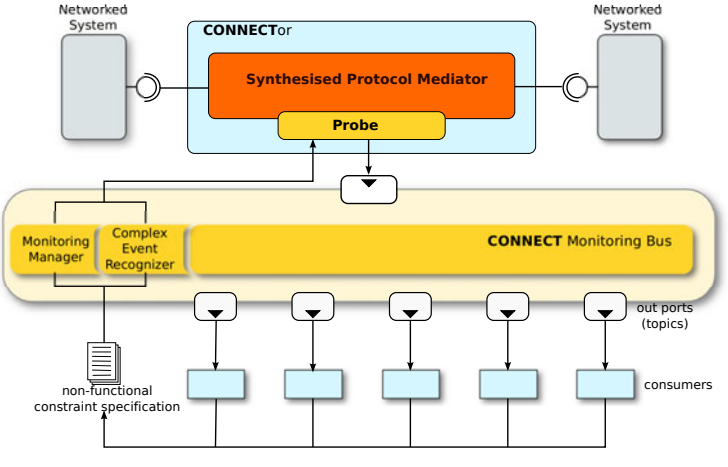


Fig. 5. CONNECT monitoring infrastructure

of monitoring information over the monitoring bus (i.e., from the probes to the event recognizer and then to the consumers interested in specific complex events).

As an example, in this paper we consider a constraint that defines the acceptable latency for operations used by the RPC photo-sharing client. As already mentioned, other properties can be checked using the same framework, as long as they can be translated onto the complex event specification language used in the CONNECT monitoring system.

We assume that latency constraints are expressed in terms of operations belonging to the interface of the NS. For example, the already mentioned constraint imposed by the RPC client on the time it takes to get a list of photos that match a query, can be expressed as:

$$\Delta(Call(SearchPhotos,\text{-},\text{-}),ReceiveReply(SearchPhotos,\text{-},\text{-}) < X$$

which means "the time elapsed from calling the operation SearchPhotos to the receiving a reply for that invocation must be less than X time units".

This policy is expressed in terms of high-level functionalities (i.e., operations included in the public interface of the NS) and using the RPC middleware primitives (in this example, *Call, ReceiveReply*). It must be processed in order to derive a specification that is used to configure the monitoring system so that the policy can be checked.

In order to do so, the policy is converted into its corresponding formulation in terms of *complementary* actions performed by the mediator. In this example, *Call(SearchPhotos,-,-)* translates to *ReceiveCall(SearchPhotos,-,-)* and analogously *ReceiveReply(SearchPhotos,-,-)* translates to *Reply(SearchPhotos,-,-)*. The resulting constraint on the behavior of the mediator is therefore:

$$\Delta(ReceiveCall(SearchPhotos,\text{-},\text{-}),Reply(SearchPhotos,\text{-},\text{-}) < X'$$

Although in general, $X' = X + X_n$, where X_n is the delay due to the network, for the sake of simplicity, here we assume that this delay is negligible (i.e., $X_n = 0$) and therefore $X' = X$.

In real-life scenarios, X_n is typically not negligible, so a client policy that requires the delay to be less than X (observed on the client) translates into a requirement that the delay observed on the mediator be less than $X - X_n$. This is a general problem entailed by observing latency (or other time-related properties) in networked settings. However, this is beyond the scope this paper and we do not discuss it further.

Finally, the expression obtained for the constraint on the mediator is translated into a language that is readily understood by the event-correlation engine. The example in Listing 1.1 shows the latency constraint expressed the specification language used by Drools Fusion [6], an open source rule engine with complex event processing capabilities.

The rule *PhotoSearchLatency* matches a *ReceiveCall* event followed by a *Reply event* (lines 4 and 8 respectively), ensuring that both refer to the same session

```
1  rule "PhotoSearchLatency"
2  when
3      // this is the complementary of Call
4      $call: ReceiveCall(
5          operation == "SearchPhotos";
6          $session : session_id)
7          from entry-point "PhotoSharingMediator"
8      $reply: Reply(
9          session_id == $session;
10         operation == "SearchPhotos";
11         this after[1200ms] $call )
12         from entry-point "PhotoSharingMediator"
13 then
14     // inject alarm on the monitoring bus
15 end
```

Listing 1.1. Sample rule for checking a latency policy

(lines 6 and 9) and that the latter happens no earlier than 1200 ms[2] after the former (line 11). If all these conditions are verified, an alarm is injected into the monitoring bus (line 14) so that the subscribers for that kind of complex event can be notified.

5 Conclusion

The high degree of heterogeneity in the current digital system's underlying infrastructures thwarts the realization of the long-standing Ubiquitous Computing vision. Interoperability is a key requirement in such systems, where both functional and non-functional aspects expressed by Networked Systems have to be met while making them able to work together.

The continuous evolution characterizing the Ubiquitous environment, asks for on-the-fly approaches rather than relying on interoperability mechanisms fixed a-priori that are not adequate to completely address the problem.

In order to achieve a complete CONNECTion (functional and non-functional interoperability), this paper presented a combined interoperability approach. It is made by the integration of an automated technique for the synthesis of mediators with a monitoring mechanism. The mediators provide functional interoperability and the monitors make it possible to assess the non-functional characteristics of the connected system at runtime that cannot be assessed statically at synthesis time.

As future work, we plan to investigate the following aspects that are important in the larger CONNECT picture [2].

We need to propose a language to express non-functional constraints and properties.

[2] In this example, we assume 1200 ms is the concrete value for X'.

We need to provide reaction policies or reaction policy patterns that can be undertaken when something wrong is detected by the monitoring. Examples are: to use predictive approaches that try to prevent the wrong behaviors; to adapt the CONNECT architectural infrastructure, if possible, for improving the provided connection; eventually, to notify the Networked Systems about the unexpected behavior, and let them directly handle the problem.

As a long-term goal, we will work towards including reasoning about non-functional properties into the synthesis process [8].

References

1. Bennaceur, A., Blair, G., Georgantas, N., Grace, P., Inverardi, P., Issarny, V., Pathak, A., Saadi, R., Spalazzese, R.: Revisiting the Middleware Paradigm: On-the-fly Interoperability in Highly Complex Distributed Systems. Technical Report, INRIA Rocquencourt - Paris (May 2010)
2. Bennaceur, A., Blair, G.S., Chauvel, F., Georgantas, N., Grace, P., Howar, F., Inverardi, P., Issarny, V., Paolucci, M., Pathak, A., Spalazzese, R., Steffen, B., Souville, B.: Towards an architecture for runtime interoperability. In: ISoLA 2010, Part II. LNCS, vol. 6416, pp. 206–220. Springer, Heidelberg (2010)
3. Bertolino, A., Di Giandomenico, F., Di Marco, A., Issarny, V., Martinelli, F., Masci, P.M., Matteucci, I., Saadi, R., Sabetta, A.: Dependability in dynamic, evolving and heterogeneous systems: the CONNECT approach. In: 2nd International Workshop on Software Engineering for Resilient Systems SERENE 2010, London, U.K (2010)
4. Bertolino, A., Inverardi, P., Pelliccione, P., Tivoli, M.: Automatic synthesis of behavior protocols for composable web-services. In: ESEC/FSE 2009: Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, pp. 141–150. ACM, New York (2009)
5. Bianculli, D., Ghezzi, C.: Monitoring conversational web services. In: IW-SOSWE 2007: 2nd international workshop on Service oriented software engineering, pp. 15–21. ACM, New York (2007)
6. Browne, P.: JBoss Drools Business Rules. Packt Publishing (2009)
7. Costa, G., Matteucci, I.: Enforcing private policy via security-by-contract. Special issue Identity and Privacy Management. UPGRADE Journal XI(1), 43–53 (February 2010)
8. Di Giandomenico, F., Kwiatkowska, M., Martinucci, M., Masci, P., Qu, H.: Dependability analysis and verification for connected systems. In: ISoLA 2010, Part II. LNCS, vol. 6416, pp. 263–277. Springer, Heidelberg (2010)
9. Gibson, J.J.: The ecological approach to visual perception. Houghton Mifflin (1979)
10. Howar, F., Jonsson, B., Merten, M., Steffen, B., Cassel, S.: On handling data in automata learning: Considerations from the connect perspective. In: Margaria, T., Steffen, B. (eds.) ISoLA 2010, Part II. LNCS, vol. 6416, pp. 221–235. Springer, Heidelberg (2010)
11. Inverardi, P., Issarny, V., Spalazzese, R.: A theory of mediators for eternal connectors. In: ISoLA 2010, Part II. LNCS, vol. 6416, pp. 236–250. Springer, Heidelberg (2010)
12. Inverardi, P., Mostarda, L.: Desert: a decentralized monitoring tool generator. In: ASE 2007: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, pp. 529–530. ACM, New York (2007)

13. Issarny, V., Steffen, B., Jonsson, B., Blair, G., Grace, P., Kwiatkowska, M., Calinescu, R., Inverardi, P., Tivoli, M., Bertolino, A., Sabetta, A.: CONNECT Challenges: Towards Emergent Connectors for Eternal Networked Systems. In: 14th IEEE International Conference on Engineering of Complex Computer Systems, Postdam Germany (2009)
14. Keller, R.M.: Formal verification of parallel programs. ACM Commun. 19(7), 371–384 (1976)
15. Luckham, D.C.: The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems. Addison-Wesley Longman Publishing Co., Inc., Amsterdam (2001)
16. Spalazzese, R., Inverardi, P., Issarny, V.: Towards a formalization of mediating connectors for on the fly interoperability. In: Proceedings of the Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture (WICSA/ECSA 2009), pp. 345–348 (2009)
17. Vaculín, R., Sycara, K.: Towards automatic mediation of OWL-S process models. In: IEEE International Conference on Web Services, pp. 1032–1039 (2007)
18. Weiser, M.: The computer for the 21st century. Scientific American (Septmeber 1991)
19. Williams, S.K., Battle, S.A., Cuadrado, J.E.: Protocol mediation for adaptation in semantic web services. In: Sure, Y., Domingue, J. (eds.) ESWC 2006. LNCS, vol. 4011, pp. 635–649. Springer, Heidelberg (2006)

Dependability Analysis and Verification for CONNECTed Systems*

Felicita Di Giandomenico[1], Marta Kwiatkowska[2], Marco Martinucci[1], Paolo Masci[1,3], and Hongyang Qu[2]

[1] Information Science and Technologies Institute, CNR, Pisa, Italy
[2] Oxford University Computing Laboratory, Oxford, UK
[3] Dept. of Information Engineering, University of Pisa, Italy

Abstract. The CONNECT project aims to enable the seamless composition of heterogeneous networked systems. In this context, Verification and Validation (V&V) techniques are sought to ensure that the CONNECTed system satisfies dependability requirements. Stochastic model checking and state-based stochastic methods are two appealing V&V approaches to accomplish this task. In this paper, we report on the application of the two approaches in a typical CONNECT scenario. Specifically, we make clear (i) how the two approaches can be employed to enhance the confidence in the correctness of the analysis, and (ii) how the complementarity of these approaches can be fruitfully exploited to extend the analysis.

1 Introduction

The CONNECT project [14] aims at dropping the barriers that prevent heterogeneous networked systems from being CONNECTed, by enabling their seamless composition in spite of technological evolution. To achieve this aim, CONNECT intends to dynamically synthesise the CONNECTors that allow the networked systems to communicate. The resulting emergent CONNECTors compose and adapt interaction protocols run by the CONNECTed systems.

In addition to functional properties, CONNECTors generally need to satisfy non-functional properties as well. Therefore, an evaluation to assess whether the CONNECTor specification is adequate to satisfy the dependability requirements is highly desirable. Indeed, Verification and Validation (V&V) techniques are sought in CONNECT to ensure that networked systems, as well as the generated bridging CONNECTors, satisfy specified levels of accomplishment for dependability requirements, according to pertinent dependability metrics. Note that, in CONNECT, dependability is used as a term inclusive of several non-functional properties [4], e.g., including also performance aspects.

In CONNECT we are interested in quantitative, or probabilistic, dependability evaluation. To this purpose, as indicated in [1], the two main approaches are modelling and (evaluation) testing. Since evaluation testing assumes that a test

* This work is supported by the European FP 7 project CONNECT (IST 231167).

T. Margaria and B. Steffen (Eds.): ISoLA 2010, Part II, LNCS 6416, pp. 263–277, 2010.

suite is run on the system under test, which in the CONNECT vision is not a priori available, for the analysis we focus on modelling.

Modelling is composed of two phases: (i) building a model for the system from the elementary stochastic processes that represent the behaviour of the components of the system and their interactions (these elementary stochastic processes relate to failures, to repair, service restoration and possibly to system duty cycle or phases of activity); (ii) processing the model to obtain the expressions and the values of the dependability measures of the system.

Research in dependability analysis has developed a variety of modelling techniques, each of which focuses on particular levels of abstraction and/or system characteristics. As reported in [18], important classes of model representation include combinatorial methods (such as Reliability Block Diagrams), model checking, and state-based stochastic methods. We are not interested in combinatorial methods, which are simpler approaches and do not easily capture certain features, such as stochastic dependence and imperfect fault coverage. Therefore, in the context of CONNECT, we consider as evaluation techniques:

- Stochastic model checking, which is a formal verification technique for the analysis of stochastic systems. It is based on the construction of a probabilistic model from a precise, high-level description of a system's behaviour.
- State-based stochastic methods, which use state-space mathematical models expressed with probabilistic assumptions about time durations and transition behaviours. They allow explicit modelling of complex relationships (e.g., concerning failure and repair processes), and their transition structure encodes important sequencing information.

In this paper, these two approaches are applied to a CONNECT scenario to perform various dependability analyses, thus showing their complementarity in assessing dependability properties. First, both approaches are used to validate two basic dependability properties. Next, extra properties are checked by the appropriate approach, selected according with its ability to cope with the specific type of analysis. Indeed, the different formalisms and tools implied by the two methods allow: (i) on the one hand, to complement the analysis from the point of view of a number of aspects, such as level of abstraction/scalability/accuracy, for which the two approaches may show different abilities to cope with; and (ii) on the other hand, through the inner diversity, provide cross-validation to enhance confidence in the correctness of the analysis itself. The rest of the paper is structured as follows. Section 2 introduces the tools implementing the approaches, and properties supported by the tools. A CONNECT scenario, namely, a model of a distributed market scenario, is presented in Section 3, and analysed in Section 4. We conclude the paper in Section 5.

2 Analysis and Verification Tools

In this section, we present a brief description of PRISM and Möbius, which implement stochastic model checking and state-based stochastic methods respectively.

2.1 PRISM

PRISM [12] is a popular probabilistic model checker, which can handle discrete and continuous time Markov chain models (DTMCs and CTMCs), as well as Markov decision processes (MDPs). DTMC and MDP models may be verified against probabilistic temporal logic formulae given in terms of PCTL (Probabilistic Computational Tree Logic) [11,6], as well as cost/reward-based properties, and LTL (Linear Temporal Logic) formulae [19]. CTMCs may be verified against CSL (Continuous Stochastic Logic) [2,3] formulae. Both states and transitions in a system can be associated with rewards, which allow for the checking of both instantaneous and cumulative properties.

So far PRISM has been applied to numerous probabilistic models, such as network protocols, security protocols, randomised distributed algorithms, biological processes, etc.

Modelling formalism. As CONNECTed systems evolve in a manner where a system stays in a state for a certain period of time and then moves to a successor state, we model such systems using CTMCs because they preserve the memoryless property. For CTMCs, the memoryless property not only requires that the probability of firing a transition totally depends on the current state, but also asks the probability to be independent of the elapsed time so far. The only continuous probability distribution exhibiting this property is the exponential distribution, which associates a *rate* to each transition in CTMCs. The rate can be understood as the average number of times we can execute the transition per unit of time. The probability of executing a transition from the current state within t time units is $1 - e^{-\lambda \cdot t}$. The rates associated with all transitions in a CTMC can be stored in a *transition rate matrix* $\mathbf{R}$, where each entry represents a rate between a pair of states. A transition can only occur from state s to state s' if $\mathbf{R}(s, s') > 0$. If more than one transition can be executed in state s, the successor state is determined by the first transition being taken. Let S be the set of states in a CTMC. The amount of time for which the system stays in s before any transition occurs is governed by an exponential distribution with rate $E(s)$ such that $E(s) \stackrel{def}{=} \sum_{s' \in S} \mathbf{R}(s, s')$. The probability of going to successor state s' from state s is calculated as follows.

$$\mathbf{P}(s, s') = \begin{cases} \mathbf{R}(s, s')/E(s) & \text{if } E(s) \neq 0, \\ 1 & \text{if } E(s) = 0 \text{ and } s = s', \\ 0 & \text{otherwise.} \end{cases} \quad (1)$$

Properties of interest. Using formula (1), we can compute the probability of reaching a set of target states through all paths. *Steady-state* behaviour is another interesting property for CTMC models. The steady-state probability for a state s is the probability of being in s in the long run, which can be used to infer the percentage of time that the model spends in s in the long run.

In addition to path and steady-state probabilities, we consider two additional types of reward for instantaneous and cumulative rewards separately. Every

transition is associated with an instantaneous reward and every state has a cumulative reward. The former is the actual reward obtained when the system executes a transition, and the latter is the coefficient, at which the reward is computed in a state, for the amount of time spent in that state. We can define the expected reward of reaching a set of target states F through paths. The reward for a path that does not pass any target state is set to ∞. Thus, the expected reward of reaching a state in F from state s is finite if all non-zero probability paths starting from s pass a state in F.

2.2 Möbius

Möbius [8] is a popular software tool that provides a comprehensive framework for model-based dependability and performance evaluation of systems. The main features of the tool include: (i) multiple high-level modelling formalisms, including, among others, Stochastic Activity Networks (SANs) [20] and PEPA fault trees [10]; (ii) a hierarchical modelling paradigm, allowing one to build complex models by first specifying the behaviour of individual components and then by combining the components to create a model of the complete system; (iii) customised measures of system properties; (iv) distributed discrete-event simulation, to evaluate measures using efficient simulation algorithms to repeatedly execute the system and gather statistical results of the measures; (v) numerical solution techniques, to obtain exact solutions for Markov models.

Modelling formalism. We model the system with Stochastic Activity Networks (SANs). SANs are stochastic extensions of Petri Nets; they have a graphical representation and consist of four primitive objects: *places*, *activities*, *input gates* and *output gates*. Places in SANs have the same interpretation as in Petri Nets, i.e., they hold tokens. The number of tokens in a place is referred to as the marking of that place, and the marking of the SAN is the set of all place markings. There are two types of activities: instantaneous and timed. Timed activities represent actions that have a duration that impacts the performance of the modelled system, e.g., message transmission time, recovery time, time to fail. The duration of each timed activity is expressed via a time distribution function. Both instantaneous and timed activities may have *case probabilities*. Each case probability stands for a possible outcome of the activity, and can be used to model probabilistic aspects of the system, e.g., probability for a component to fail. Input gates control the enabling of activities, and output gates define the state change that will occur when an activity completes.

SAN models can be composed with *Join* and *Rep* operators. Join is used to compose two or more SANs. Rep is a special case of Join, and is used to construct a model consisting of a number of replicas of a SAN. Models in a composed system interact via *Place Sharing*. Place Sharing is a composition formalism based on the notion of sharing places via an equivalence relation.

Properties of interest. Properties of interest are specified with *reward functions*. Each reward function is a C++ function that specifies how to measure a

property on the basis of the marking of the SAN. There are two kinds of reward functions: *rate reward* and *impulse reward*. Rate rewards can be evaluated at any time instant. Impulse rewards are associated with specific activities and they can be evaluated only when the associated activity completes. Measurements can be conducted at specific time instants, over periods of time, or when the system reaches the steady state.

3 The Distributed Market Scenario

We consider a case study based on a distributed market, where consumers execute a discovery protocol to gather information on the products sold by merchants. The discovery phase is performed in two steps. In the first step, consumers interoperate with all merchants to gather a list of all available products. In the second step, consumers select a product type and continue to interoperate with a subset of merchants (those that sell the selected product type) to gather additional information on the product.

Since consumers and merchants have heterogeneous devices that execute different protocols, interoperability among them is obtained via a CONNECTor that bridges the functional mismatches between the protocols. Without loss of generality, we assume that all merchants have the same protocol $P1$, and that all consumers have the same protocol $P2$ ($P2 \neq P1$).

Figure 1 illustrates the LTSs (Labelled Transition Systems) for consumer, merchant, CONNECTor and CONNECTed system. For simplicity, in Figure 1(c), we assume that there are two merchants in the market, one of which sells the product requested by a consumer. A larger number of merchants is handled by the CONNECTor in the same way. In the first discovery step, the CONNECTor receives the consumer's request `rdgBrowse` and sends a message `mSearch` to all merchants. Each merchant responds with a message `resp`. Note that the CONNECTor can handle any order of responses from the merchants. The CONNECTor then sends back to the consumer a message `tupleListBrowse`. In the second step, the CONNECTor converts the consumer's request `rdgGetInfo` into two requests `httpGet` and `httpGetResp` to the merchant selling the product, and obtains the responses `soapReqGetInfo` and `soapRespInfo` respectively. In the end, the CONNECTor returns a response `tupleListInfo` to the consumer. The LTS for the composed system is illustrated in Figure 1(d). The complete description of the scenario for a CONNECTed system with one consumer and one merchant is presented in [13].

Basic dependability properties. In this paper, we consider two basic properties that will be analysed using both PRISM and Möbius. In Section 4, further properties will be analysed in order to extend the analysis in accordance with the capabilities of the approaches.

- *Coverage in the first step.* We are interested in the percentage of merchants that can give a response to the CONNECTor in a given time interval during the first step. This property is affected by the network characteristics in the

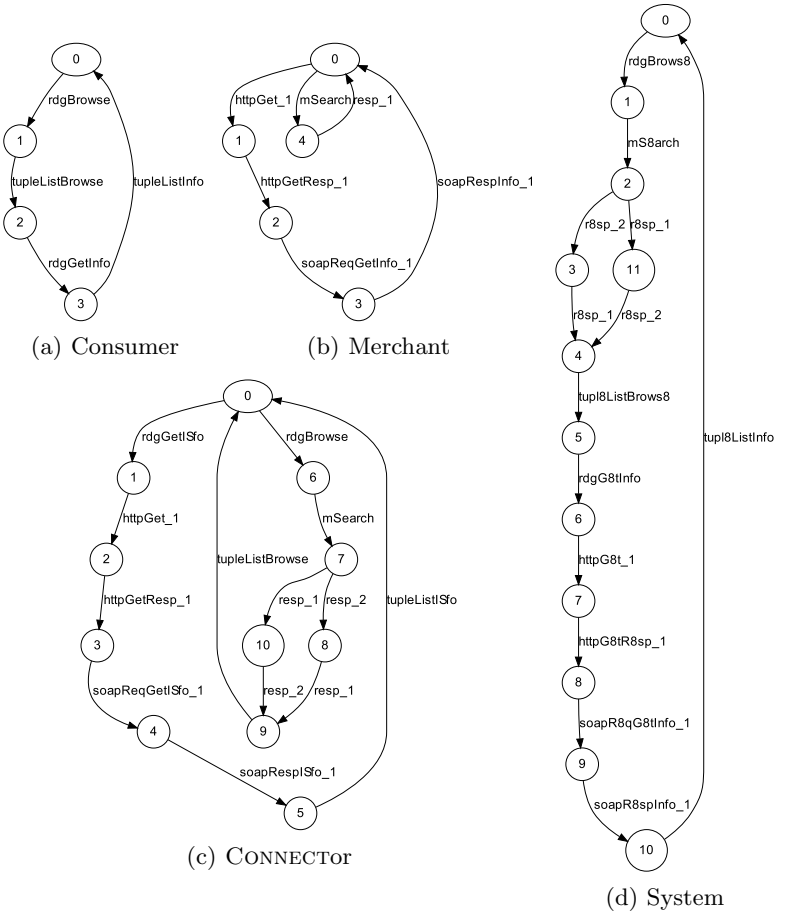


Fig. 1. LTS models for distributed market scenario

market, e.g., the number of consumers, the number of merchants, and the reliability of the channel. In the rest of the paper, we analyse the result for the case in which there is one consumer and three merchants. The transmission speed is modelled by different values of the rate associated with the transitions among consumer, CONNECTor, merchants.

– *Latency in the second step.* Latency is more interesting in the second step, as the communication between the CONNECTor and the merchants involves more message exchanges. Similarly to coverage, we assume there is a single consumer in addition to three merchants, all of which are selling the requested product. We measure the time spent by the consumer from when it starts to send `rdgGetInfo` to the arrival of `tupleListInfo`.

4 Dependability Analysis and Verification

In this section, we model the CONNECT scenario using PRISM and Möbius respectively, and perform dependability analysis on the models.

4.1 PRISM Models

The LTSs for the scenario in Figure 1 can be translated into the PRISM CTMC model in a straightforward manner. In detail, each component LTS in Figure 1(a)-1(b) is translated into a PRISM module in the following way. We define a variable in each module, whose domain is the set of states in the corresponding LTS and whose initial value is the initial state of the LTS. Each transition in the module has the same label as the corresponding one in the LTS. Since the LTSs do not contain information for rates, we deliberately assign rate $R1 = 1$ to all transitions between the consumer and the connector, and assign $R2$ (which may vary) to those between the connector and the merchants.

In order to check the basic properties specified in Section 3, we need to add the timeout mechanism to the model. Two timeouts T1 and T2 are introduced to model the maximum time for the first step and second step respectively. However, the deterministic delay in timeout breaks the basic rule of CTMCs: all delays in a CTMC model respect exponential distributions, and this makes the model difficult to verify. In this paper, we use an Erlang distribution to approximate a deterministic delay T by a sequence of transitions, each of which has an exponential distribution of rate k/T, where k is the number of transitions in the sequence. The accuracy of the approximation, as well as the verification time, increases as k increases. In the experiments, we choose k to be $T \times 10$, i.e., the rate in the Erlang distribution is 10, which is a reasonable trade-off between speed and accuracy.

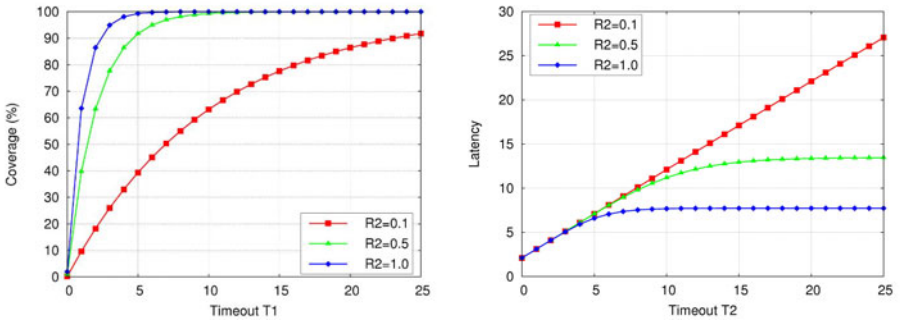
4.2 Stochastic Verification

In this section, we first show the verification results for the basic properties, and then discuss additional properties that can be verified using CSL. For each property, we construct a set of experiments by choosing different values for timeouts $T1$ and $T2$, and letting $R2$ range over values 0.1, 0.5 and 1.0 respectively. This way, we can illustrate the trend as $T1$ (resp. $T2$) increases.

Coverage. This property is specified by the following CSL reward formula on the reward structure *Coverage*:

$$\mathcal{R}_{=?}[C^{\leq T}], \tag{2}$$

where $C^{\leq T}$ represents the cumulative reward up to time bound T. The structure *Coverage* associates the real value m/n to states where, among the total number n of merchants, m merchants send back their response to the connector within $T1$ time units after they receive the request `mSearch`. We choose T to be $T1 +$



(a) Coverage (b) Latency

Fig. 2. Verification results

10 to take into account the time for transmitting message `rdgBrowse`. This formula calculates the expected cumulative reward within T time units, and the verification results are shown in Figure 2(a).

Latency. In the second step of the discovery, the connector triggers a timeout after $T2$ units of time when it receives message `rdgGetInfo` from the consumer. When the timeout occurs, the connector does not wait for pending responses from merchants, and returns `tupleListInfo` to the consumer. To verify this property, we use formula (2) on the reward structure *Latency*, which assigns $1/(T2/k) = 10$ to each transition used to approximate the timeout. The results are depicted in Figure 2(b).

Probability of receiving replies from all merchants in the second step. In addition to the basic properties, we are also interested in the probability of receiving responses from all merchants contacted within deadline $T2$ in the second discovery step. This property is formulated as follows:

$$\mathcal{P}_{=?}[F^{\leq T}(m = n)], \tag{3}$$

where $T = T2 + 2$, n is the total number of merchants contacted and m is the number of merchants that reply before the timeout. This formula computes the probability of all paths that can reach a state satisfying $m = n$ within T units.

In formula (3), $T = T2 + 2$ is a reasonable bound to take into account the transmission time for `rdgGetInfo` and `tupleListInfo`, given $R1 = 1$ for these two messages. However, it is still possible that it takes longer than 2 units of time to transmit these messages, which generates a small error in the experimental results. Steady state probability can be used to overcome this problem. If we ignore truncation errors, formula (4) gives an accurate value for the required probability at a cost of longer verification time.

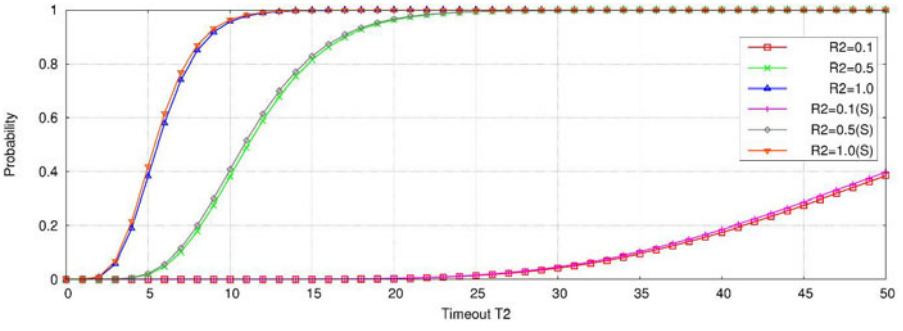


Fig. 3. Verification results for probability and steady-state probability

$$\mathcal{S}_{=?}[m = n] \tag{4}$$

In Figure 3, the curves labelled $R2 = 0.1$, $R2 = 0.5$ and $R2 = 1.0$ are probabilities computed using formula (3), and the others are computed using formula (4) for the same rates, respectively. Note that the accurate result for the coverage property can be computed by the steady-state reward formula:

$$\mathcal{R}_{=?}[\mathcal{S}] \tag{5}$$

on the corresponding reward structure. However, the difference between the values computed by formulae (2) and (5) in this example is negligible.

Maximum probability of receiving replies from all merchants in the second step. In the second discovery step, a certain number $n1$ of merchants, instead of the total number n of merchants, are contacted for the information of the request product. This number $n1$ may vary depending on many factors, such as the traffic in the network. The property considers the maximum probability of receiving responses from all $n1$ merchants in the following situation. At the

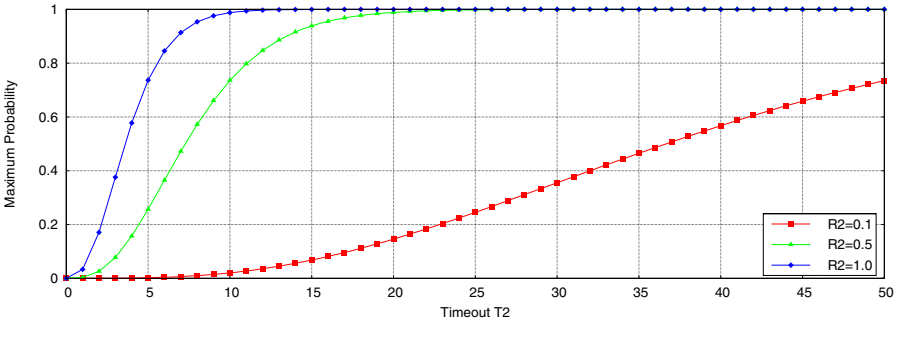


Fig. 4. Verification results for maximum probability

beginning of the second step, when the consumer is in state 2 in Figure 1(a), (formulated as $\texttt{Consumer} = 2$), the number of merchants that will be contacted is beyond a certain bound, i.e., $n1 \geq n \cdot a$ ($a \in (0, 1]$). Let $m1$ be the number of responses received before timeout occurs at $T2$. The property can be checked by formula (6):

$$\mathcal{P}_{=?}[F(m1 = n1)\{n1 \geq n \cdot a \wedge \texttt{Consumer} = 2\}\{\texttt{max}\}]. \tag{6}$$

The results for $a = \frac{1}{3}$ and $n = 3$ are illustrated in Figure 4.

4.3 SAN Models

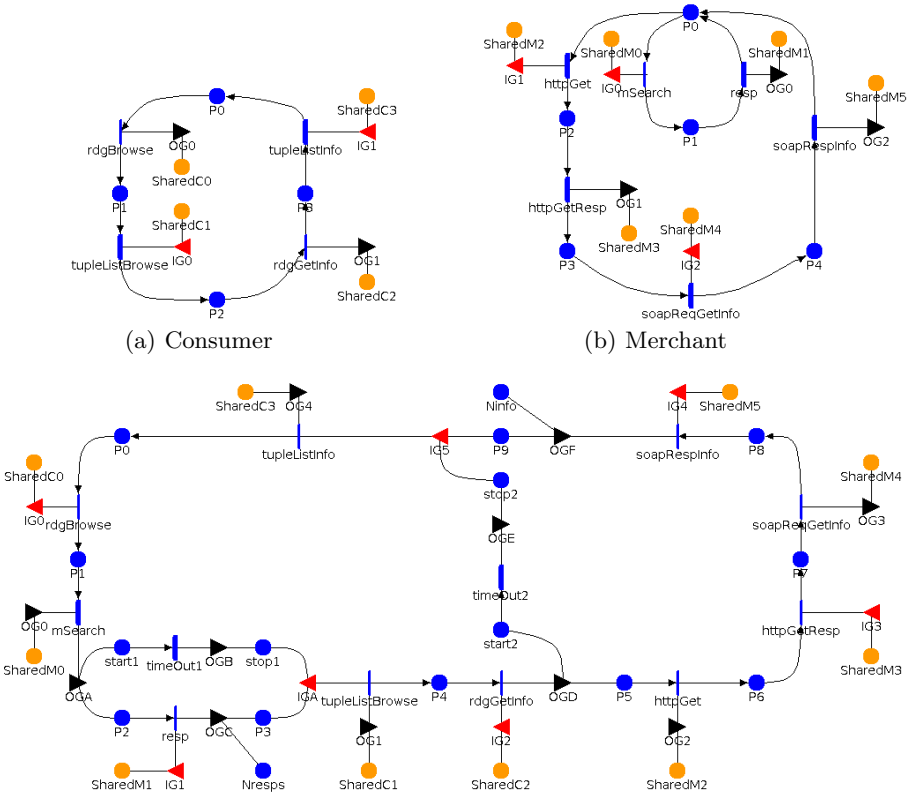
The SAN models of merchant, consumer and CONNECTor are shown in Figure 5. The model of the CONNECTed system is obtained by composing, via place sharing, the SAN models of consumer, CONNECTor and merchants (the SAN model of the merchants is obtained by replicating a merchant with the Rep operator). There is a shared place for each pair of activities that represent send/receive actions: send activities add tokens in the shared place, while receive activities remove tokens from the shared place and use the marking of the shared place as enabling condition. Note that, in general, a send activity may control $n > 1$ receive activities (e.g., in the case of a message with multicast/broadcast addresses); in this case, the send activity will add n tokens to the shared place to allow the simultaneous enabling of the receive activity of n receivers.

Timing aspects for send/receive actions are taken into account in the SAN models as follows: when n receive activities complete simultaneously after a send action completes, the receive activities are instantaneous and the send activity is timed; when n receive activities complete independently after a send action completes, the receive activities are timed and the send activity is instantaneous. Timeouts are modelled with timed activities that force the enabling of other activities.

In the following we describe in detail the behaviour of the model of CONNECTed system during the first step. In the description, we will use the prefixes `C`, `M[i]`, and `CON` to disambiguate the names of local places, activities and gates of consumer, merchants, and CONNECTor.

Initially, all places in the models have zero tokens, except `P0`, which contains one token in all models. The consumer starts the communication, because `C.rdgBrowse` is the only enabled activity. When `C.rdgBrowse` completes, one token is placed in `C.P1` and one token in `SharedC0`. At this point, `CON.rdgBrowse` is enabled. When `CON.rdgBrowse` completes, one token is moved from `SharedC0` to `CON.P1`, and `CON.mSearch` becomes enabled.

When `CON.mSearch` completes, the marking changes as follows: n tokens are placed in `SharedM0`, because n merchants must be involved in the communication; n tokens are placed in `CON.P2`, because the CONNECTor must wait for one `resp` from each merchant; one token is placed in `CON.start1`, because the CONNECTor has a timeout on the maximum waiting time.



(a) Consumer (b) Merchant

(c) CONNECTor

Fig. 5. SAN models

Each token in `SharedM0` enables the instantaneous activity `M[i].mSearch` of a merchant. All such activities complete immediately[1] and enable `M[i].resp`. Whenever a `M[i].resp` completes, a token is placed in `SharedM1` to enable the instantaneous activity `CON.resp` of the CONNECTor, which places a token in `CON.P3` and `CON.Nresps`. The number of tokens in `CON.Nresps` represents the number of merchants that will participate to the interactions during the second step. This behaviour continues until `CON.tupleListBrowse` becomes enabled, i.e., either when `CON.timeOut1` completes (one token is placed in `CON.stop1`), or n responses are received from the merchants (n tokens are present in `CON.P3` and `CON.Nresps`).

4.4 State-Based Stochastic Analysis

In this section, we present the analysis performed with Möbius: first, we cross-validate the results obtained by PRISM for coverage and latency; second, we

[1] When instantaneous activities and timed activities are enabled at the same time, all instantaneous activities complete first.

scale up to large systems with hundreds of merchants; third, we refine the SAN models to take into account some real-world aspects that have an impact on coverage and latency, such as traffic patterns and communication failures.

Cross validation. The reward functions are expressed as follows.

Coverage. This property is specified by accumulating over time the following impulse reward on `CON.resp` (`NMerchants` is a parameter of the composed model, and holds the number of merchants):

```
double coverage() {
   if ( CON->start1->Mark() > 0 ) { return 1.0/NMerchants; }
   return 0;
}
```

Latency. This property is specified by accumulating over time the following rate reward function:

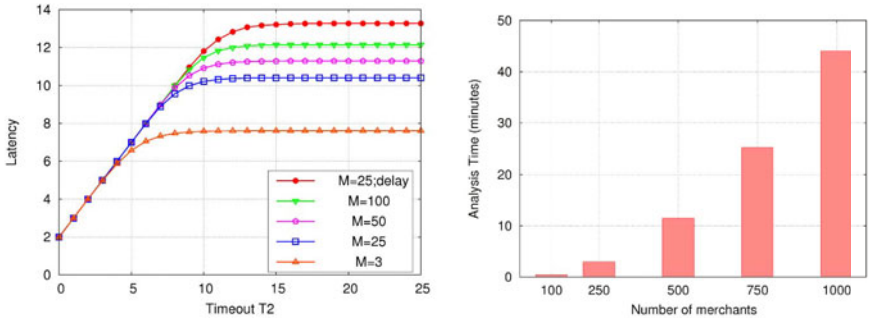
```
double latency() {
   if ( CON->P4->Mark() > 0 || CON->start2->Mark() > 0
        || CON->stop2->Mark() > 0) { return 1; }
   return 0;
}
```

We were able to successfully reproduce with Möbius the verification results of PRISM. We used simulation, and the relative difference between the average results was always below 2%.

Scalability of the models. CONNECTed systems may include an arbitrary large number of networked systems. Therefore, we investigated the scalability of the SAN model of the CONNECTed system by analysing large networks. The developed SAN model of the CONNECTed system is parametric with respect to the number of merchants: networks with different number of nodes can be modelled by changing only one model parameter.

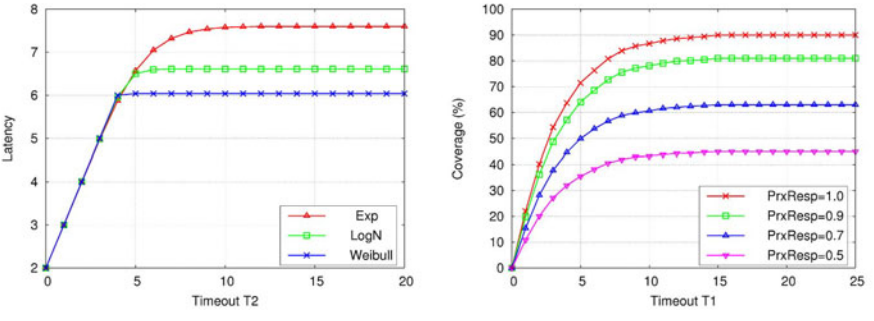
We successfully assessed coverage and latency for scenarios with hundreds of merchants. Figure 6(a) shows the analysis results for latency in scenarios with at most 100 merchants. The number of batches needed to reach a confidence level of 95% and a confidence interval of 10% for the considered models was always below $10K$, because the models are relatively simple. Figure 6(b) reports the average time to complete $10K$ simulation batches for different number of merchants on a system with a 2.8GHz Intel Core2 Quad processor.

Latency for different traffic patterns. CONNECTed systems are expected to be a mix of heterogeneous user applications, each of which may have different characteristics and requirements. Currently, there is no single traffic distribution that can efficiently capture the traffic characteristics of all types of networks under every possible situation. A large number of empirical studies have shown that network traffic is self-similar and that it generally exhibits multiple time-scale



(a) Latency (b) Analysis Time

Fig. 6. Latency and time required for the analysis for different system size



(a) Latency for different traffic patterns (b) Coverage for different P_{rxResp}

Fig. 7. Latency and coverage in different settings

behaviour [16]. These aspects can be modelled with subexponential distributions, such as Weibull and Lognormal.

We investigated the effect of different subexponential distributions on latency by changing the probability distribution function of the timed activities. For a fair comparison, we have chosen distribution parameters that allow the same mean value in all cases. The analysis results are shown in Figure 7(a). We can notice that different traffic patterns lead to different latency profiles.

Coverage in the case of failures. Communication in the real-world can be subject to failures. Therefore, failure modes need to be accounted for when setting up the system model. Failure modes can pertain the value domain (e.g., wrong output), and/or the time domain (e.g., omission). In this section, we assess coverage in the case of omission failure of the multicast search received by the merchants (`M[i].mSearch`) and omission failure of the responses sent

by the merchants (`M[i].resp`). Figure 7(b) shows the coverage profiles for different probability P_{rxResp} of failures of `resp`; in the figure, the probability of failure of `M[i].mSearch` depends on the timeout value and is derived from the analysis results reported in [17]; e.g., for $T1 = 10$, the probability of failure of `M[i].mSearch` is 0.87.

5 Conclusions

We have shown two approaches to analyse dependability properties for a CONNECT scenario. As can be seen in the previous section, the experimental results produced by one approach match those by the other approach for a significant range of properties. Each approach has its own advantages regarding modelling capability, specification of properties, scalability, etc. For example, bounded until formula $\phi_1 \mathcal{U}^I \phi_2$, steady state formula $\mathcal{S}_{\bowtie p}[\phi]$ and more complex CSL formulae can be verified in PRISM without manually augmenting models, while Möbius can deal with larger sized models and can mix exponential distributions with other distributions.

During the case study, we also found that the cross validation was particularly useful to improve the confidence in the correctness of the models. For example, by analysing the mismatches between results produced by PRISM and Möbius on the common properties, we were able to remove subtle non-deterministic behaviours that were erroneously present in the models, and eliminated the mismatches.

In the future, we are planning to apply both approaches to a more complex case study, and explore further the cross-fertilisation capabilities of PRISM and Möbius. In particular, we would like to exploit the assume-guarantee reasoning method [15,9] implemented in PRISM to analyse large models. We were unable to apply this here because the assume-guarantee approach has only been developed for Markov decision processes and safety properties at present. In addition, we are also interested in speeding up our approaches via incremental verification and analysis, and developing online techniques to provide support for on-the-fly CONNECTor synthesis [5], such as those based on [7].

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. Aziz, A., Sanwal, K., Singhal, V., Brayton, R.: Verifying continuous time Markov chains. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 269–276. Springer, Heidelberg (1996)
3. Baier, C., Katoen, J.-P., Hermanns, H.: Approximate symbolic model checking of continuous-time Markov chains. In: Baeten, J.C.M., Mauw, S. (eds.) CONCUR 1999. LNCS, vol. 1664, pp. 146–161. Springer, Heidelberg (1999)
4. Bertolino, A., Di Giandomenico, F., Di Marco, A., Issarny, V., Martinelli, F., Masci, P., Matteucci, I., Saadi, R., Sabetta, A.: Dependability in dynamic, evolving and heterogeneous systems: the connect approach. In: Proc. 2nd International Workshop on Software Engineering for Resilient Systems, SERENE2010 (2010)

5. Bertolino, A., Inverardi, P., Issarny, V., Sabetta, A., Spalazzese, R.: On-the-fly interoperability though automated mediator synthesis and monitoring. In: ISoLA 2010, Part II. LNCS, vol. 6416, pp. 251–262. Springer, Heidelberg (2010)
6. Bianco, A., de Alfaro, L.: Model checking of probabilistic and nondeterministic systems. In: Thiagarajan, P.S. (ed.) FSTTCS 1995. LNCS, vol. 1026. Springer, Heidelberg (1995)
7. Calinescu, R., Grunske, L., Kwiatkowska, M., Mirandola, R., Tamburrelli, G.: Dynamic QoS management and optimisation in service-based systems. IEEE Transaction on Software Engineering (to appear)
8. Clark, G., Courtney, T., Daly, D., Deavours, D.D., Derisavi, S., Doyle, J.M., Sanders, W.H., Webster, P.G.: The Möbius modeling tool. In: 9th Int. Workshop on Petri Nets and Performance Models, pp. 241–250. Aachen, Germany, Los Alamitos (September 2001)
9. Feng, L., Kwiatkowska, M., Parker, D.: Compositional verification of probabilistic systems using learning. In: Proc. 7th International Conference on Quantitative Evaluation of Systems (QEST 2010). IEEE CS Press, Los Alamitos (2010)
10. Gulati, R., Dugan, J.B.: A modular approach for analyzing static and dynamic fault trees. In: Annual Reliability and Maintainability Symposium, pp. 57–63. IEEE Computer Society Press, Los Alamitos (1997)
11. Hansson, H., Jonsson, B.: A logic for reasoning about time and reliability. Formal Aspects of Computing 6(5), 512–535 (1994)
12. Hinton, A., Kwiatkowska, M., Norman, G., Parker, D.: PRISM: A tool for automatic verification of probabilistic systems. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, Springer, Heidelberg (2006)
13. Inverardi, P., Issarny, V., Spalazzese, R.: A theory of mediators for eternal CONNECTors. In: ISoLA 2010, Part II, 2010. LNCS, vol. 6416, pp. 236–250. Springer, Heidelberg (2010)
14. Issarny, V., Steffen, B., Jonsson, B., Blair, G., Grace, P., Kwiatkowska, M., Calinescu, R., Inverardi, P., Tivoli, M., Bertolino, A., Sabetta, A.: Connect Challenges: Towards Emergent connectors for Eternal Networked Systems. In: 14th IEEE International Conference on Engineering of Complex Computer Systems, pp. 154–161. IEEE Computer Society, Los Alamitos (2009)
15. Kwiatkowska, M., Norman, G., Parker, D., Qu, H.: Assume-guarantee verification for probabilistic systems. In: Esparza, J., Majumdar, R. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. LNCS, vol. 6015, pp. 23–37. Springer, Heidelberg (2010)
16. Leland, W.E., Taqqu, M.S., Willinger, W., Wilson, D.V.: On the self-similar nature of ethernet traffic (extended version). IEEE/ACM Transactions on Networking 2(1), 1–15 (1994)
17. Masci, P., Chiaradonna, S., Di Giandomenico, F.: Dependability analysis of diffusion protocols in wireless networks with heterogeneous node capabilities. In: 8th European Dependable Computing Conference (EDCC2010), pp. 145–154. IEEE Computer Society, Los Alamitos (2010)
18. Nicol, D.M., Sanders, W.H., Trivedi, K.S.: Model-based evaluation: from dependability to security. IEEE Transactions on Dependable and Secure Computing 1, 48–65 (2004)
19. Pnueli, A.: The temporal logic of programs. In: Proc. 18th Annual Symposium on Foundations of Computer Science (FOCS 1977), pp. 46–57. IEEE Computer Society Press, Los Alamitos (1977)
20. Sanders, W.H., Meyer, J.F.: Stochastic Activity Networks: formal definitions and concepts, pp. 315–343 (2002)

Towards a Connector Algebra*

Marco Autili[1], Chris Chilton[2], Paola Inverardi[1],
Marta Kwiatkowska[2], and Massimo Tivoli[1]

[1] Dipartimento di Informatica - Università degli Studi di L'Aquila, Italy
{marco.autili,paola.inverardi,massimo.tivoli}@di.univaq.it
[2] Oxford University Computing Laboratory, Parks Road, Oxford, OX1 3QD, UK
{chris.chilton,marta.kwiatkowska}@comlab.ox.ac.uk

Abstract. Interoperability of heterogeneous networked systems has yet to reach the maturity required by ubiquitous computing due to the technology-dependent nature of solutions. The CONNECT Integrated Project attempts to develop a novel network infrastructure to allow heterogeneous networked systems to freely communicate with one another by synthesising the required connectors on-the-fly. A key objective of CONNECT is to build a comprehensive theory of composable connectors, by devising an algebra for rigorously characterising complex interaction protocols in order to support automated reasoning. With this aim in mind, we formalise a high-level algebra for reasoning about protocol mismatches. Basic mismatches can be solved by suitably defined primitives, while complex mismatches can be settled by composition operators that build connectors out of simpler ones. The semantics of the algebra is given in terms of *Interface Automata*, and an example in the domain of instant messaging is used to illustrate how the algebra can characterise the interaction behaviour of a connector for mediating protocols.

1 Introduction

Ubiquitous computing is an emerging paradigm that is rapidly changing the way we use technology to perform everyday tasks. The widespread availability of digital systems, together with the introduction of new communication infrastructures, make it possible to run and interact with software systems on a variety of networked devices. However, computing and networking technologies have yet to reach the maturity required by ubiquitous computing since technology-dependent limitations reduce the effectiveness of integrating and composing heterogeneous networked systems.

The CONNECT Integrated Project[1] attempts to develop a novel networking infrastructure to allow heterogeneous networked systems to freely communicate with one another. This would be achieved by the synthesis of emergent connectors on-the-fly. Towards this aim, a key objective of CONNECT is to build

* This work is partly supported by the CONNECT European Project No 231167, and EPSRC project EP/D076625/2.
[1] http://connect-forever.eu/

T. Margaria and B. Steffen (Eds.): ISoLA 2010, Part II, LNCS 6416, pp. 278–292, 2010.

a comprehensive theory of composable connectors, by devising an algebra that can model complex interaction behaviours with respect to both functional and non-functional properties. The algebra will serve as a baseline for automated reasoning and learning about system interaction behaviours, in addition to automated synthesis, matching, refinement, composition, evolution, and (possibly partial) re-use of connectors. This also concerns finding an adequate formalism to express and quantify, for each connector, the desired Quality of Service levels for end-to-end properties of the networked systems.

The comprehensive characterisation of such a connector algebra is our long-term goal. In this paper, as a starting point for developing such an algebra, we focus only on the functional behaviour of connectors that act as *protocol mediators*. Consequently, our algebra will characterise the behavioural mismatches that occur during interactions among heterogeneous networked systems. Quantitative aspects of networked systems and connectors are not considered hereafter; that is left as future work.

As discussed in [14,19] (and references therein), a possible approach to protocol mediation involves the categorisation of recurring protocol mismatches that must be solved by means of *mediator patterns*. For each type of mismatch, a pattern can be defined as a solution to the interaction incompatibility. Clearly, a catalogue of such problems and their related solutions would not solve all possible mismatches, but combining multiple sub-solutions should facilitate their solution.

Inspired by the set of basic mediator patterns described in [19], this paper presents a high-level algebra that reasons about protocol mismatches. Solutions to basic mismatches are modelled as primitives of the algebra, while complex mismatches can be solved by combining suitable primitives in a variety of ways. The semantics of the algebra is given in terms of *Interface Automata* (IA) of de Alfaro and Henzinger [11]. Thus, composition of terms in the algebra reduces to composition of the underlying IA. Some of these compositions are already defined on IA, but we will also present specific compositions that we conjecture are necessary for a meaningful connector algebra.

Our choice of using IA for the semantics of the algebra is heavily influenced by a previous survey of connector notations we conducted as part of CONNECT [1]. In that report we surveyed a number of formalisms against eight dimensions deemed to be of particular interest to the project. These were compositionality, incrementality, scalability, compositional reasoning, reusability, evolution, ability to express and reason about non-functional properties, and the existence of a specialised notation supported by automated tools for architectural analysis.

The choice of formalisms was decided to give a thorough coverage of the space of connectors. To give an indication: some formalisms were control-oriented [2], while others were data-flow based [7]; some supported hierarchy [9], whereas others provided mobility [18]. In the spirit of CONNECT, we also surveyed quantitative extensions of [7]; these included an extension involving discrete probabilities with non-determinism [6], a stochastic extension [4], and an extension with Quality of Service attributes [5]. Beyond the surveyed approaches, we also

investigated a number of other formalisms, but space limitations prevents us from elaborating upon them here.

Summarising the results of our survey and investigations led us to the opinion that IA are the most suitable formalism for modelling connectors. This was based on the fact that IA can be extended to support reasoning on non-functional properties, together with compositionality results implying reuse and evolution of connectors. IA are by no means complete in satisfying the properties we require of a connector algebra, but they do give us a good starting point and seem extensible enough to gain a good coverage of the dimensions of interest.

The remainder of this paper is organised as follows. Section 2 recalls background notions concerning IA, and offers justification for choosing these devices. Section 3 describes a scenario that we will use to demonstrate the effectiveness of our algebra, with models given in terms of IA. Following on, Section 4 introduces the algebra in a formal way and concludes by relating the algebra to the case study presented in Section 3. Finally, Section 5 summarises our work and discusses possible future research directions.

2 Semantics for Connectors

As clarified in Section 1, it is our intention to ascribe semantics to our algebra in terms of *Interface Automata* (IA) [11]. As we shall see later on, IA do not give us all of the desired functionality and properties that we require to interpret our algebra, but they do give us a good starting point.

IA may be seen as finite state machines whose actions are partitioned into input and output sets. At the syntactic level this classification has no significance, but examination of the semantics reveals a notion of communication well suited to components interacting in a heterogeneous environment.

Definition 1. *An Interface Automaton is a tuple* $(V, v_0, \mathcal{A}_I, \mathcal{A}_O, \rightarrow)$*, where:*

- V *is a set of states*
- $v_0 \in V$ *is the designated initial state*
- $\mathcal{A} = \mathcal{A}_I \cup \mathcal{A}_O$ *is the set of actions.* $\mathcal{A}_I$ *and* $\mathcal{A}_O$ *are disjoint sets referred to as the input actions and output actions respectively.*
- $\rightarrow : V \times \mathcal{A} \rightharpoonup V$ *is the transition (partial) function.*

For brevity, we write $v \xrightarrow{a} v'$ iff $\rightarrow (v, a) = v'$. The partial transition function ensures that the automaton is loosely deterministic (there is at most one successor for each state-action pair). This contrasts with the I/O automaton model of Lynch and Tuttle [16], where every state must be fully input-enabled.

The automaton behaves in a manner similar to that for finite state machines; it starts in the initial state and evolves over time according to its transition function. However, special consideration must be given to the transition types. Following the finite state case, a transition labelled by an input action may only be picked when the environment is offering that action. Output actions are non-blocking, so may be taken at any time if they are enabled in the current state.

Since outputs are non-blocking, it follows that the environment must be willing to accept any action it is offered.

Until now, IA have appeared as marginally generalised I/O automata. Their overarching power becomes apparent when we consider the composition of multiple IA, and observe how they interact with the environment.

From hereon let $\mathcal{C} = (V^C, v_0^C, \mathcal{A}_I^C, \mathcal{A}_O^C, \rightarrow_C)$ and $\mathcal{D} = (V^D, v_0^D, \mathcal{A}_I^D, \mathcal{A}_O^D, \rightarrow_D)$ be two IA. $\mathcal{C}$ and $\mathcal{D}$ may only be composed if their action sets are compatible with each other. Consequently, not every pair of IA can be composed.

Definition 2. *IA $\mathcal{C}$ and $\mathcal{D}$ are said to be* composable *just if $\mathcal{A}_O^C \cap \mathcal{A}_O^D = \emptyset$. Outputs must be disjoint to prevent synchronisation.*

To define the (parallel) composition on composable IA we begin by constructing the product of the automata. Unfortunately, this can introduce a number of illegal states, whereby one of the automata is willing to offer an output action in the common alphabet, but the second is not able to offer the corresponding input action. This is a consequence of not requiring IA to be fully input-enabled.

Definition 3. *The product of $\mathcal{C}$ and $\mathcal{D}$ is an interface automaton $\mathcal{C} \otimes \mathcal{D} = (V^C \times V^D, (v_0^C, v_0^D), \mathcal{A}_I, \mathcal{A}_O, \rightarrow)$, where:*

- $\mathcal{A}_I = (\mathcal{A}_I^C \cup \mathcal{A}_I^D) \setminus ((\mathcal{A}_I^C \cap \mathcal{A}_O^D) \cup (\mathcal{A}_O^C \cap \mathcal{A}_I^D))$
- $\mathcal{A}_O = \mathcal{A}_O^C \cup \mathcal{A}_O^D$
- *The transition function is given by*

$$(s,t) \xrightarrow{a} (s',t') \Leftrightarrow \begin{cases} s \xrightarrow{a}_C s' \wedge t \xrightarrow{a}_D t' & \text{if } a \in \mathcal{A}^C \cap \mathcal{A}^D \\ s \xrightarrow{a}_C s' \wedge t = t' & \text{if } a \in \mathcal{A}^C \setminus \mathcal{A}^D \\ t \xrightarrow{a}_D t' \wedge s = s' & \text{if } a \in \mathcal{A}^D \setminus \mathcal{A}^C. \end{cases}$$

We must now identify the illegal states in the product $\mathcal{C} \otimes \mathcal{D}$, denoted by $Illegal(\mathcal{C}, \mathcal{D})$. The kernel of the illegal states is taken to be the set of states (p, q) for which there is some $a \in (\mathcal{A}_I^C \cap \mathcal{A}_O^D) \cup (\mathcal{A}_O^C \cap \mathcal{A}_I^D)$ such that one of p and q can make an a-labelled output transition, but the other cannot match it with the corresponding input transition. The illegal set is then taken to be all those states in the kernel, plus those that can reach a state in the kernel by a sequence of transitions labelled by output actions.

Definition 4. *The* composition *of two composable IA $\mathcal{C}$ and $\mathcal{D}$, written as $\mathcal{C} \parallel \mathcal{D}$, is defined to be $\mathcal{C} \otimes \mathcal{D}$ after pruning all states in $Illegal(\mathcal{C}, \mathcal{D})$, providing the initial state (v_0^C, v_0^D) is contained within the remaining automaton. Otherwise, the composition is undefined.*

In the words of de Alfaro and Henzinger [11], the pruning yields an optimistic notion of composition. The pruning is equivalent to making the assumption that the environment will not issue inputs that can lead to illegal states. Consequently, composed IA place an assumption on the environment about what inputs will be issued and when. This is in stark contrast to I/O automata where the composition must handle any input the environment offers.

From a system designer's point-of-view, we are interested in modelling concrete connectors by means of a term in our algebra. We could certainly develop an algebra that addresses this sole goal, however we are looking for something considerably more high-level. To that extent, and conflating the desiderata from Section 1, we believe that our algebra should also support abstractions of connectors, or more precisely specifications.

IA have an explicit correspondence with specification theories, which is why we are using them to express the semantics of our algebra. In [11], de Alfaro and Henzinger define a refinement relation $\sqsubseteq$[2] on IA in terms of alternating simulation[3] [3]. Consequently, we have a test for when a connector can be safely substituted with another. Since refinement on IA is a congruence with respect to composition (see next theorem), substitution can be performed compositionally.

Theorem 1. *For suitable restrictions on the composability of $\mathcal{C}$, $\mathcal{C}'$ and $\mathcal{D}$, if $\mathcal{C} \parallel \mathcal{D}$ is defined and $\mathcal{C} \sqsubseteq \mathcal{C}'$, then $\mathcal{C}' \parallel \mathcal{D}$ is defined and $\mathcal{C} \parallel \mathcal{D} \sqsubseteq \mathcal{C}' \parallel \mathcal{D}$.*

Besides parallel composition, a conjunctive operator $\wedge$ can be defined on IA. $\mathcal{C} \wedge \mathcal{D}$ yields the least specified interface automaton that refines both $\mathcal{C}$ and $\mathcal{D}$. This is of direct use in defining a connector that must satisfy multiple specifications. More verbosely, this means we can build up specifications in a distributed and compositional manner. This supports the *separation of concerns* principle, and again allows for compositional development.

IA can be synthesised in a specification theory by means of a quotienting operator $\backslash$, as described in [8]. Given IA $\mathcal{D}$ and $\mathcal{E}$, $\mathcal{E} \backslash \mathcal{D}$ yields the least specified interface automaton $\mathcal{C}$ such that $\mathcal{E} \sqsubseteq \mathcal{C} \parallel \mathcal{D}$. Thus, given a specification of what a connector should do, together with an implementation of a connector that implements part of that behaviour, we can synthesise a connector that when composed with the partial connector fulfils the desired behaviour. The advantages of such a feature are self-evident.

Applicability to notions of specification was a key justification for adopting IA, yet another essential reason relates to their extensibility. We would eventually like to develop an algebra that can handle a number of quantitative extensions, such as time and probability. A timed-extension of I/O automata has already been developed [10], and the model completely supports all of the specification constructs and relations we have mentioned. Furthermore, there is a probabilistic extension of I/O automata [20], although this is not given in terms of a specification theory. Nevertheless, recent work has augmented interactive Markov chains with specification notions [21], with a bridge between these and probabilistic I/O automata seeming highly plausible.

Since IA and I/O automata are closely related to each other, we would like to combine the aforementioned quantitative extensions with the optimistic composition techniques of IA. Accordingly, IA appear to support many of the features

[2] $P \sqsubseteq Q$ if, and only if, P is refined by Q. Refinement tends to be one of the most prevalent relations in any specification theory.

[3] The simulation must be alternating as the inputs and outputs of a refining automaton must be related contravariantly to those of the original.

we would like of a connector algebra. Section 4 will demonstrate just how effective they are when we assign semantics to the algebra.

3 Case Study

To illustrate in Section 4 the efficacy with which our algebra can model and reason about protocol mediators, we present a simple yet challenging example of universal instant messaging inspired by [14].

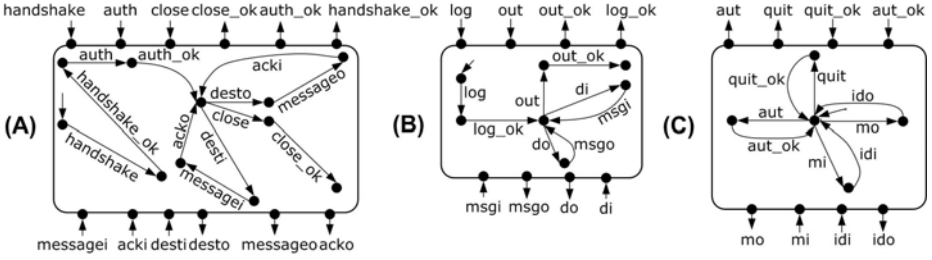


Fig. 1. (A) MSN messaging service. (B) XMPP messaging service. (C) CFring client.

Fring[4] is an instant messaging program that allows one to exchange text messages between a predefined set of heterogeneous messaging services. At present, the service supports connection to the *MSN Messenger* and *XMPP Messenger* services, amongst many others, with the collection of supported services being static. This contrasts with the evolving world of Connect, where messaging services to be bridged should not be known *a priori*. We therefore propose a generalisation of Fring, let it be called *CFring*, where connectors between unknown messaging services are generated on-the-fly. This generalisation will allow us to express the full power of the algebra.

As the interface automaton in Figure 1.C shows, the CFring service provides only core functionalities for "abstract" authentication and message exchange. In particular, when receiving (resp., sending) a message *mi* (resp., *mo*), CFring expects to receive (resp., send) the identity *idi* (resp., *ido*) of the sender (resp., receiver) as well.

The behavioural models of MSN and XMPP, which are unknown to CFring, are expressed as IA in Figures 1.A and 1.B. In contrast to the behavioural model for CFring, both MSN and XMPP when receiving (resp., sending) a message expect to have provided (resp., provide) the identity of the sender (resp., receiver) first. Furthermore, unlike the others, MSN expects to receive (resp., send) an acknowledgement for the message sending (resp., receiving).

[4] http://www.fring.com

It is obvious that the MSN and XMPP services should both be able to interoperate with CFring, since they amount to supporting authentication and then message exchange. This requires "specialising" the CFring communication protocol in order to mediate the communication between the other messaging services. Note that this is far from trivial, especially if one wishes to rigorously characterise the achieved interoperability (e.g., for supporting automated reasoning, detecting possible mismatches, etc.). Nevertheless, in Section 4 we realise such a connector that mediates the communication between CFring and MSN/XMPP as a term of our algebra.

4 Towards a Connector Algebra: Primitives and Operators

Section 1 briefly mentioned a number of existing connector formalisms that we had surveyed in [1]. The formalisms vary quite considerably; some support hierarchical development, whereas others have resolute granularity. Some support mobility, while others assume a fairly static environment. In essence, the formalisms have niche environments where they work well, while outside their haven of assumptions the quality of modelling is often variable.

In CONNECT, we are interested in generating connectors on-the-fly to bridge communication inconsistencies at both the application and middleware layers. For the purposes of this paper, we are concerned with exchanging structured messages between components, rather than worrying about, say, data transfer at the transport level.

In [19], the authors attempt to characterise mismatches between functionally equivalent yet behaviourally different protocols that wish to communicate. For each type of communication discrepancy they provide a mediating connector that can handle and resolve the mismatch. This is a high-level approach to analysing and addressing interoperability issues.

Following this insight, it is our intention to develop a high-level algebra for reasoning about mismatches. We shall model each basic mismatch solution as a primitive of our algebra, with semantics given in terms of IA. Complex mismatches can be decomposed into combinations of basic ones, and so an algebraic connector for a complex mismatch can be obtained by composing primitive connectors. For most cases, composition of terms in the algebra will reduce to composition of the underlying IA as described in Section 2. However, as we shall see, we will also require our own specific operators on connectors.

Components wishing to communicate with each other are modelled by arbitrary IA, which we assume have disjoint action sets[5]. We will treat the action sets associated with IA as sets of message ports that can send (resp., receive) a signal depending on whether they are an output (resp., input) port. Hence, at this stage we do not allow for the exchange of data over a domain.

[5] Under the aegis of CONNECT, equivalence of actions is assumed to be specified in an ontology. This allows us to assume disjoint component actions.

From hereon let $\mathcal{A}$ be a global set of message ports. The primitives of the connector algebra $\mathcal{AP}(\mathcal{A})$ corresponding to the mismatches in [19] are described below.

1. **Extra send.** This first mismatch considers a component that generates a redundant message a. Such a mismatch may be resolved by introducing a consumer that swallows the superfluous message. We model this by a parameterised primitive $Cons(a)$.
2. **Missing send.** This mismatch describes the case in which a component expects a message a that is not sent by another component. A mismatch of this type may be resolved by introducing a producer that generates the required message. This may be modelled by a parameterised primitive $Prod(a)$.
3. **Signature mismatch.** There are occasions when a message to be exchanged between two components is functionally compatible yet syntactically inconsistent. In the case of CONNECT, the functional equivalence of the messages a and b is assumed to be specified in an ontology. Such a mismatch may be resolved by means of a translating primitive $Trans(a, b)$ that accepts message a as input and produces message b as output.
4. **Split message mismatch.** A component may expect to receive a message a as a sequence of fragments of a. If message a can be decomposed into $a_1, \ldots, a_n$, then the mismatch may be resolved with a primitive $Split(a, [a_1, \ldots, a_n])$ which accepts message a as input and offers $a_1, \ldots, a_n$ as output in that order.
5. **Merge message mismatch.** Similar to the previous case, some components expect to receive a single message a in place of a fragmented version of a. If messages $a_1, \ldots, a_n$ can be composed into a, then the mismatch may be resolved with a primitive $Merge([a_1, \ldots, a_n], a)$ which accepts messages $a_1, \ldots, a_n$ as input in that order, and generates a as output.
6. **Ordering mismatch.** A component can expect to receive messages in an order different from the order used by the sending component. The mismatch can be resolved by introducing an ordering primitive $Order([a_1, \ldots, a_n], \pi, [a'_1, \ldots, a'_n])$, where π is a permutation of $\{1, \ldots, n\}$. The primitive accepts inputs from one component in the order $a_1, \ldots, a_n$, and produces outputs for the other in the order $a'_{\pi(1)}, \ldots, a'_{\pi(n)}$. Note that port a_i is related to port a'_i.

Besides the mismatch primitives above, a further primitive is required to force the algebra to work in a sensible way. The primitive does not perform any interactions, so is fittingly called $NoOp$. Equipped with all of the basic primitives, a term s of the algebra is given by the following grammar:

$$s ::= s \odot s \mid s + s \mid s \wedge s \mid s \setminus s \mid s^{\perp} \mid (s) \mid p$$

$$p ::= NoOp \mid Cons(a) \mid Prod(a) \mid Trans(a, b) \mid Split(a, [a_1, \ldots, a_n]) \mid Merge([a_1, \ldots, a_n], a) \mid Order([a_1, \ldots, a_n], \pi, [a'_1, \ldots, a'_n])$$

where $a, a_i, a'_i, b \in \mathcal{A}$ and π is a permutation of $\{1, \ldots, n\}$. The symbols $\odot$, $+$, $\wedge$, and $\setminus$ are binary operators called *plugging*, *alternation*, *conjunction* and *quotienting* respectively, and $^{\perp}$ is a unary operator called *inversion*.

The semantics of the algebra $\mathcal{AP}(\mathcal{A})$ is given in terms of a function $[\![\cdot]\!] : \mathcal{AP}(A) \rightarrow \mathcal{IA} \cup \{Err\}$, where $\mathcal{IA}$ is the universal set of IA and Err represents the undefined IA. For any term s, the denotation $[\![s]\!]$ is defined inductively.

First and foremost, if s is a primitive, then $[\![s]\!]$ is the corresponding interface automaton defined in Fig. 2, providing the parameters are well-defined (otherwise the semantics of the primitive is taken to be Err). The parameters of each primitive are single or lists of uninterpreted message ports of $\mathcal{A}$. Lists must have finite length, and message ports in the parameters of a primitive must be pairwise disjoint. In the case of $Merge$ and $Split$, we require that a and $a_1 \ldots a_n$ are equated in the ontology. Furthermore, for the case of the $Order$ primitive, we require that both lists have the same length.

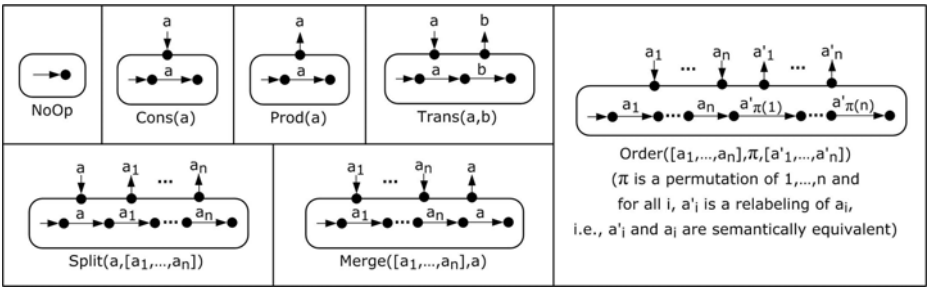


Fig. 2. Semantics of the primitives

If s is a compound term (i.e., consists of operators), then $[\![s]\!]$ is given by the mappings below. However, an informal description is in order first. An operator on terms of the algebra induces a behaviour on the behaviours of the operands. The operator $\odot$ connects terms of the algebra on common message ports; this is equivalent to plugging the corresponding IA into each other, or synchronising them. On the other hand, the operator $+$ behaves like alternation in regular expressions; a connector defined in terms of $+$ behaves like one of its operands. The operators $\wedge$ and $\setminus$ were both defined in Section 2. Finally, $^{\perp}$ acts like the inverse of its operand by interchanging inputs and outputs.

- $s = t \odot u$. If either of $[\![t]\!]$ or $[\![u]\!]$ is equal to Err, then $[\![s]\!] = Err$. Alternatively, if $[\![t]\!]$ and $[\![u]\!]$ are not composable or $[\![t]\!] \,||\, [\![u]\!]$ is not defined, then $[\![s]\!] = Err$. Otherwise, $[\![s]\!] = [\![t]\!] \,||\, [\![u]\!]$.

- $s = t + u$. If either of $[\![t]\!]$ or $[\![u]\!]$ is equal to Err, then $[\![s]\!] = Err$. Otherwise, $[\![t]\!]$ and $[\![u]\!]$ are IA $\mathcal{C}$ and $\mathcal{D}$. If $\mathcal{A}_I^C \cup \mathcal{A}_I^D$ and $\mathcal{A}_O^C \cup \mathcal{A}_O^D$ are not disjoint, then

the alternation is not defined and $[\![s]\!] = Err$. For the case when the action sets do agree, $[\![s]\!] = Determinise(\mathcal{C} + \mathcal{D})$, where $\mathcal{C} + \mathcal{D}$ is defined to be the IA $(V, v_0, \mathcal{A}_I^C \cup \mathcal{A}_I^D, \mathcal{A}_O^C \cup \mathcal{A}_O^D, \rightarrow)$ such that:

- $V = V^C \dot\cup V^D \dot\cup \{v_0\}$ i.e., V^C, V^D and $\{v_0\}$ are pairwise disjoint
- $\rightarrow = \rightarrow_C \cup \rightarrow_D \cup \{(v_0, a, v) : v_0^C \xrightarrow{a}_C v\} \cup \{(v_0, a, v) : v_0^D \xrightarrow{a}_D v\}$.

$Determinise(\mathcal{C} + \mathcal{D})$ is the deterministic IA equivalent to the possibly non-deterministic IA $\mathcal{C} + \mathcal{D}$. Thus, $Determinise$ is a function on IA which implements a suitable variant of the algorithm described in [13], that determinises a finite state automaton.

- $s = t \wedge u$. If either of $[\![t]\!]$ or $[\![u]\!]$ is equal to Err, then $[\![s]\!] = Err$. Otherwise, $[\![t]\!]$ and $[\![u]\!]$ are IA $\mathcal{C}$ and $\mathcal{D}$. If $\mathcal{A}_I^C \cup \mathcal{A}_I^D$ and $\mathcal{A}_O^C \cup \mathcal{A}_O^D$ are not disjoint, then the conjunction does not exist and $[\![s]\!] = Err$. For the case when the action sets do agree, $[\![s]\!]$ is an IA $(V^C \times V^D, (v_0^C, v_0^D), \mathcal{A}_I^C \cup \mathcal{A}_I^D, \mathcal{A}_O^C \cap \mathcal{A}_O^D, \rightarrow)$, where $\rightarrow$ is the smallest relation satisfying the following rules:
  1. If $p \xrightarrow{a}_C p'$ and $q \xrightarrow{a}_D q'$ with $a \in \mathcal{A}_O^C \cap \mathcal{A}_O^D$, then $(p, q) \xrightarrow{a} (p', q')$
  2. If $p \xrightarrow{a}_C p'$ with $a \in \mathcal{A}_I^C \setminus \mathcal{A}_I^D$, then $(p, q) \xrightarrow{a} (p', q)$
  3. If $q \xrightarrow{a}_D q'$ with $a \in \mathcal{A}_I^D \setminus \mathcal{A}_I^C$, then $(p, q) \xrightarrow{a} (p, q')$
  4. For $a \in \mathcal{A}_I^C \cap \mathcal{A}_I^D$:
     (a) If $p \xrightarrow{a}_C p'$ and $q \not\xrightarrow{a}_D$, then $(p, q) \xrightarrow{a} (p', q)$
     (b) If $p \not\xrightarrow{a}_C$ and $q \xrightarrow{a}_D q'$, then $(p, q) \xrightarrow{a} (p, q')$
     (c) If $p \xrightarrow{a}_C p'$ and $q \xrightarrow{a}_D q'$, then $(p, q) \xrightarrow{a} (p', q')$.

- $s = t \setminus u$. Based on [8], it follows that quotienting is a derived operator of the algebra. Thus, $[\![s]\!] = [\![(t^\perp \odot u)^\perp]\!]$.

- $s = t^\perp$. If $[\![t]\!] = Err$, then $[\![s]\!] = Err$. Otherwise, $[\![s]\!]$ is equal to $[\![t]\!]$ with the input and output sets exchanged in the signature of $[\![t]\!]$.

- $s = (t)$. Simply $[\![s]\!] = [\![t]\!]$.

The operators $\odot$, $+$, $\wedge$, $\setminus$ and $^\perp$ satisfy a number of axioms, as we briefly elaborate below.

1. Plugging $\odot$ is commutative and associative, but is not idempotent. It has an identity element $NoOp$, so $(\mathcal{AP}(\mathcal{A}), \odot, NoOp)$ is a commutative monoid (i.e., an abelian semigroup with an identity). Plugging does not distribute over $+$, $\wedge$ nor $\setminus$.
2. Alternation $+$ is commutative, associative, and idempotent. The identity of $+$ is also $NoOp$, so $(\mathcal{AP}(\mathcal{A}), +, NoOp)$ is a commutative monoid. Alternation does not distribute over $\odot$ nor $\setminus$, however it does distribute over $\wedge$.
3. Conjunction $\wedge$ is commutative, associative, and idempotent. The operator does not have an identity element in the algebra, and does not distribute over $\odot$ nor $\setminus$, but it does distribute over $+$.

4. Quotienting $\setminus$ is not associative, commutative, nor idempotent. *NoOp* is a right-identity element for the operator. Quotienting does not left or right distribute over $\odot$, $+$, nor $\wedge$.
5. Inversion $^{\perp}$ distributes over $+$, but not $\odot$, $\wedge$ nor $\setminus$. Double inversion of a term is an identity function on that term.

As we remarked in Section 2, a desirable property of a connector algebra is its ability to support notions of specification. Consequently, there should be a concept of refinement on terms, and indeed our algebra does support this.

Definition 5. *Let s and t be terms of the algebra, and let $\sqsubseteq$ be the alternating simulation refinement relation defined on IA $[\![s]\!]$ and $[\![t]\!]$. Term t refines term s, written as $s \trianglelefteq t$, iff the denotation of t refines the denotation of s at the semantic level or $[\![s]\!] = Err$. Formally, $s \trianglelefteq t \Leftrightarrow [\![s]\!] \sqsubseteq [\![t]\!] \vee [\![s]\!] = Err$.*

Establishing refinement on terms allows us to define equivalence. Semantic equality of terms is too strong for equivalence of connector behaviours, so we choose to express it in terms of the weaker refinement relation.

Definition 6. *Term s is said to be equivalent to term t, written as $s \equiv t$, if, and only if, $s \trianglelefteq t$ and $t \trianglelefteq s$.*

Our choice of equivalence seems most natural for connector behaviours, as it allows for seamless substitutivity. However, it is unfortunate that the equivalence is expressed in terms of the underlying semantics, rather than the syntax of the terms. Nevertheless, this correspondence with the IA semantics means that the algebra is trivially both sound and complete, even after equating all incompatible and undefined terms with Err.

Theorem 2. *Let $\stackrel{\circ}{=}$ denote the equivalence of IA (i.e. mutual refinement). For any terms s and t it holds that $s \equiv t \Leftrightarrow ([\![s]\!] \stackrel{\circ}{=} [\![t]\!]) \vee ([\![s]\!] = Err = [\![t]\!])$.*

Having defined equivalence and established that the algebra is sound and complete, it is an easy consequence that our axiomatisation is correct. The reason for $\odot$ failing to be idempotent is closely related to the reason that $[\![s]\!] \,||\, [\![s]\!]$ is not defined in general, because of restrictions on composability.

The formal definition of the semantics, as well as the example of idempotence failing, has a notable consequence for the algebra. If s and t are well-formed terms whose semantics are not equal to Err, then it is not the case that the semantics of $s \odot t$, $s + t$, $s \wedge t$ and $s \setminus t$ are not equal to Err, because of the restrictions imposed by these operators. This seems undesirable, but it is a direct consequence of IA.

This shortcoming might seem unpalatable at first, but we do not believe it to be a problem; in fact, it is an advantage. In the context of CONNECT, we are concerned with generating connectors in a compositional manner. If we take two connectors, each of which is expressed by a term in the algebra, we can combine the two terms and observe if the outcome is equal to Err. If it is, then it follows

that the two connectors do not work with each other. Thus the algebra allows for compositional reasoning.

Another requirement of CONNECT is the ability of our algebra to serve as a baseline for automated connector synthesis, as stated below. This is closely related to the quotienting operator defined in [8], but requires suitable modification to be applicable to the algebra.

> CONNECTOR SYNTHESIS
> *Instance:* Components $\mathcal{E}$ and $\mathcal{F}$ represented by IA.
> *Problem:* Find a term $x \in \mathcal{AP}(\mathcal{A})$ such that $inv(\mathcal{E}) \sqsubseteq \mathcal{F} \parallel [\![x]\!]$ and $inv(\mathcal{F}) \sqsubseteq \mathcal{E} \parallel [\![x]\!]$, where inv inverts input and output actions.

The connector synthesis problem aims to find a connector x expressible in our algebra that can mediate interoperability incompatibilities between components represented by arbitrary IA. We require that (i) every interaction exhibited by $inv(\mathcal{F})$ is allowed by $[\![x]\!] \parallel \mathcal{E}$, and (ii) every interaction exhibited by $inv(\mathcal{E})$ is permitted by $[\![x]\!] \parallel \mathcal{F}$. This allows us to formally characterise *interoperability* between components in our algebra.

CFring example. A connector for the scenario described in Section 3 may be expressed in terms of the algebra $\mathcal{AP}(\mathcal{A})$ as the following compound term:

```
((Trans(aut,handshake) ⊙ Cons(handshake_ok) ⊙ Prod(auth) ⊙ Trans(auth_ok,aut_ok)) ⊙
(Trans(quit,close) ⊙ Trans(close_ok,quit_ok)) ⊙
(Order([mo,ido],(2,1),[mo',ido']) ⊙ Trans(ido',desti) ⊙ Trans(mo',messagei) ⊙ Cons(acko))⊙
(Order([desto,messageo],(2,1),[desto',messageo']) ⊙ Trans(messageo',mi) ⊙
Trans(desto',idi) ⊙ Prod(acki)))
+
((Trans(aut,log) ⊙ Trans(log_ok,aut_ok)) ⊙
(Trans(quit,out) ⊙ Trans(out_ok,quit_ok)) ⊙
(Order([mo,ido],(2,1),[mo',ido']) ⊙ Trans(ido',di) ⊙ Trans(mo',msgi)) ⊙
(Order([do,msgo],(2,1),[do',msgo']) ⊙ Trans(msgo',mi) ⊙ Trans(do',idi)))
```

This expression seems quite complex, but it is worthwhile noticing how it can easily be decomposed into distinct portions corresponding to the original protocols. There is close correspondence between the four branches of the CFring protocol shown in Figure 1.C, and the paths of the MSN and XMPP protocols shown in Figures 1.A and 1.B, respectively. Each of these branches neatly map onto a sub-term of the connector expression above. This suggests that connector terms can be defined by analysis of the corresponding protocols' transition systems. Unfortunately, the connector only allows the sending and receiving of a single message, but we shall elaborate on this observation further in Section 5.

Relating the connector synthesis problem to our example, we built our connector by constructing two sub-connectors x' and x''. The term x' was used to mediate MSN and CFring. Note how $inv(MSN) \sqsubseteq [\![x']\!] \parallel CFring$ and $inv(CFring) \sqsubseteq [\![x']\!] \parallel MSN$. Analogously, x'' mediates XMPP and CFring.

We combined x' and x'' by means of a suitable composition operator (i.e., $+$), thus obtaining x. Automating this kind of reasoning represents a specific area that we wish to explore, in order to develop a comprehensive theory of composable connectors in CONNECT.

5 Concluding Remarks

In this paper, we formalised an initial high-level connector algebra for reasoning about protocol mismatches. Solutions to basic mismatches are modelled as primitive terms and complex mismatches are solved by combining primitives of the algebra by means of different composition operators. The semantics of the terms and operations on them are given by IA, which is a suitable candidate for expressing the behaviour of connectors that reside within a highly heterogeneous environment, as we shall remark later.

Our formulation sets the scene for a yet-to-come comprehensive algebra facilitating the rigorous characterisation of complex interaction behaviours. Such an algebra should allow reasoning with respect to both functional and non-functional properties, in addition to supporting automated reasoning and learning about system interaction behaviour, as well as automated synthesis, matching, refinement, composition, evolution, and (possibly partial) re-use of connectors.

The case study highlights a shortcoming of our current algebra, in that we cannot construct a connector that exhibits looping behaviours. The form of our algebra dictates that for any term whose semantics are equivalent to an IA (as opposed to Err), the structure of the automaton is a directed graph in which every state is visited at most once. Such a restriction on the behaviour of the connector is unduly restrictive. This is evident from our case study, where the connector only supports the sending and receiving of a single message. If the number of messages to be sent and received is known in advance, then we can build a connector whose size is related polynomially to the number of messages to be transmitted. This, however, is inadequate.

In a future version of the algebra, the restriction on looping would need to be lifted. We already have an idea of how this could be done, by introducing looping equivalents of the primitives. It also seems likely that we would need a fix-point operator to encode complex looping behaviours in the algebra, beyond those at the primitive level. It would be interesting to see whether looping operators make it easier to model connectors in our algebra or not. Naturally, we would hope so.

This shortcoming of the algebra is not to say that IA are a bad choice of model for assigning semantics to terms; after all, it is the algebra that is restricting the behaviour of IA. As a consequence of having chosen IA as the semantic model, our algebra supports specifications of behaviours, which we claim are necessary for building scalable connectors. Furthermore, IA support a notion of refinement which is a congruence with respect to a number of our composition operators. Accordingly, substitution (e.g., for connector reuse or evolution) can be performed compositionally. We have also defined a notion of equivalence over the terms of the algebra and, based on this, we established that the algebra

is sound and complete. These properties advocate the adoption of IA as the semantic model for the algebra, based on how closely they align with the key dimensions of CONNECT that we specified in Section 1.

Our case study shows that the algebraic term representing a connector maps intuitively onto the models of the protocols to be mediated. The purpose of the algebra was to give system designers a high-level tool for specifying and reasoning about connector behaviours, which is why we favoured the utilisation of high-level primitives rather than developing yet another low-level process calculus. It seems that our high-level algebra allows a designer to easily and intuitively specify complex connectors, although further justification would be required for this claim based on further case studies.

We have not considered quantitative aspects of connectors as part of our algebra. As a minimum we would like to support time and probability. Although we have not introduced such aspects to our algebra yet, we have been looking at interactive Markov chains [21] and quantitative extensions of I/O automata [10,20]. Further work in this area involves looking at how these extensions may be carried across to IA. Besides having a quantitative model for expressing the semantics of our algebra, we would also need to consider how the syntax of our algebra would change. Clearly, primitives of the algebra will need to be annotated with quantitative values, but it will also be necessary to see whether it is meaningful to combine these values under the operators of the algebra.

In addition to IA, we are looking at modal specifications as a formalism for connectors [15]. These models are to some extent dissimilar to IA, yet they do share some common functionality [17]. We would like to compare and contrast these models with IA so that we can try to combine the best features of both for our algebra. Besides this future work, it is also our intention to determine whether the set of identified primitives is complete enough. Increasing the set of basic solutions should allow us to increase the types of behaviours that our algebra can capture. Discovery of such primitives is likely to come from analysis of further scenarios.

As broached at the end of Section 4, we need to take a closer look at automated connector synthesis. This is likely to be a definitive area on which our algebra is judged as to whether it has made a meaningful contribution to component-based design. We already have ideas relating to this in terms of rewriting systems [12], although the details require further fleshing out.

Thus, our preliminary algebra has raised a number of issues that we should take account of in formalising a comprehensive algebra to meet the demands imposed by CONNECT. Moving on from here, we intend to combine the positive features of our current algebra and refine its limitations in order to develop an algebra suitable for modelling connectors that reside in a truly heterogeneous world of ubiquitous devices.

References

1. CONNECT consortium. CONNECT Deliverable D2.1: Capturing functional and non-functional connector behaviours. CONNECT EU project no. 231167, http://connect-forever.eu/

2. Allen, R., Garlan, D.: A Formal Basis for Architectural Connection. ACM Trans. Softw. Eng. Methodol. 6(3), 213–249 (1997)
3. Alur, R., Henzinger, T.A., Kupferman, O., Vardi, M.Y.: Alternating Refinement Relations. In: Sangiorgi, D., de Simone, R. (eds.) CONCUR 1998. LNCS, vol. 1466, pp. 163–178. Springer, Heidelberg (1998)
4. Arbab, F., Chothia, T., Mei, R., Meng, S., Moon, Y., Verhoef, C.: From coordination to stochastic models of QoS. In: Field, J., Vasconcelos, V.T. (eds.) COORDINATION 2009. LNCS, vol. 5521, pp. 268–287. Springer, Heidelberg (2009)
5. Arbab, F., Chothia, T., Meng, S., Moon, Y.-J.: Component connectors with QoS guarantees. In: Murphy, A.L., Vitek, J. (eds.) COORDINATION 2007. LNCS, vol. 4467, pp. 286–304. Springer, Heidelberg (2007)
6. Baier, C.: Probabilistic models for reo connector circuits. Journal of Universal Computer Science 11(10), 1718–1748 (2005)
7. Baier, C., Sirjani, M., Arbab, F., Rutten, J.J.M.M.: Modeling component connectors in reo by constraint automata. SCP 61(2), 75–113 (2006)
8. Bhaduri, P., Ramesh, S.: Interface synthesis and protocol conversion. Form. Asp. Comput. 20(2), 205–224 (2008)
9. Bliudze, S., Sifakis, J.: The Algebra of Connectors - Structuring Interaction in BIP. IEEE Transactions on Computers 57(10), 1315–1330 (2008)
10. David, A., Larsen, K.G., Legay, A., Nyman, U., Wasowski, A.: Timed I/O Automata: A Complete Specification Theory for Real-time Systems. In: HSCC 2010, pp. 91–100 (2010)
11. de Alfaro, L., Henzinger, T.A.: Interface-based Design. In: Engineering Theories of Software-intensive Systems. NATO Science Series: Mathematics, Physics, and Chemistry, vol. 195, pp. 83–104. Springer, Heidelberg (2005)
12. Dershowitz, N., Jouannaud, J.-P.: Rewrite Systems. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science. Formal Models and Sematics, vol. B, pp. 243–320. Elsevier, MIT Press (1990)
13. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. Prentice-Hall, Englewood Cliffs (2007)
14. Inverardi, P., Issarny, V., Spalazzese, R.: A Theory of Mediators for Eternal Connectors. In: Margaria, T., Steffen, B. (eds.) ISoLA 2010, Part II. LNCS, vol. 6416, pp. 236–250. Springer, Heidelberg (2010)
15. Larsen, K.G.: Modal specifications. In: Sifakis, J. (ed.) CAV 1989. LNCS, vol. 407, pp. 232–246. Springer, Heidelberg (1990)
16. Lynch, N.A., Tuttle, M.R.: An Introduction to Input/Output Automata. CWI Quarterly 2, 219–246 (1989)
17. Raclet, J.-B., Badouel, E., Benveniste, A., Caillaud, B., Legay, A., Passerone, R.: Modal interfaces: unifying interface automata and modal specifications. In: EMSOFT 2009, pp. 87–96. ACM, New York (2009)
18. Schmitt, A., Stefani, J.-B.: The kell calculus: A family of higher-order distributed process calculi. In: Priami, C., Quaglia, P. (eds.) GC 2004. LNCS, vol. 3267, pp. 146–178. Springer, Heidelberg (2005)
19. Spalazzese, R., Inverardi, P.: Mediating connector patterns for components interoperability. In: ECSA 2010, LNCS (to appear, 2010)
20. Wu, S.-H., Smolka, S.A., Stark, E.W.: Composition and Behaviors of Probabilistic I/O Automata. Theor. Comput. Sci. 176(1-2), 1–38 (1997)
21. Xu, D.N., Gössler, G., Girault, A.: Probabilistic Contracts for Component-based Design. In: ATVA 2010 (to appear, 2010)

Certification of Software-Driven Medical Devices

Mark Lawford, Tom Maibaum, and Alan Wassyng

McMaster Centre for Software Certification
Department of Computing and Software
McMaster University, Hamilton, Canada L8S 4K1
{lawford,wassyng}@mcmaster.ca, tom@maibaum.org

Abstract. This track focuses on the issue of certification for modern medical devices. These devices rely more and more on software and are paradigmatic examples of safety critical systems. Existing approaches to software safety and certification are invariably process based; at best, this gives us only indirect, statistical evidence of safety. Thus, they do not offer the kinds of product based guarantees expected by engineers in other classical disciplines as a basis for product certification. This track focuses on the state of the art and proposals for improvement in this crucial area of research. The track includes four papers on relevant topics and a panel discussing the crucial scientific issues involved in making certification judgements.

Keywords: software certification, safety critical systems, safety cases, assurance cases, formal methods, medical devices.

1 The Certification of Software-Driven Medical Devices Track

Software certification is in the news. From automotive recalls to radiation device malfunctions, deaths caused by faulty software have woken people up to the fact that software embedded in devices of all kinds has the capability of both helping us and killing us. It is quite obvious to many that software is an incredible enabling technology. It is so good in fact, that there is now almost no new device/technology on the market that does not depend on software in some way. We have also been remarkably successful in building huge numbers of software-enabled devices, with a rather limited number of known serious problems. However, this is quite misleading, and has resulted in severe over confidence, both on the part of manufacturers and the public at large. As software and devices become increasingly complex and safety features get further intertwined with functional features, the chance of creating serious disasters also dramatically increases. Every now and again, just to remind us, software faults in critical applications feature prominently in the world's news - but establishing software certification and regulation as the norm is not on most people's horizon, never mind establishing real improvements in software certification. [1]

Certifying software based systems for critical applications remains a serious challenge for the software engineering research community, not just for industry.

T. Margaria and B. Steffen (Eds.): ISoLA 2010, Part II, LNCS 6416, pp. 293–295, 2010.

The problems for industry are very serious, due to lack of sufficient engineering principles to make this possible and due to the focus by regulators and industry on process based approaches to quality when, typically, classical engineers take a very product focused approach to quality and certification. Medical devices have become dependant on software for their operation and have proven to be an increasing challenge for regulators and manufacturers. This track focuses on approaches that the formal methods community could recommend to regulators and to industry to improve the predictability of the certification process for both the regulator and the manufacturer, as well as the effectiveness of the certification itself.

Process based certification is common because we have been able to do it and software engineering research has largely focused on it; it gives us the illusion, rather than a guarantee, that we have produced a safe system. It is a lot easier to evaluate a development process than it is to decide on required attributes in dependable software products and then be able to measure them effectively. Because process based standards model the products of the process superficially, if at all, it is impossible to characterise properly the properties of the entity we are actually interested in, namely the application we have built. Any process-based definition of quality ends up guaranteeing only that certain steps and activities were undertaken, but does not offer direct evidence of the presence of desirable properties. A high quality process is not a reliable indication of a high quality product. Nevertheless, we believe that it is essential that the product be built by an organization that has excellent processes and excellent people, because this is likely to result in good products. It follows that certifying agents will be interested in these aspects, but they can usually be audited in a straightforward manner, often by a third-party with no specific knowledge about the products developed by the audited organization, or its potential problems in relation to safety. As a result, because process- based approaches are inherently unable to guarantee what a regulator wants, it is product-focused certification that should currently be more of a focus - and essential to evaluating the quality of the actual product. [1]

One approach that exhibits elements of both process based and product focused certification, is that of the safety case, and its cousin the assurance case. A safety case provides a structure in which the producer makes claims related to the safety of the product, and presents an argument as to why the claims are valid, using evidence related to and/or derived from the product. Latterly, argumentation theory has been proposed as a way of better presenting/structuring the safety case. This may make the safety case more comprehensible, but there are some serious issues remaining about the scientific basis for safety (and assurance) cases, as well as their objective evaluation by regulatory bodies.

The track consists of four technical papers, followed by a panel discussion on one of the most important topics we can think of in the current state of software certification: *In general, the formal methods and software engineering communities have not established an adequate scientific understanding of the coverage obtained through applying various inspection, testing and mathematical verification*

strategies and techniques to a specific problem. One specific consequence of this is that there appears to be no commonly accepted basis for making a scientific judgement about the adequacy of assurance cases. Judgement is based, more or less completely, on expert opinion and educated guesses rather than objective, scientific criteria.

We believe that the technical papers presented in this track are representative of different levels of formality as well as different points of view as to what are important problems to tackle in relation to software certification.

The first paper by Michaela Huhn and Axel Zechner, puts forward a proposal for a framework that uses international standards to derive quality criteria that can then be used by software developers to build the software/system and the accompanying assurance case, and by assessors from certification authorities to evaluate the assurance case. The idea is to take a recommended process based standard like IEC 62304 for the design of medical device software and use techniques and measures recommended in IEC 61508-3 to populate identified process activities in the former with proven techniques from the latter. They can thus build a quality assessment model based on this and apply it to assess both the software and the assurance case associated with it.

The second paper is by Dominique Mèry and Neeraj Kumar Singh. This paper represents a development paradigm built on ideas from formal methods and model driven engineering. The development progresses from informal requirements, through formal requirements and refinement based verification and validation steps. The paper also discusses the application of the methodology in the modelling of a cardiac pacemaker.

The third paper by Sebastian Fischmeister and Akramul Azim examines how design choices can positively or adversely affect the difficulty in mathematically analyzing the system, and also the black box predictability of the system. An example that involves distributed monitoring of the human cardiovascular system is used to demonstrate their ideas.

Finally, the fourth paper is by Eunkyoung Jee, Insup Lee, and Oleg Sokolsky. This paper considers the problem of developing an assurance case for a real time cardiac pacemaker. It focuses on the construction of an assurance case for real time software developed using a model driven safety assured process based on formal modeling, rigorous code generation from the verified model, and subsequent validation of the timing characteristics of the developed code. The authors ultimate goal is to arrive at a commonly accepted assurance case template that can be applied to a variety of safety critical, software based systems.

Reference

1. Hatcliff, J., Heimdahl, M., Lawford, M., Maibaum, T., Wassyng, A., Wurden, F.: A software certification consortium and its top 9 hurdles. In: SafeCert 2008 Proceedings (2008) (to appear in ENTCS)

Arguing for Software Quality in an IEC 62304 Compliant Development Process

Michaela Huhn and Axel Zechner

Institut für Informatik
Technische Universität Clausthal
Clausthal-Zellerfeld, Germany
{michaela.huhn,axel.zechner}@tu-clausthal.de

Abstract. Safety regulations for medical device software are stipulated in numerous international standards. IEC 62304 addresses software life-cycle processes and identifies core processes, software development activities, and tasks that aim for high-integrity software as a prerequisite for dependability of medical devices controlled by this software. However, these standards prescribe neither a process model nor particular software engineering methods to accomplish the normative requirements. Hence, the manufacturer has to argue in the software development and quality management plans that the selected methods cover the required tasks and are appropriate in order to accomplish high-quality artifacts.

We propose a method for assessing quality- and engineering-centric arguments in dependability cases to assure IEC 62304-compliant software development. Our method is based on an activity-based quality model representing the *impact* of facts about methods and design artifacts on development activities. The impact makes the relation between characteristics of design artifacts and activities contributing to the software safety process explicit. It is derived from state-of-the-art software engineering knowledge and best practices recommended in current safety standards like IEC 61508-3.

1 Introduction

Software-controlled medical devices are an emerging market. New applications are developed, increasing the diversity in the field in several dimensions: Medical devices range from small systems with restricted functionality like pace makers or infusion pumps to complex medical imaging for diagnostics or surgery navigation. We find one of a kind compounds of devices, e.g. in intensive care wards in specialized medical centres, and the mass market of simple living assistance and health monitoring systems for elderly and disabled people.

As software has become an integral part in realizing a medical device's functionality, software dependability is gaining importance. This is an immediate consequence of regulatory obligations and product liability, but also of the increasing number of software related faults and product recalls [18,4].

Besides hundreds of standards that comprise functional requirements on particular medical products, there are also a number of standards addressing system

T. Margaria and B. Steffen (Eds.): ISoLA 2010, Part II, LNCS 6416, pp. 296–311, 2010.

safety and software development: The process standard for medical devices IEC 62304 [12] is accompanied by standards for quality and risk management, ISO 14971 and ISO 13485, respectively, whereas the medical device product standards IEC 60601-1 and IEC 61010-1 direct the development at the system level.

Compared to dependability standards addressing other domains, like avionics RTCA-Do-178B [1] or the IEC 61508 as the fundamental standard for functional safety of E/E/EP systems [6], the standard for software life-cycles for medical devices IEC 62304 is more explicit with respect to the goals that shall be achieved by a certain process activity. But IEC 62304 does not name software engineering methods and techniques that are considered adequate for the required activities.

Thus, manufacturers have the freedom to tailor the process and to select software engineering methods according to their specific needs in the medical device sector. On the other hand, a lack of concrete technical advice on process and product qualities leads to uncertainness about the appropriateness of software development measures on both sides, the manufacturers who have to argue for their processes in the quality management plan and for the dependability of the product in the associated safety case when they apply for the approval of a product, as well as the certification authorities which have to assess these documents (see [9,2,3] for a detailed discussion of these difficulties).

Here, we propose an assessment method for quality-centric arguments provided in the certification procedure of dependable medical device software. The method adapts our previous work on dependability case arguments in the railway domain [11]. The core of our method is an activity-based quality model relating facts about artifacts to detailed development activities. The refined list of activities as well as the concrete software engineering techniques are taken from the IEC 61508-3 which is recommended in the IEC 62304, annex C.7. By associating the goals given in IEC 62304 with concrete activities in a meaningful way, arguments and hence the selection of techniques can be evaluated for sufficiency wrt. the required software safety classification. We refine and extend the assessment of arguments by checking for coverage and dependencies of artifacts and activities which reflects requirements that are stated explicitly in IEC 62304.

The paper is structured as follows: In Sec. 2 we sketch the IEC 62304 and contrast it with the approach taken in the IEC 61508. Section 3 is dedicated to our assessment method and describes how to derive a quality matrix. The usage of the quality model is demonstrated in Section 4. In Section 5 we discuss our approach in the context of related work and conclude.

2 Software Quality Assurance in Safety Standards

2.1 IEC 62304 - Process Requirements for Medical Device Software

IEC 62304 standardizes life-cycle processes of medical device software. For each subprocess (SW-development, maintenance, risk-management, ..., change) it identifies key activities, subdivided into tasks, recommended in a safety-related

software process. IEC 62304 follows a risk-based approach to quality management, meaning that the fundamental classification determining the required rigor of software quality assurance is inferred from the risk that emanates from a medical device or its malfunctioning, respectively. To avoid controversy about the probability of software defects originating from systematic errors, only the severity of software induced hazards is considered. The software safety classification ranges from *A* - no harm or injury - to *C* - death or severe injury is possible. The classification defines the principle level of rigor, and consequently the efforts to be undertaken, required for all software development and maintenance activities.

IEC 62304 sketches a process-based argumentation towards software safety: software does not contribute to hazardous failure of the system because its development and maintenance follows a systematic safety-oriented process and a dependable implementation of functional requirements can be guaranteed by carefully performing the required activities. The development process consists of the well-known phases (requirement analysis, architecture, design, implementation and integration) which help to control and cope with the complexity of the development. For each phase IEC 62304 recommends activities to *plan, track, control and communicate* possible problems to prevent the risk of systematic errors, following a generic risk management paradigm as described in [5].

2.2 IEC 61508-3 - Software Safety in E/E/EP Systems

ISO/IEC 61508 [6] constitutes a generic standard for functional safety of E/E/EP systems. It provides a generic development approach in order to achieve a rational and consistent technical policy for all electrically-based safety-related systems. Part 3 addresses software. It defines a life-cycle for safety-critical software considering best practices and recommendations from early phases of requirements and development to operation, maintenance and disposal. In contrast to IEC 62304, IEC 61508-3 complements activities by recommendation lists (see Fig. 1) of specific artifacts, engineering methods and technologies giving detailed information about which tasks to perform and how to perform them in a dependable software's life-cycle.

Table C-A.2 – Properties for systematic integrity. Software design and development: Software Architecture Design

Technique/Measure		Properties							
		Completeness with respect to Software Safety Requirements Specification	Correctness with respect to Software Safety Requirements Specification	Freedom from intrinsic design faults	Simplicity and Understandability	Predictability of Behaviour	Verifiable and Testable Design	Fault Tolerance	Defence against Common Cause Failure from external events
3b	Diverse monitor techniques (with independence between the monitor and the monitored function in the same computer)	-	-	R2 Diverse monitor implements only the minimum safety requirements	R2 Diverse monitor provides for implicit diversity	R2 Diverse monitor implements in a simple manner only the minimum safety requirements)	R2 Diverse monitor implements only the minimum safety requirements	R1 (R2 if coverage targets are defined, justified and met.)	R1 (R2 if coverage targets are defined, justified and met.)
3c	Diverse monitor techniques (with separation between the monitor computer and the monitored computer)	-	-	R2 Diverse monitor implements only the minimum safety requirements	R2 Diverse monitor provides for implicit diversity	R2 Diverse monitor implements in a simple manner only the minimum safety requirements)	R2 Diverse monitor implements only the minimum safety requirements	R1 (R2 if coverage targets are defined, justified and met.)	R1 (R2 if coverage targets are defined, justified and met.)

Fig. 1. Recommendations on Software Architecture Design from IEC 61508-3 CDV

2.3 Comparison

Both safety standards agree on a process- and quality-oriented line of argumentation and a risk-based approach to quality management. They coincide on the principal design phases and activities in the development process. However, IEC 62304 focuses on the principal activities, artifacts and goals. Compared to IEC 61508-3, it defines the role of artifacts wrt. the preceding and subsequent process steps more explicitly, but it is less concrete concerning properties of artifacts and how they may be achieved by performing specific software engineering techniques. This observation will shape the quality model and the assessment wrt. coverage and dependencies of activities (see Sec. 4.1). For illustration, we compare the statements on software architecture.

IEC 62304 mentions six subactivities for the architectural design step, namely (1) realization of the requirements, (2) interface design, (3) specification of functional and non-functional SOUP components, (4) specification of the environment of SOUP (SW of unknown provenance) components, (5) partitioning due to the risk mitigation strategy, and (6) verification of the architecture. Generic quality goals are stated like consistency, consideration of dependencies and traceability for all artifacts, but also exemplified for specific artifacts. For the software architecture, risk and safety analysis is recommended to be performed on the architectural design and these analyses shall consider the identified components which are specified wrt. to their structure and behavior (Annex B), etc.

IEC 61508-3 contains three main objectives for software architecture design, one of them addresses the design activity itself and is further subdivided into six parts: (1) selection of techniques and measures to satisfy the safety requirements, (2) partitioning, (3) software/hardware interaction, (4) unambiguous representation of the architecture, (5) treatment of safety integrity of data, and (6) specification of architecture integration tests. Besides generic quality goals, IEC 61508-3 lists input and output documents for architectural design which refer to process dependencies and concrete characteristics of the architecture, i.e. completeness and correctness wrt. the requirements, freedom from intrinsic design faults, simplicity and understandability, predictability of behaviour, verifiable and testable design, fault tolerance and defense against common cause failure. Annex A contains a table of 27 recommended software engineering techniques for architectural design related to software integrity levels (SILs). The upcoming revised version [7] supports the selection of appropriate techniques even further: Each technique is ranked wrt. the rigour with which it is considered capable of achieving a stated characteristic (see Annex C and Fig. 1).

Both standards have strong points and blind spots, i.e. to align the argument with the principal activities and overall quality goals (IEC 62304) vs. directing engineers to proven, well-accepted software engineering methods and stressing their contribution to concrete properties of the required artifacts (IEC 61508-3).

From the view-point of software safety assurance - for medical devices or in general - a seamless, holistic argumentation has the following cornerstones: (1) principal goals and activities of a software safety process at the top-level, (2) achieved by performing a mature, harmonized set of software engineering

techniques, which lead to (3) high-quality artifacts with defined safety-related properties and for which (4) evidence is provided in an agreed way.

3 A Quality Model for Argumentations

3.1 A Staged Assessment Process

According to Kelly [15], assessment of software assurance can be thought to consist of, at least, four consecutive steps: (1) Argument Comprehension, (2) Well-formedness checks, (3) Expressive sufficiency checks, (4) Argument criticism and defeat. The presented approach addresses all four steps and especially concentrates on the latter steps in two parts:

1. Structure and Well-Formedness Analysis
2. Conclusiveness of Argumentation

The first part - Structure and Well-Formedness Analysis - covers the syntactical aspect of assessment. The argumentation, often represented as a quality assurance plan, is revised for key claims, assumptions and evidence and put into a unified representation for arguments (here GSN [14]). This representation is then automatically evaluated for well-formedness of an argumentation.

The second part deals with validation of the arguments regarding conclusiveness and coverage. The structured arguments, evidence and facts about development artifacts are fed into an activity-based quality model. The model represents domain knowledge and expert judgement on expressiveness and relevance of arguments which can be derived from best practices, norms and standard for software engineering. Evaluation, as described in Sec. 4, will then yield possibly flawed arguments, missing evidence and implicit trade-offs.

3.2 Structure and Well-Formedness Analysis

Structured Representation of Quality Arguments. As a first step towards assessment, the line of arguments being presented has to be understood. Therefore it is inevitable that the basic elements of the argumentation will be identified: claims, assumptions, strategies, context of the system, and argumentation and evidence. Then, links between those elements have to be captured in order to represent the structure of the argumentation.

For easier and tool-supported handling of the presented argument the elements and links shall be represented using a unified structure. We have chosen the Goal Structuring Notation (GSN) as specified by Kelly in [14]. GSN is a well-accepted notation for safety-case arguments and provides infrastructure for claims as *Goals*, assumptions, strategies, context and evidence as *Solutions*, and representation of the links.

To benefit from model-integration, we propose a UML-profile (see Fig. 2) for GSN. The profile constitutes the types of argumentation elements (*Goal*, *Solution*, etc.) of the GSN with *Stereotypes*.

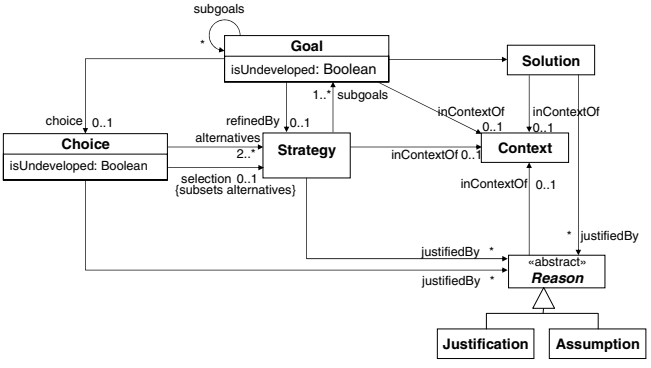


Fig. 2. Basic meta-model for the representation of goal structures

Structural Well-formedness. Next we are able to check for structural correctness of arguments. For example, a cycle in argumentation is definitely wrong, or unrelated arguments indicate an unclear line of of argumentation. This knowledge is captured in rules: Each claim (Goal) must be

- either directly backed by evidence (Solution),
- or immediately refined by sub claims decomposing the higher level claim,
- or, GSN-specifically, refined following a Strategy which in turn decompose into subgoals.
- No other type of element but a Goal may be the root of a goal structure.

The rules concerning the relations between elements are encoded in the profile itself and have been implemented using the Object Constraint Language (OCL) resulting in 19 formulas. Details about the profile can be found in [20].

Argumentation Patterns. In [13,10,14] it is proposed to reuse well proven arguments from the engineering domain as more abstract "argument patterns".

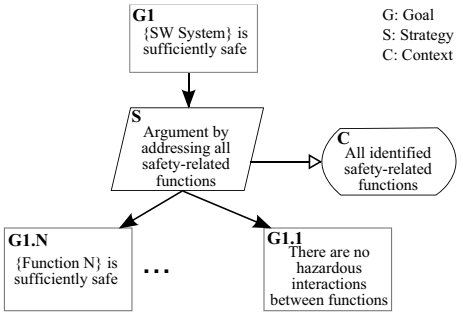


Fig. 3. Example of an argument pattern: Functional Decomposition

Argument patterns describe a fragment of an argumentation, where several place-holders (pattern roles) must be replaced with appropriate arguments or evidence. An example pattern is "Functional Decomposition" (see Fig. 3).

To assure that a pattern is instantiated correctly, it has to be proven that there exists a proper embedding from the pattern roles onto the goal structure: (1) For each pattern role there must exist at least one corresponding instance in the goal structure. (2) The goal structure respects the pattern relations, i.e. whenever two pattern roles A and B are related, then from each instance argument associated with A there exists a path to an instance associated with B respecting the direction of the relation. We extend the profile for GSN with concepts for pattern specification, pattern instance, pattern roles and refinement mapping. Rule checking is also realized via OCL formulae (see [20]).

3.3 Conclusiveness of Argumentation

Completeness, validity and strength of the argumentation have to be inspected next. Software assurance cases are usually assessed by experienced experts providing an adept judgement on the line of argument and its foundation on best practices in software engineering. In part, this knowledge is captured in domain-specific standards, by defining processes and activities, as shown in Sec. 2.

We seize the argument given by Maibaum and Wassyng in [16] that an assessment procedure should focus on *activities* and *products* to be reviewed from a domain-specific point-of-view relying on profound engineering expertise: Does the development plan consider all relevant risks for systematic error by including suitable activities? Are the selected techniques appropriate for their purpose in activities and safety classification? Do the provided evidence and facts about the software development artifacts[1], support strength and validity of the arguments?

To answer this we developed an activity-based quality model [11]. Our key ideas of an activity-centric assessment are (1) to associate a claim with those life-cycle activities it addresses and (2) to evaluate whether facts have either supportive or prejudicial impact on the activities. Interpreting this impact leads to a judgement on the compelling power of the lines of argument.

A Quality Model for Quality-Oriented Arguments. The quality model is inspired from Deissenboeck et al. [8] and presents activities and facts on the system and the process in a matrix. The first dimension lists facts derived from evidence provided in the case and other information known in the system context, from standards or domain specific best practices. *Facts* characterize evaluable and measurable properties of artifacts of the development process. The type of a fact's value can be quantitative or qualitative. Qualitative means either nominal values (e.g. existent, non-existent) or ordinal values (e.g. low, high).

Along the second dimension we align detailed *activities* which are carried out within the life-cycle: e.g. in the argument "Functional safety *(goal)* is supported by

[1] Like e.g. investigation of the identified risks, of the software architecture, of test reports, or other verification results.

IEC 61508-3 properties (IEC 61508-7) Modelling Language	IEC 62304 SW Architecture Activities: document & define structure and components 5.3.1 view **B/C**	document & define interfaces between components 5.3.2 view **B/C**	document & verify SW architecture against requirements 5.3.6 view **B/C**
data / control flow / logic			$\times \mapsto -, \checkmark \mapsto ++$
input/output		$\times \mapsto -, \checkmark \mapsto ++$	$\times \mapsto -, \checkmark \mapsto ++$
function blocks / actions	$\times \mapsto -, \checkmark \mapsto ++$	$\times \mapsto -, \checkmark \mapsto ++$	
hierarchy	$\times \mapsto -, \checkmark \mapsto ++$		
states/places		$\times \mapsto -, \checkmark \mapsto ++$	$\times \mapsto -, \checkmark \mapsto ++$
time		$\times \mapsto 0, \checkmark \mapsto +$	$\times \mapsto 0, \checkmark \mapsto +$
animation			$\times \mapsto -, \checkmark \mapsto ++$
automatic checks			$\times \mapsto -, \checkmark \mapsto ++$

Fig. 4. Mapping of IEC 61508 criteria to IEC 62304 with effect relation

a straight architectural design *(subgoal)*" the subgoal addresses the "architectural design" activity qualified as "straight"'. The performance-degree of activities has a principal impact on what we regard as a quality. Hence, we will annotate the required quality level as an attribute of the activity.

The connection between a fact and an activity is called an *effect*. An effect describes the impact of an evaluated fact on the activity denoted as entry in the quality matrix. The fact's value has to be translated to the effect's domain, e.g. by a table. For simplicity, we consider the ordinal scale (e.g. $--, -, 0, +, ++$) as the set of effect values, expressing strength and direction of the effect. A positive effect value improves, a negative impairs the performance of an activity:

$$\text{Fact} \,|\, \text{Value} \overset{+/-\text{Strength}}{\longrightarrow} \text{Activity}$$

The effect relation (translation table) encodes domain knowledge which is made explicit this way. Fig. 4 depicts an excerpt from our quality model. The column headers contain activities from IEC 62304, the rows contain relevant properties of the modelling language[2] from IEC 61508-3. The cells hold the *effect relationship*.

The basic assessment is performed on arguments and evidence presented in the dependability case only. The qualified activities are taken from the arguments and arranged along the activity-dimension, evidence along the facts-dimension. Filling the matrix yields a set of effects on each activity column. The analysis discovers inconsistencies and tacit trade-offs: Uniform negative impact corresponds to strong counter-evidence that the activity is performed adequately. Mixed directions of effects indicate implicit rebuttal of arguments. Even a uniform positive

[2] Taken from Table A.2 semi-formal methods, properties from part 7 B.2.3.

effect may be below the quality level required for that activity which alludes to lack of strength of backing arguments resulting in undercutting defeat. A conclusive argumentation relies on activities supported by uniform impact from evidence to the required quality level.

Example: We consider a part of an argumentation regarding meaningful backward and forward traceability to create requirement-related test cases. By evaluating the coverage of traceability links on the development documents, the metric *full depth and coverage* yields [DECO | 100%Coverage]. Measuring the fan-out[3] of requirements links in the traceability matrix evaluates to [FAN-OUT, 30% req. $\gg Max_{\text{FAN-OUT}}$], meaning that for 30% of the (functional) requirements fan-out significantly exceeds the recommended maximum. In the quality model, a senior validator recorded the effect of *full-depth coverage* and *FAN-OUT* on *Forward Tracing* and *Backward Tracing* activities. Evaluation results in "+" on *Forward/Backward Tracing*, but "−−" on the *Create Requirement-related test cases* activity. Mixed effect directions occur: Although requirements are traced forward and backward along the development, the argumentation is seriously weakened by the fact of lack of coherence from requirements to implementation units, probably indicating an inept, poorly testable architecture design.

Representing Engineering Knowledge as Quality Model Criteria. In [11] we showed how to derive the dimensions of activities and facts using a dependability taxonomy as a guideline. In general, activities are deduced from the whole life-cycle of a system, including operation, but we examine in detail the activities (identify, analyse, plan, track and control) of dependability management to avoid the risk of systematic errors. Artifacts, i.e. the products of the development process, modelling language, modelling infrastructure, modelling strategies (where modelling also includes detailed design or programming, e.g. C++, UML, written text) and verification strategies serve as input for facts.

As mentioned in Sec. 2, standards regarding functional safety and software provide a rich source of software engineering knowledge on techniques and methods which are *accepted as appropriate in a specific domain.* IEC 62304 comprises recommendations on activities to be performed with a certain quality, in particular to achieve higher levels of dependability (safety classification B,C in IEC 62304). Hence, we take these activities for the activity dimension of the quality model. Requirements for certain qualities are stated as a minimum level of performance for the activities. An argument referring to such an activity is regarded as *conclusive* if the facts contributing with their effects to the activity establish at least the defined quality level.

Example: IEC 62304 declares structural decomposition and behavioural specifications at the architectural design level as mandatory for safety-critical software to assign a risk mitigation measure and for safety analysis and assurance. Thus, the sub-activities *assign risk mitigation measures* and *perform safety analysis* and *safety assurance* are added as parts of the architectural design activity to

[3] FAN-OUT measures the dissemination of functional requirements over design components.

the quality model (even if they are not mentioned in the case) and we impose a requirement that an impact of ++ has to be contributed for these activities from the facts about the architecture. Consequently, an acceptable argument needs backing evidence (facts) whose effect evaluates to ++.

We adapted the quality model to EN 50128 and IEC 61508-3 specific views in a similar manner in [11]: We processed the description of the life-cycle processes to identify activities. The facts were obtained from the recommendations in the annexes (e.g. Fig. 1). The resulting quality model for EN 50128 is a matrix of 67 facts by 54 activities; most entries can be evaluated in a check-list manner. The effect relation was derived from the rating of recommended techniques.

Model Criteria for IEC 62304. The previously described approach cannot immediately be adapted to IEC 62304 since the standard does not explicitly recommend any methods or techniques. So what is needed is an alternative source (1) which provides us with facts related to the goals and activities required by the standard, (2) for which sound, expert-based ratings for their appropriateness exist and (3) which could be accepted by authors and assessors of software assurance cases in practice. As already indicated by the discussion in Sec. 2, we choose IEC 61508-3 to gather facts on techniques. We justify our approach using the following reasons: Firstly, IEC 61508-3 represents a generic standard for E/E/EP systems providing recommendations and ratings on techniques, measures and tools. Secondly, as an international standard, it can be taken as widely accepted among experts and assessors. Thirdly, a closer look in IEC 62304 Annex C.7 reveals that IEC 61508 can be used as a source for appropriate methods and techniques to realize the activities of the life-cycle processes.

Examination of IEC 62304 yields the list of activities grouped by the corresponding phase of the development process (see column heading of Fig. 4). Since activities and goals of IEC 61508-3 cannot be directly mapped to any in IEC 62304, we have chosen to map activities and facts via the process phases. IEC 61508-3 recommends certain measures in annexes B and C for the realization of each process phase. In most cases the measures (e.g. semi-formal methods) are refined to concrete techniques (e.g. Time Petri nets). Relevant details of the methods are explained in part 7 of the standard. From these descriptions we extract the facts (e.g. hierarchy) and then combine similar concepts (e.g. data flow, control flow, logic) to one fact. For the mapping of facts to activities we compare the descriptions of an activity in IEC 62304 to the description of the fact in the context of the related measure or technique in IEC 61508-3. If intended use and recommendation of a measure matches the goal of the activity, we have identified an effect relation. This procedure is systematically applied to the IEC 62304 list of activities.

Example: In sections 5.3.1 & 5.3.2 (IEC 62304) we find the activities: *document & define structure and components* and *document & define interfaces between components*, both belonging to the *software architecture* phase. The IEC 61508-3 recommends in table A.2, e.g., *Structured methods, semi-formal methods ...* exhibiting facts like *systematic partitioning of a system* and *hierarchy*. As

proposed above we deduce: *an appropriate technique for documenting and defining the structure and components* and *the documentation of structure* must provide *systematic partitioning of a system* and *hierarchy*.

To derive the effect relation we propose to map the level of recommendation (NR,-,R,HR) of a certain technique to the ordinal values (–,0,+,++). To abstract from specific techniques in the quality model, a technique is not directly taken as a fact, but we map the values to the above identified set of key facts about it. Thereby, we are able to integrate and evaluate "new" technologies not mentioned in the standards like model-based development on the basis of the existing facts. In case of a binary fact, presence will be evaluated to the associated value of the level of recommendation, in case of absence the effect direction is reverted. Sometimes there is a choice between techniques recommended for an activity, but the techniques are not equivalent wrt. the facts they support. Then we adjust the emphasis: a missing fact is assigned the neutral value 0 and for presence the effects magnitude is lowered.

In addition to recording the effect of facts we also collect the relation between activities which IEC 62304 states as prerequisites for later activities (see Sec. 2.1). These relations will be evaluated further as explained in Sec. 4.2.

Now we have both missing points: (1) relevant, accredited facts to evaluate the activities of an argumentation and (2) an approved rating of the facts, with a potential to evaluate unlisted techniques. In this way, we were able to populate our quality matrix with *facts*, *activities* and the *effect* relationship.

4 Assessment Procedure

We assume that the underlying quality model has been stated. The assessment of the conclusiveness of quality management plans and dependability case arguments is carried out in three phases:

1. Prepare the assessment
   (a) Go through the argument structure and fill in the facts and performed activities addressed in the arguments (see (1.) and (2.) in Fig. 5)
   (b) Add the requirements according to the view representing the assigned safety classification (see (3.) in Fig.5)
2. Evaluate the argument wrt. the quality model
   (a) Evaluate effects according to an assessment strategy (see (4.) in Fig.5)
3. Review potentially flawed arguments. (see (5.) in Fig.5)

To prepare for the assessment, the quality model has to be populated with activities and facts from the argumentation or a view, respectively. The underlying reference model then yields a set of effects on each activity.

Evidence is naturally deduced from properties gained from the subject of investigation. The representation of evidence either is itself a value or can be evaluated to a value. The interpretation of evidence is left to the argument.

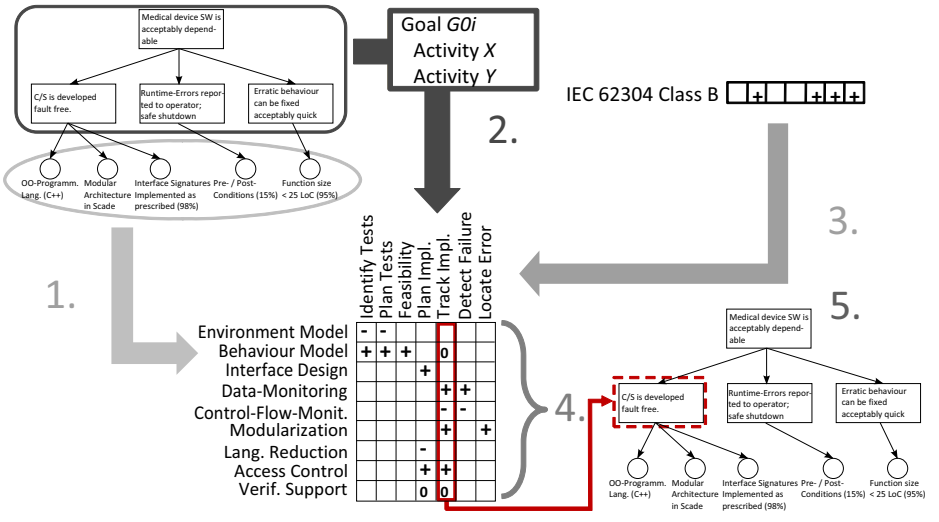


Fig. 5. Illustration of the Assessment

4.1 Using the Quality Model for Assessment

For the analysis, the assessor associates the argument to the concrete activity. That way, the evaluation of the impact of an activity is traceable to an argument. The evaluation strategy depends on the purpose of the assessment:

Evaluate Consistency and Strength of Argumentation. The evaluation is performed only on evidence and arguments presented in the case and it embraces the validity check of the provided artifacts. This strategy aims at discovering intrinsic deficiencies in the line of argument as it is provided. Findings may be:

- missing evidence and defective artifacts, e.g. an incomplete test report;
- the techniques are insufficient or neutralize each other, i.e. supportive impact is missing or below the required quality level, see the example in Sec. 3.3;
- a technique misses a quality fact required for an activity, e.g. a design model is verified against a partial (=incomplete) specification by model checking.

Evaluate Arguments against Available Evidence. In addition, all available facts about the system are represented in the quality model and taken into account. This strategy will additionally find possible counter-evidence that is recorded in the factual basis but neglected in the dependability case, i.e.

- properties of techniques and artifacts that have a negative impact on certain activities that have not been considered in the argumentation.

Check Coverage of Activities wrt. a Normative View. Now the quality model explicitly highlights activities and fact classes recommended by the specific standard it is derived from and compares them to those addressed in the case. The comparison may yield

– activities required by the view, but neglected in the argumentation or not adequately performed;
– some facts about the system for which the standard recommends a particular treatment and evidence has to be provided, e.g. using SOUP components requires a specification of the functional and non-functional enviroment of that component on the architectural level (IEC 62304 Sec. 5.3.3).

Check Dependencies and Usage of Artifacts wrt. a Normative View. The quality model, in particular if induced by the IEC 62304, will state dependencies between activities and a proper usage of artifacts. In the quality model a dependency is reflected by fact classes about artifacts or techniques that have to be evaluated wrt. their effects on the dependent activities. If this required impact is missing, it indicates,

– a lack of integration of activities, artifacts or techniques, e.g. abnormal conditions were specified but not decomposed on the architectural design level;
– a lack of completeness wrt. verification requirements, e.g. testing does not cover all classes of acceptance criteria stated for software components.

The importance of the latter two kinds of assessment is emphasized by the empirical study on recall data of medical device software [18]: A majority of failures arose either from incomplete artifacts (e.g. recommended aspects of the requirements like value ranges, default values, exceptions were not specified) or from losing sight of these issues during the subsequent phases.

4.2 Argument Review

Negative findings in the evaluation only indicate a possible flaw in the presentation of arguments. Further interpretation is needed to decide whether the underlying rationale has to be revised or a better presentation of the arguments is needed or a recalibration of effects can remedy the problem. Nevertheless, the evaluation directly points to a cause in terms of facts and activities.

Based on the evaluation, three steps are performed: (1) look for reasons explaining the impairment, (2) reevaluate arguments, (3) create a report.

The evaluation yields suspicious activities for which the conclusiveness of evidence may be doubted because uniform supportive impact from the facts is missing. These activities trace to arguments associated with the population of the quality model. The assessor then looks for additional explanations putting even strong findings into a perspective: Often, contextual information defines and constrains the actual system requirements. Claims may be additionally supported by implicit or explicit assumptions and justifications. Some arguments may already suffice (e.g. mixed directions in weak effects, cumulative effect of weak facts) or may be justified by providing new evidence. The assessor reevaluates the adjusted arguments and facts. Results are captured in the review report.

4.3 Discussion

Assuring high confidence in the software controlling safety-critical systems is the basis for software certification. Principles such as the need for a rigid safety-oriented process with extensive verification activities or the documentation of the safety arguments in quality management plans and safety cases are agreed. Also the advantages of formal methods in providing conclusive product-specific evidence are becoming more and more accepted.

However, when instantiating such principles for a product development, i.e. when manufacturers have to frame a standard-compliant process or an assessor has to evaluate a software assurance case, the conclusiveness, strength and completeness of arguments may be challenged: Controversy about the right answers arises in particular in the following situations: (1) the corresponding standard does not recommend concrete techniques, artifacts, and quality characteristics like the IEC 62304, (2) manufacturers and certification authorities do not agree on a de-facto standard procedure, or (3) the technical state-of-the-art is passing by the standard's state and new techniques shall be introduced. Since there exist hardly any systematic studies on the impact of methods and techniques applied in industrial software development on software safety, a scientific fundament is still missing. Moreover, the long standing (simplified) credo of "more formal methods will bring more safety" has to be revised as recent work indicates [19].

For medical device software these difficulties are discussed by Maibaum and Wassyng [16] and Feldmann et al. [9]. An answer heading in the same direction as ours is given by Manleitner [17] who used the software quality standard ISO 9126 to derive a quality model from the IEC 62304. As our approach, Manleiter's quality model goes beyond the analysis of structural or logical fallacies, and purely process-oriented arguments. Both quality models offer a systematically constructed, transparent starting point founded on shared software engineering expertise stipulated in standards for assessing software assurance.

What is intentionally left open is a mapping of software engineering techniques recommended in IEC 61508-3 for the different SILs to the software safety classification of IEC 62304.

5 Conclusion

We adapted our 2-phase method for the assessment of IEC 62304-compliant software assurance cases for medical devices. Based on a representation of the arguments in the Goal Structuring Notation, syntax and well-formedness of the argumentation can be automatically checked against common, logical and structural fallacies. To assess the conclusiveness of argumentation, our approach features an activity-based quality model describing the effect of facts on activities relevant to the development of a dependable system. By encoding the impact of properties of the system, artifacts or techniques on the process activities and their required qualities, the assessor's trust in the conclusiveness of an argumentation are made transparent and founded on a software engineering rationale.

The investigation of IEC 62304 and IEC 61508-3 leads to an integrated view. We combine in a quality matrix the concrete facts about artifacts and techniques gathered from IEC 61508-3 with the dependencies and goals stated in IEC 62304. Based on that matrix, the assessment may reveal inconsistent or missing arguments or even neglected counter-evidence. Negative findings can be traced back to arguments and facts about the process or the system. One may object that with the impact relation still a core element depends on experts judgement. However, this judgement is brought to a statement about concrete facts and their impact on a detailed development activity with a specified goal. From that, concrete steps for rebuttal or improvement can be deduced in terms of a detailment of the justification or additional safety-oriented measures which we consider a major advantage of our approach.

References

1. RTCA: Software considerations in airborne systems and equipment certification (December 1992)
2. Abdeen, M.M., Kahl, W., Maibaum, T.: Fda: Between process & product evaluation. In: Joint Workshop on High Confidence Medical Devices, Software and Systems and Medical Device Plug-and Play Interoperability. pp. 181–186 (2007)
3. Becker, U.: Model-based development of medical devices. In: Buth, B., Rabe, G., Seyfarth, T. (eds.) SAFECOMP 2009. LNCS, vol. 5775, pp. 4–17. Springer, Heidelberg (2009)
4. Bliznakov, Z., Mitalas, G., Pallikarakis, N.: Analysis and Classification of Medical Device Recalls. In: World Congress on Medical Physics and Biomedical Engineering - Imaging the Future Medicine. Springer, Heidelberg (2007)
5. Carr, M., Kondra, S., Monarch, I., Ulrich, F., Walker, C.: Taxonomy-based risk identification. Tech. Rep. CMU/SEI-93-TR-006, CMU/SEI (93)
6. Commission, I.E.: IEC 61508: Functional safety of electrical / electronic / programmable electronic safety-related systems (1998)
7. Commission, I.E.: 65A/524/CDV: IEC 61508-3: Functional safety of electrical/electronic/programmable electronic safety-related systems Part 3: Software requirements, Committee Draft for Voting (2008)
8. Deissenboeck, F., Wagner, S., Pizka, M., Teuchert, S., Girard, J.F.: An activity-based quality model for maintainability. In: Proceedings of the 23rd International Conference on Software Maintenance, ICSM 2007 (2007)
9. Feldmann, R.L., Shull, F., Denger, C., Höst, M., Lindholm, C.: A survey of software engineering techniques in medical device development. In: Joint Workshop on High Confidence Medical Devices, Software and Systems and Medical Device Plug-and-Play Interoperability, pp. 46–54 (2007)
10. Graydon, P., Knight, J.: Success arguments: Establishing confidence in software development. Tech. Rep. CS-2008-10, University of Virginia (2008)
11. Huhn, M., Zechner, A.: Analysing dependability case arguments using quality models. In: Buth, B., Rabe, G., Seyfarth, T. (eds.) SAFECOMP 2009. LNCS, vol. 5775, pp. 118–131. Springer, Heidelberg (2009)
12. International Electrotechnical Commission: Medical device software - Software lifecycle processes, IEC62304:2006 (2006)
13. Kelly, T.P., McDermid, J.A.: Safety case construction and reuse using patterns. In: Intl. Conf. on Computer Safety and Reliability (SAFECOMP), pp. 55–69 (1997)

14. Kelly, T.: Arguing Safety – A Systemic Approach to Managing Safety Cases. Ph.D. thesis, University of York (1998)
15. Kelly, T.: Reviewing assurance arguments - a step-by-step approach. In: Proceedings of Workshop on Assurance Cases for Security - The Metrics Challenge, Dependable Systems and Networks (DSN) (July 2007)
16. Maibaum, T.S.E., Wassyng, A.: A product-focused approach to software certification. IEEE Computer 41(2), 91–93 (2008)
17. Manleitner, M.: Quality attributes in IEC 62403 - a practical implementation of a process standard (2010)
18. Wallace, D., Kuhn, D.R.: Failure modes in medical device software: An analysis of 15 years of recall data. Intern. Journal of Reliability, Quality and Safety 8(4) (2001)
19. Yang, F., Jacquot, J.P.: Prouvé? et aprés? In: Actes10es Journées Francophones Internationales sur les Approches Formelles dans l'Assistance au Développement de Logiciels - AFADL 2010, pp. 133–147 (2010)
20. Zechner, A., Huhn, M.: Structural analysis of safety case arguments in a model-based development environment. In: Tagungsband Modellbasierte Enwicklung eingebetteter Systeme V, MBEES 2009 (2009)

Trustable Formal Specification for Software Certification

Dominique Méry and Neeraj Kumar Singh

Université Henri Poincaré Nancy 1
LORIA, BP 239, 54506 Vandoeuvre lès Nancy, France
{mery,singhnne}@loria.fr

Abstract. Formal methods have emerged as a complementary approach to ensuring quality and correctness of high-confidence medical systems, overcoming limitations of traditional validation techniques such as simulation and testing. In this paper, we propose a new methodology to obtain certification assurance for complex medical systems design, based on the use of formal methods. The methodology consists of five main phases: first, informal requirements, resulting in a structured version of the requirements, where each fragment is classified according to a fixed taxonomy. In the second phase, informal requirements are represented in formal modeling language, with a precise semantics, and enriched with invariants and temporal constraints. The third phase consists of refinement-based formal verification to test the internal consistency and correctness of the specifications. The fourth phase is the process of determining the degree to which a formal model is an accurate representation of the real world from the perspective of the intended uses of the model using model-checker. Last phase provides an animation framework for the formal model with *real-time data* set instead of *toy-data*, and offers a simple way for specifiers to build a domain specific visualization that can be used by domain experts to check whether a formal specification corresponds to their expectations. Furthermore, we show the effectiveness of this methodology for modeling of a cardiac pacemaker system.

1 Introduction

Since software plays an increasingly important role in medical devices and more generally in healthcare-related activities, regulatory agencies such as the US Food and Drug Administration and certification bodies (FDA QSR (Quality System Regulation) and ISO's 13485) [1,2] need effective means for ensuring that the developed software-based healthcare device is *safe* and *reliable*. Regulatory agencies, as well as medical device manufacturers, have been striving for a more rigorous engineering-based review strategy providing this assurance.

Providing assurance guarantees for medical devices makes formal methods appealing. Formal model-based methods have been successful in targeted applications [3,4,5] of medical devices. Over the past decade, there has been considerable progress in the development of formal methods [6] to improve confidence in complex software-based devices.

T. Margaria and B. Steffen (Eds.): ISoLA 2010, Part II, LNCS 6416, pp. 312–326, 2010.

Formal methods are usually used in analysing assumptions, relationships, and requirements of a system. To apply formal methods for developing high-confidence medical devices, we consider basic objectives as follows,

- Requirements and metrics for certifiable assurance and safety.
- Establishing a unified theory of medical device development.
- Building a comprehensive and integrated suite of tools for a medical device that support verification activities, including formal specification, model validation and real-time animation.
- Refinement-based formal development to achieve accurate models, easier specification for system and reuse of such specification for further designs.
- Evidence-based certification

Only simulation and testing are usual validation techniques [7] for given specification of any high-confidence medical devices. This is an operational way to check whether a given system realization conforms to an abstract specification. By nature, testing can be applied only after a prototype implementation of the system has been realized. Formal verification, as opposed to testing, works on models (rather than implementation) and amounts to mathematical proof of correctness of a system. There are several other approaches that provide higher level modeling and verification solutions for medical devices. Software verification is a core technology for formal methods. The role of verification and validation is very important in the development of safety critical medical devices. Functional testing and environmental modeling start verification from the requirements analysis stage where design reviews and checklists are used for the validation stage. The results of the verification and validation process is an important component in the safety case, which is heavily used to support the certification process.

Although formal methods are part of the standard recommendations [2] for developing and certifying medical devices, how to integrate formal methods into the certification process is, in large part, unclear. Especially challenging is how to demonstrate that the final developed system, behaves safely. This paper describes formal methods and tools that we have applied to produce evidence for the certification, based on the Common Criteria (CC) [8], of a medical software based device. It also describes the most effective aspects of our methodology for certification and research that could significantly increase the utility of formal methods in software certification.

Software certification as performed by e.g. the FDA [1,2] does not prove correctness. If a product receives certification, it simply means that it has met all the requirements needed for certification. It does not mean that the product is *fault free*. Therefore, the manufacturer cannot use certification to avoid assuming its legal or moral obligations.

We propose a certification model that does focus on correctness. In summary, we can say that we do not encounter any model addressing product quality in the same sense that we do, related to correctness and consistency. There are however many approaches to software certification, that mostly rely on formal verification or expert reviews to determine the product quality.

The contribution of our paper is to propose a refinement-based methodology for medical devices. This methodology provides the solutions for all the requirements enumerated above. A refinement-based combined approach of formal verification, model validation using a model checker and real-time animation [9] is proposed in this methodology for designing high confidence medical devices [2]. It can help a specifier gain confidence that the model that is being specified, refined and implemented, does meet the domain requirements. The formal verification and model validation offer to obtain that challenge of complying with FDA QSR and ISO's 13485 quality system directives [1]. According to the FDA QSR, validation is the "confirmation by examination and provision of objective evidence that the particular requirements for a specific intended use can be consistently fulfilled." Verification is "confirmation by examination and provision of objective evidence that specified requirements have been fulfilled" [2]. An assessment of the proposed methodology is given through a case study, relative to the development of a cardiac pacemaker [10,11].

Considering related works, C.L. Heitmeyer et. al have presented an approach for software certification using formal methods [12,13]. They describe how formal methods are used to produce evidence in a certification, based on facts of a security-critical software system. The evidence includes a top level specification (TLS) of the security-relevant software behavior, a formal statement of security requirements, proofs that the specification satisfied properties, and a demonstration that the source code, which has been annotated with preconditions and postconditions, is a refinement of the TLS. A research report [14] is presented by John Rushby, which is based on certification issues for advanced technology. Its purpose is to explain the use of formal methods in the specification and verification of software and hardware requirements, designs, and implementations, to identify benefits, weaknesses, and difficulties in applying these methods to digital systems used in critical applications, and to suggest factors for consideration when formal methods are offered in support of certification. Software certification as performed by e.g. the FDA [1,2] does not prove correctness. If a product receives certification, it simply means that it has met all the requirements needed for certification. It does not mean that the product is *fault free*. Therefore, the manufacturer cannot use certification to avoid assuming its legal or moral obligations. We propose a certification model that does focus on correctness. In summary, we can say that we do not encounter any model that address product quality in the same sense that we do: related to correctness and consistency. There are however many approaches to software certification, that mostly rely on formal verification or expert reviews to determine the product quality. We believe that our approach adds value with its comprehensiveness (to obtain trustable formal model), its focus on correctness and by establishing a standard to perform software certification that also includes formal verification and real-time animation [9] under domain experts review. It uniformly establishes what to check and how to check it and provides certain evidence of correctness.

Section 2 presents the related work. Section 3 presents a verification and validation methodology for a high confidence medical device. An assessment of the

proposed methodology is given through a case study, relative to the development of a cardiac pacemaker in Section 4. Section 5 describes benefits of our proposed methodology. Finally, section 6 presents concluding remarks.

2 Overview of the Methodology

In recent years, medical devices have grown more complex and providing certification assurance, is a common crucial issue for certification body [1,2]. Under consideration of all kind of requirements of certification body, we propose a novel methodology that addresses the issue of certification for all high-confidence medical devices. Different phases of the methodology are shown in Fig.1 and this is used in development process of critical system. Our methodology consists of the following five main phases,

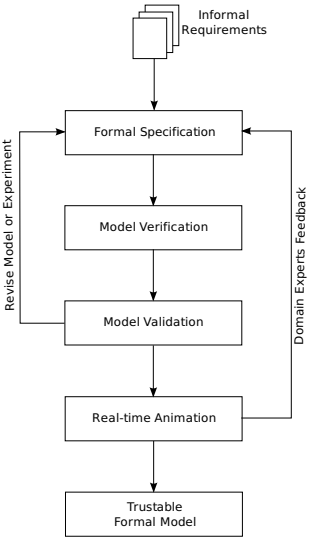


Fig. 1. Trustable verification and validation methodology

Informal Requirements
This phase of our methodology presents informal requirements of a given system. Software requirements specifications are widely used in restricted form of natural language. Natural language is convenient because it allows non-technical users to understand systems requirements. On the other hand, the lack of precise semantics increases the possibility of errors being introduced due to interpretation mistakes and inherent ambiguity. Under or over specification are also common problems when using natural language. Software requirement specification consists of the categorization and structuring of the informal requirement fragment described in the requirements document to produce categorized requirement fragments. An objective of informal requirements is to provide a precise, yet

understandable description of the safety-relevant behavior of the system and to make explicit assumptions on which the safety of the system is based.

Formalization Phase
In our methodology, the required security requirements are formally expressed as properties of the state-based model that underlies the informal requirements. The categorized requirement fragments are described through the set of formal notations in any formal language like Event-B , Z,VDM, etcetera. Formal specification languages have a mathematical (usually formal logic) basis and employ a formal notation to express system requirements. The formal specification is typically a mathematically based description of system behavior, using state tables or mathematical logic. Using the formal notation, precision and conciseness of specifications can be achieved. Formal specification will not normally describe lowest level software, such as mathematical subroutine packages or data structure manipulation, but will describe the response of the system to events and inputs to a degree necessary to establish critical properties.

Formal Verification Phase
This phase has a very important role in formal development. To demonstrate that informal requirements satisfy the safety properties of interest, the informal requirements and the properties are passed to a theorem prover and the prover applied to prove formally that the informal requirements satisfy the properties. A formal notation can be analysed and manipulated using mathematical operators, mathematical proof procedures can be used to test (and prove) the internal consistency (including data conservation) and syntactic correctness of the specifications. Furthermore, the completeness of the specification can be checked in the sense that all enumerated options and elements have been specified. However, no specification language can ensure completeness in the sense that all of the users requirements have been met, because of the informal human-intention nature of the requirements specifications [15]. Finally, the implementation of the system will be in a formal language (i.e., the programming language), it is easier to avoid misconceptions and ambiguities in crossing the divide from formal specifications to formal implementations. A formal verification phase is done to ensure that the model is designed correctly, the algorithms have been implemented properly and the model does not contain errors, oversights, or bugs. In summary, we can say that verification ensures that the specification is complete and that mistakes have not been made in implementing the model. But verification does not ensure that the model solves an important problem, meets a specified set of model requirements and correctly reflects the workings of a real world process.

Formal Validation Phase
Formal validation phase is the process of determining the degree to which a model is an accurate representation of the real world from the perspective of the intended uses of the model that is not covered by formal verification. It consists of the identification of a subset of the formalized requirement fragments for an automatic validation analysis. Validation ensures that the model meets

its intended requirements in terms of the methods employed and the results obtained. The ultimate goal of model validation is to make the model useful in the sense that the model addresses the right problem, provides accurate information about the system being modeled, and to makes the model actually used.

Model checking [16] is a complementary technique to validate the formal model. Model checkers attempt to make formal techniques easier to use by providing a high degree of automation at the expense of generality. Inputs to a model checker are typically a finite state model of a system, along with a set of properties that are expected to be preserved by the system. Properties to be verified can usually be categorized as one of the following,

1. Correct sequences of events
2. Proper consequences of activities
3. Simultaneous occurrences of particular events
4. Mutual exclusion of particular events
5. Required precedence of activities

The model checker explores all possible event sequences of the model to determine that system is always holding required safety properties. If properties hold, the model checker confirms correctness of the system. If a property fails to hold for some possible event sequences, the tool produces counter-examples, i.e., traces of event sequences that lead to failure of the property [17,18].

In Fig.1, this phase gives the feedback to the formalization phase in case of any error is occurring or model is not satisfying expected behavior of the system. The feedback approach is allowed to modify the formal model and verify it through formal verification phase and finally validate it using a model checker tool [16,17]. The verification and validation processes are applied continue until not find the correct formal model according to the expected system behavior.

Real-Time Animation Phase

This phase is the new validation technique to verify the formal system in real-time environment using *real-time data* set instead of using *toy-data* set. This phase is applied for rigorous testing of the formal model under domain experts. Real-time animation shows the behaviors of the system using real environment in early phase of the development without generating the source code. Such kind of techniques are very useful when a domain experts are also involved in system development [9].

We give a brief introduction for creating visual animations from formal models. The main contributions of this phase are: these animations are driven by real-time datasets instead of "*toy world*" datasets; the approach allows for simulation and validation early in the development process. The implementation of real-time animation is described in the form of a toolchain [9]. Fig.2 depicts the overall functional architecture that can use the real-time data set to animate the Event-B model without generating source code of the model in any target language (C, C++, VHDL etc.). This architecture has six components, which are described as follows,

• **Data acquisition** is the process of sampling of real world physical conditions and conversion of the resulting samples into digital numeric values. The

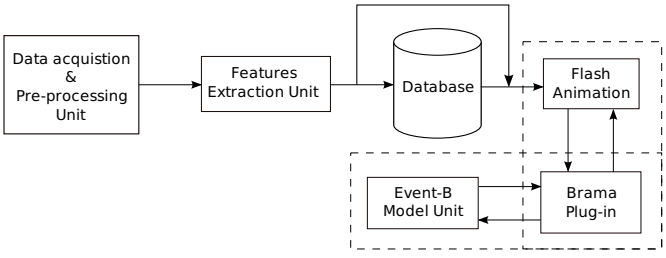


Fig. 2. Real-time animation architecture

data acquisition hardware can vary from environment to environment (i.e camera, sensor etc.).

• **Data preprocessing** transforms the data into a format that will be more easily and effectively processed for the purpose of the user.

• **Features extraction** unit is a set of algorithms that is used to extract the parameters or features from the collected data set. Theses parameters or features are numerical values that are used by animated model at the time of animation.

• **Database** stores the feature or parameter values in the database file in any specific format. This database file of parameters or features can be used in future to execute the model.

• **Macromedia Flash** tool is used to create the animated model of the physical environment and use the Brama plug-in to connect the Flash animation and the Event-B model.

• **Brama** provides the elements required to connect your Flash animation and Event-B model and animating and inspecting a model using Flash animations.

• **Event-B** is a formal method for system-level modeling and analysis using set theory modeling notation with incremental refinement approach at different abstraction levels to verify a given system.

In Fig.1, this phase also gives the feedback to the formalization phase in case of unexpected behavior of the system. The feedback approach is allowed to modify the formal model and verify it using any theorem prover tool and finally validate it using a model checker tool. The verification, validation and real-time animation processes are applied continue until not find the correct formal model according to the expected system behavior. In this phase of the formal development, most of the errors are discovered by the domain experts.

Most simulation researchers agree that animation may be dangerous too, as the analysts and users tend to concentrate on very short simulation runs so the problems that occur only in long runs go unnoticed. Of course, good analysts, who are aware of this danger, will continue the run long enough to create a rare event until not cover all possible events, which is then displayed to the users.

Each phase of the methodology is supported by a specific tool. In the following sections, we show the effectiveness of the methodology using as grand

challenge example a cardiac pacemaker specification in term of obtaining a certification assurance.

3 Case Study: A Cardiac Pacemaker

A challenging problem is offered by Boston Scientific in the area of system specification of a cardiac pacemaker [19]. They have released a specification that defines functions and operating characteristics, identifies system environmental performance parameters, and characterizes anticipated uses. We have used the same specifications to test the effectiveness of our methodology for obtaining trustable formal model that can help to obtain certification assurances, as follows.

Informal Requirements
We start by analysing the specification of the cardiac pacemaker described at [19]. An informal requirement of the cardiac pacemaker is proposed by the software quality research laboratory at McMaster University as a pilot project for the Verified Software Initiative [19,20]. We have used this informal requirement of a cardiac pacemaker for generating the formal specification.

Formalization Phase
Our methodology requires proceeding with the formalization of the categorized requirement fragment version of the requirements produced as artifact of the informal analysis phase. The formalization phase presents formal specification of each requirement fragment identified in the informal requirements documents using Event-B modeling language [6,17].

In one- and two-electrode pacemaker, *pacing* and *sensing* activities are defined abstractly using *action-reaction* [6] and *time patterns* [21]. We apply the actions-reaction and time patterns in modeling to synchronize the sensing and pacing stimulus functions of the cardiac pacemaker in continuous progressive time constraint. We present here only summary informations about each refinement of one- and two-electrode pacemakers and omit detailed formalization and proof details. To find more detailed information see the technical reports [22,23]. The following outline is given about every refinement level to understand the basic formal notion of the cardiac pacemaker model.

We present a block diagram (see Figure 3) of hierarchical tree structure of the possible bradycardia operating modes for a pacemaker. The hierarchical tree structure depicts a stepwise refinement from abstract to concrete model of formal development for a pacemaker. Each level of refinement introduces new features of a pacemaker as functional and parametric requirements.

The root node indicates a cardiac pacemaker system. The next two branches show two classes of pacemaker, namely one-electrode pacemaker and two-electrode pacemaker. The one-electrode pacemaker branch is divided in two parts to indicate different chambers of the heart, namely atrium and ventricular. Atrium and ventricular are the right atrium and the right ventricular. The atrium chamber uses the three operating modes; AOO, AAI and AAT. Similarly, the ventricular

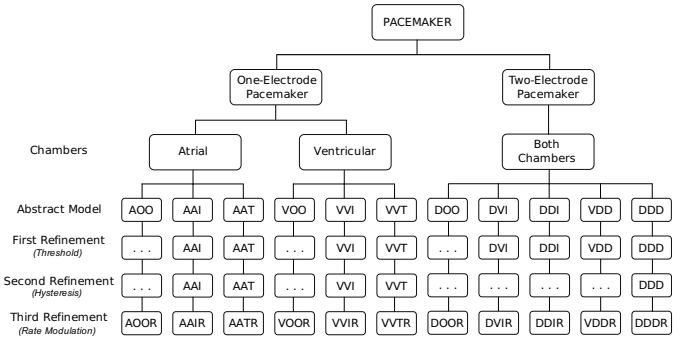


Fig. 3. Refinement structure of bradycardia operating modes of the pacemaker

chamber uses three operating modes: VOO, VVI and VVT. In the part of two-electrode pacemaker, there is only one branch for both chambers. Both chambers of the heart use the five operating modes: DOO, DVI, DDI, VDD and DDD. In the abstract model, we introduce the bradycardia operating modes of the pacemaker abstractly with required properties. From first refinement to last refinement, there is only one branch in every operating modes of the pacemaker. In one and two-electrode pacemaker pacemaker, there are three refinements. First *threshold* refinement; second *hysteresis* refinement; and third *rate adaptive or rate modulation* refinement. The subsequent refinement models introduce new features or functional requirements for the resulting system. The triple dots (...) in the hierarchical tree represents that there is no refinement at that level in particular operating modes (AOO, VOO, DOO etc.). In last refinement level, we have achieved the additional rate adaptive operating modes (i.e. AOOR, AAIR, VVTR, DOOR, DDDR etc). These operating modes are different from the previous levels of operating modes. This refinement structure is very helpful to model the functional requirements of the cardiac pacemaker.

The formal development of the one- and two-electrode cardiac pacemaker is made up of the following models from abstract to concrete models through refinement steps.

Abstract Model: Specifies the *pacing* and *sensing* under real-time properties using *action-reaction* and *real-time* patterns for defining abstractly initial events like *Pace_ON*, *Pace_OFF*,*Sense_ON*, *Sense_OFF* and *tic* events.

Refinement 1: This refinement introduces additional features for filtering the exact sensing value through the pacemaker's sensor by introducing standard threshold constants for both atrial and ventricular chambers, and new events are introduced as refinement of *skip* for capturing the sensors value from the single or both chambers. A pacemaker has a stimulation threshold measuring unit which measures a stimulation threshold voltage value of a heart and a pulse generator for delivering stimulation pulses to the heart. The pulse generator is controlled by a control unit to deliver the stimulation pulses with respective amplitudes related to the measured threshold value and a safety margin.

Refinement 2: This refinement introduces a *hysteresis* operating mode to prevent constant pacing. The *hysteresis* is a programmed feature whereby the pacemaker paces at a faster rate than the sensing rate. For example, pacing at 80 pulses a minute with a hysteresis rate of 55 means that the pacemaker will be inhibited at all rates down to 55 beats per minute, having been activated at a rate below 55, the pacemaker then switches on and paces at 80 pulses a minute [24,25]. The main purpose of hysteresis is to allow the patient to have his or her own underlying rhythm as much as possible. In this refinement only new variables are introduced for applying *hysteresis* operating modes.

Refinement 3: In the final refinement, we describe a rate adapting pacing technique of the cardiac pacemaker. Rate modulation term is used to describe the capacity of a pacing system to respond to physiologic needs by increasing and decreasing pacing rate. The rate modulation mode of the pacemaker can progressively pace faster than the lower rate, but no more than the upper sensor rate limit, when it determines that heart rate needs to increase. This typically occurs with exercise in patients that can not increase their own heart rate. The amount of rate increase is determined by the pacemaker on the basis of maximum exertion performed by the patient. This increased pacing rate is sometimes referred to as the *sensor indicated rate*. When exertion has stopped, the pacemaker will progressively decrease the paced rate down to the lower rate. Two new events are introduced as refinement of *skip* for increasing and decreasing the pacing rate using accelerometer.

Formal Verification Phase

Table 1 shows proof statistics for the formal development of the pacemaker using the RODIN platform [17]. These statistics measure the size of the model, the proof obligations generated and discharged by the RODIN prover, and those are interactively proved. The complete development of the cardiac pacemaker results in 781(100%) proof obligations, in which 674(86%) are proved automatically by the RODIN tool. The remaining 107(14%) proof obligations are proved interactively using RODIN tool. In Event-B models, many proof obligations are generated due to the introduction of new functional behaviors and their parameters (*threshold*, *hysteresis* and *rate modulation*) under real-time constraints.

Table 1. Proof statistics

Model	Total number of POs	Automatic Proof	Interactive Proof
One-electrode pacemaker			
Abstract Model	159	134(84%)	25(16%)
First Refinement	44	40(91%)	4(9%)
Second Refinement	36	24(66%)	12(34%)
Third Refinement	80	80(100%)	0(0%)
Two-electrode pacemaker			
Abstract Model	166	125(76%)	41(24%)
First Refinement	211	190(90%)	21(10%)
Second Refinement	18	15(90%)	3(10%)
Third Refinement	67	66(99%)	1(1%)
Total	781	674(86%)	107(14%)

In order to guarantee the correctness of these functional behaviors, we have established various invariants in stepwise refinement. Proofs are quite simple, and achieved with the help of *do case* operation. Guards of some events are very complex, so for proving invariants and theorems, we simplify guards using *do case*. The stepwise refinement of the cardiac pacemaker helps to achieve a high degree of automatic proofs.

Formal Validation Phase

Model analysis, which is done by the ProB tool, consists in exploring traces or scenarios of our consistent Event-B models. For instance, the ProB [17] may discover possible deadlocks or system behaviors that are not expressed by generated proof obligations. "Validation" refers to the activity of gaining confidence that the developed formal models are consistent with the requirements, expressed in the requirements document [19]. We have used the ProB tool [17] that supports *automated consistency checking* of Event-B machines via model checking [16] and constraint-based checking [18]. ProB is used to validate the Event-B pacemaker formal model. This tool assists us to find potential problems, to improve invariants expressions in our Event-B models, for instance by generating counter-examples when it discovers an invariant violation. ProB may help in improving invariant expression by suggesting hints for strengthening the invariant and each time an invariant is modified, new proof obligations are generated by the RODIN platform. It is the complementary use of both techniques to develop formal models of critical devices, where high safety and security are required. More errors are corrected during the elaboration of the specifications while discharging the proof obligations and careful cross-reading than during the animations. A model animation using model checker helps to discover some unexpected behaviors of the device and this information is used as feedback information to correct the formal specification (see Fig.1). We have validated all operating modes of the pacemaker in each refinement of models. The pacemaker specification is developed and formally proved by the RODIN tool.

ProB was very useful in the development of the pacemaker specification, and was able to animate all of pacemaker models and able to prove the absence of error (no counter-example exist). The ProB model checker also discovered several invariant violations, e.g., related to incorrect responses or unordered pacing and sensing activities.

Real-Time Animation Phase

This phase shows an implementation of real-time data sets with formal model of the pacemaker. Fig.4 represents an implementation of the proposed architecture for the formal model of a cardiac pacemaker case study. According to the architecture, data acquisition unit collects the ECG signal and features extraction are done by the feature extraction or parameter estimation unit. The extracting features are stored in the database in XML file format. Macromedia Flash tool helps to design the animated graphics of the heart and pacemaker. In next unit, Brama plug-in helps to communicate between animated graphics and Event-B formal model of the cardiac pacemaker. Finally, we have tested a *real-time data* set in the formal models without generating the source code with the help of Brama existing animation tool [9].

One- and two-electrode pacemaker's pacing and sensing behaviors are validated through cardiologist experts using real-time data; ECG signal. We found some unexpected behaviors of the model according to the cardiologist experts in visualization. We have modified the pacemaker model according cardiologist experts in this animation phase of our methodology. So, we consider that this

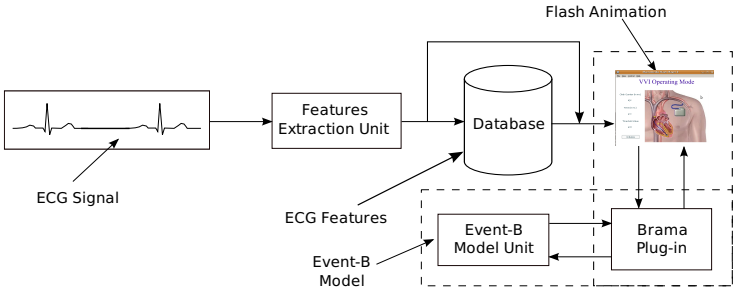


Fig. 4. Real-time animation of cardiac pacemaker

phase is very important role in development of formal methods and it can help to obtain the trustable formal model, which can be helpful to obtain the certification assurance.

4 Benefits of Using Our Proposed Approach

In our methodology, we have provided an architecture to obtain a trustable formal model using formal verification, validation and real-time animation (see Fig.1). Our methodology has the potential for increasing safety of certifying high confidence medical-critical systems. Specific benefits include improving requirements, reducing error introduction, improving error detection, and reducing cost. Secondly, the proposed architecture of methodology allows us to carry out rigorous analyses. Such analyses can verify useful properties such as consistency, deadlock-freedom, satisfaction of high level requirements, correctness of a proposed system design and expected system behavior according to the domain experts using real-time environment in early phase of the development without generating the source code.

Improving Requirements. Using our methodology to capture requirements provides a simple validation check in early stage of medical-system development. Requirements expressed in a formal notation can also be analysed early to detect inconsistency and incompleteness for removing errors that are normally found later in the development process.

Reducing Error Introduction. Formalized requirements prevent misunderstandings due to ambiguities that lead to error introduction. As development proceeds, compliance can be continually checked using a formal analysis to ensure that errors have not been introduced. A further advantage of using our methodology at the requirements level is the ability to derive or refine from these requirements the code itself, thus ensuring that no error is introduced at this stage. Alternatively their use at the requirements level allows formal analysis to establish correctness between requirements and final generated source code of the complex system.

Improving Error Detection. Our methodology can provide exhaustive verification at whatever levels it is applied, high level requirements or low level

requirements. Exhaustive verification means that the whole structure is verified over all possible inputs and states. This can detect errors that would be difficult or impossible to find using only a test based approach.

Reducing Development Cost. Our methodology is based on formal techniques. In general, software errors are less expensive to correct the earlier in the development lifecycle they are detected in high confidence medical devices. The effort required to generate formal models is generally more than offset by the early identification of errors. That is, when formal methods are used early in the lifecycle, they can reduce the overall cost of the project development. When requirements have been formalized the costs of downstream activities are reduced. Formal notations also reduce cost by enabling the automation of verification activities.

5 Conclusion

One valuable byproduct of applying formal methods in software certification is that the process produces a formal specification of the required software behavior. Developing this specification has at least two benefits. First, a formal specification can be valuable when a new version of the software is developed. Second, the process of developing a formal specification by itself may expose errors.

This paper has described methodology to applying formal methods and to achieve correct model for certification of high confidence medical devices. Building on existing software certification standards, such as ISO's 13485 and the Common Criteria [8], more and improved approaches which use formal methods in software certification are needed. Applying these new approaches for highly critical systems should have many benefits; the exposure of errors which might have not been detected without formal methods.

That guidance, as proposed by NITRD [2], allows adoption of formal methods into an established set of processes for development and verification of a high confidence medical device to be an evolutionary refinement rather than an abrupt change of methodology. Formal methods might be used in a very selective manner to partially address a small set of objectives, or might be the primary source of evidence for the satisfaction of many of the objectives concerned with development and verification.

In this article, we have presented evidence of proposed methodology for modeling of a cardiac pacemaker. Formal development of the cardiac pacemaker and refinement based hierarchical structure (see Fig.3) of operating modes are presented in this case study. Most important contribution in certification process is the introduction of real-time animation using *real-time data* set. The pacemaker case study suggests that such an approach can yield a viable model that can be subjected to useful validation against system-level properties at an early stage in the development process. Cardiac pacemaker case study shows the usefulness of this methodology in a high-confidence medical device. This methodology is not limited to only medical devices but it can be applied to design a highly critical device for certification assurance.

Two reliable facts of formal methods have demonstrated by last decades of research and experience - they are not the "*silver bullet*" to eliminate all software failures, but neither are they beyond the budget constraints of software developers. In critical system, formal methods are commonly demonstrating the absence of undesired behaviors and preserving essential properties. A model checkers, theorem provers and real-time animation make it possible to analyses of formal specifications in an automated or semi-automated mode, making these tools for certification use. On the other hand, the ability to generate complete test cases from formal specifications can result in overall savings, despite the cost of developing the specification. The process of developing a specification is often the most valuable phase of a formal verification, and "lightweight formal methods" approaches make it possible to formally analyse partial specifications and early requirements definitions. Experience with mandated use of formal techniques and other standards provides empirical evidence that these methods can be successfully incorporated into the development process for hight confidence medical devices and other critical systems.

References

1. Keatley, K.L.: A review of the fda draft guidance document for software validation: guidance for industry. Qual. Assur. 7(1), 49–55 (1999)
2. A Research and Development Needs Report by NITRD: High-Confidence Medical Devices: Cyber-Physical Systems for 21st Century Health Care, http://www.nitrd.gov/About/MedDevice-FINAL1-web.pdf
3. Bowen, J., Stavridou, V.: Safety-critical systems, formal methods and standards. Software Engineering Journal 8(4), 189–209 (1993)
4. Jetley, R.P., Carlos, C., Iyer, S.P.: A case study on applying formal methods to medical devices: computer-aided resuscitation algorithm. STTT 5(4), 320–330 (2004)
5. Jetley, R., Purushothaman Iyer, S., Jones, P.: A formal methods approach to medical device review. Computer 39(4), 61–67 (2006)
6. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, Cambridge (2010)
7. Magee, J.H.: Validation of medical modeling & simulation training devices and systems. Stud. Health Technol. Inform. 94, 196–198 (2003)
8. Common Criteria, http://www.commoncriteria.org/
9. Méry, D., Singh, N.K.: Real-time animation for formal specification. In: Complex Systems Design & Management (CSDM), Paris (2010)
10. Hoare, T.: The verifying compiler: A grand challenge for computing research. J. ACM 50(1), 63–69 (2003)
11. Goldman, B.S., Noble, E.J., Heller, J.G., Covvey, D.: The pacemaker challenge. CMAJ 110(1), 28–31 (1974)
12. Heitmeyer, C.L.: On the role of formal methods in software certification: An experience report. Electr. Notes Theor. Comput. Sci. 238(4), 3–9 (2009)
13. Heitmeyer, C.L., Archer, M., Leonard, E.I., McLean, J.: Applying formal methods to a certifiably secure software system. IEEE Trans. Software Eng. 34(1), 82–98 (2008)
14. Rushby, J.: Formal methods and their role in the certification of critical systems. Technical report, Safety and Reliability of Software Based Systems (Twelfth Annual CSR Workshop) (1995)

15. Jackson, M.: The problem frames approach to software engineering. In: APSEC, p. 14 (2007)
16. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (1999) ISBN 978-0262032704.
17. Project RODIN: Rigorous open development environment for complex systems: RODIN Toolset and ProB. (2004), http://rodin-b-sharp.sourceforge.net/
18. Jackson, D.: Alloy: a lightweight object modelling notation. ACM Trans. Softw. Eng. Methodol. 11(2), 256–290 (2002)
19. Boston Scientific: Pacemaker system specification, Technical report, Boston Scientific (2007), http://www.cas.mcmaster.ca/sqrl/SQRLDocuments/PACEMAKER.pdf
20. Macedo, H.D., Larsen, P.G., Fitzgerald, J.S.: Incremental development of a distributed real-time model of a cardiac pacing system using vdm. In: Cuellar, J., Maibaum, T., Sere, K. (eds.) FM 2008. LNCS, vol. 5014, pp. 181–197. Springer, Heidelberg (2008)
21. Cansell, D., Méry, D., Rehm, J.: Time constraint patterns for event b development. In: Julliand, J., Kouchnarenko, O. (eds.) B 2007. LNCS, vol. 4355, pp. 140–154. Springer, Heidelberg (2006)
22. Méry, D., Singh, N.K.: Functional behavior of a cardiac pacing system. International Journal of Discrete Event Control Systems 1(2) (in Press, 2010)
23. Méry, D., Singh, N.K.: Technical Report on Formal Development of Two-Electrode Cardiac Pacing System (2010), http://hal.archives-ouvertes.fr/inria-00465061/en/
24. Malmivuo, J.: Bioelectromagnetism. Oxford University Press, Oxford (1995) ISBN 0-19-505823-2
25. Hesselson, A.: Simplified Interpretations of Pacemaker ECGs. Blackwell Publishers, Malden (2003) ISBN 978-1-4051-0372-5

Design Choices for High-Confidence Distributed Real-Time Software

Sebastian Fischmeister and Akramul Azim

Department of Electrical and Computer Engineering
University of Waterloo, Canada
sfischme@uwaterloo.ca, aazim@uwaterloo.ca

Abstract. Safety-critical distributed real-time systems, such as networked medical devices, must operate according to their specification, because incorrect behaviour can have fatal consequences. A system's design and architecture influences how difficult it is to provide confidence that the system follows the specification. In this work, we summarize and discuss three design choices and the underlying concepts that aim at increasing predictability and analyzability. We investigate mandatory resource reservation to guarantee resource availability, separation of resource consumptions to better manage resource inter-dependency, and enumerative reconfiguration. We use the example of a distributed monitoring system for the human cardiovascular system to substantiate our arguments.

1 Introduction

Networked medical devices are good examples of distributed real-time systems with safety-critical functionality. They assist medical staff by automatically measuring physiologic parameters such as blood pressure, oxygen level, and heart rate, or actively influence the patient's parameters by means of infusion pumps for analgesia and insulin or breathing support. As a consequence, incorrect behaviour of the system can result in fatal outcomes for the patient. As such, patients must have confidence that the devices operate according to their specification. One way to establish confidence in the system is by making systems predictable and analyzable, which permits developers and certification authorities to inspect the system before deployment.

Many researchers have looked into the problem of how to make a system predictable and analyzable. By predictable, we mean that an external observer, for example the developer, can predict the system's behaviour with respect to input values and their timing without knowing the internal state. This allows the developer to build a system that implements a specification with strict constraints. By analyzable, we mean that the system can be subjected to formal analysis methods such as model checking which allows the developer to formally check the correct behaviour of the system.

In this work, we summarize and discuss three design choices made in previous and related works that aim at raising the predictability and analyzability of real-time systems. Our contribution is to abstract from these systems and provide

T. Margaria and B. Steffen (Eds.): ISoLA 2010, Part II, LNCS 6416, pp. 327–342, 2010.

a general description of the concept underlying the design choices. This allows developers to quickly understand the choices and adapt the concept for their own system. The following paragraphs introduce the necessary concepts of resources, resource reservation, and resource consumption. We then discuss the three design decisions: mandatory resource reservation (in Section 2), separation of resource consumptions (in Section 3), and enumerative reconfiguration (in Section 4). We illustrate all three concepts with an example of a distributed patient monitoring system for the human cardiovascular system.

Applications require resources to execute. Classical resources include computation time (i.e., access to a processing unit to execute instructions), memory (i.e., temporary or permanent data storage), and communication bandwidth (i.e., access to a shared medium to transmit information to remote stations). One can extend this concept to logical resources such as locks or peripherals.

Before an application can use such resources, it must acquire them. A resource broker mechanism usually provides resources to applications. For some resources, the system implicitly allocates resources to applications. For example, when considering computation time, the dispatcher in the operating system decides at each scheduling point which process is ready to execute. For other resources, the application must explicitly request them. For example, programs usually make system calls to request memory during their execution or to statically request memory at their start time.

Systems can include mechanisms to reserve resources for applications. In such systems, the developer can specify that the system must provide a certain amount of resources to an application. For example, the developer might specify that a station can always receive 50kB/s of communication bandwidth to guarantee that the station can communicate the video stream of the surgical device or other patient data. Resource reservation schemes are well studied across the different resource types and come in great variety. For example, for computation time there are scheduling algorithms (e.g. [1,2,3]) and for communication bandwidth there is quality of service (e.g. [4,5]).

For this work, we assume that the resources are reservable, meaning that we can build a resource reservation policy. For all examples involving networking, we assume that the system consists of a set of stations (e.g., patient monitors, biometric sensing devices, nurse workstation) and they are interconnected through a shared bus network.

2 Mandatory Resource Reservation

Using resources without an appropriate reservation scheme can make systems unpredictable. For example in networks without resource reservation, message transmission time can be unbounded. In Ethernet [6], developers are unable to predict the duration it will take to send an updated value from the sensor to the monitor or an alarm message from the monitor to the nurse station. One problem causing this is the Ethernet capture effect [7] that results in transient or short-term unfairness. This effect leads to incorrect behaviour, because Ethernet was designed to provide fair access to all stations, and during these periods of

unfairness a single station can monopolize the channel. Thus, the developer is unable to predict how long it will take to transmit a message and thus is unable to know whether the system correctly implements a specification that requires a time bound on the transmission delay.

Bandwidth reservation as a means of resource reservation can solve this problem. Using bandwidth reservation, the developer can allocate bandwidth for each station and bound the transmission delay. Several different real-time protocols on top of Ethernet have demonstrated that this is technically feasible by extending the drivers in the stations [8,9,10,11] or switches [12,13,14].

Resource reservation can thus increase the predictability of medical device software. Using resource reservation, developers can rely that sufficient resources will be available for the application whenever it needs these resources. Therefore, the application will never have to wait for the system to free up resources.

Resource reservation can be either *mandatory* (driven by the system) or *discretionary* (driven by the applications). Mandatory means that the system guarantees the resource reservation for applications, and applications are unable to alter these reservations. In contrast to this, discretionary resource reservation allows applications to request more or less resources at run time. For example, the partitioning scheme specified by ARINC 653 [15] and cyclic executives [3] implement mandatory resource reservation. Works such as FTT-Ethernet [10] and RETHER [11] provide discretionary resource reservation, since applications can choose to request changes for their present reservations.

Mandatory resource reservation fits well for networked medical devices. Since mandatory resource reservation prohibits applications from changing their reservations, it is easier to provide evidence on the behaviour of such systems than systems with discretionary resource reservation. Mandatory resource reservation remains static and provides a complete specification of how the broker will distribute resources at run time. The resource reservation itself can then become evidence for establishing confidence in the system's correctness. Examples of this type can be a fully specified time-triggered schedule as found in TTP [5], the dispatch table of a cyclic executive [3], or the tree schedule encoded in a Network Code program [9].

We now exemplify the concept of mandatory resources reservation by looking at our previous work on tree schedules [16,9]. We assume a set of network stations that exchange messages with each other. Stations store messages in queues before they can transmit them. A message can either contain arbitrary contents or a specific variable v. Message ordering in queues is local to each station.

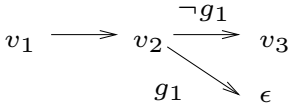
We assume that time is given in discrete units, and that time is measured on a global clock. The communication medium provides an atomic broadcast service, so either all of the stations or none of them receive a message. All messaging behaviours for which developers want to give guarantees are known a-priori.

An informal description of a tree schedule is then a tree structure with a root and a set of leaves where each vertex in the structure specifies a messages to be transmitted and each edge a possible state transition. Edges contain enabling conditions. At run time, for each vertex exactly one edge is enabled at any given

time. Whenever an execution reaches a leaf of the tree, it will loop back to the root. For a formal definition, we refer the reader to related work [16,9].

Figure 1 shows a tree schedule. Labels on vertices show which variable needs to be communicated. An assignment of ϵ means that nobody will transmit. To simplify the example, we assume that each location has a duration of one time unit and that the system is already synchronized.

The system executes the tree schedule as follows: First one station transmits v_1 followed by a message containing v_2. Then, the enabling condition g_1 determines which edge to follow. If $\neg g_1$ holds, then v_3 will be transmitted; otherwise, the system will leave the medium idle for one time unit.



$$g_1 := (v_1 < v_{thr} \wedge v_2 < v_{thr}) \vee (v_1 \geq v_{thr} \wedge v_2 \geq v_{thr})$$

Fig. 1. Example of a tree schedule

Tree schedules can still lead to unbounded communication delays, because the tree schedule itself may encode collisions on the medium and thus force retransmissions. Developers must choose the right type of communication to prevent this. Tree schedules can model and execute two different types of traffic: guaranteed and best effort. Also, developers can increase the level of detail by either communicating individual variables or using general message passing. The difference between these types of communication lies within the ownership of the queues, meaning which stations know the different types of queues.

For example, the communication type of *guaranteed variable updates* will occur, if only one station transmits in that state of the tree schedule, and the transmission is specifically bound to a variable. The update is guaranteed since no other station will transmit and thus the communication will be free of collisions. On the other hand, *best effort messaging* will occur, if more than one station is permitted to transmit data from their send queue in the state of the tree schedule. If more than one station has a message in its send queue, then communication problems such as collisions or packets drop might occur. These different types of communication are visible from the specification of the tree schedule, and the system also directly executes the tree schedule as it gets encoded in the Network Code language [9].

Since the system will execute the tree schedule at run time, developers can use the tree schedule itself and state-space exploration on the schedule as evidence that the system works correctly. In the later sections, we will demonstrate the advantages of hard coded enabling decisions. Here, we only argue that the schedule enables developers to, for example, provide upper bounds on the resource allocations for specific applications. For the tree schedule in Figure 1, the

developer can claim that the variables v_1 and v_2 will always receive bandwidth and stations will always receive updated values every three time units.

3 Separation of Resource Consumptions

Another element reducing predictability and analyzability is the high degree of internal dependencies of resources within programs. A program requires many different resources and uses them as the program code specifies. Consequently, consecutive lines in the program code can use different resources. This causes a dependency between the resources that is hidden in the program. While such dependencies are of no concern in traditional systems, they become a major concern for safety-critical systems, because variations in the use of resources in one line can affect subsequent lines.

```
1  thread_run() {
     float *d=NULL;
3    while ( 1 ) {
       d = malloc(sizeof(float)*10);  // mem: allocating memory
5      acquireFilteredValues(d);      // cpu: computing
       msg_send(d);                   // net: communicating
7      free(d);                       // mem: deallocating memory
       milliSleep(100);               // time: controlling time
9  }}
```

Listing 1.1. Sample C program for computing a value and transmitting it

Listing 1.1 shows a short example of a program that performs a simple task, but it is hard to predict the timing behaviour. The program first allocates memory to read some sensor values. Then, it sends them to another station through the network, and frees the memory again. The program then delays for 100 milliseconds before it repeats this behaviour. Now, the interesting question is: Does the program really send a new message every 100 milliseconds?

Unfortunately, several plausible scenarios can prevent the program from sending a message every 100 milliseconds. The scenarios range from memory allocation to preemption, to collisions on the network to the clock granularity. For example, in Lines 4 and 7, the program executes memory operations. Depending on the current state of the memory manager, allocating memory might take more or less time. For example, the memory manager might need to swap out processes to free a memory frame, it might decide to flush buffered pages, or it might change the resident set sizes for processes. In Line 8, the exact time of the delay depends on the clock granularity supported by the operating system and the actual crystal. The actual duration of the milliSleep() call varies depending on these factors. Worst of all, the individual effects influence each other, so for example, the program might send the message late (see Line 6), because of a delay in the memory allocation. Individual small modification can cause ripple effects throughout the system and manifest at parts of the program. This complicates tracing the effect back to the source.

Industry and academia know this problem and provide approaches for individual effects. For example, work on synchronous languages [17] addressed the

precision problem by reducing reaction intervals to reaction instants. Another work investigated jitter of conventional sleep functions in operating systems [18]. Other work addresses the problem of predictability of execution at the hardware level [19]. Industry uses static allocation of memory and other resources to minimize the dependencies. The Ravenscar profile [20], RavenSPARK [21] and work on MISRA-C [22] provide evidence for this.

With the observation that resource dependencies cause problems, we argue for factoring out the reservation and consumption parts into separate programs. While overall, the number of inter-dependencies remains the same, encapsulating them and joining them through a well-defined mechanism makes the program predictable and more analyzable. The approach mimics divide-and-conquer in that it splits the whole program into several pieces where each piece has a minimal set of resource dependencies—in the optimal case only dependencies for one resource—we make the programs easier to understand and analyze. After the developers specify the pieces, they can join them together for example through specified timed interactions such as timed interfaces [23] and retain predictability. For medical system software, this means that the developers write independent pieces of code with little resource interdependencies and then, for example, join them through a pipe and filter architecture with well-known temporal behaviour.

Several systems have already tried to lower the resource interdependency by encapsulating resource use in separate layers. For example, Giotto [24] separates the value and execution domain. In this system, the reading and writing of values is independent of the program execution. Changing one does not necessarily require changes in the other. TTP [5], similar to other communication protocols, separates the communication from the execution domain. A schedule defines when nodes send and receive messages which are independent when tasks running on the nodes produce new values. In PEACOD [25], the authors provide a framework for specifying resource consumptions for small pieces of code to provide compositionality and predictable behaviour for multiple resources.

In the following, we show how we separate computation and communication in the Network Code framework [9] that implements tree schedules. Figure 2 provides the overview of the architecture. Computation tasks on the top implement the application logic. The communication tasks on the bottom implement the communication behaviour. Both layers interact through buffers and queues. The typical data flow is as follows: the computation tasks produce new data and write it into the buffers. The communication tasks read and encapsulate this data in messages and transmit them on the communication medium. At the remote station, the communication tasks will receive the messages and write their contents into the buffers. Finally, the computation tasks at the remote station will process the new data.

The computation tasks can only use computation and associated resources such as memory. Computation tasks never directly access the medium. Communication tasks only use the communication medium as a shared resource, all other resources need to be provided separately.

Building such a system is feasible and robust [26] as the hardware implementation shows. It isolates the computation part from the communication part in

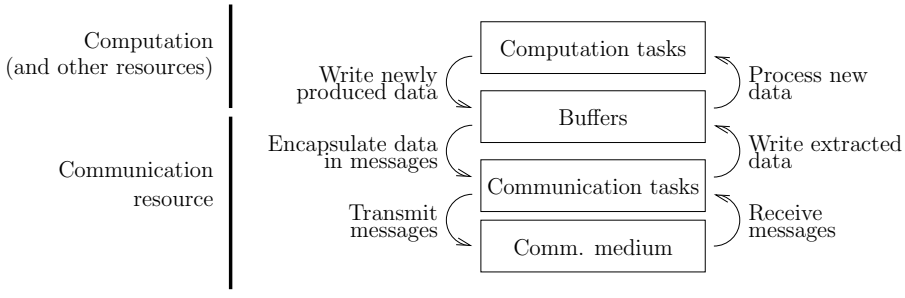


Fig. 2. Overview of the Network Code framework

the program. Since we use tree schedules to specify the communication tasks, we can verify the communication behaviour on the shared medium and for example performing static checks for collisions, buffer underrun, buffer overruns, sender/receiver pairing, and incorrect messaging lifecycle.

For the example shown in Listing 1.1, the developer will specify a communication task on the sending station that reads the value of `d` from the buffers and transmits it precisely every 100 milliseconds. The receiving stations will run the matching tasks that receive the transmitted value and store it in the buffers. On the computation layer, tasks will now only be concerned with memory and computation resources, which the developer can easily resolve by statically allocating the memory and then performing schedulability analysis for the computation parts. We acknowledge that such a system still contains jitter caused by, for instance, hardware effects, however, we argue that the developers can place more confidence in the system. This increase in confidence originates from the better handling of the dependencies and using the interaction between the tasks and the buffers as well as the tree schedule as evidence.

4 Enumerative Reconfiguration

Reconfiguration in systems has been shown to allow developers to build systems that can adapt to new use cases, increase system survivability [27], and improve efficiency in the use of system resources [28]. Any mass-produced safety-critical device benefits from these properties. Medical devices also benefit, as reconfiguration enables an integrated clinical environment [29] which improves service quality and reduces cost. However, the increase in complexity by providing a reconfiguration mechanism must not compromise the system's correctness, so an important question is how to provide the reconfiguration mechanism and still establish confidence in the system's correctness.

We want systems to be reconfigurable, but without knowing whether the system works in a different configuration, the system will be unusable for safety-critical applications. Reconfigurable systems can either be space constrained (bounded state-space) or unconstrained (unbounded state-space). In general, space unconstrained reconfiguration schemes provide more flexibility but are

unable to provide evidence that the system behaves as it should, which makes it hard to provide guarantees on the behaviour of the systems. For this reason, we argue that reconfiguration must be bounded by constraints.

Bounding the reconfiguration space can either use a constraint-based or an enumerative approach. In the constraint-based approach, the developer specifies constraints at a high-level within which the system can choose its point of operation. For example, the developer can specify a range of acceptable data rates for transmitting the patient's parameters to the monitor. Then at run time depending on the actual situation the system will choose a data rate within the range. The advantage of constraint-based reconfiguration is that it provides a large state space within which the system can choose the best point of operation for the current situation. In the enumerative approach, the developer exhaustively lists all possible configurations and the system picks one configuration at run time. In the example with the data rates, the developer will, for instance, specify three possible data rates and the system will select one of the three rates. Systems with multiple modes of operation usually provide exhaustive lists in which a mode usually realizes a system's functionality for a particular configuration.

One advantage of using the enumerative approach instead of the constrained-based approach is the guarantee that the system supports reconfiguration but remains analyzable. If system's state-space is small, then verifying the system will be tractable. However, the size depends on the constraints and the application, and a system with loose constraints can easily run into the state-space explosion problem. In the enumerative approach, the state space is automatically constrained by the requirement to list all possible reconfigurations.

Note that fast checks for safe reconfiguration are available [30,31], however, they only apply to the resource reservation parts and developers still need to establish confidence in the functional correctness for all possible configurations once they have sufficient resources.

Several systems support reconfiguration. For example TTP/C [5] supports up to 30 modes with a safe mode-change protocol. Tree schedules [9] can encode modes in the structure that can have safe transitions. Endochronous clocked graphs [32] can encode different modes similarly to tree schedules.

Section 2 demonstrates how we can encode different configurations in tree schedules. The tree schedule shown in Figure 1 includes two modes of operation: one where values v_1 and v_2 agree and the other where the two values disagree. Similar to this configuration, we can encode the list of configurations in the tree structure and verify properties as already mentioned in the previous sections.

5 Illustrative Example

We use the following example to summarize and substantiate our point about the three concepts mentioned in the previous sections. The aim is to integrate all three concepts into one example and show how one can provide evidence using formal verification. We go through the following steps of building the system: (1) we build the resource reservation scheme using tree schedules, (2) we provide evidence that the resource reservation works by means of formal verification enabled by the separation of resource consumptions and the enumerative

reconfiguration, and (3) we show the simulation framework for tree schedules in Matlab Simulink to test tree schedules before deployment.

5.1 Overview

We assume a distributed patient monitoring system in which body sensors transmit physiological parameters to the patient monitor. When the pulmonary vascular resistance (PVR) of the patient passes a given threshold, the patient monitor will send an alarm message to the nurse station within bounded time.

PVR [33] is the resistance in the pulmonary vascular bed against which the right ventricle must eject blood. To calculate the pulmonary vascular resistance, the patient monitor requires the left atrial pressure (LAP) or the pulmonary capillary wedge pressure (PCWP), the pulmonary artery pressure (PAP), and the cardiac output (CO). PCWP provides an indirect estimate of LAP. PCWP is measured by wedging a catheter into a small pulmonary artery tightly enough to block flow from behind. LAP can be measured by placing a special catheter into the right atrium and then pushing through the inter-atrial septum. Since the patient monitor only requires the LAP or the PCWP, we can create several modes for the operation of the monitor:

- Configuration 1: The patient monitor uses the PAP, CO, and LAP.
- Configuration 2: The patient monitor uses the PAP, CO, and PCWP.
- Configuration 3: The patient monitor uses the PAP, CO, and LAP. If an alarm is pending, then the monitor will make a safety check and also acquire the PCWP, before signaling the nurse alarm. This will lower the number of false alarms as it eliminates the problem of incorrect LAP measurements.
- Configuration 4: This is similar to configuration 3 but the patient first uses PAP, CO, and PCWP, and then uses LAP for the fail safe.

We treat calculating the pulmonary vascular resistance as a single transaction. This means that the system should always complete all data transmissions that the patient monitor requires before reconfiguring (e.g., changing configuration). This assumption is important, because we model setting the configuration with a physical button which the nurse can press with a frequency of at most once in a fixed amount of time. In addition, the patient monitor must signal the nurse alarm within a bounded time when the pulmonary vascular resistance exceeds a specific threshold.

5.2 Developing the Tree Schedule

Based on the specification in Section 5.1, we can develop the tree schedule for the resource reservation. We assume that communicating one value takes one time unit, and the inter-arrival time of button pressed events is set accordingly. Figure 3 shows the tree schedule that implements the specification (or so we claim). A vertex labeled ϵ takes zero time and we use it to encode branches with more than two choices or for early termination of the schedule. The PVR monitoring station can operate in four configurations. In any of the four configurations, the monitoring system at first receives the value of the PAP from the circulatory

system to calculate the PVR. If the received value is out of the normal range for PAP values (i.e. 10-20 $mmHg$), the system will enter the *safety interlock* state. In the *safety interlock* state, the system checks the important functions of the human cardiovascular system such as the patient's pulse rate while resting (60-100 beats per minute) to determine the patient's safety. This assumption is implicit and not shown in the Figure 3. After receiving the value of CO within the normal range (4 L/min-8 L/min), the system can either receive LAP (normal range 6-12 $mmHg$) or PCWP (normal range 6-12 $mmHg$) based on the current configuration. The patient monitor will receive PCWP after PAP, if the system uses configuration 3. On the other hand, the monitor will receive LAP after the PAP, if the system runs in the default configuration (i.e., any configuration other than 1, 2, and 3). The system will enter into the safety interlock state for out of the normal range of CO, LAP, or PCWP. The system will generate an alarm and notify the nurse, when the PVR exceeds normal value (> 250 $dyn.s/cm^5$). The nurse can change the configuration of the monitoring system at any point in time, but not in the middle of a transaction.

Guards g_1 and g_2 define the enabling conditions whether the PVR value of the patient exceeds the defined threshold thr. Guards g_3 to g_6 are enabling conditions depending on the configuration setting. We assume configuration 4 to be the default configuration.

For demonstration purposes, we walk through one configuration for which we assume $conf = 4$ and PVR $\geq thr$. In the root location labeled ϵ_0, only g_6 will be enabled. The tree schedule specifies that the next three messages on the bus will be PAP, CO, and PCWP. At that point PVR exceeds the threshold thr (PVR $\geq thr$), so g_1 is true and the patient monitor will also receive the LAP measurement. Finally, g_1 will again be true and the patient monitor will signal the nurse alarm before the tree schedule restarts at its root location.

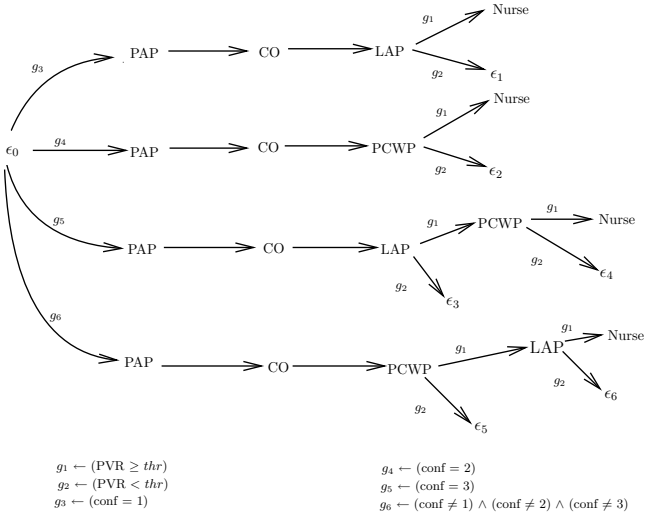


Fig. 3. The tree scheduling for the patient monitoring system

5.3 Verifying the Tree Schedule

To provide evidence that our system meets the specification with respect to the communication requirements, we provide the following guarantees for our reservation mechanism. Note that we can verify these properties, because we separate communication from computation in our framework (see Section 3) and we can enumerate all configurations (see Section 4).

- **P1:** In every t time units and in all configurations, the PVR monitoring system will receive all data necessary to compute and display the new PVR value.
- **P2:** When PVR $\geq thr$, then the nurse will be notified no later than x time units in all configurations.
- **P3:** Calculating PVR is atomic and when the system is in a particular configuration mode, it is not possible to switch to different modes.
- **P4:** The system will always make progress and never gets stuck.

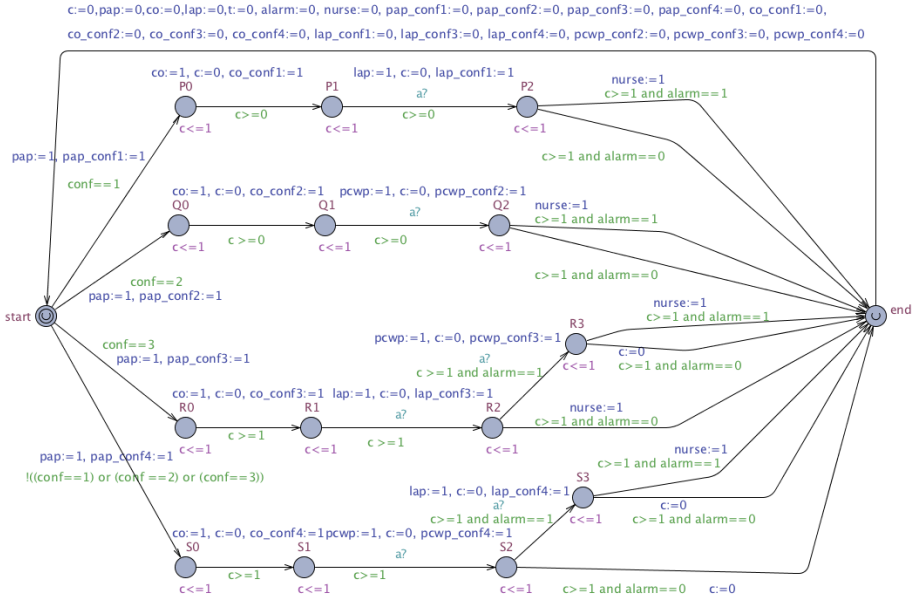


Fig. 4. Modeling PVR monitoring system in UPPAAL

To provide evidence that these properties hold in our system, we encode the tree schedule in a timed automaton and check the properties using UPPAAL. UPPAAL [34] is a timed-automata based model checker that allows formal verification of temporal logic properties in finite systems. Figure 4 shows the tree schedule part of the UPPAAL model. The whole system comprises three different processes: one modeling the tree schedule, one modeling the nurse, and

one modeling the alarm condition. All three processes run in parallel. The nurse process can alter the conf variable at most once every time unit. The nurse alarm will sound, if the variable nurse is set to one. We use channel a to synchronize the alarm process with the tree schedule process. The alarm process implements the non-deterministic choice whether an alarm happened or not. In Figure 4, the clock c constraints state changes (one transmission requires one time unit), the clock t counts for the cycle, and the clock tall always increases. We use t and tall for verification purposes only. We can now check the properties defined above using UPPAAL's query language:

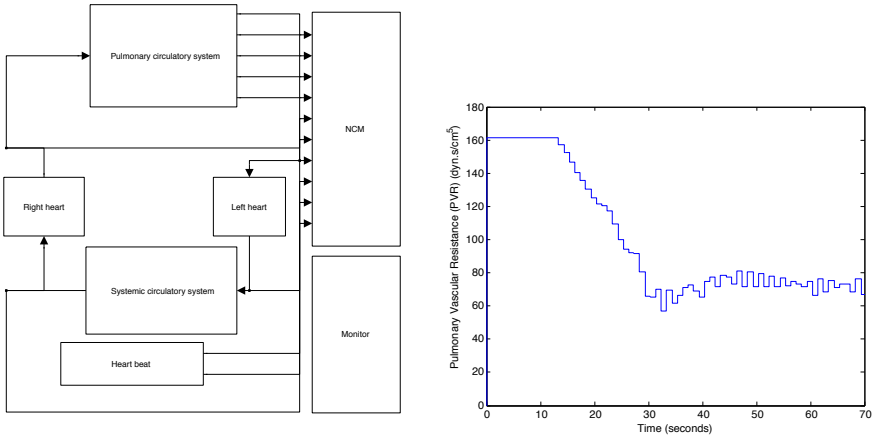
- **P1:** $A\Box(P.end \rightarrow (pap = 1 \wedge co = 1 \wedge lap = 1) \vee (pap = 1 \wedge co = 1 \wedge pcwp = 1)) \wedge (P.end \rightarrow (t \leq 5))$: Whenever the system reaches the end, PAP, CO, and LAP or PCWP have been transmitted. And, the system always reaches the end withing *five* time steps.
- **P2:** $A\Box((adly \geq 3) \wedge (adly \leq 4) \rightarrow ((alarm = 0) \vee (nurse = 1)))$: The system will notify the nurse within three to four time units after an alarm happened.
- **P3:** $A\Box((pap\_conf1 = 1) \vee (co\_conf1 = 1) \vee (lap\_conf1 = 1)) \rightarrow \neg((pap\_conf2 = 1) \vee (co\_conf2 = 1) \vee (pcwp\_conf2 = 1) \vee (pap\_conf3 = 1) \vee (co\_conf3 = 1) \vee (lap\_conf3 = 1) \vee (pcwp\_conf3 = 1) \vee (pap\_conf4 = 1) \vee (co\_conf4 = 1) \vee (lap\_conf4 = 1) \vee (pcwp\_conf4 = 1))$: When the system is in configuration 1, the system cannot enter into configuration 2, 3, or 4. Therefore, The system cannot be switched to different configurations while it is in a configuration mode.
- **P4:** $A\Box\neg deadlock$: The system will always make progress and not deadlock.

Note, compare the complexity involved in checking such properties for general programs that mix computation and communication in a programming language like C. We can easily now connect the communication layer with the computation layer through buffers and a specification when values get read and written in these buffers (as we have done in [9]). Also note that we generate the schedule from high-level specifications [35].

5.4 Simulating the System

We use Simulink to simulate our patient monitoring system. We implement the patient monitor and connect it to a model of a human cardiovascular system [36] using TrueTime [37]. TrueTime supports simulating network communication for real-time control systems. The human cardiovascular system implements a heart model and produces different physiological parameters of the heart.

In our simulation, we implement the tree schedule defined in Section 5.2. Figure 5(a) provides an overview of the Simulink model and Figure 5(b) shows the resulting PVR value of a sample run of the simulation. The human cardiovascular system abstracts the implanted body sensors that report PAP, LAP, CO, and PCWP to the external PVR monitoring system. The tree schedule runs inside the network code machine (NCM) implemented on top of TrueTime. We can implement the tree schedule inside a state machine using the StateChart block.



(a) Human cardiovascular system model in Simulink

(b) Monitoring the PVR

Fig. 5. Simulation of tree schedules for the human cardiovascular system example

The human cardiovascular system components connect with the TrueTime network, and the monitoring system receives the physiological parameters through the network. The basic elements of TrueTime are the TrueTime send block, the TrueTime network block, and the Network Code Machine block that implement the tree schedule. The monitoring system use TrueTime receive blocks to receive data over the network. If we enter the subsystems of the system model, we will see the detailed interactions of Matlab, Simulink, and TrueTime elements for each subsystem. Before starting the simulation, we set the parameters of different elements of the system model such as network type, number of nodes, data rate and frame size in the TrueTime network block.

6 Conclusion

In this work, we discussed three useful concepts following design decisions that aim at increasing the predictability and analyzability of real-time systems: resource reservation, different types of resource consumptions, and constrained reconfiguration. Our work on tree schedules created the guiding example for each of these three mechanisms. Finally, we showed an illustrative example of a distributed patient monitoring system in which we went through the phases of specifying, checking, and finally simulating the system.

The mentioned concepts open up many avenues for future work. One can explore each of the mentioned concepts in more detail for different resource types and investigate how to join them together into one framework for multiple resources. Another interesting problem is compositionality of resource consumptions and evidence. This points to the question of how can one place confidence in the whole system from having evidence that all the individual modules work as specified.

Acknowledements

We would like to thank the members of the Embedded Software group at the University of Waterloo for their feedback on an early draft as well as Hiren Patel for in-depth discussions and Steven Dain for information on the medical parts of the work. This research was supported in part by NSERC DG 357121-2008, ORF RE03-045, and ISOP IS09-06-037.

References

1. Buttazzo, G.: Hard Real-Time Computing Systems. Kluwer Academic Publishers, Dordrecht (2000)
2. Leung, J. (ed.): Handbook on Scheduling. CRC Press, Boca Raton (2004)
3. Liu, J.: Real-Time Systems. Prentice-Hall, New Jersey (2000)
4. Coulouris, G., Dollimore, J., Kingberg, T.: Distributed Systems: Concepts and Design. Queen Mary and Westfield College, University of London (1996)
5. Kopetz, H.: Real-time Systems: Design Principles for Distributed Embedded Applications. Kluwer Academic Publishers, Dordrecht (1997)
6. Metcalfe, R.M., Boggs, D.R.: Ethernet: distributed packet switching for local computer networks. Commun. ACM 19(7), 395–404 (1976)
7. Ramakrishnan, K., Yang, H.: The Ethernet Capture Effect: Analysis and Solution. In: Proc. 19th Local Computer Networks Conference (1994)
8. Court, R.: Real-time Ethernet. Comput. Commun. 15(3), 198–201 (1992)
9. Fischmeister, S., Sokolsky, O., Lee, I.: A Verifiable Language for Programming Communication Schedules. IEEE Transactions on Computers 56(11), 1505–1519 (2007)
10. Pedreiras, P., Almeida, L., Gai, P.: The FTT-Ethernet protocol: merging flexibility, timeliness and efficiency. In: Proceedings of the 14th Euromicro Conference on Real-Time Systems (ECRTS), pp. 134–142. IEEE Press, Los Alamitos (June 2002)
11. Venkatramani, C., Chiueh, T.: Supporting real-time traffic on Ethernet. In: Proceedings of Real-Time Systems Symposium (RTSS), pp. 282–286. IEEE Press, Los Alamitos (December 1994)
12. Carvajal, G., Fischmeister, S.: A TDMA Ethernet Switch for Dynamic Real-Time Communication. In: Proc. of the 18th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), Charlotte, United States (May 2010)
13. Jasperneite, J., Neumann, P., Theis, M., Watson, K.: Deterministic Real-Time Communication with Switched Ethernet. In: Proceedings of 4th IEEE International Workshop on Factory Communication Systems, WFCS (2002)
14. Steinhammer, K., Grillinger, P., Ademaj, A., Kopetz, H.: A Time-Triggered Ethernet (TTE) Switch. In: Proceedings of the Conference on Design, Automation and Test in Europe (DATE), 3001 Leuven, Belgium, Belgium, European Design and Automation Association, pp. 794–799 (2006)
15. Aeronautical Radio, I.A.: ARINC 653 (Avionics Application Standard Software Interface). ARINC Standard (2003)
16. Anand, M., Fischmeister, S., Lee, I.: Composition Techniques for Tree Communication Schedules. In: Proceedings of the 19th Euromicro Conference on Real-Time Systems (ECRTS), Pisa, Italy, pp. 235–246 (July 2007)

17. Halbwachs, N.: Synchronous Programming of Reactive Systems. Kluwer, Dordrecht (1997)
18. Dubey, A., Karsai, G., Abdelwahed, S.: Compensating for Timing Jitter in Computing Systems with General-Purpose Operating Systems. In: Proceedings of the IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), Tokyo, Japan (March 2009)
19. Lickly, B., Liu, I., Kim, S., Patel, H., Edwards, S., Lee, E.: Predictable Programming on a Precision Timed Architecture. In: Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES), pp. 137–146. ACM, New York (2008)
20. Dobbing, B., Burns, A.: The Ravenscar Tasking Profile for High Integrity Real-time Programs. In: Proceedings of the 1998 annual ACM SIGAda international conference on Ada (SIGAda), pp. 1–6. ACM, New York (1998)
21. Systems, P.C.: SPARK 95 - The SPADE Ada 95 Kernel (including RavenSPARK). RavenSPARK S.P0468.73.62 version 4.8 (January 2008)
22. McCall, G.: Misra-C: 2004. MIRA Limited, Warwickshire, United Kingdom (2004)
23. de Alfaro, L., Henzinger, T., Stoelinga, M.: Timed Interfaces. In: Sangiovanni-Vincentelli, A.L., Sifakis, J. (eds.) EMSOFT 2002. LNCS, vol. 2491, pp. 108–122. Springer, Heidelberg (2002)
24. Henzinger, T.A., Kirsch, C.M., Horowitz, B.: Giotto: A Time-triggered Language for Embedded Programming. In: Henzinger, T.A., Kirsch, C.M. (eds.) EMSOFT 2001. LNCS, vol. 2211. Springer, Heidelberg (2001)
25. Anand, M., Fischmeister, S., Lee, I.: Resource Scopes: Toward Language Support for Compositional Determinism. In: Proceedings the 12th IEEE International Symposium on Object/component/service-oriented Real-time Distributed Computing (ISORC), Tokyo, Japan, pp. 295–304 (May 2009)
26. Fischmeister, S., Trausmuth, R., Lee, I.: Hardware Acceleration for Conditional State-Based Communication Scheduling on Real-Time Ethernet. IEEE Transactions on Industrial Informatics 5, 3 (2009)
27. Shelton, C., Koopman, P.: Improving System Dependability with Functional Alternatives. In: Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN 2004), p. 295. IEEE Computer Society, Los Alamitos (2004)
28. Buttazzo, G.C., Lipari, G., Caccamo, M., Abeni, L.: Elastic Scheduling for Flexible Workload Management. IEEE Transactions on Computers 51(3), 289–302 (2002)
29. Schrenker, R.: Software engineering for future healthcare and clinical systems. Computer 39(4), 26–32 (2006)
30. Real, J., Crespo, A.: Mode Change Protocols for Real-Time Systems: A Survey and a New Proposal. Real-Time Systems 26(2), 161–197 (2004)
31. Almeida, L., Anand, M., Fischmeister, S., Lee, I.: A Dynamic Scheduling Approach to Designing Flexible Safety-Critical Systems. In: Proceedings of the 7th Annual ACM Conference on Embedded Software (EMSOFT), Salzburg, Austria, pp. 67–75 (October 2007)
32. Potop-Butucaru, D., de Simone, R., Sorel, Y., Talpin, J.: Clock-driven Distributed Real-time Implementation of Endochronous Synchronous Programs. In: Proceedings of the 7th ACM International Conference on Embedded Software (EMSOFT), pp. 147–156. ACM, New York (2009)
33. Abbas, A.E., Fortuin, F.D., Schiller, N.B., Appleton, C.P., Moreno, C.A., Lester, S.J.: A Simple Method for Noninvasive Estimation of Pulmonary Vascular Resistance. Journal of the American College of Cardiology 41(6), 1021–1027 (2003)

34. UPPAAL—An Integrated Tool Environment for Modeling, Validation, and Verification of Real-Time Systems, http://www.uppaal.com (visited June 2010)
35. Potop-Butucaru, D., Azim, A., Fischmeister, S.: Semantics-preserving Implementation of Synchronous Specifications over Dynamic TDMA Distributed Architectures. In: Proceedings of the 10th International Conference on Embedded Software, EMSOFT (2010)
36. Hu, Z., Diao, Y.: Primary Model of Heart-systemic-pulmonary System. Journal of Tongji University 30(1), 61–65 (2002)
37. Henriksson, D., Cervin, A., Årzén, K.E.: TrueTime: Real-time Control System Simulation with MATLAB/Simulink. In: Proceedings of the Nordic MATLAB Conference (2003)

Assurance Cases in Model-Driven Development of the Pacemaker Software*

Eunkyoung Jee, Insup Lee, and Oleg Sokolsky

PRECISE Center
Department of Computer and Information Science
University of Pennsylvania, Philadelphia PA 19104, USA
eunkjee@seas.upenn.edu, {lee,sokolsky}@cis.upenn.edu

Abstract. We discuss the construction of an assurance case for the pacemaker software. The software is developed following a model-based technique that combined formal modeling of the system, systematic code generation from the formal model, and measurement of timing behavior of the implementation. We show how the structure of the assurance case reflects our development approach.

Keywords: assurance case, pacemaker challenge, model-driven development, real-time software.

1 Introduction

We consider the problem of developing an assurance case for the real-time cardiac pacemaker software, representative of life-critical systems in which many complex timing constraints are imposed. This work was motivated by the Pacemaker Grand Challenge, the first certification challenge problem issued by the Software Certification Consortium (SCC) [1]. Boston Scientific has released into the public domain the system specification for a previous-generation pacemaker to have it serve as the basis for a challenge to the formal methods community. In [2], we proposed a safety-assured approach for the development of pacemaker software. In this paper, we consider how the features of our development process are reflected in the structure of the assurance case.

When we develop a real-time system, guaranteeing timing properties on its implementation is an important but non-trivial issue. It becomes essential if the real-time system is a safety-critical one in which violation of timing properties can result in loss of life. We focus on how to systematically implement time-guaranteed real-time software from a given model and how to convincingly demonstrate the safety of the software.

Several concepts and approaches can be effectively integrated to contribute to the development of safety-assured real-time software. The model-driven development (MDD) approach is steadily gaining popularity in the development of embedded software. According to the MDD concept, we create a formal model

* This research was supported in part by NSF CNS-0834524 and NSF CNS-0930647.

T. Margaria and B. Steffen (Eds.): ISoLA 2010, Part II, LNCS 6416, pp. 343–356, 2010.

of the real-time system, verify the model, and generate an implementation code from it. In order to validate the result and check the timing constraints on the implementation, we perform measurement-based timing analysis on the implementation and revise the implementation and the model according to the timing analysis result, repeating the verification process if necessary.

Many safety critical systems, such as avionics systems and medical devices, are subject to regulatory approval. Once the system is implemented, it is necessary to present development documentation to the regulators for review. Currently, this process is lengthy and expensive. Certification costs constitute a significant fraction of the development costs for regulated systems.

Assurance cases are currently seen to be holding a promise of both reducing certification costs *and* improving the quality of certification by tying it to the evidence. An assurance case is a documented body of evidence that provides a convincing and valid argument that a specified set of critical claims about a system's properties are adequately justified for a given application in a given environment [3]. Yet, there are few commonly accepted ways of constructing assurance cases. There is evidence that a poorly structured assurance case can hamper the evaluation process, rather than help it [4]. Clearly, there is no "one size fits all" structure, and software developed through different processes is likely to require different arguments about its safety. The case study put forth in this paper aims to discover appropriate structures for one development approach, namely model-driven development.

The contribution of this paper is the construction of an assurance case for real-time software developed using a model-driven safety-assured process based on formal modeling, rigorous code generation from the verified model, and subsequent validation of the timing characteristics of the developed code. We believe that other model-driven development frameworks will be amenable to similarly structured assurance cases. Our ultimate goal is to arrive at an accepted assurance case template that can be applied to a variety of safety-critical software-based systems. Having such a template will simplify regulatory approval of these systems, by making the argument easier for the evaluators to follow. While this goal still lies ahead of us, this work can be seen as the first step in the right direction.

The remainder of the paper is organized as follows: Section 2 explains the background of the case study. Section 3 presents the overview of our development process and demonstrates its application to the development of the pacemaker software. Section 4 presents the assurance case for the pacemaker software in its relation to the evidence generated during the development. We discuss related issues in Section 5 and present a review of previous work related to topics addressed in this paper in Section 6. Section 7 concludes the paper.

2 Pacemaker Operation

2.1 Heart

A human heart has four chambers: right and left atria, and right and left ventricles. De-oxygenated blood from the body is collected in the right atrium and

then pumped into the lungs via the right ventricle. In the lungs, carbon dioxide in the blood is replaced with oxygen. This oxygenated blood then passes through the left atrium and enters the left ventricle, which pumps it out to the rest of the body.

From an electrical point of view, the heart is a pump made up of muscle tissue, controlled by an intrinsic electrical system. An electrical stimulus generated periodically (normally about 60-100 times per minute) by the sinus node, located in the right atrium, travels through the conduction pathways and causes the heart's chambers to contract and pump out blood. The atria are stimulated and contract shortly before the ventricles are stimulated and contract.

Under some conditions, this intrinsic cardiac system does not work properly and the heart rate becomes overly fast or slow, or irregular. In these situations, the body may not receive enough blood, which causes several symptoms such as low blood pressure, weakness, and fatigue. To avoid these symptoms, a pacemaker can be used to regulate the heartbeat [5].

2.2 Pacemaker

A cardiac pacemaker is an electronic device implanted into the body to regulate the heart beat by delivering electrical stimuli over leads with electrodes that are in contact with the heart. These stimuli are called *paces*. The pacemaker may also detect natural cardiac stimulations, called *senses*. We refer to cardiac paces and senses collectively as *events*.

A pacemaker must satisfy three fundamental medical requirements: the rate at which the cardiac chambers contract must not be too high; the rate at which the cardiac chambers contract must not be too low; the ventricles must contract at a particular interval after the atria contract. These general requirements are concretized by setting specific values or ranges to configurable parameters for the pacemaker.

The pacemaker can operate in a number of modes, distinguished by which chambers of the heart are sensed and paced, how sensed events will affect pacing, and whether the pacing rate is adapted to the patient state. In this paper, we concentrate on the VVI mode, in which the pacemaker senses only ventricular contractions and performs only ventricular pacing. In this mode, pacing is inhibited if ventricular contractions are sensed.

A pacemaker in the VVI mode operates in a timing cycle that begins with a paced or sensed ventricular event. The basis of the timing cycle is the *lower rate interval* (LRI), which is the maximum amount of time between two consecutive events in one chamber. If the LRI elapses and no sensed event occurs since the beginning of the cycle, a pace is delivered and the cycle is reset. At the beginning of each cycle, there is a *ventricular refractory period* (VRP), usually 200-350 ms. Chaotic electrical activity in the heart immediately following a pace may lead to spurious detection of sensed events that can interfere with future pacing. For this reason, sensing is disabled during the VRP period. Once the VRP period is over, a sensed ventricular event inhibits the pacing and resets the LRI, starting the new timing cycle. Hysteresis pacing can be enabled in the VVI mode, when

the pacemaker will delay pacing beyond the LRI to give the heart a chance of resuming normal operation. In that case, the timing cycle is to a larger value, namely the *hysteresis rate interval* (HRI). In our implementation, hysteresis pacing is applied after a ventricular sense is received, and disabled after sending a pacing signal.

3 Model-Driven Development of Pacemaker Software

3.1 Overall Process

We propose a safety-assured development process for real-time software. The proposed process follows a model-driven development approach with the emphasis on ensuring that the implementation satisfies timing properties that are satisfied in the model. Fig. 1 shows the overall process.

During the requirements and design phases of the software life cycle, developers first start from formal modeling with timed automata of the real-time software. Second, model checking is performed on the timed automata model with respect to desired properties using a real-time model checker such as UPPAAL. We focus on safety properties, especially timing properties which require that a certain event should happen no later than a specific delay. Given a verified formal model, an implementation code is synthesized in the third step.

In the fourth step, we check to see if the same properties checked on the model are still satisfied by the code running on a target platform. If some timing properties are not satisfied by the code, we measure how much actual time deviates from the expected. During the fourth step, we find a *timing tolerance* value, Δ, through the measurement-based timing analysis. Guards in the code are modified with this Δ to make the code satisfy timing properties. Once it is confirmed that the code satisfies the desired timing properties with the Δ, changes of the code, i.e., modified guards with the Δ, are reflected to the model in the fifth step. If the modified model still satisfies all the properties, the overall process ends. Otherwise, the process is repeated by revising the problematic model and the code. We describe each step with the pacemaker example in the following subsections.

3.2 Formal Modeling

We used the Boston Scientific's system specification for a pacemaker [6]. Because timing constraints are so prevalent in the specification of the pacemaker, it is intuitive and straightforward to use timed automata [7] as our modeling language. Here we use the UPPAAL tool [8] to specify a timed automata model of the pacemaker in VVI mode.

We extracted properties to be satisfied by the VVI mode pacemaker from the system specification. LRI, HRI, and VRP are considered the most important timing periods which should be guaranteed by the VVI mode pacemaker. Fig. 2 shows two automata for Ventricle and Heart, representing the controller for ventricular pacing in the VVI mode and a heart model as the environment for model

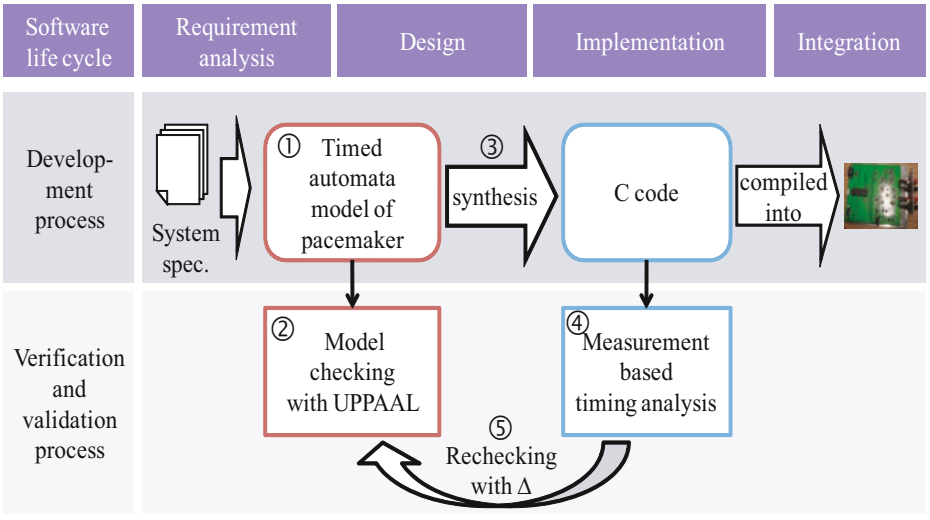


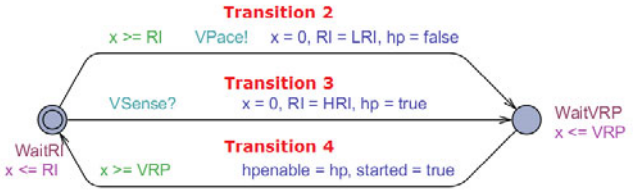
Fig. 1. Overall process of a safety-assured development for a real-time pacemaker software

verification, respectively. Our heart model is the most permissive environment that is ready to accept a pacing signal whenever it is sent and can choose to deliver a sensing signal at any time.

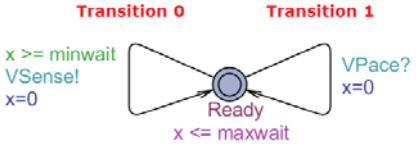
The Ventricle automaton shown in Fig. 2(a) represents sensing signals from the ventricle and emission of ventricular pacing signals to the heart, according to the LRI, HRI, and VRP timing periods. Values of these intervals are captured as parameters of the automaton.

Event channels are used to communicate between pacemaker and its environment. VPace and VSense are channels for sending pacing signals and for receiving sensed events, respectively. A question mark after the channel name represents input from the channel, while an exclamation mark denotes output to the channel. The automaton has two states, WaitRI and WaitVRP, described below.

- WaitRI: The pacemaker starts from this state (denoted by the double circle) and waits for a ventricular sensing or pacing event. If sensing does not occur before the RI period ends, the ventricle controller sends a pacing signal to the heart (Transition 2) and the timer x is reset. The RI value is reset to LRI and hp is set to false, indicating that hysteresis pacing is not used in this case. When a ventricular sense occurs, Transition 3 is taken, where the timer x is reset, hp is set to true and HRI is assigned to RI, which allows a longer period to elapse before pacing . Once the ventricle is paced or sensed, current state is changed to WaitVRP.
- WaitVRP: In this state the pacemaker waits for a VRP period to elapse. It returns to the WaitRI state after a VRP period by setting hpenable to hp and started to true. hpenable and started are auxiliary variables to be used



(a) Ventricle controller model



(b) Heart model

Fig. 2. Uppaal model for a pacemaker in VVI mode

in property description for model checking. started is initially false and holds true after the first visit of WaitVRP.

3.3 Formal Verification

We mapped the timing requirements to the following verification queries in Uppaal. Below, A□ means that the property must hold in every state along every execution. Notation P.x denotes a variable x defined in the automaton P.

- PropDeadlock: A□($\neg$*deadlock*). This property expresses the deadlock freedom in the model.
- PropLRI: A□($\neg$Ventricle.hpenable $\Rightarrow$ Ventricle.x $\leq$ Ventricle.LRI). When hysteresis pacing is disabled, the LRI period should not be exceeded between any two pacing or sensing events.
- PropHRI: A□(Ventricle.hpenable $\Rightarrow$ Ventricle.x $\leq$ Ventricle.HRI). When hysteresis pacing is enabled, the HRI period is used in place of LRI.
- PropVRP: A□(Ventricle.WaitRI $\wedge$ Ventricle.started $\Rightarrow$ Ventricle.x $\geq$ Ventricle.VRP). Except the initial state, the pacemaker can be in the state WaitRI, where sense signals are accepted, only after the VRP period expires.

When we performed model checking on the model shown in Fig. 2 with the above four properties, we confirmed that the model satisfied all these properties.

3.4 Code Generation

We implemented the pacemaker software on a hardware reference platform of the Pacemaker Formal Method Challenge [1], which is based on a Microchip

8-bit PIC18F4520 MicroController Unit (PIC18 MCU) [9] running at 40 MHz clock speed. We generated a single-threaded code where the timed automata models are implemented inside a single loop. The code checks the current enabled transitions and takes one of them in each iteration.

The code generation algorithms adapts the techniques used in the Times tool [10] to produce code for the PIC18 MCU board. While the platform is substantially different from the one supported by Times, the code structure is essentially the same and we can reuse the correctness properties of the Times algorithm.

3.5 Validation of the Generated Code

We utilized MPLAB SIM, a software simulator for PIC18 MCU in the MPLAB Integrated Development Environment (IDE) [9] to execute the code and measure its timing. We tested the generated code under a variety of testing scenarios that cover all sequences of sensing and pacing events of length two that are qualitatively different with respect to the VRP and LRI periods.

Timing analysis of the observed event sequences was used to validate the code. An iteration of the validation cycle (see Fig. 1) was necessary to obtain the bounds on event processing delay, update the model to reflect these delays, repeat the verification, and re-generate the code. Testing of the re-generated code did not reveal any violations of the timing properties. Details of the validation process and timing analysis can be found in [2].

4 Assurance Cases

We created an assurance case to demonstrate that the implemented code is safe to operate, with the intention of providing a guiding example of assurance cases to be possibly used in the certification process of pacemaker software. The assurance case went through multiple review cycles within our group until we were satisfied that no unaddressed arguments result in significant risk to the pacemaker software.

Fig. 3 shows the top-level goal (G1) that the pacemaker software for the VVI mode, implemented as described in Section 3, is acceptably safe. The assurance case is implemented using the goal-structuring notation (GSN) [11]. It concentrates on the pacemaker software, assuming that the hardware platform is reliable (A1). Two context references (C1) and (C2) were added to clarify the goal statement. The assurance case is intended to be a part of the larger case that considers the overall system and makes claims about the assumptions made here.

The element (S1) describes the strategy we are using to argue the goal (G1): it is achieved by satisfying requirements, assuming that the designer extracted all the important properties related to the software safety from the system specification (A2). With this strategy, the goal (G1) is converted into the goal (G2), to show that the implementation satisfies all the desired safety properties within

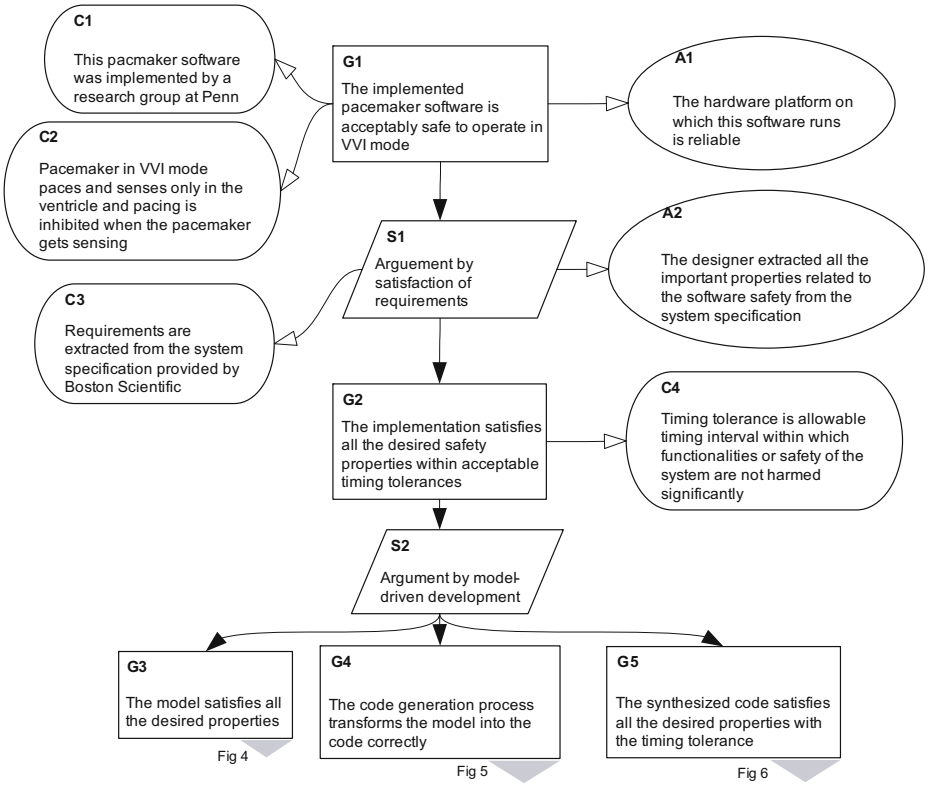


Fig. 3. The pacemaker assurance case - the pacemaker software is acceptably safe

acceptable timing tolerances. Context reference (C4) clarifies the meaning of timing tolerances in this context. Arguing by following the model-driven development approach (S2), the goal (G2) is supported by three subgoals: the model satisfies all the desired properties (G3), the code generation process transforms the model into the code correctly (G4), and the synthesized code satisfies all the desired properties with timing tolerance (G5).

Fig. 4 presents the argument for goal (G3). The model (M1) is the timed automata model of the pacemaker shown in Fig. 2. Four desired properties described in (C5) are described in Section 3.3. Conformance of the model to each property is argued by a separate subgoal ((G6)–(G9)), using model checking results as evidence.

Fig. 5 argues the goal (G4). Two strategies (S4) and (S5) were used to split the goal (G4). One of the subgoals supporting (G4) is that, in the context of using the TIMES tool (C10), the code synthesis of the TIMES tool for the verified model is correct (G10). Correctness arguments for the the code synthesis of the TIMES tool given in [12] are used as evidence (Ev5) to support (G10). Since we had to manually modify the code generated by the TIMES tool to port it on the

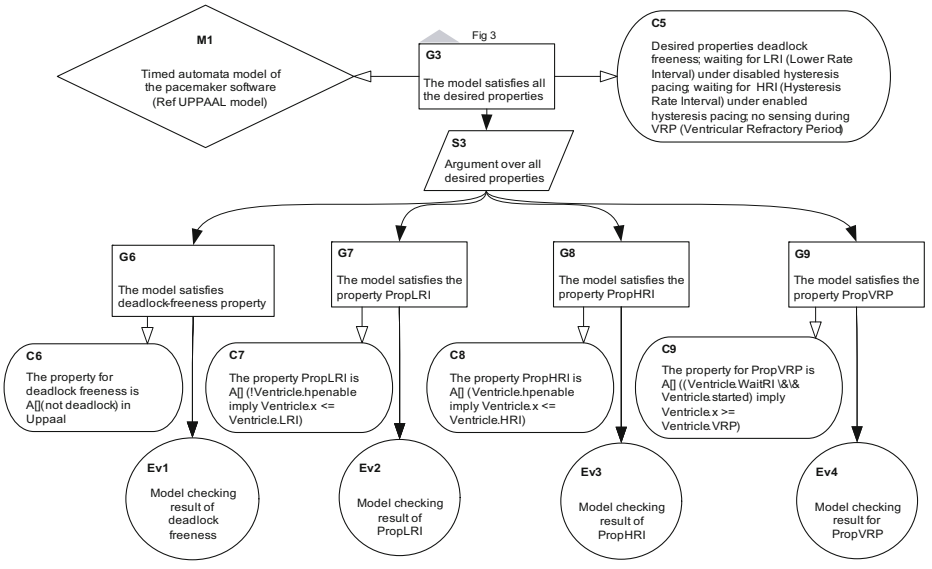


Fig. 4. The pacemaker assurance case - The model satisfies all the desired properties

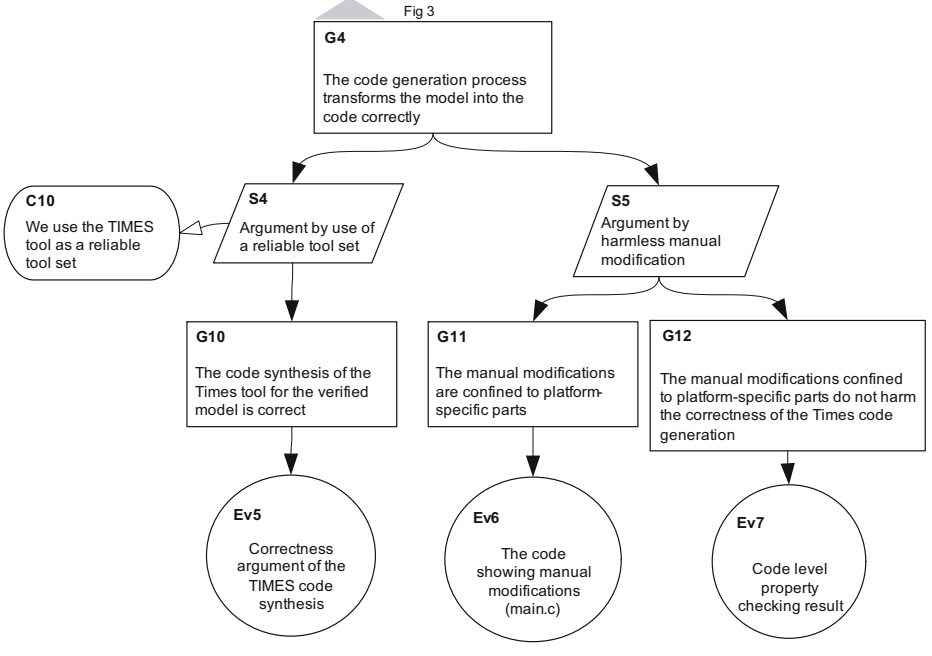


Fig. 5. The pacemaker assurance case - The code generation process is correct

pacemaker platform, we have to supplement this argument with the claim that manual modifications do not alter correctness of the code. Two subgoals, (G11) and (G12) identify the nature of the modifications, with code review results as evidence (Ev6), and demonstrate that they do not affect the functionality. In the latter case, results of code validation are used as evidence (Ev7).

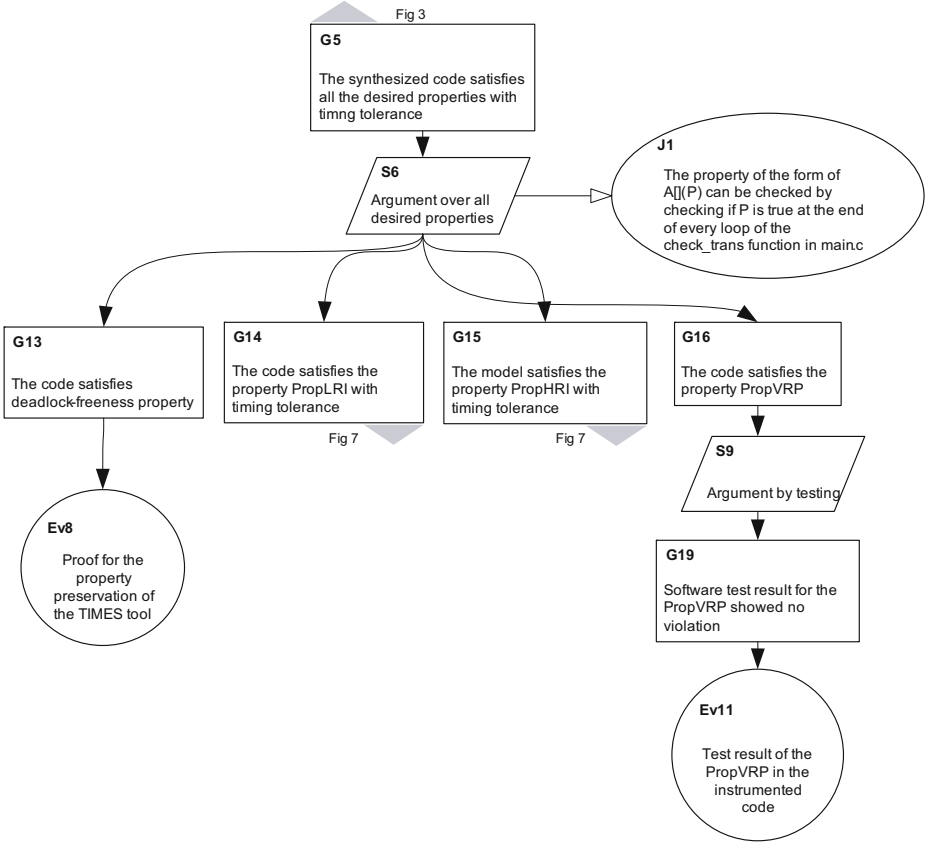


Fig. 6. The pacemaker assurance case - the code satisfies the properties

Fig. 6 addresses the third subgoal of (G2). It argues that the synthesized code satisfies all the desired properties with the timing tolerance (G5). Again, the argument is presented as a separate subgoal for each of the properties. The deadlock freedom property (G13), which does not involve tolerances, is ensured by the guarantees provided by the TIMES tool, which is used as evidence (Ev8). The other three subgoals, (G14)–(G16), are established through the code level checking based on the justification (J1) that a property in a form of $A\Box(P)$ can be checked in the code by checking if P is true at the end of every loop with a set of test cases. As shown in Fig. 6 and Fig. 7, (G14), (G15), and (G16) are argued by testing and rephrased by subgoals (G17), (G18), and (G19), respectively.

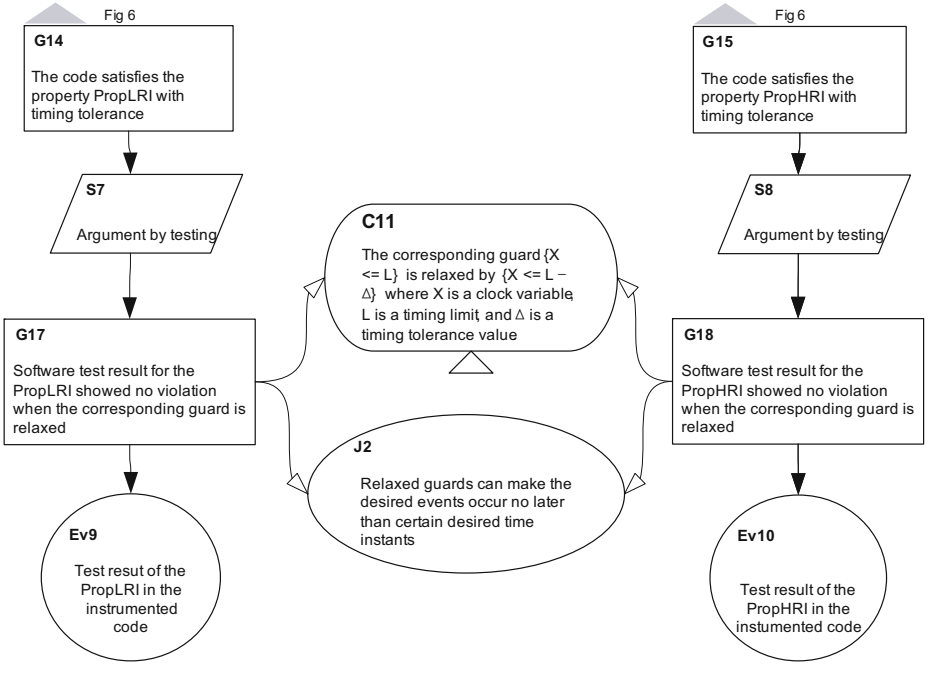


Fig. 7. The pacemaker assurance case - the code satisfies PropLRI/PropHRI

Note that the argument structure for the claim (G16) is simpler than the ones for (G14) and (G15) because the property PropVRP had been satisfied in the code all the time and no alterations were made to the corresponding guards in the code and the model. On the other hand, the properties PropLRI and PropHRI were satisfied in the modified code which involves relaxation in the corresponding guards (See (G17) and (G18) in Fig. 7). The context information for the guard relaxation was described in (C11) which can be instantiated with concrete values.

5 Discussion

Limits of the case study. We begin the construction of the assurance case with the requirements phase of the development. In a real system, the safety argument would also cover hazard analysis and offer claims that hazards are appropriately mitigated by the requirements. We omitted this phase to concentrate on the model-drive aspect of the development process. This decision also matches the current setting of the Pacemaker challenge, which begins with the pacemaker requirements by Boston Scientific. It makes sense to assume that the requirements were properly engineered with respect to hazard.

Similarly, we assume the nominal behavior of the underlying platform. We thus omit the questions of fault tolerance both in the development process and

in the assurance case construction. An assurance case for a complete system will of course have to deal with these issues.

Alternative ways to organize the assurance case. There can be alternative ways to construct the assurance case. When an assurance case has the same or similar structures within it, those common structures can be possibly merged and placed in an upper level. For example, the argument structures for (G14), (G15), and (G16) are similar and they have a common strategy "Argument by testing" as found in (S7)–(S9). "Argument by testing" can be placed in an upper level of (G14)–(G16), accompanied with logically consistent modifications to other parts.

Similarly, "Argument over all desired properties" are also commonly found under (G3) in Fig. 4 and under (G5) in Fig. 6 because we used the same strategy for arguing the property satisfaction in the code as well as by the model. It is possible to change the overall structure of the assurance case by placing "Argument over all desired properties", found in (S3) and (S6), above "Argument by model-driven development" (S2) and modifying other parts consistently. Note that these modifications do not change the logic of the argument, but may affect the size of the assurance case as we combine common nodes in different branches.

Alternative sources of evidence. In general, argument for a claim can vary and be supported by different kinds of evidence. For example, in our case study we relied on testing to establish timing properties of the generated code. If a higher level of safety is desired, we would resort to more rigorous worst-case execution time analysis using, for example, the aiT tool [13]. However, this change in technology affects only one claim, and the overall structure of the assurance case is not affected.

Ideally, when multiple alternatives can be used as evidence, we should aim to quantify the level of assurance each alternative brings and match it against the level of assurance required for the system. However, quantitative comparison cannot be achieved given today's state of the art. Even qualitative comparison of alternatives is difficult in many cases. This is an important direction of future work for our group.

Significance of the work. We believe that our case study is the first step towards developing assurance case templates for systems developed through model-driven processes. Model-drive development typically includes stages of modeling and model verification, code generation (manual or automatic) with respect to the model, and validation of the generated code and the whole system. In our approach, each of these stages correspond to a separate claim (or, in general, a set of claims) in the assurance case. This structure makes it more intuitive to follow during the evaluation and provides a clear connection to the evidence obtained in each phase.

6 Related Work

In [14], the authors considered a practice of using assurance cases in the development and approval of medical devices and addressed some of the important

issues surrounding the possible adoption of assurance cases by the medical device community. It was mentioned that a set of agreed argumentation patterns (templates) would be useful to manufacturers and reviewers. They suggested that creating and publishing a series of FDA-approvable archetypes for different kinds of medical devices be undertaken to ease the transition of assurance cases into the medical device community. With the same intention as theirs, we took a step forward by developing an argumentation template for another medical device, the pacemaker.

The process of assurance case construction and reuse can become more systematic through documentation of reusable safety case elements as patterns. In [15], 'Safety Case Patterns' for the reuse of common structures in safety case arguments were suggested. Assurance case patterns for security have been studied [16]. Our approach to the assurance case construction presented in this paper may lead to the development of assurance case patterns for model-driven development.

There are other case studies for assurance cases. In [17], the authors described an industrial application of assurance cases to the problem of ensuring that a transition from a legacy system of the Global Positioning System (GPS) to its replacement will not compromise mission assurance objectives. The assurance case demonstrated to the Air Force that the transition posed no major mission assurance concerns and this conclusion was validated by a successful transition.

7 Conclusion

We presented an approach for the construction of assurance cases for the model-driven development of safety-critical software. As a case study, we considered software for a cardiac pacemaker in the VVI mode. The assurance case ties together all the evidence collected during the development process. Several simplifications were applied in the process of constructing the assurance case, to keep the size of the case study under control and to concentrate on the aspects specific to model-driven development.

Future work includes the development of rigorous methods for the evaluation of assurance cases. For bigger systems, we also plan to study compositional construction of assurance cases. This will allow us to simplify certification of component-based systems based on product-line architectures.

References

1. Software Quality Research Laboratory: Pacemaker formal methods challenge, http://sqrl.mcmaster.ca/pacemaker.htm
2. Jee, E., Wang, S., Kim, J.K., Lee, J., Sokolsky, O., Lee, I.: A Safety-Assured Development Approach for Real-Time Software. In: The Proceedings of the 16th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, pp. 133–142 (August 2010)
3. Adelard: ASCAD – The Adelard Safety Case Development (ASCAD) Manual (1998)

4. Wassyng, A., Maibaum, T., Lawford, M.: Software certification: The case against safety-cases. In: Proceedings of the Workshop on Modeling, Development, and Verification of Adaptive Computer Systems (to appear, April 2010)
5. Oregon Health and Science University: Overview of pacemakers, http://www.ohsu.edu/health/health-topics/topic.cfm?id=10395
6. Boston Scientific: Pacemaker system specification (January 2007), http://sqrl.mcmaster.ca/_SQRLDocuments/PACEMAKER.pdf
7. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science 126, 183–235 (1994)
8. Behrmann, G., David, A., Larsen, K.G.: A Tutorial on Uppaal (November 2004)
9. Microchip: PIC18 family microcontroller, http://www.microchip.com/
10. Amnell, T., Fersman, E., Mokrushin, L., Pettersson, P., Yi, W.: TIMES: a tool for schedulability analysis and code generation of real-time systems. In: Proceedings of the 1st International Workshop on Formal Modeling and Analysis of Timed Systems, pp. 60–72 (September 2003)
11. Kelly, T., Weaver, R.: The goal structuring notation – a safety argument notation. In: Proceedings of the Dependable Systems and Networks 2004 Workshop on Assurance Cases (2004)
12. Amnell, T., Fersman, E., Pettersson, P., Yi, W., Sun, H.: Code synthesis for timed automata. Nordic Journal of Computing 9(4), 269–300 (2002)
13. Ferdinand, C., Heckmann, R.: aiT: Worst-case execution time prediction by static programm analysis. In: Jacquart, R. (ed.) IFIP Congress Topical Sessions, pp. 377–384. Kluwer, Dordrecht (2004)
14. Weinstock, C.B., Goodenough, J.B.: Towards an assurance case practice for medical device. Technical Report CMU/SEI-2009-TN-018, CMU/SEI (October 2009)
15. Kelly, T., McDermid, J.: Safety case construction and reuse using patterns. In: Proceedings of the 16th International Conference on Computer Safety, Reliability and Security, pp. 55–69. Springer, Heidelberg (1997)
16. Bloomfield, R.E., Guerra, S., Miller, A., Masera, M., Weinstock, C.B.: International working group on assurance cases (for security). IEEE Security and Privacy 4(3), 66–68 (2006)
17. Nguyen, E.A., Greenwell, W.S., Hecht, M.J.: Using an assurance case to support independent assessment of the transition to a new GPS ground control system. In: Proceedings of the International Conference on Dependable Systems and Networks, Anchorage, Alaska (June 2008)

Improving Portability of Linux Applications by Early Detection of Interoperability Issues

Denis Silakov and Andrey Smachev

Institute for System Programming at the
Russian Academy of Sciences, Moscow, Russia
{silakov,biga}@ispras.ru
http://www.ispras.ru

Abstract. This paper presents an approach aimed at simplifying development of portable Linux applications, suggesting a method of detecting compatibility problems between any Linux application and distribution by means of static analysis of executable files and shared libraries.

In the paper we concern the idea of successful launching of application in a distribution. A formal model is constructed that describes interfaces invoked during the program launching. A set of conditions is derived that should be satisfied by application's and distribution's files in order to make it possible for application to successfully launch in distribution. The Linux Application Checker tool is described that supports the approach and allows to detect portability problems of applications at early stage of development.

Keywords: Software portability, Software maintenance, Linux.

1 Introduction

Nowadays hundreds of Linux distributions exist. Most of them are based on the same set of components – Linux kernel, GNU utilities and libraries, KDE or Gnome desktop environment, etc. Many of these components follow the "Release Early, Release Often" principle, and new versions can be released every month or even every week. As a result, different distributions provide different versions of the same components. Unfortunately, though changes between successive versions can be relatively small, it is not uncommon for developers to break backward compatibility. In addition, distribution vendors often modify original code of components in order to fix known issues, to increase performance or to add some features that would be unique for their distribution. Thus, there are a lot of differences in functionality of the same components in different Linux distributions. This significantly complicates the task of development of Linux applications that could be launched on as many Linux distributions as possible.

Developers of open source applications usually rely on distribution vendors – most distributions have a *maintainer* for every application, who is responsible for correct functionality of the application in the distribution and can modify its code, if necessary. However, application should be rather popular to be included

T. Margaria and B. Steffen (Eds.): ISoLA 2010, Part II, LNCS 6416, pp. 357–370, 2010.

in many distributions; and even for open source programs, increasing portability allows to save maintainers' efforts. Finally, sometimes modifications introduced by distribution developers are criticized by original application authors.

Developers of proprietary software cannot rely on distribution vendors, since they don't publish the code and only provide compiled binary files. In this case, it is application vendor who is responsible for proper functionality of software product on all supported platforms. However, large amount of existing Linux-based systems makes systematic testing of application on every platform quite expensive even for large vendors. Moreover, not only hundreds of different Linux distributions exist, but many Linuxes run on dozen of hardware architectures. Many hardware platforms are not broadly accessible, while they are still potentially interesting for application vendors (for example, IBM zSeries platform is even more interesting for some developers of large enterprise software then the 'usual' x86).

Very often impossibility of guarantying proper functionality of the product in many distributions leads to limitation of officially supported systems – usually to a very few number of distributions that have the major market share, such as SUSE Enterprise Linux (SLES) or Red Hat Enterprise Linux (RHEL). That is, vendors do not tend to cover the whole Linux market. But as for end users, they normally want to have applications for 'Linux', not for 'RHEL' or 'SLES'.

Thus, it is desired for application developers to be able to achieve compatibility of their product with as many Linux distributions as possible. Since manual testing of applications in every existing system is a very hard task, it is important to detect as many portability problems as possible without performing runtime testing. One of the possible approaches that can help here is static analysis of application binary files. In general, such analysis may not guarantee the full compatibility of application and distribution, but can allow developers to save their efforts by decreasing time of problem detection.

The remainder of the paper is structured as follows: Section 2 gives a review of existing approaches to allow Linux application developers to support as many distributions as possible. Section 3 introduces a formal approach to analysis of compatibility of any pair of application and distribution by means of static analysis of their executable files and shared libraries. Section 4 presents the Linux Application Checker tool that supports the suggested approach and allows to detect issues in compatibility between any given application and most popular Linux distributions. Finally, Section 5 summarizes the main ideas.

2 Existing Approaches

Importance of improving portability of Linux applications is realized by all members of the Linux community and all participants of the market – distribution vendors, developers of open source projects, independent vendors of proprietary software and end users. Naturally, initiatives and approaches to increasing application portability come from different categories of community members. In this

section we consider the most popular ways for application vendors to achieve compatibility of their product with existing Linux distributions.

2.1 Using a Testing Farm

The most straightforward way for developers to guarantee compatibility of their program with some distribution is to thoroughly test the program in this distribution. In order to perform such testing, one should set up a separate machine (either physical or virtual), install the target distribution there, then install the program itself and run the tests. During the development process, developers should have a set of machines with target distributions, where their application is periodically built and subjected to testing. Such a set of machines is often referred to as a *testing farm*.

Large set of target platforms requires large set of machines to be set up. Even if machines are virtual, this can consume significant resources. Testing farms are usually served by automated scripts that schedule regularly builds, testing and other actions. Such infrastructure does not require much efforts for maintenance, but in general it increases the duration of 'detect problem – fix it – test again' chain (if compared to the case when developer is able to perform testing directly on his machine).

In addition, if developers want to cover hardware platform which is not broadly accessible, then they can find that time of real machines is expensive, while performance of emulators (if any) is usually poor.

Thus, though runtime testing in all target platforms is a really important thing, it is desired to detect as many problems as possible before running the tests inside the testing farm. It is especially important to detect critical issues whose presence will make the further testing useless.

2.2 OpenSUSE Build Service

An interesting initiative is provided by the OpenSUSE Build Service (OBS) – an infrastructure that can used by developers to build their applications for different distributions without direct access to the target systems. Nowadays, the OBS supports OpenSUSE, Mandriva, Fedora, Debian and Ubuntu; more systems to be added in future [4]. Though it is clear that periodic build of a large variety of applications requires significant resources, so it is not cheap to add a new target distribution.

As for portability, the service allows to handle several kinds of differences between distributions, such as package format or system file location. However, many aspects (e.g., differences in the behavior of the same function) are out of its scope. Surely, developers can integrate execution of automated tests in the build process – this will allow them to use OBS as a testing farm, whose disadvantages were discussed above.

Finally, the OBS approach makes sense only for developers of the open source products, since one have to provide application source code to the service.

2.3 Creating Standard-Compliant Applications

One more way to create a portable application is to focus on standards for operating systems that specify the set of interfaces that every compliant system should provide. The advantage of this approach is that standards not only guarantee the presence of interfaces, but also specifies all their characteristics that should be accessible by applications (e.g., behavior of functions). In order to be able to give such guarantees to its users, every standard usually provides a set of tests (usually referred to as *certification test suite*) that perform thorough testing of the standardized interfaces and should be passed by every implementation in order to confirm its compliance with the standard. Thus, if application uses only standard interfaces, it is guaranteed that application will demonstrate the same behavior in all standard-compliant distributions.

In the Linux world, the most famous standards are POSIX and Linux Standard Base (LSB), that concern Application Programming Interface (API) and Application Binary Interface (ABI) correspondingly. Both POSIX and LSB take into account existing systems (UNIX-like systems in case of POSIX and Linux in case of LSB) and try to standardize only those items that are implemented in all major distributions and proved to be mature, stable and useful. In those cases when some interface is provided by all systems but with slightly different characteristics, both POSIX and LSB try to specify only those aspects that are common for all implementations; relying on other aspects is not recommended for developers.

However, no standard can cover all possible interfaces – for example, POSIX only concerns the core system libraries (specifying about 1.000 of functions) and utilities. LSB covers more libraries, including some desktop and multimedia ones; however even its scope is still not broad enough for many programs – LSB 4.0 contains specifications for 57 libraries and about 38.000 functions, but a usual Linux distribution on a single DVD disc provides about several thousands of libraries and hundreds of thousands of functions [6]. Finally, trying to specify the 'common core' of existing systems often leaves many interface characteristics unspecified or declared to be 'implementation defined'.

Thus, among the existing methods described above, only the latter approach allows developers to avoid runtime testing of their applications in every target distribution. However, the main disadvantage of the approach (small coverage of existing interfaces) is hard to fix – standardization can be even more complicated and expensive task than development of tests, since creation of tests is usually only a part of the standardization process [10].

In this paper we suggest an approach that can be used to detect certain portability issues without runtime testing, allowing to decrease the time spent on problem detection. To detect portability problems, a lightweight static analysis of binary files of application and target distributions is used. The approach can be also used in cooperative with the standard-oriented method in order to analyze portability of those application parts that are not covered by existing standards.

3 Static Analysis of Interfaces Involved in Interaction between Distributions and Applications

Let *Distros* and *Apps* be the sets of all the Linux distributions and applications respectively. Below we will consider compatibility aspects of some application $A \in Apps$ and some distribution $D \in Distros$. First, let's clarify what do we mean under compatibility.

In general, interaction between two systems is performed by means of *interfaces* – one participant provides interfaces, the other uses them. For example, operating systems provides libraries that export functions, applications load these libraries and call their functions.

If distribution D provides a set of interfaces I_p and application A uses a set of interfaces I_r, then in order for successful interaction between A and D to be possible, the following condition is necessary:

$$I_r \subseteq I_p \tag{1}$$

This is a very general criterion – in every particular case one should point out concrete kinds of interfaces that should be taken into account. Provided that all possible kinds of interfaces and their properties are taken into account, this is also a sufficient condition.

For practical usage, it is important to have a way to analyze which interfaces are provided by distribution and which are used by application. Every interface may have a large set of different properties, all of which can be divided on two groups:

1. Properties that can be checked statically, without a need to invoke the interface (e.g., function signature).
2. Properties that require runtime testing (e.g., function behavior).

Surely, this classification is not ultimate – for example, one may claim that function behavior can be verified statically if its source code is available; and on the opposite side, even checking function signature in some situations may require function invocation (e.g., when there is no access to its declaration, but only to binary library that exports this function).

In this paper, we consider only those interfaces between Linux distributions and applications for which the condition 1 can be checked statically at a relatively low cost. Though the resulting set of interfaces is quite limited (in particular, it doesn't include any behavioral aspects), satisfaction of the condition 1 for this set guarantees that the application can be *successfully launched* in the distribution. To start with, let's clarify what we mean under these two words.

In our work, we consider *binary* applications – that is, applications that consist of binary executable files and shared libraries (*shared objects*, in Linux terminology). Any application in our model is a set of binary files, every of which is either a shared library or an executable file. Similarly, every Linux distribution is considered to be a set of shared libraries and binary executables.

Let's say that the application A *successfully launches* in distribution D, if dynamic loader in this distribution is able to form an executable image of the application in memory and pass the control to the application's entry point.

In Linux, for executable files and shared objects the Executable and Linking Format (ELF) is used, described in [1] and [2]. An ELF file can be self-sufficient in that sense that it may not require any external interfaces to be present in the system and work directly with hardware (maybe using low-level kernel interfaces, if direct access to hardware is not allowed). In this case dynamic loader just loads the file into memory and right after that passes control to its entry point; however, such files are used rarely and are not interesting for us.

Most programs nowadays use the advantage of dynamic linking, leaving the task of implementing routine functions for system libraries and concentrating only on unique features of their own. In this case dynamic loader has to perform much more actions to form a memory image for application. The precise algorithm is quite complex [5], but we are interested only in the following steps where the process can fail because distribution doesn't provide interfaces with required properties:

- Check that ELF files participating in dynamic linking have format acceptable for the loader – don't contain unknown ELF sections, have proper target hardware architecture, etc.
- Resolve dependencies on libraries – detect which shared libraries should be loaded to satisfy application's needs.
- Check that all dependencies on versions of binary symbols required for the loaded files are satisfied.
- Resolve addresses of external binary symbols of every loaded file – for every such symbol the actual implementation should be found among the loaded libraries.

After these tasks are complete, a memory image is constructed and dynamic loader passes control to the entry point of the file being launched. If this point is reached, we can say that the application has been successfully launched (in our terms).

Using the terms from the dynamic loading algorithm description, we can make our representation of Linux applications and distributions more accurate – every application and every distribution is considered as a set of ELF files, every of which can provide and require libraries, binary symbols and symbol versions. Using this representation, in the next sections we will derive a set of conditions that should be satisfied in order for the application A to be successfully launched in the distribution D. All these conditions can be checked statically by analyzing ELF files of application and distribution without a need to actual installation and launching. Moreover, there is no need to emulate the work of the dynamic loader – all conditions can be checked in a much more simple way and a lightweight analyzer can be developed to automate this process.

Now let's consider every kind of interfaces involved in the dynamic loading process.

3.1 ELF Sections

The ELF format evolves quite slowly (if compared to other parts of the Linux ecosystem). However, from time to time significant additions are introduced and files that use these additions cannot be used in older systems. The most notable example of the last years is introduction of the **.gnu.hash** section aimed to provide hashing with higher performance than the 'usual' **.hash** section. Introduced in 2006, this change led to the fact that programs compiled in new generation of distributions (such as RHEL 5 or Fedora 6) failed to run in previous releases of the same systems (RHEL 4, Fedora 5) [3].

Though such significant changes are introduced very rare, one should remember about them and check that dynamic loaders in target distributions support all aspects of the ELF format that are used in the application files. Thus, if $SysSupportedELF(D)$ is a set of ELF features supported by the distribution D, and $FileReqELF(f)$ is a set of ELF features used by the file f, then the following condition should be satisfied in order to launch application A in D:

$$\forall f \in A \rightarrow FileReqELF(f) \subseteq SysSupportedELF(D) \tag{2}$$

3.2 Shared Libraries

Let $SharedLibs$ be a set of all shared objects in the Linux ecosystem. Every library $lib \in SharedLibs$ is characterized by its *soname* (a special name visible to dynamic loader which may or may not be equal to the library file name) and hardware architecture: $lib = (soname, arch)$. It is the soname that is 'required' by ELF files; however, when looking for library that provides the requested soname, dynamic loader takes into account both library's soname and name of its file – if any of them matches the requested soname, than the loader picks the library up to satisfy the request. Thus, if soname of some library differs from its file name, then this library should be represented in the $SharedLibs$ set by two entities: $(soname, arch)$ and $(filename, arch)$. Let $FileProvLibs(f)$ to be a set of $SharedLibs$ elements provided by a file f (this set consists of either one or two elements).

Every ELF file can have a set of DT_NEEDED entries in its dynamic section that store sonames of libraries required by the file. Let's designate this set of required libraries as $FileReqLibs(f)$. This is a subset of our $SharedLibs$ set, with target hardware architecture of every $FileReqLibs(f)$ element been equal to target architecture of the file f itself (that can be detected on the basis of **Class** and **Machine** fields of the ELF header, as described in [6]).

For every distribution D, we are interested in the whole set of libraries provided by it, which is a union of libraries provided by all distribution files:

$$SysProvLibs(D) = \bigcup_{f \in D} FileProvLibs(f)$$

For applications, on the opposite, one should build a set of required libraries that are not provided by the application itself and thus are expected to be present in the system:

$$AppReqLibs(A) =$$

$$= \bigcup_{f \in A} FileReqLibs(f) \setminus \bigcup_{f \in A} FileProvLibs(f)$$

The second necessary condition that should be satisfied in order for the application A to be launched in the distribution D is that all libraries required by A should be provided by D:

$$AppReqLibs(A) \subseteq SysProvLibs(D) \tag{3}$$

3.3 Symbol Versions

A specific feature of ELF files in Linux is possibility of assigning a particular *version* to any binary symbol – different (from the source code point of view) functions or global variables can be made visible on the binary level under the same name but with different versions. This allows to keep backward compatibility with old applications when library developers decide to change function behavior – the old function implementation in this case becomes frozen and visible on the binary level with the same name as before. The new implementation is also visible under this name, but with a different version.

Every version is a simple literal string. Information about versions exported by file and versions required by it is stored in the appropriate ELF sections. When loading a file into memory, the system loader compares the set of required versions with versions provided by libraries loaded as file dependencies. Let's designate the latter set as $FileLoadedLibs(f, D)$; it is built using the following algorithm:

1. Set $FileLoadedLibs(f, D) = \emptyset$.
2. Put all $FileReqLibs(f)$ elements to $FileLoadedLibs(f, D)$ and to a temporary $AddedLibs$ set.
3. For each library $l \in AddedLibs$, calculate dependencies $FileReqLibs(l)$ of the ELF file that represents the library, calculate its difference with the $FileLoadedLibs(f, D)$ and union such differences to a new set:

$$FileIndirectDeps(f, D) =$$

$$= \bigcup_{l \in AddedLibs} FileReqLibs(l) \setminus FileLoadedLibs(f, D)$$

4. If $FileIndirectDeps(f, D)$ is not empty, then put all its elements to the $FileLoadedLibs(f, D)$ set.
   Rebuild $AddedLibs$ to be equal to $FileIndirectDeps(f, D)$.
   Set $FileIndirectDeps(f, D) = \emptyset$ and go to step 3.
5. Otherwise, everything is done and $FileLoadedLibs(f, D)$ is built.

Thus, the $FileLoadedLibs(f, D)$ set consists of libraries directly required by the file (that is, $FileReqLibs(f) \subseteq FileLoadedLibs(f, D)$), expanded with libraries

recursively loaded as dependencies of these libraries in a particular distribution. Since dependencies of system libraries are specific to a particular distribution, the $FileLoadedLibs$ set for the same file f can be different on different systems. That's why we write that this set is a function of both file f and distribution D.

Let $SysProvVers(f, D)$ to be a union of versions provided by files from $FileLoadedLibs(f, D)$. With $FileReqVers(f)$ standing for versions required by the file f, we can formulate the following necessary condition that should be satisfied for the application A to be launched in the distribution D:

$$\forall f \in A \rightarrow FileReqVers(f) \subseteq SysProvVers(f, D) \tag{4}$$

3.4 Binary Symbols

If all previous checks are completed successfully, the dynamic loader proceeds with resolution of external binary symbols for the files been loaded. Every binary symbol is unambiguously identified by name and version: $s = (name, version)$. The resolution process is similar to the one for versions of binary symbols – the loader takes a set of binary symbols required by file ($FileReqSyms(f)$) and then compares it with $SysProvSyms(f, D)$ – a set of symbols provided by libraries from the $FileLoadedLibs(f, D)$ set. So one more necessary condition for the successful launch is like the following:

$$\forall f \in A \rightarrow FileReqSyms(f) \subseteq SysProvSyms(f, D) \tag{5}$$

Strictly speaking, there can be a situation that if $f \in A$ is a shared object, then it is never launched directly, but only loaded along with other files. In this case some symbols required by it can be provided not by libraries from the $SysProvSyms(f, D)$ set, but by libraries from $SysProvSyms$ sets for files that are loaded together with f, since finally all these files are joined to a single image. Build tools in Linux allow programmers to perform such tricks. However, dependencies of the same library in different systems can vary (and may change as time goes by), so these tricks are not considered to be a good practice, especially from portability point of view. In our work, we ignore such possibility and treat 5 as required condition.

Now let's consider the fact the sets of required and provided versions are constructed automatically during application or library build as unions of versions of required and provided symbols respectively. That is,

$$version \in FileReqVers(f) \Leftrightarrow \exists s = (name, version) \in FileReqSyms(f)$$

$$version \in SysProvVers(f) \Leftrightarrow \exists s = (name, version) \in SysProvSyms(f, D)$$

So if 5 is satisfied, then 4 is also satisfied, that is, 5 ⇒ 4, and it is actually enough to only check the condition 5. However, in real systems a number of required symbols is usually much more greater than number of required versions, so 4 can be checked much more faster. Thus, it may still make sense to check 4 in order to detect possible problems at early stage, without deep analysis of binary symbols.

3.5 Sufficient Requirement

Up to this moment, we have considered the four necessary conditions 2, 3, 4 and 5 that should be met in order for the application A to be successfully launched in the distribution D. Since we have considered all interfaces involved in the launching process, then the sufficient condition of the successful launching is a conjunction of these conditions. As we have shown above, 5 $\Rightarrow$ 4, so the final sufficient condition can be formulated as a conjunction of 2, 3 and 5:

$$\forall f \in A \rightarrow FileReqELF(f) \subseteq SysSupportedELF(D)$$
$$AppReqLibs(A) \subseteq SysProvLibs(D)$$
$$\forall f \in A \rightarrow FileReqSyms(f) \subseteq SysProvSyms(f, D)$$

3.6 Method Value

It is clear that the approach suggested allows to detect only a limited set of compatibility problems; many kinds of issues (such as runtime function behavior) are out of its scope. In order to estimate the value of the approach in the real world, we have performed investigation of issue trackers of several Open Source projects and calculated percentage of issues that could be avoided if our approach were applied before the product release. We took into account errors concerning missing libraries and symbols; errors concerning symbol versions are relatively rare so they are joined with other symbol-related issues in the 'Failed symbols' column. Finally, it was found that issues related to the ELF format are very rare, so we haven't included them in the table below.

For our analysis, we have selected three popular applications that are broadly used in all Linux distributions – OpenOffice.org, Firefox and MySQL. In addition, we have investigated issues reported in the Launchpad software portal that provides hosting for more than 18,000 of Open Source projects. In our research, we have considered only critical bugs (either issues with severity set to 'Critical' or 'Blocker', or issues with the highest priority). Results of the analysis are shown in Table 1. The 'Total issue' column contains number of all critical bugs reported for the project; it would be also useful to calculate total number of bugs that concern portabilty, but this will require detailed investigation of every issue and is too time-consuming, so we haven't gathered such statistics.

Table 1. Number of issues in different projects that could be detected using the approach

Product	Total issues	Failed symbols	Failed libraries	Percentage
OpenOffice.org	20,000	190	110	1.5
Firefox	19,000	254	114	1.9
MySQL	6,400	98	20	1.8
Launchpad	42,000	95	60	0.3

It is clear that large applications that are actively used on almost all Linux distributions have greater percentage then relatively small projects hosted at the Launchpad whose target audience is, in general, much more smaller. Some projects from the Launchpad are not the binary ones, but created using interpreted languages (such as Perl or Python). For such programs, our approach is not applicable. Also note that our investigation only concerned bugs detected by customers, not by developers – that is, these are primarily bugs in the released products missed by QA teams.

Thus, using the approach suggested, large applications could decrease the number of critical issues discovered by customers by 1-2%. At the first glance, this is not a very high number, but it can be achieved at a very low cost – for example, the Linux Application Checker tool described below can perform all necessary analysis fully automatically, so developers will only have to spend several minutes launching the tool and checking the reports.

4 Linux Application Checker

The approach presented in this paper is implemented into an automated tool called Linux Application Checker. The tool can be used to analyze application binary files (in a form which is usually distributed to users) and check conditions 2, 3 and 5 for application and every distribution known to the tool. For every distribution, a verdict is given if application can be successfully launched there, and if not, a detailed list of compatibility problems is provided.

Data about distributions is collected by separate automated tools as described in [6] and is shipped with the tool, so the set of supported distributions is fixed for every tool version. The data collection process is fully automated and doesn't require for distribution to be installed – the analysis is performed on the basis of distribution installation packages (RPM and Deb packages are supported). In order to collect information about binary symbols and their versions exported by distribution libraries, these libraries are analyzed using 'readelf' and other tools from GNU Binary Utilities [7]. The same tools are used by the Application Checker to analyze application binary files on the user side.

The set of supported distributions is constantly updated; as of April, 2010, the tool contains knowledge about 63 distributions on Intel x86 architecture and 51 systems on x86-64 one (also known as AMD64). The tool supports five more hardware architectures: PowerPC, PowerPC64, IA64 (Itanium), S390 and S390X (IBM zSeries); for every of these platforms, knowledge about two dozens of distributions is collected (this number is smaller than for x86 and x86-64 due to the fact that many Linux variations don't support these platforms).

Finally, since the data collection process is fully automated and all necessary tools (including the Application Checker itself) are open, it is not hard for users to add data about any distribution they like.

It is important to note that Application Checker contains knowledge not about all libraries in every distribution, but only about a limited set of them (about 1,500 for every system). The thing is that according to empirical studies, there

are a lot of libraries in the Linux ecosystem that are used very rarely (usually only by their developers), so it is unlikely that they will be required by any third-party applications [9].

In addition to data necessary to check the conditions 2, 3 and 5, the Application Checker also contains information about libraries and functions included in the latest version of the LSB specification. For LSB-compliant distributions, these libraries and functions are guaranteed not only to be present in the system, but to provide all the functionality required by LSB. Thus, if application under analysis uses only interfaces included in LSB, developers can be sure that the application will not only successfully launch, but will also be able to work properly in all distributions compliant with LSB. For symbols and libraries that were once considered as LSB candidates but were rejected by the LSB workgroup (usually due to their deprecated status or known bugs in the existing implementations), the tool provides developers with appropriate warning messages, suggesting an alternative, if possible.

While a positive result from the Application Checker does not guarantee that application will run correctly in all distributions, it can notably reduce the porting and testing costs; the tool it is easy to use, doesn't require much user actions and still able to detect a noticeable set of errors. Data from the LSB knowledge base makes the tool even more valuable, allowing people to combine standard-oriented development with the approach suggested in this paper.

Nowadays, Linux Application Checker is recommended by the Linux Foundation [8] to all software developers who want to improve cross-distribution portability of their applications. Moreover, it is possible for application vendors to apply for LSB certification using Application Checker reports – the tool has possibility to automatically submit reports to the LSB Certification System.

5 Conclusion

Existence of Linux applications that can be used in any Linux distribution without modifications are greeted with applause by all members of the Linux community – distribution vendors (who don't want to maintain huge sets of patches for every application in their systems), independent software vendors (who would like to support as many platforms as possible) and end users (who don't want to be bound to a particular distribution just because their favorite software doesn't support the other systems). However, large variety of existing distributions makes it hard to develop and to support a product that would run everywhere, especially for proprietary vendors who cannot rely on any third parties when solving portability problems.

Development of products compliant with some standard may help in some respect, but existing standards cover only a small piece of the Linux ecosystem. So systematic runtime testing on every target platform remains the most popular approach for guarantying compatibility, but in case of large variety of platforms it becomes too expensive.

In this paper, we have suggested an approach to detect portability problems of applications by means of lightweight static analysis, without a need for runtime

tests. In general, the approach doesn't guarantee full compatibility of application and distribution, but it allows to check that the application at least can be *successfully launched* in the distribution. Problems concerning impossibility of launching of application are not rare in the real world.

The approach is based on the formal model of interfaces involved in the process of application launching. In the paper we use formalization to derive unambiguous conditions (both necessary and sufficient) of this process. Existence of clearly formulated and unambiguous requirements allows to create automated tools that can check these requirements.

The Linux Application Checker tool combines two techniques of increasing application portability without runtime testing. First, it allows to analyse program compatibility with different distributions using the approach described in this paper. Next, it supports development of applications that meet requirements of the Linux Standard Base specification by checking application compliance with LSB and suggesting alternatives for libraries and functions that are not included in the standard.

In our work, we have considered only programs represented as binary executable files and shared objects in ELF format. Applications that are written using interpreted languages (such as Java, Perl or Python) are beyond the scope of this paper. The approach can be extended to cover interpreted languages, too, since they also usually have some analogues of libraries exporting sets of functions that are used by applications. However, the concept of *successful launch* is not so clear for such applications, since there is no analogue of dynamic loader that resolves all external dependencies before passing control to the application itself. It can appear that even if a system doesn't provide all necessary interfaces, the application still can function correctly inside it until the missing interface is invoked. Nevertheless, it can still be desired to require that all interfaces that can be potentially invoked by the application should be present in the system. Our approach can be extended to cover this area, but derivation of formal necessary and sufficient conditions of compatibility will require investigation of interaction between application and interpreter, and this interaction can be actually specific to every particular interpreter language.

References

1. System V Application Binary Interface Draft (April 24, 2001), http://refspecs.linuxfoundation.org/elf/gabi4+/contents.html
2. Linux Standard Base Core Specification 4.0. Executable And Linking Format (ELF), http://refspecs.linuxfoundation.org/LSB_4.0.0/LSB-Core-generic/LSB-Core-generic/elf-generic.html
3. Proffitt, B.: More Compatibility Issues Easily Managed With LSB. Linux Developer Network (October 2008), http://ldn.linuxfoundation.org/node/7141
4. Schroter, A.: OpenSUSE.org Build Service – a Short Introduction. In: Free and Open Source Software Developers' European Meeting, FOSDEM (2008), http://files.opensuse.org/opensuse/en/2/21/FOSDEM2008-OBS-short-introduction.pdf

5. Drepper, U.: How To Write Shared Libraries. Red Hat, Inc. (August 20, 2006), http://people.redhat.com/drepper/dsohowto.pdf
6. Silakov, D.: Linux Distributions and Applications Analysis During Linux Standard Base Development. In: Proceedings of the Second Spring Young Researchers' Colloquium on Software Engineering (SYRCoSE 2008), St.Petersburg, Russia, May 29-30, vol. 1, pp. 11–18 (2008)
7. GNU Binary Utilities, http://sourceware.org/binutils/docs/binutils/
8. The Linux Foundation: Building a Portable Application For Linux, http://ldn.linuxfoundation.org/lsb/check-your-app
9. Rubanov, V.: Automatic Analysis of Applications for Portability Across Linux Distributions. In: Proceedings of the Third International Workshop on Foundations and Techniques for Open Source Software Certification (OpenCert 2009), York, United Kingdom, March 28, pp. 44–53 (2009)
10. Khoroshilov, A.V., Rubanov, V.V., Shatokhin, E.A.: Automated Formal Testing of C API Using T2C Framework. In: Proceedings of the Third International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2008), Part 3, Porto Sani, Greece, October 13-15, pp. 56–70 (2008)

Specification Based Conformance Testing for Email Protocols*

Nikolay Pakulin and Anastasia Tugaenko

Institute for System Programming of the Russian Academy of Sciences,
Moscow, Russia
npak@ispras.ru, tugaenko@ispras.ru

Abstract. The paper presents a method for conformance testing of Internet electronic mail protocols. The method is based on formal specification of the standards following the approach of the contract specification, and designing tests as traversal of a state machine. The paper presents the implementation of the method for the most widely used e-mail protocols SMTP, POP3 and IMAP4 and is illustrated by the results of testing of popular e-mail servers.

1 Introduction

Emails are fundamental to modern communications between people. Hundreds of millions of emails float every day around the Internet. Reliability and correctness of the emailing infrastructure is vital to the modern information society. In this article we concern two aspects of these questions: (1) reliability of mail transfer in the Internet and (2) delivery of the email to recipient, typically a human being.

Most of emails in the Internet are transferred by means of SMTP – Simple Mail Transfer Protocol [1]. It is a text-based protocol with two parties: a client and a server. Client issues commands and server executes them, returning status code and other details if needed. SMTP has its own overlay network over TCP/IP comprised by numerous mail servers and relay agents used to forward emails between various domains. A feature of SMTP is that each physical server could operate as both SMTP client and SMTP server: being a server it accepts an incoming email and becomes a client to forward it to a next hop.

SMTP is used to send messages, but when an SMTP implementation identifies itself as the final destination of an email it stops forwarding the mail and places it in an internal implementation-specific storage. To retrieve emails from the storage end-users utilize other protocols: POP3 (Post-Office Protocol, version 3 [2]) or IMAP4 (Internet Mail Access Protocol version 4 [3]). Both protocols are text-based with distinct roles of clients and servers. Clients get access to the storage through issuing protocol commands and servers provide required information in their replies. Typical POP3 and IMAP4 implementations support only one role at a time - either the server or the client.

* This work was supported by RFBR grant 10-07-00145-a and by Rosnauka grant 2009-04-1.4-00-01-006.

T. Margaria and B. Steffen (Eds.): ISoLA 2010, Part II, LNCS 6416, pp. 371–381, 2010.

On its way from the originator to the recipient an email is processed by a number of intermediate mail agents. A typical case is that those servers come from different vendors thus having different implementations of the email protocols. The total reliability of emailing infrastructure substantially depends on the compatibility between implementations as well as functional correctness of each implementation.

Nowadays protocol conformance testing is the basic method of attesting implementations compatibility. The rationale for this statement bases on the suggestion of good protocol design: if two implementations conform to the protocol specification then they are compatible, they can correctly communicate with each other. In this paper we do not consider whether this assumption holds for the Internet mail protocols, we provide an approach to the conformance testing of such protocols.

The proposed approach allows to sequentially seamless transform test cases based tests into model based tests with saving the functionality and workability. On the first steps only testing approach transformation and testing protocol model forming take place while further steps allows easily extending of test suite for certain functionality checking.

2 Related Works and Motivation

Despite of more than twenty-year history of mail protocol service and existence of dozens of SMTP/POP3/IMAP4 implementations still there are no open and implementation-agnostic conformance test suites for those protocols. We believe there are several reasons to explain the absence of such test suites:

1. the mail protocols seems simple;
2. developers focus testing on the aspects of mail server development that are unrelated to conformance: parameter processing, security, internal storage, performance, etc.

Let's consider these reasonings in more detail. First, the simplicity of the mail protocols is seeming. One can easily see that SMTP, POP3 and IMAP4 manifest a number of non-trivial issues:

1. mail protocols are underspecified: a large part of functionality is left to implementation developers; specifications prescribe several variants of possible system behavior;
2. mail protocols are nondeterministic, the standard allows various system behavior alternatives including refusal in mail message delivering or connection tear;
3. mail protocols requirements differ in the level of obligations (MUST, SHOULD, MAY and others);
4. protocol architecture is extensible, protocols implementation may use different extensions for supplemental functionality or even overlapping functionality.

Listed features demonstrate complexity of the task of conformance test for email protocols.

Second, developers has to focus testing on another big issue of mail server development: processing of a large number of settings necessary for practical use – parameters of routing, authentication, security, mail messages depository, etc. We studied test suites developed by several open source implementations: Apache James, Sendmail, Dovecot, Qmail, Postfix. The test suites turned out to be tightly coupled with the implementations under test (IUT) and non-portable to servers from other vendors, they utilize implementations features setting parameters, access to the internal state, execution in the same process with the implementation. Tests designed for one implementation could not be applied to other implementations. Moreover, as the analysis showed, these tests are inappropriate for conformance testing, they are oriented to verification of implementation-specific settings that are not directly connected with SMTP, POP3 and IMAP4 standards. The problem is that such approach to testing doesn't guarantee servers' compatibility to the standard. The situation is even worse: our conformance tests detected a serious functional defect in James server – cycling at certain conditions – that was overlooked by the James functional tests.

Also it is necessary to note that there is a special tool developed in Apache Project – Mail Protocol Tester (MPT) – for verifying correctness of server replies. The input data for this tool is a script that specifies server stimuli and expected server replies. The program uses regular expressions for specify expected replies of IUT. If a reply mismatches the regex program stops and throws the mistake message. Apache James MPT provides only basic testing facilities and does not support branching, cycles and parameters usage in tests.

Exact relationship between tests and standard's requirements allows hard confidence in terms of a user of mail service. As the example with server James shows, thorough testing of internal functions of implementation doesn't guarantee the functional quality of implementation in real environment.

We believe that the problem of conformance testing to the standards of mail protocols has significant practical importance. The ultimate goal of our research is to propose new test system architecture and the technique for developing system elements which are suitable for specific mail protocols features. In this paper the new technology for conformance test suite development is presented. This technology includes several test development steps with steps being constructed in a way that each step may be final. The novelty of presented method is the ability of sequentially transformation of testing model in which in each step testers get workability tests (model, program, depends on step). At the beginning of test suite development one have to write test cases or take available test cases. In the further steps the interface model is building, then the addition of states to already built model is performing. After these steps have executed tester got test suite with same functionality as after first step. But on this step it is more easily to extend the test suite by extending the specification. The presented approach is model-based and focuses solely on conformance and does not consider

other aspects of email infrastructure validation, such as interoperability testing, performance testing, reliability testing etc.

As an example of proposed method application an opensource conformance test suites for basic functions of SMTP, POP3 and IMAP4 protocols was delevoped by the instrumentality of this method.

The paper is structured as follows: Section 3 gives an overview of the existing approaches to protocol conformance testing and discusses why model-based testing and UniTESK technology is used in our approach. Section 4 provides a quick introduction to UniTESK technology that lays in the basis of our approach and Section 5 introduces the approach used to develop test suite for SMTP, POP3 and IMAP4. In section 6 we present the test suites for SMTP, POP3 and IMAP4 developed so far and Section 7 discusses pros and cons of the proposed approach. Section 8 summarizes results achieved by now and highlights directions of future research.

3 Mail Protocol Testing

In the contemporary industry conformance testing of protocol implementations is mostly based on manual development of test suites consisting of independent test programs written in specialized or general-purpose programming language. Such programs are referred to as test cases; they implement stimulus construction, passing generated test inputs to the IUT, reading and analysis of observed outputs [4].

Academic researchers and practitioners typically consider test cases as one of the traditional testing methods [5,6,7] while model based testing is considered as new method solving many unsolved with traditionally methods problems (for instance, imprecise coverage of IUT functionality and manually development of requirements traceability). The method proposed in this paper considers model based testing as extension of test cases based approach. In the capacity of proposed method demonstration the development of test suites for SMTP, POP3 and IMAP protocols are presented.

Let's consider requirements for test suite. Test suite for conformance testing must possess the following properties.

1. Requirements traceability. Tests must correlate with the requirements stated in the standards. It must be clear for each requirement which test it is covered by.
2. Variety of settings for implementations features (MUST, SHOULD, MAY and others). There must be an option to define the set of requirements supported by an IUT and avoid requirements that IUT does not implement.
3. Completeness of test suite in terms of requirements coverage. Resulting test suite must cover at least all obligation requirements.

Test cases approach doesn't provide evident traceability of requirements. Requirements completeness in terms of coverage in such approach is also complicated. Approaches based on TTCN-3 [8] and JUnit [9] require external tools to

establish connections between tests and requirements such as traceability matrixes. The weakness of such tools is that the requirements are not an integral part of the tests.

Besides that large number of tests presented as separate programs results in code redundancy or complex and complicated connections. It is necessary to use methods permitting decomposition of test suites and providing requirements traceability.

Required possibilities are given by tools based on formal method approach. Utilizing of formal specifications allows to:

1. define formal connections between requirements and tests; automatically backtrace quality of testing in terms of specification coverage;
2. using model repeatedly for checking correctness of implementations behavior;
3. generate test stimuli in terms of model and automatically filter redundant stimuli.

Also when choosing a method for generating test sequences it is necessary to take into account features of mail protocols. Particularly because of protocol behavior is nondeterministic and underspecified, one should choose approaches providing test sequences generation with a glance of IUT replies. As well when testing mail protocols it is complicated to make prediction for result or define equivalence of traces. Automatic verdict generation from specifications postconditions may solve the problem of verifying correctness of IUT behavior.

There are many instruments and approaches for testing. NModel [10] represents model system in C# language and provides basic facilities for on-the-fly testing and coverage maximization. But for on-the-fly testing test developer must write separate program describing complex traversal strategy.

The previous version of SpecExplorer (2004) [11] was implementing on-the-fly testing ability but this version is no longer supported. In the last version of SpecExplorer (2010) [12] on-the-fly testing is not documented. Toolkit Conformiq Qtronic [13] does not provide on-the-fly testing, instead it generates TTCN-3 test scripts.

Toolkit UniTESK [14,15] supports formal specifications notation, automated on-the-fly test stimuli generator (code-level, there is no need to write separate program) and automated test results analysis. So we decided to use this technology for this project.

In UniTESK technology formal specifications which formalize requirements as pre- and post- conditions are used for generation of test sequences. Also for test sequences generation must be given a certain finite state machine (test state machine). The test process in UniTESK is automatic traversal of test state machine in which IUT behavior is automatically verified by test oracles; test oracles are generated from formal specification. The utilizing of formal specifications allows automating verification of behavior correctness and estimation of hard confidence; presenting test as state machine makes possible to automatically generate long and various sequences of test events.

Authors used presented method for developing test suites for protocols SMTP, POP3 and IMAP4. From tools implementing the UniTESK approach JavaTESK

[16] was chosen. JavaTESK uses the programming language Java with a number of extensions for record formal specifications and specify tests.

4 UniTESK Technology Overview

The standard format for Internet protocol standardized documentation is defined by documents RFC (Request for Comment). Requirements in these documents are stated in English and correspond informal text that describes desirable system behavior. In the UniTESK technology (Fig. 1) specialized specification languages – extensions of Java and C – are used for record requirements.

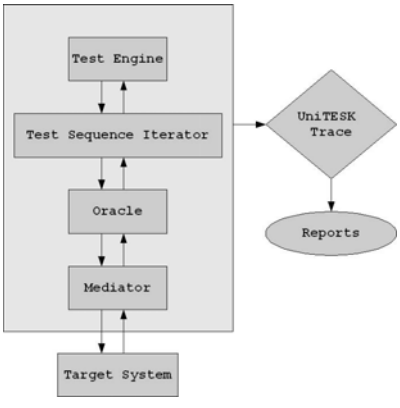


Fig. 1. UniTESK test suite architecture

Recording the informal requirements of standardized documentation in formal language represents the protocol model. In UniTESK approach the formal model is constructed in terms of finite state machine. Transitions between states may be given in explicit or in implicit way. In case of explicit definition of transition the model contains algorithm for calculating next state and protocol reaction. Presentation of implicit transition is a predicate which defines restrictions on acceptable states and protocol reactions.

Specification in JavaTESK usually consists of one or few specification classes which describe states and transitions of modeling protocol. Protocol transitions are presented as special methods (specification methods). In addition there is a possibility to define restrictions on acceptable set of states by means of type invariants (type restrictions) and state variable invariants.

Definition of implicit transitions realized as pre- and postconditions. In preconditions there are restrictions on acceptable stimulus parameters values and on states from which stimuli may be given. The IUT may react on stimuli by changing state, giving a reaction or both. Postconditions define acceptability of demonstrated behavior.

For modeling IUT behavior one uses the set of data structures which referred to as abstract states. For verdict pronouncement about the correctness of IUT behavior UniTESK uses data from model abstract state.

In UniTESK both stimuli for the IUT and its reactions are described in terms of model; model is defined by the formal specifications. Correlation between the model and the IUT is established by an intermediary – mediator – which translates stimulus parameters from model form to protocol messages, IUT reactions to model representation and if necessarily transports changes from the IUT state to the abstract state.

Test scenario defines stimulus sequence applied to IUT. The metamodel of finite state machine is used as the theoretical basis for constructing scenarios. In JavaTESK test state machine is defined in scenario class which contains the procedure for calculating current state and iterator of test stimuli. The JavaTESK tool contains test engine for constructing test stimuli sequences from test state machine description.

5 The Proposed Method for Mail Protocols Conformance Testing

Mail protocols may be in several states. When receiving certain stimulus they generate and send reactions and jump to another state or leave in current. With a glance of this fact on the basis of instrument UniTESK the method for testing mail protocols was developed. The proposed method may be considered as method with sequentially seamless transformation from test cases based testing to the model based testing (if tester already has test cases) or as method for sequentially model based tests creation. The proposed method contains the following main steps.

1. Analysis of knowledge domain. Developing of examples and elementary tests. This step doesn't give any visible results but it is important for detailed protocol understanding and helps in implementation of next steps.
2. Creation of requirements catalogue. Requirements catalogue is a database or a table with description of requirements. Catalogue's record contains not only requirement's description but also requirement's identifier, type (syntax or functional), severity, link to the place in the RFC and maybe other attributes.
3. Designing of lite protocol model. Lite protocol model includes information about commands and about possible reactions to these commands. Experimental test consists of specification, mediator and scenario classes. Specification class on this step includes only signatures of methods; all verifications are making in scenario class. Such test referred to as linear test; applying stimuli and reading reactions are process in certain order defined in scenario class.
4. Designing of conceptual protocol model. Extracting states of basic protocol. Creation of test state machines with dedicated states. Expanding experimental test – addition the block which makes transitions between states. Conceptual model defines behavior of observing system as operations on some set of abstract components and composing objects. These components are used only for behavior modeling and may not conform to the model extraction. Addition of block for making system transitions between states

turns test from linear to automatic test. In this step construction of stimulus sequence is made from test state machine traversal; applying stimuli and reading reactions are made only from acceptable states.
5. Requirements formalization. Relocation of verification of IUT responses into specification class. Checking completeness and consistency of requirements is made while formalizing requirements. The result of this step is the formal protocol specification written on one of the special program languages.
6. Enhancement of scenario and specification for covering all requirements. In this step scenario classes contain only stimuli. The order of stimuli is formed from state machine traversal and depends on applying stimuli in certain states conditions. Usually one scenario class is responsible for the certain requirements section. For covering all formal requirements few scenario classes may be needed.
7. Execution of test suites and analyzing the results. Analysis may show that not all requirements are covered by generated test suite. If not all requirements are covered then step 6 must be repeated until covering all requirements from catalogue.

If at the beginning of test suite development tester has test cases or if tester is experienced in model based tests creation then the first step may be skipped. Also we note that given order of steps is not singular. So, steps 3 and 4 may be combined. And step 5 may be performed in parallel with steps 3 and 4. Also a number of iterations may be executed within the bounds of the described steps sequence.

6 Method Application for Protocols SMTP, POP3 and IMAP4 Testing

The first step of test suite development for SMTP, POP3 and IMAP4 implementations was analysis of knowledge domain and familiarizing the testers with the protocols. Then the requirements from RFCs were elicited and categorized for commands and replies, routing, notifications, server settings, mail headers, mail body, etc. On this basis of the catalogue the lite protocol models were designed; test state machines sent commands:

1. for SMTP: `EHLO`, `HELO`, `MAIL`, `RCPT`, `DATA`, and others;
2. for POP3: `USER`, `PASS`, `LIST`, `STAT`, `RETR`, `DELE`, `TOP` and others;
3. for IMAP4: tag plus command word `LOGIN`, `EXAMINE`, `CREATE`, `DELETE`, `RENAME`, `SELECT` and others.

Protocol replies were specified – for SMTP: three digit numeric code – reply code; for POP3: "`+OK`" or "`-ERR`" replies; for IMAP4: non-obligatory tag plus "`OK`", "`NO`", "`BAD`", "`PREAUTH`" or "`BYE`".

At that step the test suites generation had a formal interface, specification classes was consisted of only methods' signatures; all IUTs behavior correctness verifications were made in scenario classes. Scenario classes consisted of methods applying

(by means of mediator classes) stimuli to the IUTs, reading servers responses and returning verdicts about correctness of severs behavior. Mediator classes transformed the IUT stimuli format to model systems format and vice versa.

Then the basic protocol states were introduced. On this step the new blocks responsible for making model system's transitions between states were added to the scenario classes. Specification classes were not changed.

In the next step blocks responsible for verification of IUT behavior correctness and blocks that make transitions between systems states were relocated from scenario classes to specification ones. Scenario classes solely applied stimuli to the IUT. From now certain commands could be passed only from acceptable states of state machines. The possibility of such verification is achieved by recording acceptable states in preconditions of specifications. Iterator every time checks the current state and whether the applying of the next command is allowed in current state. Also it gives the opportunity to check the fact that servers don't send commands from forbidden for such commands states.

7 Discussion

To the lows of the method based on formal specifications one may attribute the absence of the quick test suites updating ability. To develop new test it is necessary to thoroughly study requirements, formalize and classify them. Only after these preparations one may set out to write specification, adaptor and tests. Due to the preparation stage including specification development the period for new test development is increasing.

On the other hand, after specification, mediator and scenario classes are implemented one gets not a single test but a set of tests responsible for the corresponding requirements class. Also it is necessary to note that the lengthy preparatory stage happens only in case of incomplete formal specification. If formal specification is available tests may be generated quickly – just by modification of scenario class. Moreover, if the task is to make complex test then the utilizing of formal specification approach gives result more easily and quickly than test cases approach.

The major drawback of the proposed method is availability of an implementation: at every stage the tests and specifications are being validated against a real implementation. As a result, this method is not directly applicable for protocols that do not have existing implementations.

One of important advantages of this method is separating the verdict assignement from test sequence generation. The oracle which is generated from specification postconditions is responsible for verdict assignement. Thanks to this separation test developer does not need to handle verdicts while testing correctness of IUT behavior. Having a dedicated place (specification) in which all checks of IUT behavior correctness are kept helps reusing the oracle in different test scenarios.

Utilizing of formal specifications allows formulation the exact unambiguous hard confidence criteria – testing may be completed when all elements of appropriate formal specifications are covered. UniTESK provides support to dynamically access covered requirements and allows to define some selecting criterion

for scenarios – if applying of certain scenario doesn't increase test coverage then system misses it and moves to the next scenario.

8 Results and Further Research

For SMTP we elicited 51 basic requirements, 43 of them are related to server commands and replies (11 of them are mandatory and 4 are optional), 8 related to routing (all are mandatory). For POP3 we elicited 58 requirements for basic commands, 5 of them are mandatory and 6 are optional. For IMAP4 43 requirements were elicited for basic commands, 7 of them are mandatory and 2 are optional. The scope of the project was the basic functions of the protocols, other functions such as notifications messages for delivery failures message, mail gateways to non-SMTP domains were left out of our project.

All requirements are covered by the developed test suites. The tests were applied to open source mail servers – Apache James, hMail Server, Postfix and Dovecot. Generated tests have detected the following disagreements between protocol implementations and standards [1,2,3]:

- missing required commands;
- protocol rules violation, such as accepting commands in illegal states;
- wrong reply codes to the protocol commands;
- cycling while redirecting mail.

8.1 Further Research

The common characteristic of mail protocols is that mail protocols are extensible. Certain extensions complement the existing functionality, i.e. add new requirements which do not contradict the requirements from basic standard. But there are also such extensions which radically alter the protocol structure thereby discarding some requirements from the basic standard. For checking such extensions one should modify test suites in such a way that requirements that are amended by extensions have not been verified. Otherwise there will be no opportunity to reach 100% coverage of all obligatory requirements. Since many protocols be extensible there is a need for tools providing capability for generating test suites for various extensible protocols' implementations – both supporting extensions and supporting only basic standard functionality. The direction of the future research is to develop such tool based on JavaTESK.

9 Conclusion

The paper presents an approach to specification-based testing of mail protocol. Novelty of the method is ability to sequentially transform test cases into formal model and tests using that model as oracle. Furthermore, in each step tester gets executable test suite which cover the same functionality as tests produced at the previous step and may add more checks.

The approach belongs to model-based testing domain, it uses contract specifications to formalize protocol specification and on-the-fly test sequence generation. The implementation of the approach is based on UniTESK technology. Distinctive features of this method are automated test sequences generation on basis of formal specifications, test coverage calculating which allows constructing stimuli in optimal way and also the presence of separate component – oracle – responsible for verdict about IUT behavior correctness returning.

Developed method was applied for testing of long used mail protocols implementations. In one implementation (Apache James Server) was found a critical defect – under specific circumstances while redirecting message the server is resending the mail to itself and the message never reaches the recipient. The notification to the originator about undeliverable mail is absent. Description of steps for reproducing defect is reported to Apache Software Foundation.

References

1. IETF RFC 5321. J. Klensin. Simple Mail Transfer Protocol (2008)
2. IETF RFC 1939. J. Myers, M. Rosem, Post Office Protocol – Version 3 (1996)
3. IETF RFC 3501. M. Crispin. Internet Message Access Protocol – version 4rev1 (2003)
4. ISO/IEC 9646. Information technology – Open Systems Interconnection – Conformance testing methodology and framework – Part 1: General concepts. Geneva: ISO (1994)
5. Utting, M., Legeard, B.: Practical Model-Based Testing: A Tools Approach. Morgan Kaufmann, San Francisco (2007)
6. Blackburn, M., Busser, R., Nauman, A.: Why Model-Based Test Automation is Different and What You Should Know to Get Started. Software Productivity Consortium, NFP (2004)
7. Dalal, S.R., Jain, A., Karunanithi, N., Leaton, J.M., Lott, C.M., Patton, G.C., Horowitz, B.M.: Model-Based Testing in Practice. In: Proceedings of the ICSE 1999 (May 1999)
8. ETSI ES 201 873-1 V3.1.1. Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language. Sophia-Antipolis, France: ETSI (2009)
9. Unit testing framework, http://www.junit.org
10. Jacky, J., Veanes, M., Campbell, C., Schulte, W.: Model-based Software Testing and Analysis with C#. Cambridge University Press, Cambridge (2008)
11. http://research.microsoft.com/pubs/77383/bookChapter0nSE.pdf
12. http://research.microsoft.com/en-us/projects/specexplorer/
13. End-to-End Testing Automation in TTCN-3 environment using Conformiq Qtronic and Elvior MessageMagic (2009)
14. Kuliamin, V.V., Petrenko, A.K., Kossatchev, A.S., Burdonov, I.B.: The UniTesK Approach to Designing Test Suites. Programming and Computer Software 29(6), 310–322 (2003)
15. http://www.unitesk.com
16. Functional testing of list using JavaTESK. Moscow (2008)

Covering Arrays Generation Methods Survey

Victor Kuliamin and Alexander Petukhov

Institute for System Programming Rassian Academy of Sciences,
25 Alexander Solzhenitsyn st., Moscow, 109004, Russia
kuliamin@ispras.ru

Abstract. This paper is a survey of methods for covering arrays generation. Covering arrays are used in test data generation for program interfaces with many parameters. The effectiveness and the range of application of these methods are analysed. The algorithms used in these methods are analysed for their time complexity and memory usage. In this paper combinatorial, recursive, greedy algorithms are observed. Several heuristics for reducing the size and the time for construction of covering arrays are observed.

Keywords: covering array, interaction testing, pair-wise testing, coverage criteria, combinations of factors, interfaces with many parameters, finite fields, heuristic search, greedy algorithm.

1 Introduction

High complexity of modern computer systems and the necessity of the tasks they perform make the correctness assurance (i.e. verification) of such systems essential. To have a notion about the quality and the completeness of the verification one has to do it methodically. For the methodical verification the testing is used. Testing is a creation of special test cases and an analysis of the behaviour of the system in these cases. To assure the quality and the completeness of the verification it is required to have as many tests as possible. However, full testing is unrealisable in practice because the amount of cases in which the real world systems must be verified is infinite. That's why a finite (often rather small) amount of test cases is chosen. The test cases are chosen so that by analysing the behaviour of the system in these cases the behaviour of the whole system can be judged. The choice is usually done by dividing the whole space of test cases into equivalence classes so that the behaviour of the system varied very little inside the class and varied a lot if the class is changed. Then for each equivalence class tests are organised. During the execution of the tests the behaviour of the system is analysed and the verification is done. The quality of testing is estimated as a test coverage of equivalence classes.

The amount of classes is always chosen to be finite, because the testing is limited. However, this amount could still be too big for the effective testing of the particular project. Often the test cases and the scenarios of the system under test behaviour are classified by some set of factors, properties and parameters

T. Margaria and B. Steffen (Eds.): ISoLA 2010, Part II, LNCS 6416, pp. 382–396, 2010.

each of which can take a finite number of possible values. In this case, different test cases correspond to all possible combinations of values of selected factors.

Examples of factors used in the test case classification:

- successful/unsuccessful execution of some operation
- branching in the code of the component under test, the value of the corresponding factor is the execution of either 'if' or 'else' branch
- rules' alternatives when testing is based on grammars[33], the values in this case are possible ways to solve the alternative
- categories when testing is based on division on categories[32]

Often (including the cases above) single occurrence of a factor in the resulting test suite is not enough for a qualitative testing. It is required to check the interaction of the factors among them selves i.e. their combined influence on the system. To perform this check it is required to combine the factors to raise the quality of testing and to reduce the test suite at the same time because full examination of all combinations for real systems is very costly.

In this paper methods for combining factors which allow to create test suites covering pairs, threes etc. of factors are surveyed. These methods generate relatively small test suites and increase the quality of testing for systems possessing following properties:

- there is a sort of cases or influences on the system which has quite a lot of parameters or factors which have impact on it's work.
- the possible values of each input parameter can be divided into rather small finite number of equivalence classes so that all considerable changes in system behaviour appear only because of change in class of a single parameter.
- it is known that the errors in the system are mainly caused by some combinations of factors defined by the input parameters' values

When there is no additional information about dependence between system's errors and input parameters or other conditions on values of input parameters these methods effectively create test suites which cover a big variety of cases. Otherwise the additional information can be used to create smaller aimed tests. The methods described in this paper were successfully applied for testing different type of systems see [31,25,37]. For example, in [25] a system under test has 4 input parameters with 3 possible values each. If we try to make up all possible combinations of the parameters' values we will have $3^4 = 81$ tests. It is not many but it is obvious that the execution of all of them is quite costly.

The empirical researches [11,37] confirm that in these kind of situations the majority of errors (up to 70%) are mainly related to some combinations of values of only two factors. In other papers [20] is shown that the combinations of pairs of factors in testing improve code coverage up to 80%. In other words if the tests contain all possible combinations of pairs of parameters' values then the majority of the errors will be found. The minimum amount of tests in the example given in [25] is 9 because covering all pairs of the first two parameters takes 3*3=9 tests. A test suite for this example covering all pairs of possible parameters'

values containing 9 tests can be generated using the techniques described in this paper. The resulting test suite can be found in [25].

We can try to generate a test suite which would still be rather small but also would contain all possible threes of parameters' values. For the example above at least $27 = 3^3$ tests are required. It is possible to generate such a suite using the techniques described in this paper. Using in testing a test suite containing all possible threes of parameters' values can find even more errors[11,37].

This paper surveys methods for test suites creation containing all pairs, threes and lager sets of parameters' values. Section 2 contains definitions for covering arrays and the main information about them. Section 3 is devoted to the survey itself. Finally, conclusion summarises the paper.

2 Preliminaries

Test suites covering all pairs, threes etc. of possible values of input parameters correspond to mathematical term of covering arrays. Consider there are k factors which have influence on system behaviour. First factor has n_1 possible values, second has n_2 values etc. k-th has n_k values. Covering array depth t is a Nxk matrix, where the values of an i-th factor are in i-th column and any combination of possible values of any t factors occurs in at least one row of the matrix.

It is not essential what are the exact values, therefore they can be designated by numbers from 0 to n_{i-1}. A set of numbers $(t; k, n_1...n_k)$ is called a covering array configuration, and a set of covering arrays corresponding to this configuration is designated $CA(t; k, n_1...n_k)$. In configuration $(t; k, n_1...n_k)$ t – is a depth of a covering array, k – is an amount of parameters and n_i – is an amount of possible values of i-th parameter. All numbers in configuration are called attributes of a covering array. Covering array of depth t is also called a t-covering array.

Covering array is minimal if there does not exit covering array for the same configuration containing less amount of rows. The amount of rows in minimal covering array for configuration $(t; k, n_1...n_k)$ is designated $CAN(t; k, n_1...n_k)$.

If the amount of values for all parameters is the same, i.e. $n_1 = n_2 = ... = n_k = n$, corresponding covering array is called homogeneous. It's configuration is designated as $(t; k, n)$, a set of such covering arrays – $CA(t; k, n)$.

There are more complex cases of covering arrays - variable strength covering arrays. In such covering arrays occur all pairs of values of parameters $i_1, ..., i_{p_2} (2 \le p_2 \le k)$, all threes of parameters $j_1, ..., j_{p_3} (3 \le p_3 \le k)$ etc. all t-tuples of parameters $l_1, ..., l_{p_t} (t \le p_t \le k, t \le k)$, where $i_p, j_p, ..., l_p$ are the numbers of parameters from 1 to k. These numbers designated with different letters can coincide. For example, if there are 7 parameters we can be interested in combinations of all pairs for all 7 parameters, threes of 1st, 2nd, 3rd and 4th parameter, 4-tuples for 3rd, 4th, 5th, 6th and 7th parameter (see Fig. 1).
We will designate variable depth covering array configurations as $(t_1; k, n_1...n_k; t_2, j_1...j_{p_2}; ...; t_m; l_1, ..., l_{p_m})$. Fig. 1 gives a visual representation for configuration $(2; 7, n_1...n_7; 3; 1, 2, 3, 4; 4; 3, 4, 5, 6, 7)$.

When test suite is described using covering arrays every row of covering array corresponds to a test and the value in j-th column corresponds to the ordinal

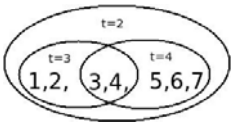


Fig. 1. Graphical representation of a variable depth covering array configuration

number of equivalence class for j-th parameter. This mapping allows us to use the mathematical theory of covering arrays for test suite creation. Because of necessity of reducing the size of test suite it is convenient to use minimal or close to minimal covering arrays.

In general case there is no effective solution for covering array generation problem. Seroussi and Bshouty in [19] proved that generation of $CA(t;k,n)$ is NP-complete using a reduction to the problem of graph 3-colouring. Lei and Tai in [18] proved that generation of $CA(2;k,n)$ is NP-complete using a reduction to the problem of finding a minimum vertex cover. Thereby generation of covering array is NP-complete.

The the amount of rows in minimal covering arrays there is an estimation [40]:$CAN(t;k,n) \leq (t-1)log(k)/log(n^t/(n^t-1))*(1+o(1))$ when k, which gives $CAN(t;k,n) \leq (t-1)n^t log(k)*(1+o(1))$ when $n^t, k \to \infty$. The obvious bottom bound for the amount of rows in minimal covering arrays is $CAN(t;k,n) \leq n^t$, which is based on the notion that all possible combinations of t values each of which can be chosen n different ways must occur in such an array. As we can see the amount of rows grows logarithmically when $k \to \infty$, which makes the usage of covering arrays effective for construction of small and qualitative test suites.

The researches on covering arrays are mainly concentrated on finding minimal covering arrays for different configurations and on creation of effective (polynomial) algorithms for generation of close to minimal covering arrays. There are many well known algorithms possessing these properties but they either work only for particular configurations, either for the majority of configurations give covering arrays far from minimal.

The goal of this paper is to analyse covering arrays generation methods, reveal their area of application where they give minimal or close to minimal covering arrays and finally introduce a new combined technique which will effectively using the most effective known method generate close to minimal covering arrays for different configurations.

3 Survey of Algorithms for Covering Arrays Generation

In this survey we tried to cover all algorithms found in papers [4,22,27,28]. Several additional algorithms were also covered: algorithm for covering arrays generation using initial homogeneous covering array with similar configuration [5] and algorithm for combining blocks of homogeneous covering arrays and additional arrays [25]. We also give such characteristics as time complexity and memory usage which absent for mentioned above survey papers.

Existing algorithms for covering array generation can be classified as follows:

- Combinatorial (direct) algorithms use the mapping between covering arrays and other combinatorial schemes (e.g. sets of Latin squares, finite groups etc.) which allows to create covering array when the corresponding scheme is simple enough.
- Recursive algorithms construct covering arrays out of other covering arrays with e.g. smaller number of parameters or smaller depth
- Reduction algorithms generate covering arrays by modifying and reducing covering arrays with larger values of the attributes.
- Greedy algorithms consider covering array generation as an optimization problem for minimizing the number of rows in generated array using techniques for local extremum search.

This survey describes known solutions in particular cases of the NP-complete problem of covering array generation. The goal of this survey is to find effective algorithms and heuristics for some particular cases because the problem in general due to it's NP-completeness most likely does not have an effective (polynomial) solution. Following properties for the methods found will be defined:

- The range of application of an algorithm i.e. for which covering arrays configuration attributes this particular method generates minimal or near minimal covering arrays and has good time complexity estimations.
- Classes of algorithms used. The dependence of these algorithms on other algorithms or methods
- Algorithm time complexity, memory usage. For recursive and reduction algorithms time complexity will be calculated only for the algorithm itself (without calculation of time for initial array construction)
- Dependence of the next row on already generated rows or the rows specified before. The possibility of generation an array row by row which will give a memory optimization.
- Possibility of addition the semantic constraints reducing the size of an array. Semantic constraints do not allow forbidden combinations for test values, which is very convenient for real world systems.

3.1 Homogeneous Covering Arrays Generation Algorithms

3.1.1 Boolean Covering Array Generation Algorithm [28,29,30]

This direct algorithm generates covering array form $CA(2; k, 2)$ for any $k \geq 1$. The algorithm description can be found in [28]. An array constructed is a minimal covering array. Time complexity is O(k), memory usage is $O(k * log_2(k))$.

3.1.2 Affine Covering Array Generation Algorithm [25,28]

This direct algorithm generates covering array form $CA(t; n+1, n)$ where n is a prime power, $n = p^k$, $k \geq 1$, and any $t \leq n+1$. If $t = 3$ and $n = 2^k$ an array from $CA(t; n+2, n)$ can be built. The algorithm description can be found in [28]. An array constructed is a minimal covering array (either for $(t; n+1, n)$, either

for $(t; n+2, n)$). Time complexity is $O(n^t)$, memory usage is $O(n)$, because an array can be built row by row. However, for the implementation it is required to simulate the arithmetic of polynomials in Galois Field, i.e. create multiplication and addition tables. The table creation will consume $O((log_p n)^4)$ operations, storing tables in memory will consume $O((log_p n)^3)$.

3.1.3 Multiplication Covering Array Generation Algorithm [22,28]
This recursive algorithm generates covering array from $CA(t; k, n_1 * n_2)$ using covering array A from $CA(t; k, n_1)$ and B form $CA(t; k, n_2)$. The number of rows in the final array is equal to multiplication of number of rows in the initial arrays. Algorithm is not effective when the number of rows is large because in this case the generated covering arrays are far from minimal. The algorithm description can be found in [28]. The final covering array may not be minimal even if the initial covering arrays are minimal. Time complexity is $O(k * (1 + N_1 + N_2))$, where N_1 is the number of rows in the first initial array and N_2 is a number of rows in the second initial array. The required memory is the same because the array may be built row by row but the initial arrays must be kept in memory.

3.1.4 Generating a Homogeneous Covering Array with Strength of 2 Using Recursive Constructions [22,28]
This recursive algorithm generates covering array form $CA(2; m*n+1, n)$ using covering array A from $CA(2; m, n)$, where n is a prime power, $n = p^k$, $k \geq 1$. If the number of rows in the initial array is N, then the number of rows in the final array is $N + p^{2^k} - p^k$. Algorithm is effective when the number of rows in the final array is logarithmically proportional to the number of parameters. The algorithm description can be found in [28]. The final covering array may not be minimal even if the initial covering array is minimal. Time complexity is $n^2 + (n+1) * log_2(n+1) + n * m * ((2*n+1)(n^3 n) + n + 1)$ [28] and required memory is $O(N + ((2*n+1)(n^3 n) + n + 1) + m)$, where N is the number of rows in initial array.

3.1.5 Generating a Homogeneous Covering Array with Strength of More than 2 Using Recursive Constructions(Roux Theorem) [2,3,4,5,6,9,22]
This recursive algorithm generates a covering array form $CA(3; 2*k, n)$ using a covering array A^3 from $CA(3; k, n)$ and A^2 from $CA(2, k, n)$. If the number of rows in the first initial array is N_3 and the number of rows in the second initial array is N_2 then the number of rows in the final array is $N_3 + (n-1) * N_2$. When $t > 3$ the algorithm generates a covering array form $CA(t; 2*k, n)$ using $t-1$ initial covering arrays respectively from $CA(t; k, n), ..., CA(t-2; k, n)$ with number of rows $N_t, ..., N_{t-2}$ respectively. The algorithm description can be found in [4]. The final covering array may not be minimal even if the initial covering arrays are minimal. Time complexity is $O(2 * N_{max} * k * (n + N_{max} * (t-3)))$ and required memory is $O(2 * k * N_{max}^2)$ (where N_{max} is the maximum number of rows in the initial arrays (N_t)).

For all the algorithms above there is no possibility of addition semantic constraints reducing the size of the array.

3.2 Heterogeneous and Variable Depth Covering Arrays Generation Methods

3.2.1 Generation Using Initial Homogeneous Covering Array with Similar Configuration [5]

This reduction algorithm generates a heterogeneous covering array basing on a close configuration homogeneous covering array e.g. a covering array from $CA(2; 6, 3, 2, 5, 4, 3, 5)$ basing on initial array from $CA(2; 6, 5)$. Further research of particular cases in which this method works effectively is required. Several ideas can be found in [5].

If we have a homogeneous covering array B from $CA(t; k, n)$, which has already been constructed, where n is chosen to be so that array B contains all rows of required to generate covering array A from $CA(t; k, n_1...n_k)$ (e.g. $n = max(n_1...n_k)$), then we can change the values from B which does not fit in possible values of A with the values which do fit in. And finally, delete all the unnecessary (e.g doubling) rows.

The resulting array depends on the way the unnecessary values are eliminated i.e. it is possible to get different covering arrays for the same configuration. The minimality of the covering array generated using this method cannot be guaranteed, however there are techniques to estimate how close to minimal is the resulting array [5]. Algorithm requires detailed analysis of arrays' configuration and unconventional heuristics. Time complexity and memory usage depend on particular case. The array construction cannot be optimized by addition of semantic constraints while it is possible to introduce the constraints. Also it is not possible to optimize the memory usage because the whole initial array must be kept in memory.

3.2.2 Heterogeneous Arrays Depth t=2 Generation Using Combinations of Blocks of Homogeneous Covering Arrays and Additional Arrays [25]

This recursive algorithm generates covering array A from $CA(2; k, n_1...n_k)$, where $k > max(n_1...n_k)$. Further research is required to generalise this method for the cases of t>2 and variable depth covering arrays. The algorithm description can be found in [25]. The final covering array may not be minimal. The examples given in [25] show that the arrays generated by this algorithm are rather small. An estimation of the final array size is $\lceil log_{n+1}(k) \rceil * (n^2 - 1) + 1$. Time complexity is $n^2 + k * log_2(k)$ and required memory is $O((n^2 * (n+1) + k * (\lceil log_{n+1}(k) \rceil * (n^2 - 1) + 1))$, where $n \geq max(n_1...n_k)$ is the smallest prime degree integer. It is not possible to optimize the memory usage because the whole array is generated at once but not row by row. It is not possible to extend the existing test suite with this method. It is not possible to add semantic constraints.

3.2.3 Recursive Generation of Covering Arrays with Depth 2 [1,14,22,28]

The Profile of an Array

Let's mark by * the entries in the covering array, which can be replaced by any value and the array would still be covering. The profile $(d_1...d_k)$ of an Nxk array is a k-tuple in which the entry d_i is the number of * entries in the i-th column.

This recursive algorithm generates covering arrays depth 2 if the following conditions are satisfied: $\exists A \in CA(2; k, v_1...v_k)$, number of rows is N, with profile $(d_1...d_k)$, and $\forall i \in [1, k] \exists B_i \in CA(2; l_i, w_{i,1}...w_i, l_i)$, number of rows is M_i, with profile $(f_{i,1}...f_i, l_i)$ and for which $w_{i_j} \leq v_i \forall j \in [1, l_i]$. Then using this algorithm a covering array from $CA(2; l_1+...+l_k, w_{1,1}...w_1, l_1, ..., w_{k,1}...w_{k,l_k})$, number of rows is $T = N+max(M_i-d_i)$, can be generated. The algorithm description can be found in [1]. The final covering array may not be minimal. In [35] a method for lowering a size of the final array is shown. Time complexity is $(N + M) * L + \sum_{i=1}^{k}(M_i * l_i)$ and required memory is $(N + M) * L + max_{i=1}^{k}(M_i * l_i)$, where N is the number of rows in the initial array, $M = max_{i=1}^{k}(M_i - d_i), L = \sum_{i=1}^{k} l_i$. It is not possible to optimize the memory usage because the whole array is generated at once but not row by row. It is not possible to add semantic constraints reducing size of the array.

Further we will consider so called greedy algorithms. This type of algorithms is often used to solve NP-complete problems. The common principle of such algorithms is making the locally optimal choice at each stage with the hope of finding the global optimum[26]. For covering arrays generation algorithms it means choosing the next row covering as many uncovered tuples as possible. Greedy algorithms generate covering arrays for any configuration, they allow extend existing test suites, they do not depend on other algorithms but can be significantly improved by crossing with other techniques and, moreover, semantic constraints addition is possible. However, the array constructed is not always minimal and often is far from minimal. For the majority of greedy algorithms the following is true: the better the time complexity the larger the resulting covering array would be. Also greedy algorithms require more memory than others because all uncovered tuples must be stored. Generated for the same configuration covering arrays may differ from each other because several values are taken randomly in process of generation[27].

A separate research should be performed to find the classes of covering arrays configurations for which particular greedy algorithm will be the most efficient. We will leave this problem out of scope in this paper. We will consider greedy algorithms are applicable for generation of covering arrays for any configuration.

3.2.4 Algorithm Based on Addition of a New Parameter (IPO) [18]

This greedy algorithm can be effectively applied for extending the existing covering array from $CA(t; k, n_1...n_k)$ to array from $CA(t; k + p, m_1...m_k...m_{k+p})$, where $n_i \leq m_i, i \in [1, k]$, and $j : j \in [1, k], n_j < m_j$ and/or $p > 0$.

First a covering array depth t for the first t parameters (all combinations) is generated. Add (m+1)st parameter. Horizontal growth: set the next parameter values into a new column to cover as many t-tuples as possible. Vertical growth: add rows to the array to cover the rest t-tuples. Carry on until all parameters are added.

In [18] the algorithms of horizontal and vertical growth for case when t=2 are described. When t>2 these algorithms can be simply modified. Time complexity is: $O(t * k^6 * d^t(d^t + k + d^t * k))$ Total required memory: $O(t * k * d^t)$, where $d = max(n_1...n_k)$. It is not possible to optimize the memory usage because the whole array is generated at once but not row by row. It is not possible to add semantic constraints reducing size of the array in the initial algorithm, but it can be modified so that the constraints would be considered.

3.2.5 Family of Greedy Algorithms Based on Choosing the Best Row from Candidates [7,22,23,24,36]

This family of greedy algorithms can be effectively used for extension of an existing covering array. To implement variable depth covering arrays generation corresponding heuristics should be introduced (see below). For particular configurations the area of application for a heuristic can be found experimentally. For creation a covering array from scratch see [7,17,21,22,23,24,36]. For extension of existing arrays – no researches were found.

The array is built row by row until all t-tuples are covered. Each row is generated N_2 times and the row covering most tuples is chosen. Row is generated by inserting values into parameters in some order. Order is formed by heuristic PR_1 which tells which parameter to be chosen first and heuristic PR_2 to solve the ties. Value for chosen particular parameter is chosen using heuristics VR_1 and VR_2. Algorithm is repeated N_1 times and the best solution is chosen.

The majority of commerce and open source test data generating tools use greedy algorithms for covering array generation. See Table 1 for the heuristics used.

Table 1. Heuristics used in programming tools for covering array generation

	PR_1	PR_2	VR_1	VR_2	N_1	N_2
AETG [7,23]	RAND	-	V_IN_MAX_UC	RAND	≤ 50	≤ 50
TCG [36]	MAX_N	BY_ORD	V_IN_MAX_UC	BY_ORD	1	$max(n_1...n_k)$
DDA [24]	δ	BY_ORD	δ	BY_ORD	1	1
Jenny[17]	RAND	-	No heuristics, exponential time	-	1	1

Symbols used: RAND – the best is chosen randomly, MAX_N – the parameter having the most possible values is chosen first, δ – the parameter which is included in some set number (=δ, see [22]) of uncovered t-tuples is chosen first (or the value covering δ uncovered t-tuples is considered to be the best), BY_ORD – pick the first by order, V_IN_MAX_UC – the value covering in the most uncovered t-tuples (the values of the previous parameters are already fixed) is considered to be the best.

Several tools for which no descriptions of heuristics used were found [21]: CATS, TestCover.com, CaseMaker, Pro-test.

Time complexity depends on heuristics used and can be estimated as: $(k * (f(PR_1) + f(PR_2)) * d * (f(VR_1) + f(VR_2)))$, where $d = max(n_1...n_k)$,$f(PR_1)$, $f(PR_2)$,$f(VR_1)$, $f(VR_2)$ – time complexities for corresponding heuristics. Required memory: $O(t * k * d^t)$, where $d = max(n_1...n_k)$. Covering array can be

built row by row but it is required to store all already built rows in memory – no possibility to optimise. Algorithm allows addition of semantic constraints.

Further greedy algorithms based on solving an optimization problem are considered. The optimization problem for covering arrays generation is defined as follows: Σ – is a set of feasible solutions, arrays which can contain uncovered tuples. Cost $c(S)$ associated with each array $S \in \Sigma$, $c(S)$ – is a number of uncovered tuples. An optimal solution corresponds to a feasible solution with overall (i.e. global) minimum cost. If c(S)=0, then S is a covering array. For each $S \in \Sigma$, a set TS of transformations (or transitions) i.e. changes in array size or array elements' values. Each of this transitions can be used to change S into another feasible solution S'. The set of solutions that can be reached from S by applying a transformation from TS is called the neighbourhood N(S) of S.

3.2.6 Hill Climbing [22]

This greedy algorithm is only effectively applied for improving (in our case - reducing) of existing covering arrays for any configuration. In this case the existing covering array must be chosen as the initial feasible solution.Random transitions are generated for current feasible solution S changing it to S'. If c(S)≤c(S), then S becomes a current feasible solution. The algorithm can get stuck in a local minimum so stopping heuristics are required. To increase the chance of forming a good solution the algorithm should be repeated N_1 times.

In general case time complexity is exponential and has exponential memory usage. It is not possible to optimize the memory usage because the whole array is generated at once but not row by row. It is not possible to add semantic constraints reducing size of the array in the initial algorithm.

3.2.7 Tabu Search [8,22]

This greedy algorithm is effective for the same cases as Hill Climbing. In this case the existing covering array must be chosen as the initial feasible solution. The algorithm for transition generation for current feasible solution can be found in [8]. The next feasible solution has to be accepted by tabu constraints. The algorithms stops when covering array is obtained and it is not possible to generate a smaller covering array for N_2 times. The whole process is repeated N_1 times.

In general time complexity is exponential. However if lower and upper bounds of final covering array size are N_{min} and N_{max} respectively then the time complexity is $N_1 * k * N_{max} * d * (k + logT) + O(d * k * N_{max} * (T + k^2))$, where $d = max(n_1...n_k)$, required memory is $O(k*N_{max}*T)$, where T number of stored arrays for tabu implementation. It is not possible to optimize the memory usage because the whole array is generated at once but not row by row. Tabu work as semantic constraints.

3.2.8 Simulated Annealing [6,10,12,22]

This greedy algorithm is effective for the same cases as Hill Climbing. Further research is required to gather statistics for this algorithm.

This algorithm is a generalization of hill climbing which allows to choose the next solution with less quality than the current solution with probability

$= e^{-(c(S)-c(S))/KT}$, where K is constant, and T is controlling temperature of the simulation. The temperature is lowered by setting $T = \alpha T$, where $\alpha \in R < 1$ is the control decrement. After an stopping condition is met, the current feasible solution is taken as an approximation to the solution of the problem.

In general case time complexity is exponential. If lower and upper bounds of final covering array size are N_{min} and N_{max} respectively then the time complexity is $N_1 * k + O(k^3 * N_{max})$, required memory is $O(k * N_{max})$. It is not possible to optimize the memory usage because the whole array is generated at once but not row by row. It is not possible to add semantic constraints reducing size of the array in the initial algorithm.

3.2.9 Great Deluge Algorithm [13,15,22]
This algorithm - great deluge algorithm is a variant thereof called threshold accepting. This algorithm is similar to simulated annealing but instead of using probability to decide on a move when the cost is higher, a worse feasible solution is chosen if the cost is less than the current threshold - water level. As the algorithm progresses, the water level would be falling in this case rather than raising as in a profit maximizing problem. Sometimes this method shows faster convergence than simulated annealing [13,15].

Algorithms based on solving an optimization problem are almost not used in industrial test data generation tools because of their high memory and time requirements. However these algorithms generate covering arrays closer to minimal than other greedy algorithms. That's why they are mainly used for research purposes. These algorithms are easily generalised for the case of variable depth covering arrays by defining another cost function.

3.2.10 Genetic Algorithms [16,22]
The main idea of genetic algorithms is to maintain a population of putative solutions and to evolve the population from one generation to the next by two operations: mutation makes small local changes in putative solutions, crossover combines part of one solution and part of another. The new generation survives if it is considered to be good enough by some criterion.

In case of covering arrays population of putative solutions is a set of arrays (which may contain uncovered tuples) with fixed size (number of rows) – N. The size of the population is kept constant, equal to M. On each step of algorithm crossovers and mutations are happening. Crossover is a selection of two random arrays and recombination of their rows for obtaining two new arrays having some properties of both parents. Mutation is a change of two new arrays by some rules [16]. Then only M best arrays are kept – the ones having the least number of uncovered tuples.

There were to few researches on application of genetic algorithms to generation of covering arrays. In paper [16] Stardom published the first of results of such an application. Time complexity is $N_1 * M * k^2(N_{max} + d^2)$, where N_1 – is the number of times algorithm is repeated, $d = max(n_1...n_k)$, N_{max} – the upper bound of array size, required memory: $O(M * k * N_{max})$. Additional research must be done to gather some statistics on this method.

3.3 Survey Results

In the table 2 all methods described above are collected. The configurations for which the methods work effectively are shown.

Table 2. Algorithms for covering array construction

Algorithm	Area of application	Algorithm class	Time complexity and memory usage	M	E	SC
Homogeneous covering arrays construction						
Boolean	(2; k, 2)	Direct	O(k), Memory:$k * log_2(k)$	+	-	-
Affine	(t;n+1,n), n-prime power, ($3;2^k$+2,2k), k-natural	Direct	$n^t + (n+1) * log_2(n+1) + O((log_p(n))^4)$, Memory: $O(n) + O((log_p(n))^3)$	+	-	-
Multiplication	$(t; k, n_1 * n_2)$ from $(t; k, n_1)$, size N_1 and $(t; k, n_2)$, size N_2	Recursive	$O(k * (1 + N_1 + N_2))$, Memory: $O(k * (1 + N_1 + N_2))$	-	-	-
Homogeneous, Rec t=2	(2;m*n+1,n) from (2;m,n),size N, n-prime power	Recursive	$n^2 + (n+1) * log_2(n+1) + n * m * ((2 * n + 1)(n^3 n) + n + 1)$, Memory: $O(N + n^2 + m)$	-	-	-
Roux Theorem	(3;2*k,n) from (3;k,n) and (2;k,n), (t;2k,n) from (t;k,n)...(t-2;k,n),$t \geq 4$	Recursive	$O(2 * N_{tmax} * k * (n + N_{tmax} * (t - 3)))$,Memory:$O(2 * k * N^2_{tmax})$, N_{tmax}-number of rows in the largest initial array	-	-	-
Heterogeneous covering arrays construction						
Reduction	Further research required	Reduction	Depends on particular case	-	+	+
Block combining	$(2; k, n_1 ... n_k)$, $k > max(n_1 ... n_k)$ Further research required	Recursive	$n^2 + k * log_2(k)$, Memory: $O((n^2 * (n+1) + k * (\lceil log_{n+1}(k) \rceil * (n^2 - 1) + 1))$, $n \geq d$–the smallest prime degree integer	-	-	-
Heterogeneous, Recursive t=2	$(2; l_1 + ... + l_k, w_{11} ... w_{1l1}, ..., w_{k1} ... w_{klk})$ from $(2; k, v_1 ... v_k)$ with profile $(d_1 ... d_k)$ and $(2; l_i, w_{i1} ... w_{ili})$ with profile $(f_{i,1} ... f_{i,li})$, $\forall i \in [1, k]$ and $w_{ij} \leq v_i \forall j \in [1, l_i]$	Recursive	$(N + M) * L + \sum_{i=1}^{k} (M_i * l_i)$, Memory: $(N + M) * L + max_{i=1}^{k} (M_i * l_i)$, $L = \sum_{i=1}^{k} l_i$, $M = max_{i=1}^{k} (M_i - d_i)$, M_i-size of initial arrays	-	-	-
IPO	Extension $(t; k, n_1 ... n_k)$ to $(t; k + p, m_1 ... m_k ... m_{k+p}), n_i \leq m_i, i \in [1, k]$, $\exists j : j \in [1, k], n_j < m_j$ and/or $p > 0$	Greedy	$O(t * k^6 * d^t (d^t + k + d^t * k))$, Memory: $O(t * k * d^t)$	-	+	+
Methods easily modified for variable depth covering arrays construction						
Greedy, Row by row	Any configurations for which there are no more effective methods. Extension of initial arrays to larger	Greedy	$(k * (f(PR_1) + f(PR_2)) * d * (f(VR_1) + f(VR_2)))$, $f(PR_1)$, $f(PR_2)$, $f(VR_1)$, $f(VR_2)$, Memory: $O(t * k * d^t)$	-	+	+
Hill Climbing	Search for minimal covering array	Optimization	Exponential time and memory	-	-	-
Tabu Search	Search for minimal covering array	Optimization	$N_1 * k * N_{max} * d * (k + logT) + O(d * k * N_{max} * (T + k^2))$, Memory: $O(k * N_{max} * T)$,T-number of arrays stored for tabu implementation	-	-	+
Simulated Anneal.	Search for minimal covering array	Optimization	$N_e * k + O(k^3 * N_{max})$, Memory: $O(k * N_{max})$	-	-	-
Great Deluge	Search for minimal covering array	Optimization	$N_e * k + O(k^3 * N_{max})$, Memory: $O(k * N_{max})$	-	-	-
Genetic	Search for minimal covering array	Genetic	$N_e * M * k^2 (N_{max} + d^2)$, Memory: $O(M * k * N_{max})$, M–size of population	-	-	-

Symbols used: M – is generated array minimal, E – is extension of given array possible, SC – is addition of semantic constraints possible, $d = max(n_1 ... n_k)$ – for configurations like $(t; k, n_1 ... n_k, ...)$, N_e – number of times of algorithms' relaunch, N_{max} – upper bound for number of rows in generated covering array.

The class of covering arrays configurations for which an effective solution exists can be extended by combining methods. To find the effective combination initial covering array configuration should be analysed. Principles for combining method basing on initial configurations are: use direct algorithms combined with recursive for the configurations where it is possible. The resulting arrays will be close to minimal and with good time complexity, use greedy algorithms for extending covering arrays constructed by direct and recursive algorithms. Use IPO for horizontal growth and DDA heuristics for vertical, use reduction algorithms to reduce covering arrays with close configurations constructed by direct and recursive algorithm. The survey has shown that none of the known industrial tools is analysing the input configuration for covering array, and hence, is not using all known techniques for optimal generation. Therefore in known particular cases the tool either will generate a covering array which is larger either will generate it consuming more resources. It seems reasonable and possible to create a tool which would analyse input configuration of a covering array and choose the most effective method or combination of methods for that particular case. This kind of tool would be the most effective solution because it would use and apply in appropriate cases all known methods for covering arrays generation.

4 Conclusion

In this paper the analysis of application area for the algorithms for covering arrays generation is done. The advantages and disadvantages of the algorithms were shown as well as time complexity and required memory. The direct, recursive, greedy algorithms were surveyed. Also the heuristics for reducing the size and the time for construction of covering arrays were surveyed. The area of application for these heuristics was analysed.

The analysis of known methods for covering arrays generation draw a conclusion that the analysis of a covering array configuration and combining methods for covering array generation allows to extend the classes of covering array configurations for which the effective solution exists. None of the existing tools for covering array generation analyses the configuration of the covering array and uses the advantages of all known algorithms and their heuristics for effective covering array generation.

One of the promising direction of further research is the analysis of a covering array configuration and choosing the most effective algorithm or a combination of algorithms for such a configuration. In this paper the principles for combining different algorithms for minimising the size of the array are formalized.

References

[1] Stevens, B., Mendelsohn, E.: New recursive methods for transversal covers. Journal of Combinatorial Designs 7(3), 185–203 (1999)

[2] Sloane, N.: Covering arrays and intersecting codes. Journal of Combinatorial Designs 1(1), 51–63 (1993)

[3] Chateauneuf, M., Kreher, D.: On the state of strength-three covering arrays. Journal of Combinatorial Designs 10(4), 217–238 (2002)
[4] Martirosyan, S., Van Trung, T.: Tran Van Trung: On t-covering arrays. Designs, Codes and Cryptography 32, 323–339 (2004)
[5] Hartman, A., Raskin, L.: Problems and algorithms for covering arrays. Discrete Math. 284, 149–156 (2004)
[6] Cohen, M., Colbourn, C., Ling, A.: Constructing Strength Three Covering Arrays with Augmented Annealing. Discrete Mathematics 308, 2709–2722 (2008)
[7] Cohen, D., Dalal, S., Fredman, M., Patton, G.: The AETG System: an approach to testing based on combinatorial design. IEEE Transactions on Software Engineering 23(7), 437–444 (1996)
[8] Nurmela, K.: Upper bounds for covering arrays by tabu search. Discrete Applied Math. 138, 143–152 (2004)
[9] Colbourn, C., Martirosyan, S., Van Trung, T., Walker II, R.: Roux-type Constructions for Covering Arrays of Strengths Three and Four Designs. Codes and Cryptography 41(1), 33–57 (2006)
[10] Stevens, B.: Transversal Covers and Packings. PhD. Thesis, Mathematics, University of Toronto (1998)
[11] Patton, G.: DAT (Defect Analysis Team)1986-1990 Overview. Internal Bellcore Technical Memo (1991)
[12] Cohen, M., Colbourn, C., Ling, A.: Augmenting Sumulated Annealing to Build Interaction Test Suites. In: Proc Intl. Symposium Software Requirements Engineering, pp. 394–405 (2003)
[13] Dueck, G.: New Optimization Heuristic – The Great Deluge Algorithm and the Record-To-Record Travel. Journal of Computational Physics 104, 86–92 (1993)
[14] Stevens, B., Ling, A., Mendelsohn, E.: A direct construction of transversal coversusing group divisible designs. Ars. Combin. 63, 145–159 (2002)
[15] Dueck, G., Scheuer, T.: Threshold Accepting: A general purpose optimization algorithm appearing superior to simulating annealing. Journal of Computational Physics 90, 161–175 (1990)
[16] Stardom, J.: Metaheuristic and the search for covering and packing arrays. Master's thesis, Simon Fraser University (2001)
[17] Jenkins, B.: Tool for pairwise testing (2005), http://burtleburtle.net/bob/math/jenny.html
[18] Lei, Y., Tai, K.: In-parameter order: A test generation strategy for pairwise testing. In: Proc. 3rd IEEE High Assurance System Engineering Symposium, pp. 254–161 (1998)
[19] Seroussi, G., Bshouty, N.: Vector sets for exhaustive testing of logic circuits. IEEE Trans. Information Theory 34, 513–522 (1988)
[20] Cohen, D., Dalal, S., Parelius, J., Patton, G.: The Combinatorial Design Approach to Automatic Test Generation. IEEE Software, 83–87 (September 1996)
[21] Pairwise Testing, Combinatorial Test Case Generation, http://www.pairwise.org/tools.asp
[22] Colbourn, C.: Combinatorial aspects of covering arrays. Le Matematiche (Catania) 58, 121–167 (2004)
[23] Cohen, D., Dalal, S., Fredman, M., Patton, G.: The Automatic Efficient Test Generator (AETG) System. In: Proceedings of 5th International Symposium on Software Reliability Engineering, November 6-9, pp. 303–309 (1994)
[24] Colbourn, C., Cohen, M., Turban, R.: A deterministic density algorithm for pairwise interaction coverage. In: Proc. of the IASTED Intl. Conference on Software Engineering, pp. 242–252 (February 2004)

[25] Williams, A.: Determination of Test Configurations for Pairwise Interaction Coverage. In: Proceedings of the 13th International Conference on Testing Communicating Systems (Test-Com 2000), pp. 59–74 (2000)
[26] Cormen, T., Stein, C., Rivest, R., Leiserson, C.: Greedy algorithms // Introduction to Algorithms, 2nd edn., vol. 16. Moscow Viliyams (2006)
[27] Grindal, M., Offutt, A., Andler, S.: Combination testing strategies: A survey. Software Testing, Verification, and Reliability 15(3), 167–199 (2005)
[28] Hartman, A.: Software and Hardware Testing Using Combinatorial Covering Suites. In: Proc. Graph Theory, Combinatorics and Algorithms: Interdisciplinary Applications, pp. 266–327 (2005)
[29] Edelman, A.: The mathematics of the Pentium division bug. SIAM Review 39, 54–67 (1997)
[30] Greene, C.: Sperner families and partitions of a partially ordered set. In: Hall Jr., M., van Lint, J. (eds.) Combinatorics, Dordrecht, Holland, pp. 277–290 (1975)
[31] Williams, A., Probert, R.: A measure for component interaction test coverage. In: Proc. ACS/IEEE Intl. Conf. on Computer Systems and Applications, pp. 301–311 (2001)
[32] Ammann, P., Offutt, J.: Using Formal Methods to Derive Test Frames in Category-Partition Testing Safety, Reliability, Fault Tolerance, Concurrency, and Real Time Security. In: Proc. Ninth Ann. Conf. Computer Assurance (COMPASS 1994), pp. 69–79 (1994)
[33] Zelenov, S., Zelenova, S.: Automated negative and positive test generation for testing a syntax analysis phase. Works of Institute for System Programming RAS, russian (2004)
[34] Documentation on Webmoney interface WM Keeper Light, http://www.wmtransfer.com/eng/about/demo/light/index.shtml
[35] Colbourn, C., Martirosyan, S., Mullen, G., Shasha, D., Yucas, J., Sherwood, G.: Products of Mixed Covering Arrays of Strength Two. J. Combin. Des. 14, 124–138 (2006)
[36] Tung, Y.-W., Aldiwan, W.: Automating test case generation for the new generation mission software system. In: Proc. IEEE Aerospace Conf., pp. 431–437 (2000)
[37] Bryce, R., Colbourn, C.: One-Test-at-a-Time Heuristic Search for Interaction Test Suites. In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO), Search-based Software Engineering track (SBSE), London, England, pp. 1082–1089 (2007)
[38] Lidl, R., Niederreiter, H.: Finite fields, vol. 2. Mir, Moscow (1988)
[39] Tables for the smallest known covering arrays, http://www.public.asu.edu/~ccolbou/src/tabby/catable.html
[40] Godbole, A., Skipper, D., Sunley, R.: t-Covering arrays: upper bounds and Poisson approximations. Combinatorics, Probability and Computing 5, 105–118 (1996)

A Scalable Approach for the Description of Dependencies in Hard Real-Time Systems*

Steffen Kollmann, Victor Pollex, Kilian Kempf, and Frank Slomka

Ulm University
Institute of Embedded Systems/Real-Time Systems
{firstname.lastname}@uni-ulm.de

Abstract. During the design iterations of embedded systems, the schedulability analysis is an important method to verify whether the real-time constraints are satisfied. In order to achieve a wide acceptance in industrial companies, the analysis must be as accurate as possible and as fast as possible. The system context of the tasks has to be considered in order to accomplish an exact analysis. As this leads to longer analysis times, there is a tradeoff between accuracy and runtime. This paper introduces a general approach for the description of dependencies between tasks in distributed hard real-time systems that is able to scale the exactness and the runtime of the analysis. We will show how this concept can be applied to a real automotive application.

1 Introduction

In the last years, designing embedded systems has become a great challenge because more and more applications have to be covered by one system. As a special issue of this development, many new features are only realizable by the connection of different controllers. New cars, for example, have up to 75 ECUs (electronic control units) connected by several buses, which leads to a highly networked system with many requirements, like energy consumption, space or timing behavior.

One specific challenge for the design process are the hard real-time constraints of many applications such as an engine management system or an ABS (anti-lock braking system). Therefore it is necessary to verify in each design iteration whether the real-time constraints of single applications are satisfied or not. A schedulability analysis can be performed in order to determine the worst-case response times of the tasks in a system. To achieve a wide acceptance of such real-time analysis techniques by industrial software developers, tight bounds of the real worst-case response times of the tasks and short runtimes of the analysis are important.

A possibility to get tight bounds is the consideration of dependencies of chained tasksets. Previous work considers different kinds of dependencies, like mutual exclusion of tasks, offsets between task stimulation, task chains or tasks competing for the same resource. But the integration of many other types of dependencies is still unsolved. Especially a holistic model as a new abstraction layer for dependencies is missing. Approximation techniques can be used to achieve low runtimes of the analysis. But this is in conflict to tight worst-case response time bounds.

* This work is supported in part by the Carl Zeiss Foundation.

T. Margaria and B. Steffen (Eds.): ISoLA 2010, Part II, LNCS 6416, pp. 397–411, 2010.

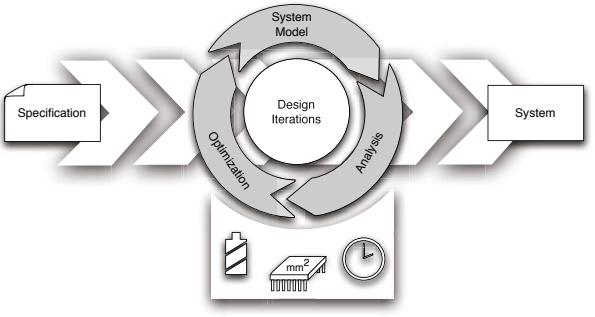


Fig. 1. Design flow of embedded systems

The integration of such real-time analysis techniques into a design process is depicted in figure 1. Starting at the specification, the system model is extracted. The system architecture is then analyzed regarding the different requirements and it is optimized based on the verification results. In early design iterations many different solutions should be considered and therefore it is necessary to have fast verification algorithms. This can be achieved by approximation techniques. In each following iteration the system model is refined and the possible design solutions are bounded. So more and more information about the architecture is available and the verification can work more accurately. While unfortunately this results in longer analysis runtimes, the runtime increase is reasonable because less design solutions have to be considered in consecutive iterations. So it is necessary to have a scalable model which can exploit this tradeoff between runtime and accuracy.

We present such a new holistic model that integrates different types of dependencies into real-time analysis and enables to adjust the level of detail. We show how this general model can be integrated into the schedulability analysis of fixed-priority systems and outline how this approach can be used to approximate the consideration of dependencies in the system and therefore improve the analysis time.

This paper is organized as follows: In section 2 an overview of the related work is given. The model and the corresponding real-time analysis are defined in section 3. Section 4 defines the limiting event stream model and shows how it can be used to describe dependencies in a system in a scalable way. The impact of the introduced analysis is illustrated by a real automotive case-study. The work closes with a conclusion.

2 Related Work

The real-time analysis for distributed systems was introduced by Tindell and Clark [16]. In this holistic schedulability analysis, tasks are considered as independent. This idea has been improved by the transaction model [15], which allows the description of static offsets between tasks. Gutierrez et al. [3] extended this work to dynamic offsets so that the offset can vary from one job of a task to another. Furthermore they have introduced an idea about mutual exclusion of tasks [4] which bases on offsets between tasks. Since

Gutierrez et al. have considered simple task chains, Redell has enhanced the idea to tree-shaped dependencies [13] and Pellizoni et al. applied the transaction model to earliest deadline first scheduling in [12].

Henia et al. used the SymTA/S approach [14] to extend the idea of the transaction model in order to introduce timing-correlations between tasks in parallel paths in distributed systems [5]. This idea has then been improved in [6]. Recently, Rox et al. [7] have described a correlation between tasks caused by an non-preemptive scheduler.

A scalable and modular approach to analyse real-time systems is the real-time calculus (RTC) as presented by Wandeler in [17]. Unfortunately it is not possible to describe dependencies like offsets or mutual exclusion of tasks with the RTC.

Because of the lack of generality or exactness the RTC and the SymTA/S approach are combined in [10], but the authors do not consider any kind of dependency. Another disadvantage of the combination of the models is that the transformation from one model into the other model leads to inaccuracy.

3 Real-Time Analysis

3.1 Model of Computation

In this section we introduce the model necessary for the real-time analysis discussed in section 3.2.

Task Model. Γ is a set of tasks mapped onto the same resource $\Gamma = \{\tau_1, ..., \tau_n\}$. A task is a tuple $\tau = (c^+, c^-, b, d, \rho, \Theta^+, \dot{\Theta}^+)$ consisting of : c^+ the worst-case execution time, c^- the best-case execution time, b the blocking time, d the relative deadline of the task, ρ the priority of the task, Θ^+ the maximum incoming stimulation and $\dot{\Theta}^+$ the maximum outgoing stimulation.

Let τ_{ij} be the j-th job/execution of task τ_i. We assume that each job of a task generates an event at the end of its execution to notify other tasks. Furthermore we define $\Gamma_{hp,\tau}$ as a taskset including only tasks having a higher priority than task τ.

Event Model. Event streams have been first defined in [2]. Their purpose is to provide a generalized description for any kind of stimulation. The basic idea is to define an event function $\eta(\Delta t, \Theta^+)$ which can calculate for every interval Δt the maximum number of events occurring within Δt (when speaking of an interval, we mean the length of the interval). The event function needs a properly described model behind it which makes it easy to extract the information.

The idea of the maximum event streams is to note for each number of events the minimum interval which can include this number of events. Therefore we get an interval for one event, two events and so on. The interval for one event is infinitely small and therefore considered to be zero. The result is a sequence of intervals showing a non-decreasing behavior. This is because the minimum interval for n events cannot be smaller than the minimum interval for $n-1$ events, as the first interval also includes $n-1$ events.

Definition 1. *An event stream is a set of event stream elements* $\theta : \Theta^+ = \{\theta_1, \theta_2, ..., \theta_n\}$ *and each event stream element* $\theta = (p, a)$ *consists of an offset-interval a and a period p. The event stream complies to the characteristic of sub-additivity:* $\eta(\Delta t_1 + \Delta t_2, \Theta^+) \leq \eta(\Delta t_1, \Theta^+) + \eta(\Delta t_2, \Theta^+)$ *and is monotonically increasing:* $\forall \Delta t_1, \Delta t_2 : \Delta t_1 \leq \Delta t_2 \Rightarrow \eta(\Delta t_1, \Theta^+) \leq \eta(\Delta t_2, \Theta^+)$

Each event stream element θ describes a set of intervals $\{a_\theta + k \cdot p_\theta | k \in \mathbb{N}_0\}$ of the sequence. With an infinite (∞) period it is possible to define a single interval in order to model irregular behavior. Event tuples having infinity as period are called aperiodic elements and event tuples having a period less than infinity are called periodic elements. The event function is defined as follows:

Definition 2. *The event function denotes for an event stream* Θ^+ *and an interval* Δt *the maximum number of events*[1]*:*

$$\eta(\Delta t, \Theta^+) = \sum_{\substack{\theta \in \Theta^+ \\ a_\theta \leq \Delta t}} \left\lfloor \frac{\Delta t - a_\theta}{p_\theta} \right\rfloor + 1 \tag{1}$$

As pseudo-inverse function we define the interval function which denotes the minimum interval in which a given number of events can occur.

Definition 3. *The interval function denotes for an event stream* Θ^+ *and a number of n events the corresponding minimum interval in which these events can occur:*

$$\Delta t(n, \Theta^+) = \inf_{\Delta t \geq 0} \{\Delta t | \eta(\Delta t, \Theta^+) \geq n\} \tag{2}$$

A detailed definition of the concept and the mathematical foundation of the event streams can be found in [1].

An event pattern can be described by the event stream model in several ways. For an efficient implementation of the approach we normalize the event streams as introduced in [8]. With this normalization, the interval function can be efficiently computed. For a detailed description on how to transform an event stream to a normalized event stream and how the interval function can be described efficiently see [8]. In the following we use definition 4.

Definition 4. *A normalized event stream* $\tilde{\Theta}^+$ *has the form*

$$\{(\infty, a_1), \ldots, (\infty, a_m), (p, a_{m+1}), \ldots, (p, a_n)\} : (1 \leq m \leq n \wedge a_i \leq a_j \Leftrightarrow i \leq j \wedge a_n - a_{m+1} \leq p)$$

which means that the event stream has first an aperiodic part and then a periodic part where each periodic element has the same period and the events occur within the same periods. All elements are sorted by their offsets. We also define that $N^{\infty}_{\tilde{\Theta}^+}$ is the number of aperiodic elements of an event stream and $N^{p}_{\tilde{\Theta}^+}$ is the number of periodic elements of an event stream. With this definition and the methodologies introduced in [8] and [9] an efficient real-time analysis for distributed systems is possible. For the rest of the paper we assume that we use only normalized event streams.

[1] Since in this case study we consider a fixed priority non-preemptive scheduler, the floor operation is used. This is also a bound for preemptive schedulers, but for these, better bounds can be given.

3.2 Holistic Real-Time Analysis

Based on previous work we define the real-time analysis with event streams. As described in [16], in each global iteration step of the real-time analysis the worst-case response time and the outgoing stimulation for each task in the system are computed until a fix-point is found. The real-time analysis with event streams is described in [9]. In that paper the computation of the worst-case response time, the best-case response time, and the outgoing stimulations are given. Since the paper at hand focuses on the improvement of the worst-case response time of a task, only that part of the analysis will be reproduced here.

Based on the worst-case response time analysis in [11] we can define the analysis for event streams as follows:

Lemma 1. *The worst-case response time of a task is bounded by:*

$$r^+(\tau) = \max_{k\in[1,\ldots,m]}\{r^+(k,\tau) - \Delta t(k,\Theta_\tau^+)\}\ m = \min_{k\in\mathbb{N}}\{k | r^+(k,\tau) \leq \Delta t(k+1,\Theta_\tau^+)\}$$
$$r^+(k,\tau) = \min\{\Delta t | \Delta t = b_\tau + k\cdot c_\tau^+ + \sum_{\tau_i\in\Gamma_{hp,\tau}} \eta(\Delta t,\tau_i)\cdot c_{\tau_i}^+\} \quad (3)$$

Proof. The proof is given in [9].

4 Considering Dependencies

The lack of the previously discussed model is that dependencies between stimulations caused by the system context are not considered during the real-time analysis. Therefore we will extend the model introduced in section 3 by the limiting event streams. Via this extension we are able to describe different kinds of dependencies with a single holistic model.

The idea of this new technique is depicted in figure 2. Four tasks are mapped onto a resource that is scheduled non-preemptively with fixed-priorities. For each task we determine the worst-case response time. Assume task τ_4 as the analyzed task. If no correlations between the stimulations are considered the worst-case response time of the task is computed by equation (3). This means the interference produced by the tasks τ_1 to τ_3 is maximal concerning τ_4.

Now assume that the stimulations have an offset dependency to each other. In order to model such a correlation, the stimulations can be considered as vertices in a graph and the correlation between them as edges. For each maximal clique in the graph of size 2 or bigger, a set S_i is build which contains all stimulations that are represented by the nodes of the clique. Note that these cliques are usually given as input and are not being searched for. To consider every possibility which is described by the correlations, the power set $\mathscr{P}(S_i)$ of such a previously mentioned set is taken. For each element $M \in \mathscr{P}(S_i)$ of such a power set a limitation for the number of events that can occur by the combined stimulations in M is computed.

This very modular concept permits to consider correlations in a scalable way by choosing only a subset of the power set which should be considered in the analysis. See the three possibilities in figure 2. In part a) only one limitation over all stimulations is

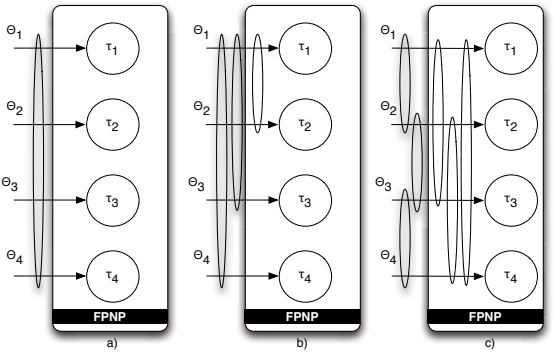


Fig. 2. Possible consideration of relations by limiting event streams

used. Out of all stimulations of the corresponding power set $\mathscr{P}(S_i)$ the second example uses only those that contain exactly the stimulations of the k-highest priority tasks that are mapped on the resource, where $2 \leq k \leq n$ and n is the number of tasks mapped on a resource. The last one c) considers only the pairwise correlations. That way the developer has the chance to determine the level of detail in the analysis. Next we will formulate the limiting event streams formally.

Definition 5. *The limiting event stream is an event stream that defines the maximum occurrence of events for a set of event streams. The limiting event stream is defined as* $\overline{\Theta} = (\Theta^+, \overrightarrow{\Theta})$. Θ^+ *describes the limitation for a set of event streams* $\overrightarrow{\Theta}$. *The limiting event stream fulfills the condition:*

$$\eta(\Delta t, \Theta^{\pm}_{\overline{\Theta}}) \leq \sum_{\Theta \in \overrightarrow{\Theta}_{\overline{\Theta}}} \eta(\Delta t, \Theta)$$

Example 1. If no correlations between event streams are defined, the limiting event stream is definded by: $\overline{\Theta} = (\cup_{\Theta \in \overrightarrow{\Theta}} \Theta, \overrightarrow{\Theta})$.

Example 2. Assume $\Theta^+_A = \Theta^+_B = \{(20,0)\}$ and an offset of 10 t. u. between these two event streams. The cumulated occurrence of events when the offset is not considered is described by $\eta(\Delta t, \Theta^+_A \cup \Theta^+_B)$. The cumulated occurrence considering the offset of the events can be described by the limiting event stream $\overline{\Theta} = (\{(20,0), (20,10)\}, \{\Theta_A, \Theta_B\})$. If we consider the event streams as independent, two events occur in an interval $\Delta t = 5$ t. u. $(\eta(\Delta t, \Theta^+_A \cup \Theta^+_B))$. In contrast, in the same interval only one event occurs if the dependency is considered $(\eta(\Delta t, \Theta^{\pm}_{\overline{\Theta}}))$.

Next we define how a limiting event stream can be calculated.

Definition 6. *Let* $\Delta\beta : \mathbb{N} \to \mathbb{R}$ *be a limiting interval function which assigns a minimal time interval from a given number of events in subject to a relationship of event streams* $\overrightarrow{\Theta} := \{\Theta_1, \ldots, \Theta_n\}$, *then a limitation for a limiting event stream* $\overline{\Theta}$ *can be determined by:*

$$\Theta^{\pm}_{\overline{\Theta}} := \nu(\overrightarrow{\Theta}_{\overline{\Theta}}, \Delta\beta(n))$$

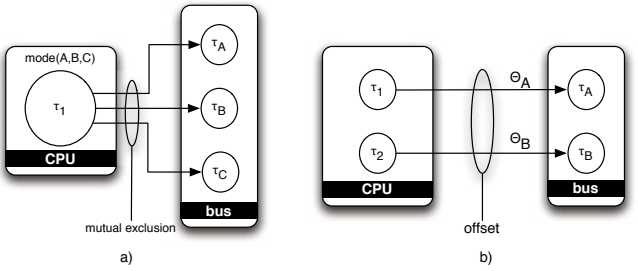


Fig. 3. Two possible dependencies in a system

Note that $\nu(\overrightarrow{\Theta}_{\overline{\Theta}}, \Delta\beta(n))$ and $\Delta\beta(n)$ are abstract functions which have to be concretized for the different types of dependencies. Next we will show how it is possible to describe two completely different kinds of dependencies with this new technique.

4.1 Mutual Exclusion

The first dependency we will consider is the mutual exclusion of event streams. Assume a task is transmitting different messages over a bus as depicted in figure 3 a). The transmission mode of the task depends on its input data. Only one message type can be sent in a transmission mode. In this case the message types exclude each other. Such a behavior can be described by the following lemma.

Lemma 2. *Let $\overrightarrow{\Theta}_{\overline{\Theta}}$ be a set of event streams having a mutual exclusion relation to each other. The limiting interval function is given by:*

$$\Delta\beta(n) = \min_{\Theta \in \overrightarrow{\Theta}_{\overline{\Theta}}} \{\Delta t(n, \Theta)\} \tag{4}$$

Proof. Mutual exclusion of event streams means that only one event stream $\Theta \in \overrightarrow{\Theta}_{\overline{\Theta}}$ is active at the time. Since the activation of these event streams is switching arbitrarily in a time interval it is necessary to determine for each time interval Δt which event stream delivers the most events in this interval. Therefore the maximum number of events in an interval is bounded by $\eta(\Delta t, \Theta^{+}_{\overline{\Theta}}) = \max_{\Theta \in \overrightarrow{\Theta}_{\overline{\Theta}}} \{\eta(\Delta t, \Theta)\}$. This can be transformed using equation (2) as follows:

$$\begin{aligned}
\eta(\Delta t, \Theta^{+}_{\overline{\Theta}}) &= \max_{\Theta \in \overrightarrow{\Theta}_{\overline{\Theta}}} \{\eta(\Delta t, \Theta)\} \\
\Delta\beta(n) &= \inf_{\Delta t \geq 0} \{\Delta t | \max_{\Theta \in \overrightarrow{\Theta}_{\overline{\Theta}}} \{\eta(\Delta t, \Theta)\} \geq n\} \\
\Delta\beta(n) &= \min_{\Theta \in \overrightarrow{\Theta}_{\overline{\Theta}}} \{ \inf_{\Delta t \geq 0} \{\Delta t | \eta(\Delta t, \Theta) \geq n\}\} = \min_{\Theta \in \overrightarrow{\Theta}_{\overline{\Theta}}} \{\Delta t(n, \Theta)\}
\end{aligned}$$

Therefore the assumption holds. □

After defining the limiting interval function we will formulate how the concrete limiting event stream can be derived. Note that mode changes of tasks result in different execution times and have influence on the outgoing event streams. Therefore the outgoing event streams of one task are not necessarily identical in their aperiodic behavior. Only the periodic behavior is the same for all.

Lemma 3. *Via lemma 2 the limiting event stream for mutual exclusion for event streams* $\overrightarrow{\Theta}_{\overline{\Theta}}$ *having the same period and equal parameters concerning* N^{∞} *and* N^{P} *can be derived as follows:*

$$\nu(\overrightarrow{\Theta}_{\overline{\Theta}}, \Delta\beta(n)) = \bigcup_{i=1}^{N_{\Theta}^{\infty}} (\infty, \Delta\beta(i)) \cup \bigcup_{i=N_{\Theta}^{\infty}+1}^{N_{\Theta}^{\infty}+N_{\Theta}^{P}} (p, \Delta\beta(i)) \tag{5}$$

where $\Theta \in \overrightarrow{\Theta}_{\overline{\Theta}}$.

Proof. Let the minimum of two event tuples be defined as:

$$\min(\theta_1, \theta_2) := (\min(p_{\theta_1}, p_{\theta_2}), \min(a_{\theta_1}, a_{\theta_2}))$$

Furthermore let $\theta_{i,j}$ be the i-th event tuple of the j-th event stream and $\Theta \in \overrightarrow{\Theta}_{\overline{\Theta}}$. According to [8] each event stream $\Theta \in \overrightarrow{\Theta}_{\overline{\Theta}}$ can be transformed so that $\forall \Theta_i, \Theta_j \in \overrightarrow{\Theta}_{\overline{\Theta}} : |\Theta_i| = |\Theta_j|$ holds. From this it follows that

$$\bigcup_{i=1}^{|\Theta|} \min_{\Theta_j \in \overrightarrow{\Theta}_{\overline{\Theta}}} \{\theta_{i,j}\}$$

is a general upper bound for mutual exclusion.

As we consider only event streams produced by the same task, all outgoing event streams have the same periodic behavior. Therefore in conjunction with [8] all event streams in $\overrightarrow{\Theta}_{\overline{\Theta}}$ can be transformed so that N_{Θ}^{p} and N_{Θ}^{∞} are equal for all $\Theta \in \overrightarrow{\Theta}_{\overline{\Theta}}$. From this we can follow:

$$\begin{aligned} \bigcup_{i=1}^{|\Theta|} \min_{\Theta_j \in \overrightarrow{\Theta}_{\overline{\Theta}}} \{\theta_{i,j}\} &= \bigcup_{i=1}^{N_{\Theta}^{\infty}} \min_{\Theta_j \in \overrightarrow{\Theta}_{\overline{\Theta}}} \{\theta_{i,j}\} \cup \bigcup_{i=N_{\Theta}^{\infty}+1}^{N_{\Theta}^{\infty}+P_{\Theta}^{\infty}} \min_{\Theta_j \in \overrightarrow{\Theta}_{\overline{\Theta}}} \{\theta_{i,j}\} \\ &= \bigcup_{i=1}^{N_{\Theta}^{\infty}} (\infty, \Delta\beta(i)) \cup \bigcup_{i=N_{\Theta}^{\infty}+1}^{N_{\Theta}^{\infty}+P_{\Theta}^{\infty}} (p, \Delta\beta(i)) \end{aligned}$$

Therefore the assumption holds. □

With the lemma stated above it is possible to consider mutual exclusion in distributed systems.

4.2 Offsets

In order to show the generality of our new approach we show how static offsets between tasks can be described by limiting event streams. For simplicity we consider here only strictly periodic stimulations, which are used in our case study. The methodology can also be used for arbitrary stimulations.

See figure 3 part b) where two different tasks are depicted each sending a message on a bus. All sending tasks have an offset to each other in order to prevent a peak load on the bus. This kind of modeling is often used for CAN bus communications. First of all we give a definition of the offsets.

Definition 7. *When a set of event streams $\overrightarrow{\Theta}_{\overline{\Theta}}$ has an offset based relation to each other, each event stream has an offset attribute ϕ_{Θ^+} describing an offset to a specific point in time.*

Next we define how the limiting interval function can be derived.

Lemma 4. *Let $\overrightarrow{\Theta}_{\overline{\Theta}}$ be a set of event streams having an offset relation to each other. And let $\tilde{\Theta}^+_\cup = \bigcup_{\Theta\in\overrightarrow{\Theta}_{\overline{\Theta}}} \bigcup_{\theta\in\Theta}(p_\theta, a_\theta + \phi_\Theta)$ be the normalized union of the offset-shifted event streams. Then the limiting interval function is given by:*

$$\Delta\beta(n) = \min_{j=1,\ldots,|\tilde{\Theta}^+_\cup|}(\Delta t(j+(n-1),\tilde{\Theta}^+_\cup) - \Delta t(j,\tilde{\Theta}^+_\cup)) \tag{6}$$

Proof. Since only static offsets are considered, the union of the offset shifted event streams describes the cumulated occurrence of the events. From this union we have to extract the minimum interval for each number of events. So we have to search over all possible combinations, which can be done by a sliding window approach. From this consideration it follows:

$$\Delta\beta(n) = \min_{j\in\mathbb{N}}(\Delta t(j+(n-1),\tilde{\Theta}^+_\cup) - \Delta t(j,\tilde{\Theta}^+_\cup))$$

So we have to show that the bound $j = 1,\ldots,|\tilde{\Theta}^+_\cup|$ in lemma 4 includes the minimal interval for a number of events. The following condition holds:

$$\Delta t(q,\tilde{\Theta}^+_\cup) - \Delta t(q - N^p_{\tilde{\Theta}^+_\cup},\tilde{\Theta}^+_\cup) = p_{\tilde{\Theta}^+_\cup} : q > |\tilde{\Theta}^+_\cup|$$

So it follows for $j > |\tilde{\Theta}^+_\cup|$ that $\Delta t(j+n-1,\tilde{\Theta}^+_\cup) - \Delta t(j+n-1-N^p_{\tilde{\Theta}^+_\cup},\tilde{\Theta}^+_\cup) = p_{\tilde{\Theta}^+_\cup}$ and $\Delta t(j,\tilde{\Theta}^+_\cup) - \Delta t(j - N^p_{\tilde{\Theta}^+_\cup},\tilde{\Theta}^+_\cup) = p_{\tilde{\Theta}^+_\cup}$ holds. From this we can follow:

$$\begin{aligned}
&\Delta t(j+n-1,\tilde{\Theta}^+_\cup) - \Delta t(j,\tilde{\Theta}^+_\cup)\\
&= \Delta t(j+n-1,\tilde{\Theta}^+_\cup) - \Delta t(j,\tilde{\Theta}^+_\cup) - p_{\tilde{\Theta}^+_\cup} + p_{\tilde{\Theta}^+_\cup}\\
&= (\Delta t(j+n-1,\tilde{\Theta}^+_\cup) - p_{\tilde{\Theta}^+_\cup}) - (\Delta t(j,\tilde{\Theta}^+_\cup) - p_{\tilde{\Theta}^+_\cup})\\
&= (\Delta t(j+n-1,\tilde{\Theta}^+_\cup) - \Delta t(j+n-1,\tilde{\Theta}^+_\cup) + \Delta t(j+n-1-N^p_{\tilde{\Theta}^+_\cup},\tilde{\Theta}^+_\cup)) -\\
&\quad (\Delta t(j,\tilde{\Theta}^+_\cup) - \Delta t(j,\tilde{\Theta}^+_\cup) + \Delta t(j - N^p_{\tilde{\Theta}^+_\cup},\tilde{\Theta}^+_\cup))\\
&= (\Delta t(j+n-1-N^p_{\tilde{\Theta}^+_\cup},\tilde{\Theta}^+_\cup)) - \Delta t(j - N^p_{\tilde{\Theta}^+_\cup},\tilde{\Theta}^+_\cup)
\end{aligned}$$

So we have shown that a minimum interval must also occur one period earlier and therefore also in the bound of the search interval. □

After having shown how the limiting interval function can be defined we will now introduce how the concrete event stream can be derived.

Lemma 5. *With the help of the last lemma we can derive the concrete limiting event stream for an offset-based dependency with* $j = N^{\infty}_{\tilde{\Theta}^+_{\cup}}$ *by the following equation:*

$$\nu(\overrightarrow{\Theta}_{\overline{\Theta}}, \Delta\beta(n)) = \bigcup_{i=1}^{j} (\infty, \Delta\beta(i)) \cup \bigcup_{i=j+1}^{j+N^{p}_{\tilde{\Theta}^+_{\cup}}} \left(p_{\tilde{\Theta}^+_{\cup}}, \Delta\beta(i)\right) \tag{7}$$

Proof. We have to show that the periodic behavior starts at the assumed value. Assume the following variables: $1 \leq j \leq |\tilde{\Theta}^+_{\cup}|$, $e = N^{\infty}_{\tilde{\Theta}^+_{\cup}} + m + kN^{p}_{\tilde{\Theta}^+_{\cup}}$ and $1 \leq m \leq N^{p}_{\tilde{\Theta}^+_{\cup}}$ and the following equation giving the minimum interval for $N^{\infty}_{\tilde{\Theta}^+_{\cup}} + m$ events where j is chosen accordingly:

$$\Delta\beta(N^{\infty}_{\tilde{\Theta}^+_{\cup}} + m) = \Delta t(j + (N^{\infty}_{\tilde{\Theta}^+_{\cup}} + m - 1), \tilde{\Theta}^+_{\cup}) - \Delta t(j, \tilde{\Theta}^+_{\cup})$$

So we have to show that the following assumption holds for e events :

$$\Delta\beta(e) = \Delta\beta(N^{\infty}_{\tilde{\Theta}^+_{\cup}} + m) + kp_{\tilde{\Theta}^+_{\cup}} \tag{8}$$

Assume $1 \leq j_1 \leq |\tilde{\Theta}^+_{\cup}|$ and $\Delta t(j_1 + (e-1), \tilde{\Theta}^+_{\cup}) - \Delta t(j_1, \tilde{\Theta}^+_{\cup})$ is the smallest interval for e events then it follows:

$$\begin{aligned}
&\Delta t(j_1 + (e-1), \tilde{\Theta}^+_{\cup}) - \Delta t(j_1, \tilde{\Theta}^+_{\cup}) \\
&= \Delta t(j_1 + (N^{\infty}_{\tilde{\Theta}^+_{\cup}} + m + kN^{p}_{\tilde{\Theta}^+_{\cup}} - 1), \tilde{\Theta}^+_{\cup}) - \Delta t(j_1, \tilde{\Theta}^+_{\cup}) \\
&= \Delta t(j_1 + (N^{\infty}_{\tilde{\Theta}^+_{\cup}} + m + kN^{p}_{\tilde{\Theta}^+_{\cup}} - 1), \tilde{\Theta}^+_{\cup}) - \Delta t(j_1, \tilde{\Theta}^+_{\cup}) + \\
&\quad \Delta t(j_1 + (N^{\infty}_{\tilde{\Theta}^+_{\cup}} + m - 1), \tilde{\Theta}^+_{\cup}) - \Delta t(j_1 + (N^{\infty}_{\tilde{\Theta}^+_{\cup}} + m - 1), \tilde{\Theta}^+_{\cup}) \\
&= \Delta t(j_1 + (N^{\infty}_{\tilde{\Theta}^+_{\cup}} + m - 1), \tilde{\Theta}^+_{\cup}) - \Delta t(j_1, \tilde{\Theta}^+_{\cup}) + kp_{\tilde{\Theta}^+_{\cup}}
\end{aligned}$$

But this cannot be smaller than assumption (8) because $kp_{\tilde{\Theta}^+_{\cup}}$ can not be smaller than $kp_{\tilde{\Theta}^+_{\cup}}$ and $\Delta t(j_1 + (N^{\infty}_{\tilde{\Theta}^+_{\cup}} + m - 1) - \Delta t(j_1, \tilde{\Theta}^+_{\cup})$ cannot be smaller than $\Delta t(j + (N^{\infty}_{\tilde{\Theta}^+_{\cup}} + m - 1), \tilde{\Theta}^+_{\cup}) - \Delta t(j, \tilde{\Theta}^+_{\cup})$. So the assumption holds. □

After having derived how static offsets can be described by limiting event streams we will now describe the worst-case response time analysis with the limiting event streams.

4.3 Worst-Case Response Time Analysis with Limiting Event Streams

In order to use limiting event streams it is necessary to adapt this new concept to the worst-case response time analysis from section 3.2. For this we have to determine the worst-case interference of tasks in an interval Δt when limiting event streams are considered. Due to space limitation we show only the sketch of the proof.

Lemma 6. *Let τ be the task under analysis and $\overline{\Theta}$ be a limiting event stream and $\Gamma_{\overline{\Theta},\tau,\tau_i} := \{\tau_j \in \Gamma_{hp,\tau} | (c^+_{\tau_j} > c^+_{\tau_i} \vee (c^+_{\tau_j} = c^+_{\tau_i} \wedge \rho_{\tau_j} > \rho_{\tau_i})) \wedge \Theta^+_{\tau_j} \in \overrightarrow{\Theta}_{\overline{\Theta}}\}$, then the worst-case response time is bounded by:*

$$r^+(\tau) = \max_{k\in[1,\ldots,m]} \{r^+(k,\tau) - \Delta t(k,\Theta^+_\tau)\} \quad m = \min_{k\in\mathbb{N}} \{k | r^+(k,\tau) \leq \Delta t(k+1,\Theta^+_\tau)\} \tag{9}$$

$$r^+(k,\tau) = \min\{\Delta t | \Delta t = b_\tau + k\cdot c^+_\tau + \sum_{\tau_i\in\Gamma_{hp,\tau}} \overline{\eta}(\Delta t,\tau_i,k,\tau)\cdot c^+_{\tau_i}\} \tag{10}$$

$$\overline{\eta}(\Delta t,\tau_i,k,\tau) = \min(\max(\min_{\forall\overline{\Theta}|\Theta^+_{\tau_i}\in\overrightarrow{\Theta}_{\overline{\Theta}}} \{\overline{\overline{\eta}}(\Delta t,\tau_i,k,\tau,\overline{\Theta})\},0),\eta(\Delta t,\Theta^+_{\tau_i})) \tag{11}$$

$$\overline{\overline{\eta}}(\Delta t,\tau_i,k,\tau,\overline{\Theta}) = \begin{cases} \eta(\Delta t,\Theta^+_{\overline{\Theta}}) - \sum_{\tau_j\in\Gamma_{\overline{\Theta},\tau,\tau_i}} \overline{\eta}(\Delta t,\tau_j,k,\tau) & \Theta^+_\tau \notin \overrightarrow{\Theta}_{\overline{\Theta}} \\ \eta(\Delta t,\Theta^+_{\overline{\Theta}}) - \sum_{\tau_j\in\Gamma_{\overline{\Theta},\tau,\tau_i}} \overline{\eta}(\Delta t,\tau_j,k,\tau) - k & \Theta^+_\tau \in \overrightarrow{\Theta}_{\overline{\Theta}} \end{cases} \tag{12}$$

Proof. Compared to (3), (10) differs only in the interference of the higher priority tasks. Equation (11) describes the number of task activations in a given interval Δt of higher priority tasks. It is the minimum of the amount described by the limiting event streams $\max(\min_{\forall\overline{\Theta}|\Theta^+_{\tau_i}\in\overrightarrow{\Theta}_{\overline{\Theta}}} \{\overline{\overline{\eta}}(\Delta t,\tau_i,k,\tau,\overline{\Theta})\},0)$ and the amount described by the regular event stream of the task $\eta(\Delta t,\Theta^+_{\tau_i})$. If none of the event streams of the higher priority tasks is part of a limiting event stream, (11) is reduced to the regular event stream

$$\min(\max(\min_{\forall\overline{\Theta}|\Theta^+_{\tau_i}\in\overrightarrow{\Theta}_{\overline{\Theta}}} \{\},0),\eta(\Delta t,\Theta^+_{\tau_i})) = \min(\max(\infty,0),\eta(\Delta t,\Theta^+_{\tau_i})) = \eta(\Delta t,\Theta^+_{\tau_i}))$$

resulting in (3) and (10) being identical.

So we have to show that the interference is maximal if limiting event streams are considered. Let $\sum_{\tau_i\in\Gamma_{hp,\tau}} \overline{\eta}(\Delta t,\tau_i,k,\tau)\cdot c^+_{\tau_i}$ be the maximum load of higher priority tasks in an interval Δt. Since every $\overline{\Theta} : \Theta^+_{\tau_i} \in \overrightarrow{\Theta}_{\overline{\Theta}}$ is a valid bound for the number of events in an interval of a task τ_i, and $\eta(\Delta t,\Theta^+_{\tau_i})$ is the regular bound for the maximum stimulation of a task in an interval Δt, we can follow:

$$\overline{\eta}(\Delta t,\tau_i,k,\tau) = \min(\max(\overline{\overline{\eta}}(\Delta t,\tau_i,k,\tau,\overline{\Theta}),0),\eta(\Delta t,\Theta^+_{\tau_i})) \tag{13}$$

where $\overline{\overline{\eta}}(\Delta t,\tau_i,k,\tau,\overline{\Theta})$ is the maximum number of events limited by the liminting event stream $\overline{\Theta}$ for task τ_i. Since for one limiting event stream the condition $\eta(\Delta t,\Theta^+_{\overline{\Theta}}) \leq \sum_{\Theta\in\overrightarrow{\Theta}_{\overline{\Theta}}} \eta(\Delta t,\Theta)$ holds we can follow $\eta(\Delta t,\Theta^+_{\overline{\Theta}}) = \sum_{\tau_i|\Theta^+_{\tau_i}\in\overrightarrow{\Theta}_{\overline{\Theta}}} \overline{\eta}(\Delta t,\tau_i,k,\tau)$. From this it follows: $\overline{\overline{\eta}}(\Delta t,\tau_i,k,\tau,\overline{\Theta}) = \eta(\Delta t,\Theta^+_{\overline{\Theta}}) - \sum_{\tau_j|\Theta^+_{\tau_j}\in\overrightarrow{\Theta}_{\overline{\Theta}}/\Theta^+_{\tau_i}} \overline{\eta}(\Delta t,\tau_j,k,\tau)$. This can be inserted in (13) and we get:

$$\overline{\eta}(\Delta t,\tau_i,k,\tau) = \min(\max(\eta(\Delta t,\Theta^+_{\overline{\Theta}}) - \sum_{\tau_j|\Theta^+_{\tau_j}\in\overrightarrow{\Theta}_{\overline{\Theta}}/\Theta^+_{\tau_i}} \overline{\eta}(\Delta t,\tau_j,k,\tau),0),\eta(\Delta t,\Theta^+_{\tau_i}))$$

In order to achieve that the interference of the higher priority tasks $\Gamma_{hp,\tau}$ is maximal, the events must be distributed so that the task with the greatest worst-case execution time is triggered as often as possible then the task with the second greatest worst-case execution

time and so on. For tasks having the same worst-case execution time, the order for the distribution is given by the priority. From this it follows that the task τ_i is subject to the maximum interference when the available events $\eta(\Delta t, \Theta^+_{\overline{\Theta}})$ are first distributed on the tasks in the taskset $\Gamma_{\overline{\Theta},\tau,\tau_i} := \{\tau_j \in \Gamma_{hp,\tau} | (c^+_{\tau_j} > c^+_{\tau_i} \vee (c^+_{\tau_j} = c^+_{\tau_i} \wedge \rho_{\tau_j} > \rho_{\tau_i})) \wedge \Theta^+_{\tau_j} \in \overrightarrow{\Theta}_{\overline{\Theta}}\}$ and the remaining events on task τ_i:

$$\overline{\eta}(\Delta t, \tau_i, k, \tau) = \min(\max(\eta(\Delta t, \Theta^+_{\overline{\Theta}}) - \sum_{\tau_j \in \Gamma_{\overline{\Theta},\tau,\tau_i}} \overline{\eta}(\Delta t, \tau_j, k, \tau), 0), \eta(\Delta t, \Theta^+_{\tau_i}))$$

Since all limting event streams are a bound for the number of events we have to take the overall minimum and we finally get:

$$\overline{\eta}(\Delta t, \tau_i, k, \tau) = \min(\max(\min_{\forall \overline{\Theta} | \Theta^+_{\tau_i} \in \overrightarrow{\Theta}_{\overline{\Theta}}} \{\eta(\Delta t, \Theta^+_{\overline{\Theta}}) - \sum_{\tau_j \in \Gamma_{\overline{\Theta},\tau,\tau_i}} \overline{\eta}(\Delta t, \tau_j, k, \tau)\}, 0), \eta(\Delta t, \Theta^+_{\tau_i}))$$

Therefore the assumption holds. The proof of equation (12) case two includes only k jobs, because the task τ under analysis is part of the limitig event stream. The proof for this case is analog to the last one. □

As initial value for the fix-point iteration in equation (10) we take $\Delta t(k, \Theta^+_\tau) + c^+_\tau$. Note that the complexity of the response time analysis is still pseudo-polynominial. The complexity to calculate the limiting event streams depends on the kind of the dependency which is considered. The analysis, however, is not affected by this problem. So it is reasonable to find upper bounds for the limiting event streams to improve the runtime performance.

5 Example and Results

In order to show the impact of the introduced new technique we consider a 500 kBit/s CAN bus. This automotive application is depicted in figure 4. The system comprises nine electronic control units transmitting data on the CAN bus. The number of different messages sent by each ECU is stated as "Tx" in the illustration. There are 47 messages in total, which all are transmitted periodically. All stimulations have an initial offset to each other in order to prevent peak load on the bus and to provide smaller response times of the messages. These offsets are defined by the designer of the system. Furthermore two pairs of messages (one pair on ECU3 and one pair on ECU6) have a mutual exclusion dependency.

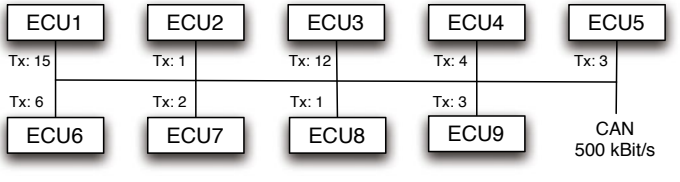


Fig. 4. Can bus communication of an automotive architecture

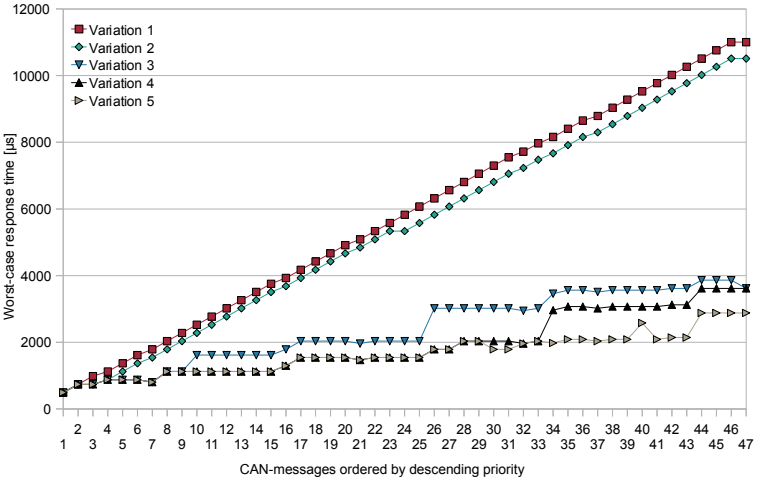


Fig. 5. Absolute worst-case response times of the messages

The system has been analyzed in five different levels of detail. In the *first variation* we have analyzed the system without considering any dependencies. As *second variation* only the mutual exclusion of the messages has been considered. The *third variation* additionally includes the offsets with only one limiting event stream over all message stimulations from an ECU as depicted in figure 2 part a). The *fourth variation* increases the level of detail of the offset consideration by the variation represented in figure 2 part b). The last and therefore the *fifth variation* increases also the level of detail of the offset dependencies by the variation c) of figure 2. The results are depicted in figure 5.

As expected, including more dependencies and therefore increasing the level of detail leads to better worst-case response times of the tasks. The consideration of the mutual exclusion leads to a nearly constant improvement of the response times of all messages, which can be seen in figure 5. This does not apply to the consideration of the offset dependencies. As can be seen for variation 3, the impact on the messages is different. The next level of detail leads to an improvement of the response times of messages 10 to 47. The last curve describes the highest level of detail and leads to a further improvement of the response times for messages with a small priority. So it can be seen that the different possibilities to integrate the dependencies into the analysis lead to very different results. For a better impression of the improvements we have described the worst-case response times of message 42 in table 1.

Based on the discussion above we now consider the corresponding runtimes shown in the table 2. All runs were performed on a machine with a 2.66 GHz Intel processor

Table 1. Worst-Case response times of message 42

Variation 1	Variation 2	Variation 3	Variation 4	Variation 5
10020 μs	9528 μs	3618 μs	3126 μs	2142 μs

Table 2. Runtimes of the different variations

Variation 1	Variation 2	Variation 3	Variation 4	Variation 5
82 ms	92 ms	301 ms	821 ms	1428 ms

and 4 GB of system memory. The runtime when considering no dependencies is the shortest. The inclusion of mutual exclusion leads to almost no increase of the runtime. The inclusion of offsets results in a noteworthy increase of the runtime. The simple case of consideration of the offset dependency leads to nearly 209 ms of more runtime. The next improvement gains an additional 520 ms of runtime with the benefit that many messages have better worst-case response times. The last variation and therefore the most detailed has a runtime of 1428 ms, but only some messages have a better response time in this variations.

Variation 3 is best suited for a design space exploration because here the quotient of the runtime increase and the average reduction of response times is minimal. As mentioned in the introduction, another possibility is to start the design process with nearly no dependency correlation and to increase the level of detail in later design iterations where more accurate results are necessary.

6 Conclusion

In this paper we have introduced a holistic model to describe task dependencies in distributed real-time systems. The new approach has been applied to a real automotive architecture. We have shown for two different dependencies how they can be described by this new model. We have cut the complexity of the dependencies from the real-time analysis, which has not been achieved in previous work. Additionally we have shown how the consideration of dependencies can be approximated very easily by this new concept.

Finally a case study has been conducted to show the improvements of the approach. It has been shown that our model works for different kinds of dependencies. Furthermore we have shown that the introduced new approximation technique can lead to significant improvements of the runtimes. In the future we will show how to integrate other dependencies into this new model.

References

1. Albers, K., Slomka, F.: An event stream driven approximation for the analysis of real- time systems. In: ECRTS 2004: Proceedings of the 16th Euromicro Conference on Real-Time Systems, pp. 187–195. IEEE, Los Alamitos (July 2004)
2. Gresser, K.: An event model for deadline verification of hard real-time systems. In: Proceedings of the 5th Euromicro Workshop on Real-Time Systems (1993)
3. Gutierrez, J.C.P., Harbour, M.G.: Schedulability analysis for tasks with static and dynamic offsets. In: RTSS, p. 26 (1998)

4. Gutierrez, J.C.P., Harbour, M.G.: Exploiting precedence relations in the schedulability analysis of distributed real-time systems. In: IEEE Real-Time Systems Symposium, pp. 328–339 (1999)
5. Henia, R., Ernst, R.: Context-aware scheduling analysis of distributed systems with tree-shaped task-dependencies. In: DATE 2005: Proceedings of the conference on Design, Automation and Test in Europe, pp. 480–485 (2005)
6. Henia, R., Ernst, R.: Improved offset-analysis using multiple timing-references. In: DATE 2006: Proceedings of the conference on Design, automation and test in Europe, pp. 450–455 (2006)
7. Jonas Rox, R.E.: Exploiting inter-event stream correlations between output event streams of non-preemptively scheduled tasks. In: Proc. Design, Automation and Test in Europe (DATE 2010) (March 2010)
8. Kollmann, S., Albers, K., Slomka, F.: Effects of simultaneous stimulation on the event stream densities of fixed-priority systems. In: Spects 2008: Proceedings of the International Simulation Multi-Conference, IEEE, Los Alamitos (June 2008)
9. Kollmann, S., Pollex, V., Slomka, F.: Holisitc real-time analysis with an expressive event model. In: proceedings of the 13th Workshop of Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (2010)
10. Kuenzli, S., Hamann, A., Ernst, R., Thiele, L.: Combined approach to system level performance analysis of embedded systems. In: CODES+ISSS 2007: Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis, pp. 63–68. ACM, New York (2007)
11. Lehoczky, J.P.: Fixed priority scheduling of periodic task sets with arbitrary deadlines. In: Proceedings of the 11th IEEE Real-Time Systems Symposium, pp. 201–209 (December 1990)
12. Pellizzoni, R., Lipari, G.: Improved schedulability analysis of real-time transactions with earliest deadline scheduling. In: RTAS 2005: Proceedings of the 11th IEEE Real Time on Embedded Technology and Applications Symposium, pp. 66–75 (2005)
13. Redell, O.: Analysis of tree-shaped transactions in distributed real-time systems. In: ECRTS 2004: Proceedings of the 16th Euromicro Conference on Real-Time Systems (ECRTS 2004), pp. 239–248 (2004)
14. Richter, K.: Compositional Scheduling Analysis Using Standard Event Models - The SymTA/S Approach. Ph.D. thesis, University of Braunschweig (2005)
15. Tindell, K.: Adding time-offsets to schedulability analysis. Tech. rep., University of York, Computer Science Dept, YCS-94-221 (1994)
16. Tindell, K., Clark, J.: Holistic schedulability analysis for distributed hard real-time systems. Microprocessing and Microprogramming 40, 117–134 (1994)
17. Wandeler, E.: Modular Performance Analysis and Interface-Based Design for Embedded Real-Time Systems. Ph.D. thesis, ETH Zurich (September 2006)

Verification of Printer Datapaths Using Timed Automata*

Georgeta Igna and Frits Vaandrager

Institute for Computing and Information Sciences
Radboud University Nijmegen, the Netherlands
{g.igna,f.vaandrager}@cs.ru.nl

Abstract. In multiprocessor systems with many data-intensive tasks, a bus may be among the most critical resources. Typically, allocation of bandwidth to one (high-priority) task may lead to a reduction of the bandwidth of other tasks, and thereby effectively slow down these tasks. WCET analysis for these types of systems is a major research challenge. In this paper, we show how the dynamic behavior of a memory bus and a USB in a realistic printer application can be faithfully modeled using timed automata. We analyze, using Uppaal, the worst case latency of scan jobs with uncertain arrival times in a setting where the printer is concurrently processing an infinite stream of print jobs.

1 Introduction

Modern embedded systems are characterized by distributed implementation platforms that include a heterogeneous mix of several processors, one or more buses for communication, and a variety of sensing and actuating devices. They have to operate in dynamic and interactive environments, and need to carry out a mix of data-intensive computational tasks and event-processing control tasks. Not only functional correctness is important, but also quantitative properties related to timeliness, quality-of-service, resource usage and energy consumption. The complexity of embedded systems and their development trajectories is thus increasing rapidly. At the same time, development trajectories are expected to deliver products that are inexpensive and performing, while meeting stringent time-to-market constraints. The complexity of the designs and the constraints imposed on the development trajectory dictate a systematic, model-driven design approach that leverages reuse and is supported by tooling whenever possible.

In multiprocessor systems with many data-intensive tasks, a bus may be among the most critical resources, and severely degrade the timing predictability. The problem is that allocation of bandwidth to one (high-priority) task may lead to a reduction of the bandwidth of other tasks, and thereby effectively slow down these tasks. If we do not want this to occur, for instance in the case of

* Research supported by the Netherlands Ministry of Economic Affairs under the Bsik program within the Octopus project, and by the European CommunityâĂŹs Seventh Framework Programme under grant agreement no 214755 (QUASIMODO).

T. Margaria and B. Steffen (Eds.): ISoLA 2010, Part II, LNCS 6416, pp. 412–423, 2010.

safety critical systems, then we may use e.g. a time division multiple access (TDMA) strategy on the buses in order to give each task a guaranteed bandwidth. However, for most systems such a solution is too expensive. According to Williams et al. [1], for the foreseeable future off-chip memory bandwidth will often be the constraining resource in system performance of multicore computers. Clearly, WCET analysis for such systems is a major research challenge. Existing performance analysis techniques are not able to accurately predict WCETs for systems with this type of highly dynamic resource behavior. Simulation of detailed models certainly provides insight, but fails to provide WCETs in settings with uncertain job arrival times, dynamic and interactive environments and/or uncertain processing times.

In this paper, we show how the dynamic behavior of a memory bus and a USB in a realistic printer application can be faithfully modeled using timed automata. In addition, we show how to compute WCETs (latencies) for the application using the model checker Uppaal [2,3,4]. The case study that we describe here originates from the Octopus [5] project. Octopus is a cooperation between Océ Technologies, the Embedded Systems Institute and several academic research groups in the Netherlands. Its objective is the development of new methods and techniques to support model-driven design space exploration for embedded systems. Some preliminary work from the Octopus project was reported in [6]. There, we considered a simplified version of an Océ printer architecture. Using this architecture, we studied the differences among three modeling formalisms and supporting tools used in the project: Uppaal [2,4,3], Colored Petri Nets [7,8] and Synchronous Dataflow Graphs [9,10]. In this paper, we present a detailed model of a realistic printer design which, in particular, includes a description of the scheduling rules used by the Océ printer controller[1]. We analyze, using Uppaal, the worst case latency of scan jobs with uncertain arrival times in a setting where the printer is concurrently processing an infinite stream of print jobs.

The purpose of this paper is to show that the Uppaal model checker can handle the complexity of dynamic memory bus behavior in a realistic model of a complex industrial application. To the best of our knowledge, no other analysis technique/tool, except maybe the hybrid method of [11], is currently able to do a performance analysis for this type of systems (involving a dynamic memory bus and uncertain arrival times). Existing techniques for WCET analysis of distributed embedded systems, such as Modular Performance Analysis [12,13], SymTA/S [14] and MAST [15] are not applicable since they lead to overly conservative analysis results. In [11], a hybrid method is proposed for analyzing embedded real-time systems that integrates modular performance analysis and timed automata. It would be interesting to use our detailed Uppaal models of the memory bus and USB as part of this hybrid method.

[1] For reasons of confidentiality, we have changed resource names, some other details and all the numerical data in our model. As a consequence, the outcomes of our analysis do not apply to any design of Océ. However, as part of the project we have succeeded to carry out a similar analysis for an actual Océ design.

The structure of this paper is as follows. The next section introduces the printer case study. In Section 3, the timed automata models are described. The analysis results is detailed in Section 4. Concluding remarks and discussions of future work follow in Section 5.

2 Case Study

The hardware architecture analyzed in this paper is depicted in Figure 1. A user can utilize this machine for copying, scanning or printing. He can either use paper or digital files. In case of paper, he must first scan it. With digital files, he can connect to Data Store both remotely and locally via USB. The upload and download through the USB behave differently depending on the bus usage: if the transfer is unidirectional, it is faster than when it is bidirectional. A user has a large variety of image processing (IP) options like zooming, rotation, or filtering, etc. Depending on these preferences, there are different components needed to process the files. A *datapath* is the complete trajectory of an image data from source (i.e. Scanner) to target (i.e. Printer). The performance of the various datapaths is of critical importance in the Océ printer design.

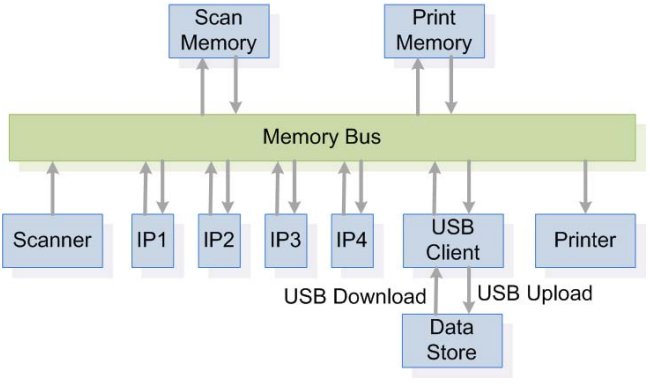
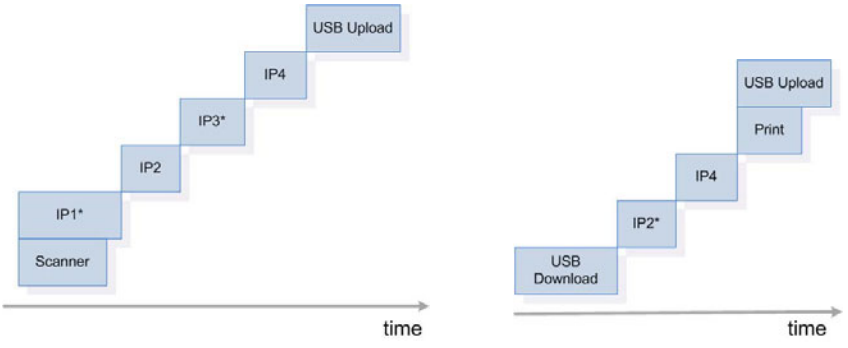


Fig. 1. An Océ Printer Architecture

Here we analyze two common datapaths[2] (Figures 2a and 2b). Since they are often used in practice, it is important to see what can happen in the worst case. In the figures, we can also observe the dependencies among the resources used. They are of three types. In the first category, two resources end in the same time (see Scanner - IP1). The second refers to the sequential dependency between two resources (e.g IP2 - IP3). The last case is the parallel execution between Upload and Printer.

A user specifies his input in the form of a job. A *job* is a tuple made of an input file, a datapath and some image processing settings. We use the term *scan*

[2] The resources marked with * are optional.



(a) Scan to Email (b) Print From Docbox

Fig. 2. Datapaths

job for a job which uses the Scan to File datapath and *print job* for a job which uses the Print from DocBox datapath. In addition, the scan jobs utilize Scan Memory and print jobs only Print Memory. These memories limit the number of concurrent images in the system.

The machine uses specific scheduling rules to solve the conflicts that may occur among the concurrent jobs. We present here the most important ones, which we have also implemented in our model.

The first rule is *files non-overtaking*: files that use the same datapath are processed in the order they enter the system.

The second rule is referred to as *bus throttling*. Memory bus is shared by all the resources when they transfer data to memories. Each resource claims different percentage of the memory bus, but the maximum bandwidth available is not enough for all the resources. Further, the execution time of a resource is limited by the bandwidth it occupies, the internal processing time being negligible. Therefore, a good bandwidth management is important for improving system's performance. When more than one job occupies the system, often incoming jobs do not have enough bandwidth available to start. In such situations, the Océ bus throttling rule is applicable. It uses a priority list to solve the bus conflicts (Figure 3). The resources at the top of the list have the highest priority. If a resource with high priority needs some bandwidth to process a job, it receives it. This potentially enforces a reduction of the bandwidth used by other running resources with lower priority, such that the total bandwidth used does not exceed a maximum (Algorithm 1). For the case when the resource has low priority, it gets the difference between the maximum bandwidth allowed and the current bandwidth used (line 11). Similarly, when one resource with high priority has finished a job, it releases the bandwidth, which is then redistributed back to other running resources based on their priority level (Algorithm 2). The bandwidth redistribution is reflected in the resource processing speed. As mentioned above, the speed of a resource directly depends on the bandwidth used. This

means that, whenever its bandwidth is modified, the expected completion time for the current job is also changed. From another perspective, this rule induces dynamic behavior in the Océ machines, which is not easy to predict.

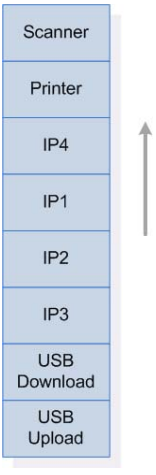


Fig. 3. Resource priority order

bandwidth allocation algorithm 1 void allocateBandwidth(resource r, bw_claimed)

```
1: bandwidth(r)=bw_claimed;
2: bandwidth available− = bandwidth(r);
3: if (bandwidth available ≤ 0 then
4:    if there are resources with lower priority in the priority list then
5:       repeat
6:          take the resources from the bottom up
7:          adjust their bandwidth
8:       until the bandwidth available gets greater or
9:       equal to 0 or we arrived at resource r
10:   else
11:      bandwidth(r)=-bandwidth available;
12:      bandwidth available = 0;
13:   end if
14: end if
```

The third rule is 'upload in order'. The USB client is one of the slowest resources in the system. Therefore often many jobs wait for uploading and they should be served in order. A list is used for keeping track of the waiting jobs. There is a strict rule when jobs are added to it. A scan job is inserted after the scanner has completed it, whereas the jobs which use the Print from Doc Box datapath are added after IP4 has processed them. The same also happens when

bandwidth deallocation algorithm 2 void deallocateBandwidth(resource r, bw_claimed)

1: $available\_bandwidth+ = bandwidth(r)$
2: bandwidth(r)=0;
3: **if** there are resources with lower priority in the priority list and their bandwidth used is less than the maximum bandwidth that they can use **then**
4: **repeat**
5: take them from top down
6: increase their bandwidth up to their maximum
7: **until** the bandwidth available gets equal to 0
8: **end if**

IP2 is shared but in this case we add scan jobs after scanning and print jobs after downloading.

The next rule modeled refers to the conflicts between the scan and the print jobs in case of shared resources. Whenever a scan and a print job claim a resource simultaneously, the print job gets priority.

Finally, we assume that all the resources in the system are non-lazy. This means that if a resource is available when a job claims it, it should be granted immediately. This restriction greatly reduces the complexity of the control software, and also the state space.

3 Model Description

The timed automata model is structured as follows. Each resource is described by a specific automaton (Figure 4), except the two memories and the memory bus, which are simply modeled as shared integer variables. A resource stays in the IDLE location until is claimed by a job. When a job grabs it, the resource computes the bandwidth it can use and its processing speed, applying the bus throttling rule when needed. Then, the resource stays in the RUNNING location until either the job is processed or its speed is changed. For the latter case, the transition between RUNNING and UPDATE_WORK is urgently taken. On the transition back to the RUNNING state, the current job's data is updated. First we under-approximate the clock which monitors the execution time of the job to the closest integer lower or equal to the clock value (select statement: i:int[0,max_exec_time] and guard: $i \leq x$ && $i+1 > x$). Then the current job's unprocessed data is computed. Using this resource template, the model is accurate. However, the state space is fragmented with every speed change, which is a problem for scalability. All the resources, except for the Printer and the Scanner, use this template. The Printer and the Scanner are never interrupted after they start. Therefore, they do not need the UPDATE_WORK location.

Each job type is modeled as a separate automaton. Figure 5 displays an automaton representing a simplified version of the Print from DocBox job. We see there actions specific to both the datapath and memory management. In addition to these, we also observe the implementation of the upload in order

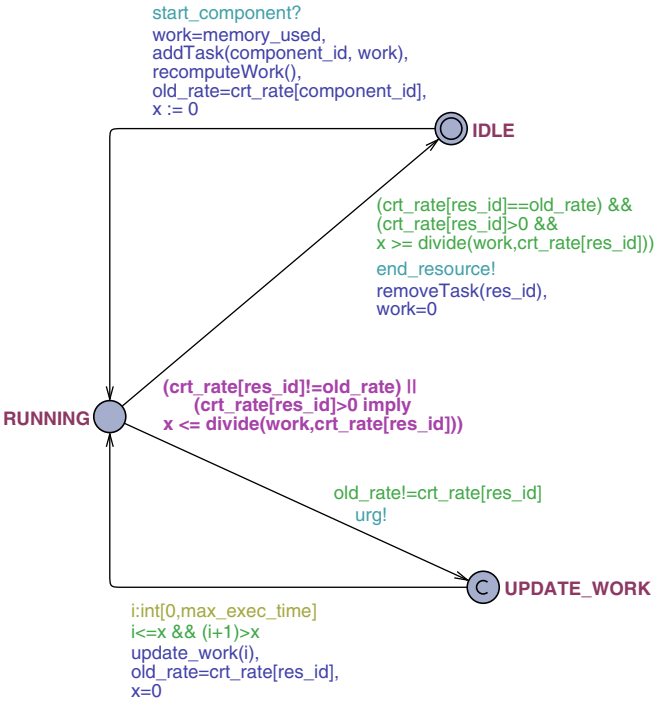


Fig. 4. Resource Automaton

scheduling rule. For simplicity, the variables regarding job non-overtaking are not specified.

4 Verification

We considered the following concurrent datapaths:

Scan To Email: Scanner ⇝ IP1 ⇝ IP2 ⇝IP4 ⇝ USB Upload

Print From Docbox: USB Download ⇝ IP4 ⇝ Print ⇝ USB Upload, with dependencies as in Figures 2a and 2b. Our goal was to find the worst case latency for the files coming from Scanner when they had uncertain arrival time and the print jobs formed an infinite stream[3].

The scheduling rules implemented in our model simplified the analysis. However, the model was nondeterministic. One cause was the uncertain arrival times of scan jobs. The other cause was the lack of partial order reduction techniques implemented in Uppaal: when multiple independent actions occurred simultaneously, Uppaal analyzed all the alternatives, and this led to state space explosion. Therefore, we searched for modalities to simplify the latter type of nondeterminism. The solution adopted was to specify priorities among all the channels

[3] All the experiments were performed using Uppaal version 4.1.2 on a Sun Fire X4440 server with 16 cores (AMD Opteron 8356, 2.3GHz), with 128 GB of DDR2 RAM.

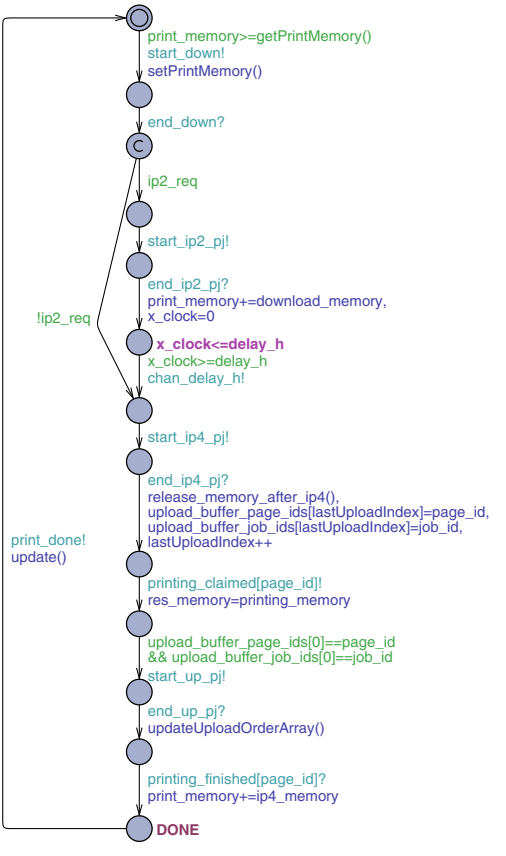


Fig. 5. Print from Doc Box Automaton

declared in the system. However, when scan and print jobs accessed shared resources, they used the same channels. Due to this, we declared separate channels and gave higher priority to the channels employed for the communication between print jobs and resources.

The property verified was

```
A[]     ((forall (i:int[0,max_scan_jobs-1])
    !A1S(i).INIT imply A1S(i).latency_clock<=worst_latency)),
```

where worst_latency was found manually.

Figure 6 shows the monotonic increase of the worst latency with the increase of the number of concurrent scan jobs. In these experiments, no print job was allowed in. As we can see, the increase stops when the machine reaches the maximum scan job capacity that it can process, 19 in this case. The last point in the figure shows that, if the system is fully loaded with scan jobs, a user has to wait more than two times for his outcome comparing to the case when there

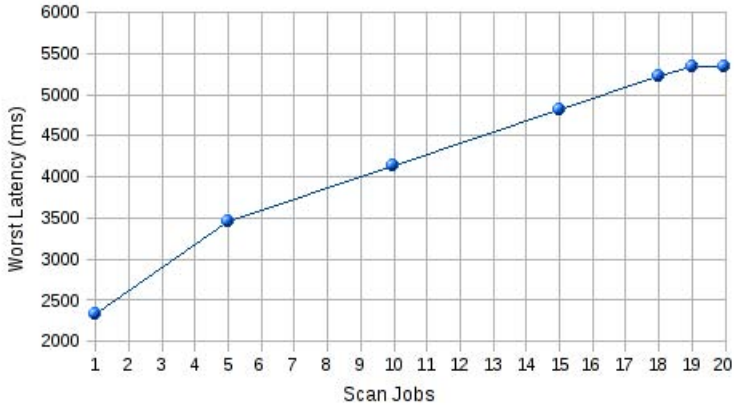


Fig. 6. Worst scan job latency without print jobs in the system

are no other concurrent jobs. In these experiments, the Uppaal running time is insignificant and bus throttling is not needed.

The analysis results of the worst scan job latency in the presence of print jobs is listed in Table 1. All these experiments contained 19 scan jobs with uncertain arrival times and a number of infinite concurrent print jobs indicated in the first column. Table 1 also shows the Uppaal analysis details. Unfortunately, due to long Uppaal running time, we could not observe the worst scan job latency in combinations with high number of infinite print jobs. The current model configuration allows far more than 20 print jobs present in the system. However, the important observation which we can make is that the influence of the print jobs upon the worst case latency of scan jobs is not negligible.

Table 1. Worst scan job latency with print jobs in the system

Infinite Print Job No	Peak Mem Usage(KB)	Running Time(s)	States Explored	Worst Latency(ms)
0	121784	4376.03	723	5341
5	473952	6459.28	349951	6711
10	1107012	15274.40	996693	9843
15	1561524	8934.50	897307	12411
20	-	no result in 24 hours	-	-

During the analysis, we observed that the bus throttling rule had two negative effects. On one hand, as Table 1 also shows, this bus arbitration rule worsened the scan job latency by slowing the execution time of the resource required within their datapath. On the other hand, it increased the analysis time due to many changes in the speed of some resources which the analysis had to explore.

Further, we searched for improvements in the bus throttling rule but firstly we dimensioned the model such that we were able to analyze it fully with Uppaal. In this sense, the scan memory was reduced by a factor of 2, whereas the print memory was reduced by a factor of 4. Except for these, nothing else was changed (the scan jobs had an uncertain arrival time and the print jobs formed an infinite stream). As a result, fewer jobs occupied the machine at some point in time. The analysis of this configuration is detailed in Table 2. This model allowed maximum 5 concurrent scan jobs and 13 concurrent print jobs.

When we searched for changes of the bus arbitration rule, we had to take into account that only the last four resources in the priority list (see Figure 3) could be interrupted while processing a job. The optimization we found, was to switch the order between USB upload and USB download. With this simple change, we obtained the worst latencies showed in Table 3. The improvement was between 4.8% and 7.6%.

Table 2. Worst scan job latency of the dimensioned model

Infinite Print Job No	Peak Mem Usage(KB)	Running Time(s)	States Explored	Worst Latency(ms)
5	195700	2082.01	147806	4941
10	315540	2232.54	294789	6426
12	384896	2315.72	464711	6767
13	434572	2478.34	555770	6940
14				6940

Table 3. Worst scan job latency with the improved bus throttling rule

Infinite Print Job No	Peak Mem Usage(KB)	Running Time(s)	States Explored	Worst Latency(ms)
5	205684	1940.63	153259	4701
10	304464	2074.39	271351	5933
12	339864	2240.14	331047	6274
13	539340	2539.96	921847	6445

To conclude, we analyzed a timed automata model built for an Océ printer architecture. The model contained many design details. We searched for the worst latency of one of the concurrent applications. During the analysis the Uppaal running time was long, on the order of days. Further, for a better understanding of the system, we dimensioned the model such that we saw the peak value of the worst latency and searched for optimization of one important scheduling rule. Currently, we discuss this improvement with the Océ engineers if it can be implemented in the controller of a printer with similar characteristics.

5 Conclusions

We have analyzed an Océ printing machine and two of its datapaths. We computed the worst latency of one datapath which has uncertain arrival time and the other datapath is infinitely often used. Our results show a strong dependency between the two datapaths.

As usual with model checking, long running time was a key issue within our case study. In order to be able to do the model checking (within reasonable time), we had to slightly scale down some of the parameters in the model. Still, the current version of Uppaal is close to the point where it can handle the complexity of industrial designs. One technical issue that we faced is that although essentially the behavior of the model is fully deterministic when all the scheduling rules are added, the resulting Uppaal model is not (and suffers from state space explosion) due to interleaving of internal actions of the various resources. We resolved this by using the channel and process priorities from Uppaal, but a better solution would be to extend Uppaal with support for confluence detection and/or partial order reduction.

We computed the worst latency by repeatedly checking an invariant property. Using a binary search we managed to find the exact value of certain parameters. However, this type of parametric analysis requires a lot of time and it would be most helpful to mechanize it using Uppaal, possibly using multiple processors to parallelize computations.

A lesson that we have learned is that it is extremely difficult to maintain correctness of the model in a setting where the object of modeling has such a high complexity. There was not a single document describing the design. In fact there was not a single person who was able to answer all our questions: the knowledge was spread over a large design team. For the engineers it is difficult to understand the intricacies of our Uppaal model. The syntax of Uppaal is not sufficiently expressive to describe the design in such a way that a small change in the design corresponds to a small change in the model. Due to these difficulties, the Octopus project has decided to develop a high level language for describing the designs, together with a translation to Uppaal: on one hand this will make it much easier to communicate with the engineers, and on the other hand it will reduce the chances of introducing errors in the Uppaal model.

References

1. Williams, S., Waterman, A., Patterson, D.: Roofline: an insightful visual performance model for multicore architectures. Commun. ACM 52(4), 65–76 (2009)
2. Behrmann, G., David, A., Larsen, K.G., Håkansson, J., Pettersson, P., Yi, W., Hendriks, M.: Uppaal 4.0. In: Third International Conference on the Quantitative Evaluation of SysTems (QEST 2006), Riverside, CA, USA, September 11-14, pp. 125–126. IEEE Computer Society, Los Alamitos (2006)
3. Larsen, K.G., Pettersson, P., Yi, W.: Uppaal in a Nutshell. Int. Journal on Software Tools for Technology Transfer 1(1-2), 134–152 (1997)

4. Behrmann, G., David, A., Larsen, K.G.: A tutorial on UPPAAL. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 200–236. Springer, Heidelberg (2004)
5. Homepage Octopus project, http://www.esi.nl/short/octopus
6. Igna, G., Kannan, V., Yang, Y., Basten, T., Geilen, M., Vaandrager, F., Voorhoeve, M., Smet, S., Somers, L.: Formal modeling and scheduling of datapaths of digital document printers. In: Cassez, F., Jard, C. (eds.) FORMATS 2008. LNCS, vol. 5215, pp. 170–187. Springer, Heidelberg (2008)
7. Jensen, K., Michael, L., Wells, K.L., Jensen, K., Kristensen, L.M.: Coloured petri nets and cpn tools for modelling and validation of concurrent systems. International Journal on Software Tools for Technology Transfer (2007)
8. Jensen, K.: Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use. EATCS Series. Springer, Heidelberg (1992)
9. Ghamarian, A.H., Geilen, M.C.W., Stuijk, S., Basten, T., Moonen, A.J.M., Bekooij, M.J.G., Theelen, B.D., Mousavi, M.R.: Throughput analysis of synchronous data flow graphs. In: Proc. ACSD 2006, pp. 25–34. IEEE, Los Alamitos (2006)
10. Stuijk, E., Geilen, M., Basten, T.: Sdf 3: Sdf for free. In: Proceedings of Application of Concurrency to System Design, ACSD 2006, pp. 276–278. IEEE, Los Alamitos (2006)
11. Lampka, K., Perathoner, S., Thiele, L.: Analytic real-time analysis and timed automata: a hybrid method for analyzing embedded real-time systems. In: Chakraborty, S., Halbwachs, N. (eds.) Proceedings of the 9th ACM & IEEE International conference on Embedded software, EMSOFT 2009, Grenoble, France, October 12-16, pp. 107–116. ACM, New York (2009)
12. Chakraborty, S., Kunzli, S., Thiele, L.: A general framework for analysing system properties in platform-based embedded system designs. In: DATE 2003: Proceedings of the conference on Design, Automation and Test in Europe, Washington, DC, USA, p. 10190. IEEE Computer Society, Los Alamitos (2003)
13. Thiele, L., Bacivarov, I., Haid, W., Huang, K.: Mapping applications to tiled multiprocessor embedded systems. In: Proceedings of the Seventh International Conference on Application of Concurrency to System Design, Washington, DC, USA, pp. 29–40. IEEE Computer Society Press, Los Alamitos (2007)
14. Hamann, A., Jersak, M., Richter, K., Ernst, R.: Design space exploration and system optimization with symta/s– symbolic timing analysis for systems. In: Proceedings of the 25th IEEE Real-Time Systems Symposium (RTSS 2004), Lisbon, Portugal, December 5-8, pp. 469–478. IEEE Computer Society, Los Alamitos (2004)
15. Harbour, M.G., García, J.G., Gutiérrez, J.P., Moyano, J.D.: Mast: Modeling and analysis suite for real time applications. In: 13th Euromicro Conference on Real-Time Systems (ECRTS 2001), Delft, The Netherlands, June 13-15, pp. 125–134. IEEE Computer Society Press, Los Alamitos (2001)

Resource Analysis of Automotive/Infotainment Systems Based on Domain-Specific Models – A Real-World Example

Klaus Birken[1], Daniel Hünig[1], Thomas Rustemeyer[1], and Ralph Wittmann[2]

[1] Harman Becker Automotive Systems, Raiffeisenstr. 34,
70794 Filderstadt, Germany
[2] Harman Becker Automotive Systems, Becker-Göring-Str. 16,
76307 Karlsbad, Germany
{klaus.birken,daniel.huenig}@harman.com,
{thomas.rustemeyer,ralph.wittmann}@harman.com

Abstract. High-end infotainment units for the Automotive domain have to provide a huge set of features, implemented in complex software on distributed embedded system hardware. In order to analyse the computational resources necessary for such systems, abstract models can be created which are used as a starting point for simulation and static analysis methods. In this paper, a domain-specific modeling method and derived analysis methods will be presented. A real-world scenario will be shown which could also serve as a challenging example for future tools and formal methods.

Keywords: Domain-specific modeling, timing simulation, resource analysis, Automotive, Infotainment.

1 Introduction

In the Automotive/Infotainment realm, there is an increasing pressure on improving quality while reducing development and per-unit cost. For system designers and software architects, this implies that the focus has to move to early design phases during the development process. Additionally, performance commitments are required in RFQ[1] phase, based only on hardware and software specs without the possibility of feasibility verification. Usually, target hardware and software implementations will not be available at this early phase. Thus, abstract models of the planned system have to be used to reason about requirements and design decisions.

For smaller Automotive-ECUs[2], it has already been shown that modeling, analysis and simulation methods can be applied successfully in industry (e.g., [1]). For infotainment devices, this is not as well-established yet. In this paper, we present a modeling methodology based on domain-specific languages and a non-trivial real-world example model (i.e., a premium-headunit startup scenario). We further show

[1] *Request for quote* phase, where manufacturer request a valid offer from their suppliers.
[2] *Electronic Control Unit.*

T. Margaria and B. Steffen (Eds.): ISoLA 2010, Part II, LNCS 6416, pp. 424–433, 2010.

how the model can be utilized in various ways to predict resource usage and gain design confidence.

2 Domain-Specific Modeling

It is an early and most important experience when starting to model real-world systems that the expressiveness of the model is defined by the features of the available language. On the other hand, the right formulation of the problem is already half-way to the solution. The model bridges the gap between a given set of requirements and the properties and APIs of the target platform which is being used.

Automotive/infotainment domain experts like system engineers or software architects should be able to create and extend their models easily and without having to learn tool details or wade through syntactic overhead. Furthermore, the models should be mergeable, modularized to leverage reuse of model parts and support fine-grained version history. Thus, we use domain-specific, textual modeling instead of general-purpose graphical modeling approaches like SysML and UML. This allows a specific and streamlined, but at the same time complete specification of the models.

As technological basis we are using an Eclipse environment, which hosts the EMF[3] framework with Xtext on top [2]. During the past years, the Xtext toolchain has gained a level of maturity which is well applicable for industrial applications; Eclipse and EMF are established in industry already. From an EBNF-like definition of a domain-specific language (DSL), Xtext is able to generate EMF-based metamodels (Ecore models). It also provides a powerful framework for generator development and a convenient text editor with many useful features, among them syntax highlighting, an outlook view and code completion – all adapted to the domain-specific language under development. Among other benefits, the toolchain is available as open-source software and offers a friendly community with optional commercial support by companies being active as committers.

In order to develop the DSL and a couple of generators for various target formats, we applied an incremental-iterative approach, where the domain experts and the modeling experts collaborated closely. All sources, i.e., the grammar defining the DSL, tool extensions, generators and also all resulting models are managed in a state-of-the-art source code control system. The resulting DSL and a non-trivial example model will be described in the following sections.

3 The Hbsim DSL for Infotainment System Models

State-of-the-art automotive infotainment systems exhibit complex behaviour, which is due to a huge set of features (e.g., about 30,000 requirements for a current high system), a multifaceted system environment (automotive busses, multiseat / multimodal user interface) and a heterogeneous, distributed hardware design. This complexity can only be handled in a simulation model by abstracting away details along several dimensions:

[3] Eclipse Modeling Framework (see http://www.eclipse.org/modeling/emf).

- Only system-relevant requirements will be taken into account.
- Hardware entities (e.g., mass storage devices, CPUs) and basic software layers (e.g., operating systems, drivers) will be represented only by their primary parameters.
- On the software side, we model abstract functional blocks instead of concrete software components and processes.

Table 1. Overview of language elements (i.e., vocabulary) of the *hbsim* DSL

hbsim element	purpose	examples
`cpu`	processors and controllers	CPU1, DSP, …
`resource`	bandwidth-bounded resource	SD-Card, harddisk, …
`pool`	allocate&free resources	RAM, DMA channels, …
`fb`	functional block, incl. dynamic behaviour and dependencies	tuner, navigation, media, browser, GUI, …
`partitioning`	mapping of functional blocks to processors	browser runs on CPU1
`usecase`	set of triggers for functional blocks	play mp3, start guidance
`scenario`	collection of concurrent use cases	startup lastmode tuner

The resulting core DSL we defined for modeling the infotainment systems is named *hbsim* [3]. The main language elements are listed in Table 1. It is focused on specification of available hardware, the functional blocks running on that hardware and the use cases and scenarios which can be executed. There is a strong focus on dynamic aspects and system behaviour, which eventually allows to execute these models either statically (with analysis tools) or operational (with simulation tools).

In order to illustrate our approach, two examples will be shown subsequently. Fig. 1 shows a screenshot of a modeled resource (here: NAND flash storage). Each resource interface models a special way to access the resource (e.g., raw device access, file system, DMA). Each resource interface is defined by its access bandwidth, the induced CPU load (e.g., by the device driver) and the context switching time (short: cst), which defines a penalty for simultaneous access from multiple clients. In general, sufficient accuracy can be gained by this level of abstraction. It is not necessary to model resources with more details, e.g., latencies of mechanical parts or low-level caches.

The second example in Fig. 2 demonstrates how the Multimedia application of an infotainment system is modeled using the functional block concept. The behaviour of each functional block is specified as a sequence of steps. Each step may define CPU load needed to accomplish this step, resource accesses (e.g., reading of data from file systems) and preconditions for this step. In actual *hbsim* models, we extended this concept by supporting multiple behaviours for each functional block. These behaviours may contain control structures (e.g., loops) and trigger other functional blocks after their execution. Multiple behaviours per functional block will also be used to express concurrent execution, e.g., processes, threads or tasks.

```
resource NAND_Flash {
    short flash;
    unit "MB";
    blocksize 16.0; // in kByte

    interface standard {
        bandwidth 7.1;
        inducedCPU 11;
        cst 30;
    };
}
```

Fig. 1. The resource NAND_Flash as modeled using the hbsim DSL (editor screenshot). Each resource may offer one or more interfaces, which can be used to read data from that resource.

```
fb Multimedia {
    "This function block is an abstraction for all multimedia stuff,
    including codecs and access to a filesystem for getting media files.";

    on load() {
        unloaded {
        }
        drivers_available {
            use 35;
            read exec_image: 0.3 via NAND_Flash.standard;
        }
        application_loaded {
            precondition OnOff::application_load2;
            use 365;
            read exec_image: 10.0 via NAND_Flash.standard;
        }
        init_done {
            use 365;
            read metaDB: 5.0 via HDD.direct_io;
        }
        mp3_playing {
            use 5;
            read mp3_file: 0.5 via HDD.standard;
        }
    }
}
```

Fig. 2. Example model (editor screenshot with syntax highlighting): functional block *Multimedia* with startup behaviour. In several steps of execution, read operations from the resource NAND_Flash (cf. Fig. 1) have been defined.

An additional, orthogonal modeling aspect is related to variability requirements. In the Automotive domain, there is a lot of variability concepts which should be 1st class citizens in our modeling language, among them OEM[4] variants on top of a product line, market variants, hardware samples, customization variants, and many more. The variants can be expressed easily and concise by using feature diagrams [4]. We are

[4] *Original Equipment Manufacturer*, i.e., the car manufacturing companies.

using an approach combining feature models and domain models similar to [5]. Variant points can be inserted at proper locations into the *hbsim* model. By defining configurations, feature sets can be selected which define slices of the actual model. Each slice represents one instance of the model configuration space and can be input for subsequent generator steps.

4 Example: Infotainment Head-Unit Startup

A real-world example which is relevant and important in Automotive/Infotainment systems is the startup process. There are multiple different scenarios depending on the wake-up reason and on persistent lastmode conditions. Examples: The infotainment system may be woken by a trigger on CAN-bus (e.g., door opened), by the driver pressing the On-key while the car is parked, or by a diagnostic session during production or in the garage. The system has to fulfil a plethora of timing requirements, ranging from safety-relevant, legal aspects of different markets, system-architecture requirements on car-level, production issues and usability expectations from the end-customer (represented by human factors and marketing experts on the OEM side).

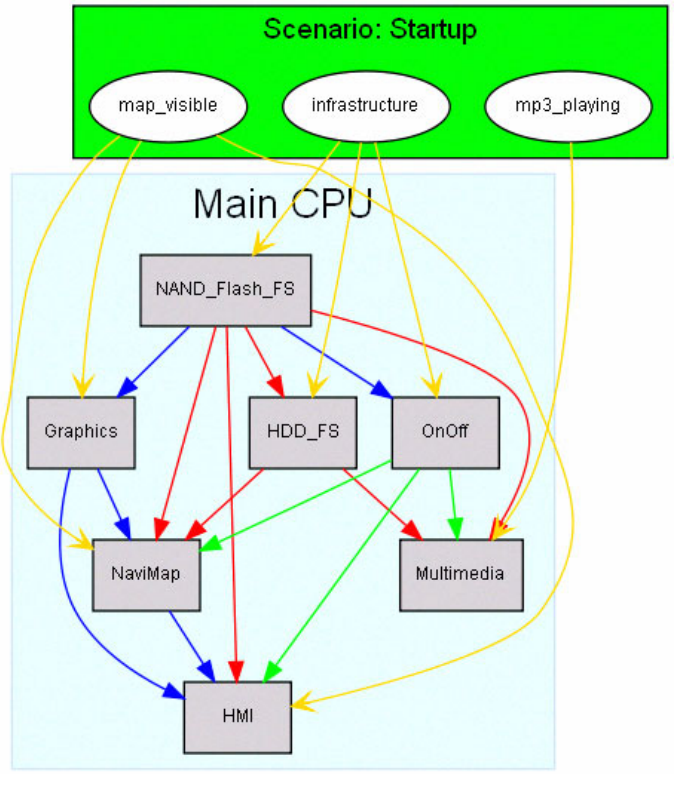


Fig. 3. Example model: System startup of a simplified Automotive/Infotainment system. Grey boxes represent functional blocks, arrows indicate dependencies between those blocks. The diagram has been automatically generated from the *hbsim* model (backend: graphviz).

The resulting model describes the startup behaviour of all relevant functional blocks, together with their resource needs and technical dependencies. A typical model defines 50-60 functional blocks and various processors and resources (e.g., flash memory, RAM, harddisk). It additionally incorporates aspects like multi-core/multi-cpu, inter-processor communication and partition-based scheduling algorithms.

Fig. 3 shows a simplified example of this kind of model. In this example, all functional blocks are deployed to a hardware processor named *Main CPU*. The arrows in the diagram indicate different kinds of inter-block dependencies, e.g., triggers (yellow, hollow arrowheads), technical preconditions (blue) or resource dependencies (red). With *hbsim*, complex simulation scenarios can be created by combining existing usecases. In the top of this diagram, it is shown how three usecases (map visible, mp3 playing, infrastructure) are combined to the scenario *Startup*. This represents a system startup where the navigation map is visible and mp3 is audible.

Analysing the dependency structure revealed in the automatically generated graph of Fig. 3, one can see that the NAND flash filesystem is a common prerequisite for all other functional blocks. We can also see that the HMI (i.e., the graphical user interface) depends on nearly all other functional blocks except the Multimedia application, which is needed for audio output only. However, there might be scenarios where the HMI is also dependent on the Multimedia application (e.g., for displaying a *current track* screen).

Thus, with this kind of models defined using the *hbsim* DSL, the structure and dynamic behaviour of infotainment systems can be defined on the appropriate level of abstraction. They collect the knowledge of various system and software experts with multiple years of domain experience each. The models represent an important part of the company's intellectual property. The hbsim DSL defines a powerful combination of executable architecture model and system design properties.

5 Model-Based Analysis and Simulation

The abstraction level of the infotainment system models (as described above) is well-suited for describing all major timing and resource usage aspects of this kind of product. By using the Xtext generator framework, the models can be transformed into a variety of target formats:

- graphical representation (based on automatic layout, e.g., graphviz [6])
- static analysis (e.g., critical-path, theoretical limits for timing requirements)
- guidance for automated trace analysis (gaining input for the model details, e.g., CPU loads of single steps)
- input for simulation tools (e.g., discrete-event or task-level simulation)

Our current main focus is on executing the model on a simulator in order to understand the dynamic behaviour of the system. We apply different simulators depending on the accuracy we want to achieve and the simulation performance we need. The interpretation of the simulation results is usually done by applying graphical tools (e.g., chronSIM by Inchron [7]) and/or extracting information from simulation traces automatically. This approach, together with other backends like visualization generation and static analysis, provides a new level of understanding for the domain

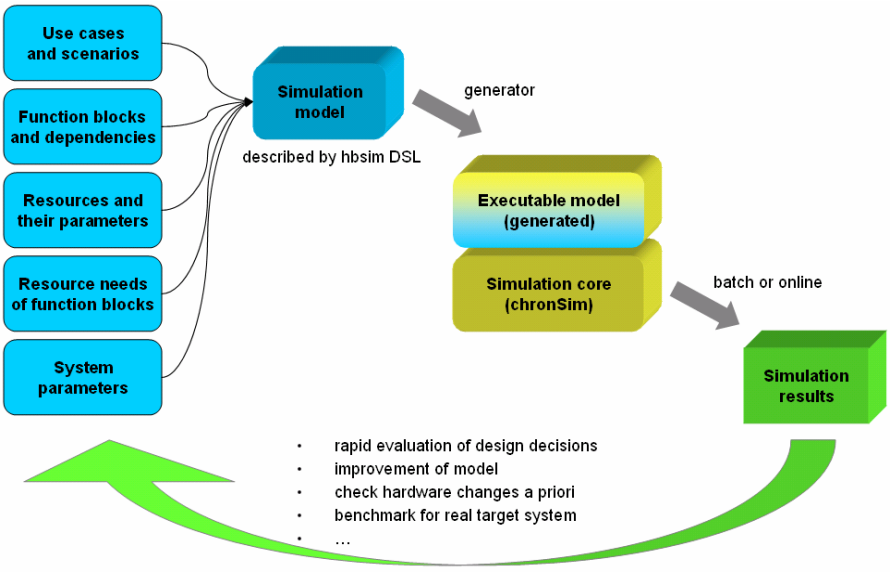


Fig. 4. Typical model/simulate/analyse cycle as applied when working with the modeling/simulation toolchain presented here

experts, architects and developers and helps to fulfil especially the non-functional customer's requirements.

After the creation of an initial model, the typical usage of the simulation toolchain is iterative (as with many other workflows in software development). Fig. 4 depicts the typical model/simulate/analyse cycle. It is important to note that the typical cycle is accomplished within 1-3 minutes (depending on simulation tool and model size), whereas implementing the real change would last at least hours, if the necessary changes are possible at all. E.g., even hardware changes can be modeled and simulated easily, but would be expensive to check in a real system.

Previous examples have shown that *hbsim* models contain detailed data describing cpu loads, resource bandwidths etc. (e.g., example model in Fig. 2). For big models, it is tedious and sometimes even not feasible to keep all those values up-to-date manually. A mechanism is needed which extracts the proper detail values from real system traces and injects them into the model.

Fig. 5 shows the principle of guided automatic analysis, which is a tool for providing real-world measurements as detail data for the simulation models. Traces from the real system are analysed automatically guided by concise information extracted from the simulation model. Basically, all resource loads detected in the real-world traces are mapped onto the functional blocks and their steps of execution. Thus, the usage of each resource is computed for each step. Moreover, correspondences between abstract markers in the model and concrete markers in the trace are evaluated by the analysis tool. This technique is indispensable for harvesting detail data for the model out of real-world traces. It allows keeping the model up-to-date with the real implementation. We will provide more information about this current subject of research in a follow-up publication.

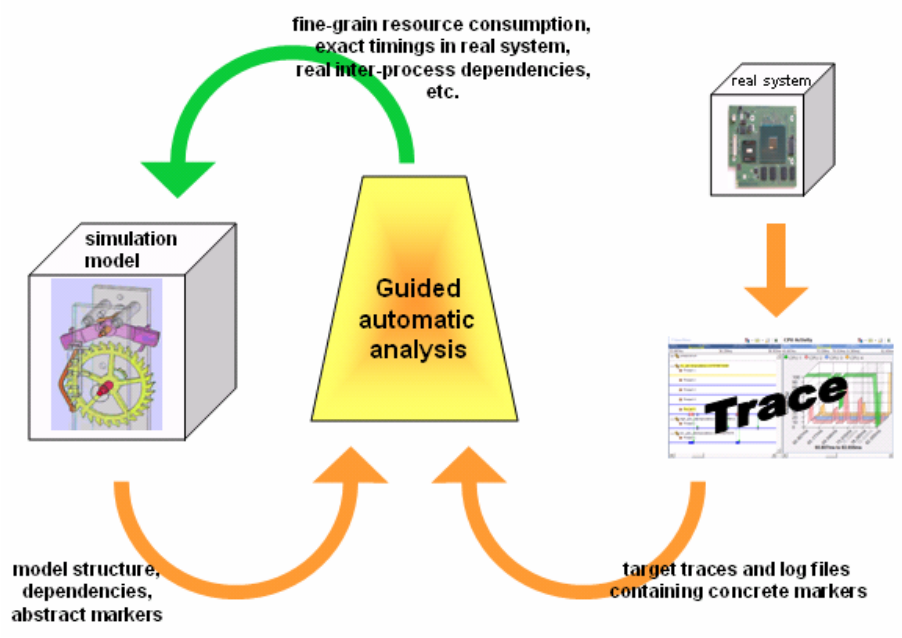


Fig. 5. Principle of guided automatic analysis

6 Example Results

Based on detailed models of infotainment unit startup scenarios, the methodology described above proved to be a valid and useful tool for aiding the system and software design. There is a variety of examples which could be described here, among them startup in diagnostic mode (needed during production and in the garage), reaction times on CAN messages (e.g., showing the rearview camera picture quickly in a park distance control usecase) and early audio usecases. In this section, we explain the latter scenario in more detail.

Typical early audio usecases require the infotainment unit to be able to support audio output very quickly after system wake-up. One relevant example is to provide audible warning messages triggered by the park-distance control ECU if the system is woken with activated reverse gear. The reaction time should not violate its limit even if the startup is loaded with additional compute-intense usecases, e.g., calculation of a navigation route. In order to ensure that the system always meets this requirement, we modeled this scenario as part of the overall infotainment startup model.

Fig. 6 shows a part of the simulation result, represented as a system state graph. The graph has been generated from the simulation trace output and layouted by the graphviz toolset. Each grey box in that graph indicates the execution of one step, which is a part of a functional block's behaviour. The label of each block gives the functional block, the behaviour currently active (both in the first line) and the name of that step (second line). For each such block, the time in milliseconds when the step starts its execution (*running*) is given as well as the time for the completion of that

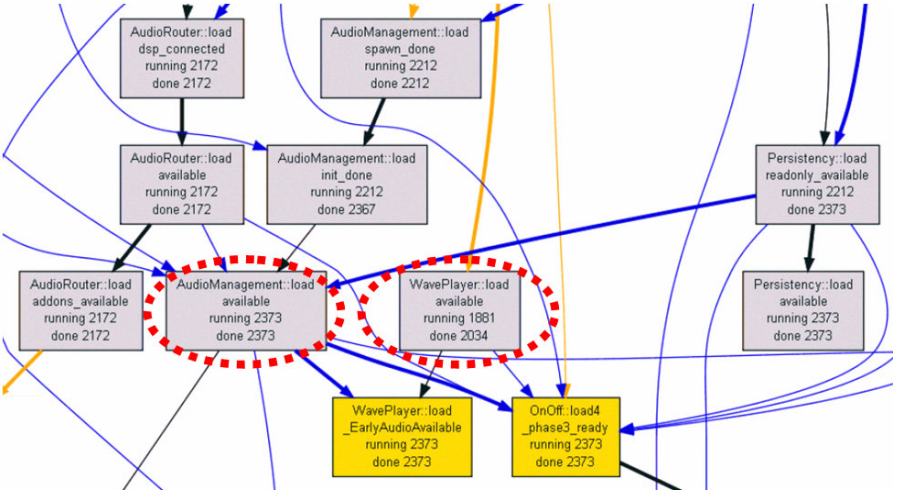


Fig. 6. Example simulation result, represented as system state change graph. In this example, the step EarlyAudioAvailable could be reached only after AudioManagement finished its *available* state. Times are in milliseconds.

step (*done*). Incoming arrows indicate all preliminaries for each step. The critical path is indicated by bold arrows.

In the example shown in Fig. 6, the target step for the early audio usecase is *EarlyAudioAvailable* (yellow box) of the functional block *WavePlayer*. This is completed at time 2,373 seconds after wakeup, which is too slow for state-of-the-art infotainment systems. In order to find out why that is too late we analyse the incoming arrows. The bold arrow leads us back along the critical path to the state *available* of the functional block *AudioManagement*. I.e., the WavePlayer component is available at 2034 milliseconds and has to wait for 2373-2034=*339* more milliseconds until AudioManagement can be accessed. If we further follow the critical path backwards, we are able to track down the causal chain based on the graphical simulation results. Eventually, the system engineer will find valid optimizations, apply them to the model and start the cycle again.

7 Conclusion and Next Steps

Applying domain-specific modeling of infotainment systems on system level is a valuable approach for analysis of timings and resource usage. The method can be applied early during the development phase, even as decision-making aid in RFQ phases. We also successfully introduced *hbsim*-based modeling and timing simulation as a continuous activity throughout the development process. This ensures that resource usage limits and boundary conditions are met. The development process is such directed towards the planned implementation as envisioned by system architects and designers. Thus, the models represent executable design documentation.

It has been shown that by this method significantly complex examples can be tackled which require a solid system design and domain know-how. The model serves as

an expert system where this kind of knowledge is accumulated. By applying various kinds of generators, the model can be transformed to useful target formats, among them graphical representations and input for simulators. This ensures that the accumulated knowledge is not a black hole, but is actively used and refined instead.

As the resulting models are defined concisely and completely based on a formally defined meta-model, we expect that formal methods and corresponding tools for verification, validation and timing analysis can be attached seamlessly. Therefore, one of our next steps is the application of state-of-the-art formal analysis methods complementing our current simulation approach.

References

1. Wirrer, G.: Scheduling Simulation for Engine Control Systems all along the Development Cycle. In: Fachkonferenz Echtzeitentwicklung 2009 (2009), http://echtzeitkongress.de
2. Eclipse/Xtext: Homepage of the Xtext project as part of the Eclipse-platform, http://www.eclipse.org/modeling/tmf/?project=xtext
3. Birken, K.: Forward-looking system development with domain-specific modelling and timing simulation. In: 2nd Real-time Development Congress (2010), http://echtzeitkongress.de
4. Czarnecki, K., Eisenecker, U.W.: Generative Programming. Methods, Tools, and Applications. Addison-Wesley Longman, Amsterdam (2000)
5. Völter, M.: Architecture As Language, Part 2: Expressing Variability (2008), http://www.voelter.de/data/articles/ArchitectureAsLanguage-Part2-PDF.pdf
6. graphviz Graph Visualization Software, http://www.graphviz.org
7. Kramer, T., Münzenberger, R.: Echtzeitverhalten simulieren und validieren, verstehen und absichern. In: Proceedings Design&Elektronik Entwicklerforum (2009), http://www.inchron.de/fileadmin/INCHRON/3-PDFs/DE/Embedded-System-Entwicklung_09_INCHRON.pdf

Source-Level Support for Timing Analysis*

Gergö Barany and Adrian Prantl**

Institute of Computer Languages
Vienna University of Technology
{gergo,adrian}@complang.tuwien.ac.at

Abstract. Timing analysis is an important prerequisite for the design of embedded real-time systems. In order to get tight and safe bounds for the timing of a program, precise information about its control flow and data flow is needed. While actual timings can only be derived from the machine code, many of the supporting analyses (deriving timing-relevant data such as points-to and loop bound information) operate much more effectively on the source code level. At this level, they can use high-level information that would otherwise be lost during the compilation to machine code.

During the optimization stage, compilers often apply transformations, such as loop unrolling, that modify the program's control flow. Such transformations can invalidate information derived from the source code. In our approach, we therefore apply such optimizations already at the source-code level and transform the analyzed information accordingly. This way, we can marry the goals of precise timing analysis and optimizing compilation.

In this article we present our implementation of this concept within the SATIrE source-to-source analysis and transformation framework. SATIrE forms the basis for the TuBound timing analyzer. In the ALL-TIMES EU FP7 project we extended SATIrE to exchange timing-relevant analysis data with other European timing analysis tools. In this context, we explain how timing-relevant information from the source code level can be communicated to a wide variety of tools that apply various forms of static and dynamic analysis on different levels.

1 Introduction

Many aspects of our lives are controlled by embedded computer systems with real-time constraints. Embedded real-time systems used in aerospace and

* This work was supported by the Austrian Science Fund (Fonds zur Förderung der wissenschaftlichen Forschung) under contracts P18925-N13, *Compiler Support for Timing Analysis (CoSTA)*, `http://costa.tuwien.ac.at/` and P21842, *Optimal Code Generation for Explicitly Parallel Processors*, `http://www.complang.tuwien.ac.at/epicopt/`, and the Commission of the European Union within the 7th EU R&D Framework Programme under contract 215068, *Integrating European Timing Analysis Technology (ALL-TIMES)*, `http://www.mrtc.mdh.se/projects/all-times/`

** The author is now at Lawrence Livermore National Laboratory, P. O. Box 808, 94551 Livermore, CA.

T. Margaria and B. Steffen (Eds.): ISoLA 2010, Part II, LNCS 6416, pp. 434–448, 2010.

automotive applications are especially delicate: Since errors in such computer systems may endanger lives, these systems are deemed safety-critical. Adherence to the specification, both functionally and in non-functional aspects such as timing, is therefore of the utmost importance. Timing analysis aims to predict worst-case timing behavior in order to give feedback to developers and provide information to validation processes.

There are many forms of timing analysis, each having a number of specific advantages and disadvantages. Static analysis predicts timing behavior based on the program's code. This form of analysis is able to cover all possible eventualities; however, static analyses must typically make some simplifying assumptions that coarsen the analysis information, suggesting possible program behaviors that cannot be realized in actual executions. This may lead to overestimations of the worst-case execution time (WCET). Various kinds of static analysis techniques can reduce overestimations, but often at considerable costs in analysis time and memory consumption and, thus, scalability. Wilhelm et al. [WEE+08, pp. 39–41] summarize studies which found that practical static analysis methods overestimate cache miss penalties by 15–30 %, and up to 50 % on more modern and complex architectures.

In contrast to static approaches, dynamic analysis observes actual executions of the program and examines dynamically collected information. Such information can include things such as traces of paths taken through the program, relationships between program paths (such as mutual exclusion), numbers of iterations of loops in the program, and execution times of basic blocks or larger pieces of code. These data points describe actual executions and are therefore very precise. However, the major difficulty of dynamic analysis is that one must ensure that measured data are also safe: The application must be run under conditions and with input data that are capable of eliciting worst-case behavior. Finding such input data can be a very difficult task.

In practice, several forms and levels of analysis can be combined: For example, static analysis may be used in the search for worst-case inputs for subsequent dynamic analysis. Dynamically measured (safe) timings of parts of a program may be combined with statically derived information on worst-case execution frequencies.

This paper provides a high-level overview of past work at Vienna University of Technology's Compilers and Languages group. We focus on the combination of analysis *levels*: The derivation of symbolic auxiliary information at the source-code level and its relevance to timing analysis (Section 3); and communication of the results to tools that perform lower-level timing analysis, usually involving the application binary (Section 4). As far as information about the program's control flow is concerned, source-level information may be invalidated by optimizing compilers during translation. For this reason, we also discuss how to perform some optimizations at the source-code level and update the flow information accordingly (Section 5).

2 Static Timing Analysis Techniques

The standard approach to static timing analysis consists of three conceptual stages (that need not be implemented separately in actual tools): high-level analysis, low-level analysis, and calculation.

High-level analysis derives information about the control- and data flow of the program. In particular, this involves finding static bounds on the number of iterations of each loop in the program; if it is not known after how many iterations some loop terminates, the WCET must be conservatively estimated to be infinite. Other kinds of flow information may include conflicting paths: For instance, whenever some path π_1 is taken at some branch, it may be impossible to take another path π_2 at a subsequent branch.

Low-level analysis is concerned with finding actual timing information for parts of the program, typically basic blocks. As a very rough approximation, this may consist simply of adding up individual worst-case instruction timings. However, this approach gives much too high numbers on modern processor architectures that feature pipelines and caching mechanisms. Sophisticated low-level analyzers therefore track models of the pipeline and cache states, which allows them to compute a safe approximation of the performance gain due to overlapped execution of instructions as well as cache hits.

Finally, the calculation phase integrates flow information with low-level timings. The common approach is the implicit path enumeration technique (IPET): The entire timing analysis problem is formulated as an integer linear program. Each basic block's execution frequency is represented by a variable in the ILP formulation; flow constraints such as loop bounds and mutual exclusions are expressed as linear inequalities. Basic block timings are encoded in the problem's objective function, which is then maximized using an off-the-shelf ILP solver. The maximal solution gives the worst-case execution time, and the values of the ILP variables give some information about the worst-case path.

Both high-level and low-level analysis may profit from various kinds of auxiliary information: For instance, precise information about pointer targets may both eliminate false paths due to indirect function calls (high-level) and improve the prediction of cache effects (low-level). In the following section, we describe our source-level analyses supporting timing analysis.

3 Source-Level Analyses for Timing Analysis

We use the SATIrE (Static Analysis Tool Integration Engine)[1] framework to perform high-level parts of timing analysis. SATIrE is a source-level analysis and transformation framework that has been under development at Vienna University of Technology since 2004.

3.1 The SATIrE Framework

SATIrE allows users to build tools that analyze or transform C (and, to some extent, C++) programs. Its internal program representation is based on the object-oriented abstract syntax tree (AST) provided by the ROSE[2] source-to-source transformation system. The AST may carry arbitrary annotations as

[1] `http://www.complang.tuwien.ac.at/satire/`
[2] `http://www.rosecompiler.org`

extracted from annotations in the program or computed by program analyzers. The AST is either built by ROSE, which is based on the C and C++ frontend by Edison Design Group[3], or by a modified version of the Clang frontend[4].

Given the AST, there are several ways of analyzing and manipulating the program it represents. ROSE includes a number of analyses and transformations, including a loop optimizer that can perform common operations such as loop unrolling. SATIrE includes a component that builds an interprocedural control flow graph (ICFG) suitable for data-flow analysis. Bindings to the Program Analyzer Generator (PAG)[5] allow the generation of such data-flow analyzers from simple functional specifications. Another component[6] can export and import ASTs represented as Prolog terms. Prolog's symbolic processing capabilities provide excellent tools for the manipulation of tree-shaped data, which makes it a good match for our purposes.

ASTs that have been transformed or annotated with analysis results (in the form of comments or `#pragma` statements) can be unparsed to C source code. This code can then be passed to any other source-based tool or regular C compiler for further processing.

3.2 High-Level Analyses Supporting Timing Analysis

SATIrE allows us to implement a number of program analyses supporting the high-level component of timing analysis. On the source-code level, we can derive information regarding pointer relationships, including function pointer targets; information about the possible ranges of the values of integer variables; and, based on these, tight bounds for the number of iterations of many loops and loop nests. This section describes our implementations of these analyses.

Points-To Analysis. SATIrE includes a flow-insensitive unification-based analysis based on Steensgaard's well-known analysis [Ste96]. The basic analysis performs a single pass over the program, assigning an 'abstract memory location' to each program variable, function, or dynamic memory allocation site. Locations representing structures have special edges to their members' locations. All elements of an array are collapsed into a single summary location. Several distinct objects may be assigned the same location, as described below.

The effects of pointer assignments are modeled using points-to edges between locations: There is a points-to edge from location ℓ_1 to location ℓ_2 if at some point during some execution of the program, one of the objects represented by ℓ_1 may hold a pointer value that points to one of the objects represented by ℓ_2.

Each location is constrained to have at most one outgoing points-to edge; if some location might point to two or more different locations, those locations are merged into a new combined location. If merged locations pointed to different

[3] `http://www.edg.com`
[4] `http://clang.llvm.org`
[5] `http://www.absint.de/pag/`
[6] `http://www.complang.tuwien.ac.at/adrian/termite/`

locations before, those target locations must also be merged recursively. This merging ensures that the analysis can be implemented in almost-linear time using a fast Union/Find data structure [Tar75]; however, it is also a source of imprecision as it may introduce spurious points-to relations that cannot be realized in any actual run of the program.

In its basic form, the algorithm suffers from imprecision because it is context-insensitive: If a function that receives a pointer argument is called at several different sites, the analysis will merge all the objects that may be pointed to at *any* call site. We made the analysis context-sensitive by analyzing each function several times (once for each context), and linking the analysis data according to the calling structure between contexts. This approach of cloning contexts is similar to Lattner et al.'s context-sensitive points-to analysis [LLA07].

The points-to analysis supports other source-level analyses in SATIrE. However, it is also directly relevant to timing: Precise points-to information can help other tools reduce the number of candidates at indirect call sites; allow better value analysis by reducing the number of candidates for indirect data accesses; and allow better modeling of cache effects.

Value Interval Analysis. SATIrE uses a flow-sensitive interval analysis (or 'value range analysis') to associate each integer variable with a value interval for each location in its scope. If at some point a variable is associated with an interval $[a, b]$, this means that at that point, the variable's value is definitely somewhere between a and b. The interval analysis is implemented as an abstract interpretation [CC77]. The declarative analysis specification is translated to an executable program by PAG.

In certain cases, the analysis can make use of assert statements in the program that were inserted by programmers with domain knowledge, or by some other program analysis/transformation. The information in a statement like `assert(x >= 0 && x <= 10);` can be used by the interval analysis to infer that at the program point following that statement, the value of variable `x` must be in the range $[0, 10]$, regardless of what was known about its value before. SATIrE includes a component that annotates a program with such assertions capturing the results of interval analysis; thus, such assertions are well suited for storing analysis information for later use without full-scale recomputation. These assertions are also useful in testing the analysis itself, and in verifying annotations provided by users or by other tools [PKK+09].

The analysis is integrated with SATIrE's points-to analysis. Integer assignments or reads through pointer expressions can therefore be resolved to sets of possibly referenced variables. This allows us to avoid some conservative assumptions that would be necessary without points-to information: Indirect reads yield the union of all involved variables' intervals, indirect writes only affect the analysis data associated with the possible pointer targets. Similarly, arrays are modeled as sets of aliased variables.

The interval analysis is inter-procedural, i. e., intervals associated with argument expressions of function calls are propagated into the corresponding functions. For programs that use indirect calls through function pointers, the points-to analysis

is consulted to propagate information to and from all possible targets of the given call. Using facilities provided by PAG, the interval analysis can be used in a context-sensitive way with arbitrarily long call strings.

In the context of timing analysis, interval analysis is mostly of interest for the computation of loop bounds (see below). It some cases, it can also identify infeasible paths: Branches on the values of function parameters may be resolved statically (in context-sensitive ways) if the analysis can identify the value ranges of actual arguments.

Loop Bounds Analysis. To implement the TuBound timing analysis tool, SATIrE was extended with a component which computes bounds for loops based on iteration variables [PKST08]. It uses results of the interval analysis and structural information about the program to build equations or set of inequalities, which are solved to yield bounds on the number of loop iterations.

The loop analyzer looks for loops preceded by the initialization of an iteration variable, a loop condition consisting of an inequality involving the variable (or a set of such inequalities connected by 'logical or' operations), and exactly one increment or decrement of the variable inside the loop with a bounded (but not necessarily constant) step size. Assuming the loop variable is i, the initialization expression is *Init*, the test expression is $i < Max$, and the minimum step size *Step* is known to be positive, we can set up an equation like $n = (Max - Init)/Step$ to describe the number of loop iterations.

This expression can be evaluated using interval arithmetic to provide an upper bound. Before numeric evaluation, we also perform a symbolic simplification step that attempts to eliminate common subexpressions between *Max* and *Init*. This allows us to handle some loops that involve unknown quantities, such as the common idiom of iterating over an array using a pointer: `for (p = a; p < a + 10; p++)`. Here, our interval analysis cannot determine an interval for `a`; however, none is needed because after simplification, no occurrences of `a` are left in the loop bound expression.

The above analysis works well for single loops, but it can overestimate nested loops with a triangular or irregular iteration space. We analyze nested loops using more general flow constraints. This analysis works for counting loops as described above, but now we require a constant step size. For each (upwards-counting) loop, we set up a system of inequalities $\{i \geq Init, i \leq Max, (i - Init) \bmod Step = 0\}$.

This translation can be performed recursively for nested loops. The set of distinct integral solutions to the resulting system of inequalities describes the entire iteration space, i. e., the set of all tuples of values that the iteration variables of the loops can take. The size of this set gives the number of iterations of the innermost loop in the nest. The *clpfd* solver distributed with SWI-Prolog[7] allows efficient computation of the number of solutions without producing them.

Precise data on loop bounds is directly relevant to timing analysis as programs typically spend most of their time in loops, and an overestimation of loop trip counts directly translates into an overestimation of the WCET. Automatic

[7] `http://www.swi-prolog.org`

analysis, especially of complex irregular loop nests, is both less time-consuming and less error-prone than manual annotation.

4 Integration of Timing Analysis Tools

This section explains how we integrate the high-level analyses described in the previous section with other timing analysis tools. Such integration is needed because actual timing information cannot be derived at the source code level: An intervening compiler is needed to produce actual machine code. (Compilation to an abstract machine may suffice if a precise timing model of the abstract machine on a given physical machine is available [HBH+07].) Such compilers may be, but need not be, aware of the real-time nature of the software they are compiling.

Here we describe SATIrE's integration with four other tools with different approaches to the WCET analysis problem: Compiler integration; dynamic analysis; static analysis on the binary; flow analysis on a lower-level representation. All of these connections make heavy use of annotation capabilities provided by the respective tools.

The integration with CalcWCET$_{C167}$ was implemented as part of the Austrian CoSTA project, while the other three integrations were part of the ALL-TIMES EU FP7 project. We have working research prototypes for each tool integration.

4.1 Integrated Compilation and WCET Calculation

One compiler designed for integrated compilation and WCET calculation is CalcWCET$_{C167}$ [Kir01] targeting Infineon's C167 family of microcontrollers. This is a modified version of GCC which understands `wcetC`, an extension of C that provides a custom syntax to specify flow constraints and loop bounds in addition to the input program. During code generation, it computes execution times for each basic block it generates; the flow information and basic block timings are used to set up an IPET problem, which is solved using standard techniques. One drawback of this approach is that the compiler is prohibited from performing optimizations that alter the control flow of the program. Section 5 discusses how we can sidestep this problem by performing optimizations on the source-code level.

CalcWCET$_{C167}$'s approach to timing analysis relies on good source-level flow annotations. Without tool support, such annotations must be placed in the program by the programmer, which is a tedious and error-prone task. The TuBound tool implemented using SATIrE is able to leverage its loop bounds analysis to compute the necessary information for many loops. Its program transformation capabilities can then be used to insert the annotations in the source code.

4.2 Annotations for Measurement-Based Analysis

RapiTime by Rapita Systems Ltd.[8] is a dynamic analysis toolkit. It instruments target applications with measurement code and uses the measurement data to

[8] `http://www.rapitasystems.com`

profile performance, provide code coverage information, and perform WCET analysis. As RapiTime uses dynamic analysis to gather information at run-time, one cannot always be sure that all possible executions of certain parts of the code have been covered by its analysis. RapiTime therefore provides the possibility for users to annotate the program's source code with high-level knowledge about issues such as points-to relations or flow constraints. SATIrE can compute some of the relevant information, as detailed below.

In order to compute a worst-case timing for a function call, RapiTime must know all the possible functions that may be called at that site (in a certain context). Since embedded system programs often contain indirect calls through function pointers, this information is typically not immediately available. During the execution of the system, the code instrumented by RapiTime can record all *observed* functions called from a certain site, but as noted above, it may not always be sure that these were all the *possible* call targets for that call site. Without this information, it must make a conservative approximation or reject the program.

SATIrE's points-to analysis statically computes conservative approximations of the sets of targets of each indirect function call. This automatic analysis is much faster and more reliable than manual annotations; this is particularly true for context-sensitive annotations. Thus SATIrE's information can tell RapiTime whether it has observed all possible call targets during its tests, or which other possible targets it must take into account. Similarly, RapiTime may observe certain numbers of iterations for loops in the application. SATIrE's static analysis of loop bounds may confirm that the observed iterations are indeed the worst case, or provide information for appropriate computation of a guaranteed time bound.

Our source-level static analysis thus helps in ensuring the safety of the dynamic analysis, or in proving that a given dynamic analysis result is indeed safe.

4.3 Annotations for Binary-Level Static Analysis

The aiT family of WCET analysis tools from AbsInt Angewandte Informatik GmbH[9] performs static analysis directly on the application binary. Using abstract interpretation, aiT derives possible value ranges of registers and memory locations; it also computes upper bounds on the WCET of basic blocks, taking cache and pipelining effects into account. The overall WCET is computed using the IPET approach.

While the analyses of aiT and SATIrE compute some similar information, they have different ways of computing that information. aiT's value analysis subsumes its pointer analysis, treating pointer values simply as integers. A value interval determined for a pointer thus implicitly denotes the set of all objects that could be addressed by that pointer. In contrast, SATIrE's points-to analysis is symbolic, identifying objects by abstract symbols, not by memory addresses. Where a pointer points to non-adjacent functions or global symbols in the program, aiT's analysis will determine that it may also point to any intervening function or object. In such cases, SATIrE's symbolic analysis can derive the possibly much more precise result

[9] `http://www.absint.com`

that the pointer may only reference one of a discrete set of functions or objects, but not any arbitrary memory address in between. This more precise pointer analysis may also add precision to SATIrE's interval analysis in some cases. We express results using aiT's existing annotation mechanism.

aiT's annotation mechanism includes a notion of 'user-defined registers', which are virtual machine registers whose values can be read or written by annotations. In particular, annotations can be made conditional on a user-defined register's value. This allows us to formulate context-sensitive annotations by encoding call string information in virtual registers. Such annotations may, in turn, tighten aiT's timing results computed for certain contexts.

4.4 Integration with Other High-Level Tools

SWEET is a research tool from Mälardalen University's WCET group[10]. Its flow analysis is based on abstract execution [GESL06] and can derive complex flow constraints relating execution frequencies for different points in the program. The integration of SWEET and SATIrE involves various issues. First, as SWEET works on programs represented in the ALF format [GEL+09], it needs translators to ALF in order to analyze source code. The connection between SATIrE and SWEET in the ALL-TIMES project therefore included a C-to-ALF compiler.[11] One advantage of having control over this translator is that it can output useful meta-information besides the ALF code.

The translator therefore outputs information mapping ALF code positions (identified by jump labels) to SATIrE's internal position identifiers as well as to source code locations. It also exports information on the call strings used by SATIrE. Using this information, SATIrE's context-sensitive analysis results regarding points-to information and value intervals can also be communicated to SWEET. In contrast to the other connections described above, SWEET and SATIrE have similar notions of program objects (ALF allows named, scoped variables like C does). Thus SATIrE's analysis information referring to program variables and pointer relations is directly useful to SWEET. The tight correspondences between program positions as well as variables allow SATIrE to exchange even flow-sensitive information with SWEET, which is not the case for the other connections. This tight integration can ease the implementation burden on SWEET's developers, who at the time of writing do not have a context-sensitive points-to analysis.

5 Source-Level Optimization and Timing Analysis

As mentioned in Section 4.1, some types of compiler optimizations alter the program's control flow and invalidate flow information annotated at the source-code level. One prominent example for this is loop unrolling: An unrolled loop will perform fewer iterations than expected from the source code, but will accordingly

[10] `http://www.mrtc.mdh.se/projects/wcet/`

[11] `http://www.complang.tuwien.ac.at/gergo/melmac/`

do more work in each iteration. For a loop unrolled by a factor of k, preserving the original annotation will result in an overestimation of the WCET by a similar factor of about k. For this reason, such optimizations must be disabled in the back-end compiler if source-level annotations are to be used. The CalcWCET$_{C167}$ compiler disables all loop optimizations that change the control flow.

However, such optimizations are desired because they can often result in considerable speedups. We have therefore combined source-level optimization with corresponding transformation of source-code annotations. This way, programs can take advantage of loop optimizations at the source level, while such optimizations are still disabled in the back-end. At the same time, annotations remain correct and tight. The result is usually a reduction in the calculated WCET.

5.1 Transformation of Flow Information

We consider flow information represented by a variable f_e for each edge e in the program's control-flow graph. Flow constraints are given as inequalities of linear expressions involving possibly scaled flow variables, which we write as $\langle n \cdot f_e \rangle$. We transform constraints by replacing expressions referring to transformed edges with other expressions [KPP10]. In general, there are two cases to consider:

- The flow f_e at edge e is split into multiple edges e'_i. We replace the corresponding scaled flow variable by appropriately scaled variables for the new edges:
$$\langle n \cdot f_e \rangle \longrightarrow \langle n_1 \cdot f_{e'_1} \rangle + \langle n_2 \cdot f_{e'_2} \rangle + \cdots$$
- The flow of several edges e_i is merged into one edge e'. For each e_i, we perform the following transformations for $\leq$ or $<$ constraints:
$$\langle n \cdot f_{e_i} \rangle \longrightarrow \begin{cases} \langle n_{\text{lhs}} \cdot f_{e'} \rangle & \text{on the left-hand-side of the constraint} \\ \langle n_{\text{rhs}} \cdot f_{e'} \rangle & \text{on the right-hand-side of the constraint} \end{cases}$$
and vice versa for $\geq$ or $>$ constraints.

In either case, the values of the newly introduced scalar factors depend on the details of the program transformation.

Transformation of flow annotations is guided by an optimization trace. The trace, a log of each modification of the control flow, is produced by the loop optimizer. We use it to access data about the original and transformed loops as well as the flow variables involved. Our implementation of optimization traces and the transformation rules can handle the correct transformation of annotations for loop unrolling, loop blocking, loop fusion, and loop interchange.

Example. Figure 1 illustrates the transformation of flow constraints for interchanged loops. Note that each occurrence of f_{l_1} is replaced by the flow variable f_{l_2}, scaled appropriately according to the analyzed lower or upper bounds of iterations of the new outer loop.

Original program	Loop-interchanged program′

`for (i = 0; i < 8; ++i) {`
`  // $l_1$`
`  for (j = 0; j < n; ++j) {`
`  // $l_2$`
`  if (even(i))`
`  // then`
`  ...`
`  else ...`
	`for (j = 0; j < n; ++j) {`
	`  // $l_1$`
	`  for (i = 0; i < 8; ++i) {`
	`  // $l_2$`
	`  if (even(i))`
	`  // then`
	`  ...`
	`  else ...`

Loop bounds	Loop bounds′
$\langle l_1, 8 \dots 8 \rangle$ $\langle l_2, 1 \dots 4 \rangle$	$\langle l_1, 1 \dots 4 \rangle$ $\langle l_2, 8 \dots 8 \rangle$
Constraints	Constraints′
$f_{l_1} \leq 8$ $f_{\text{then}} \leq f_{l_1} \cdot 2$ $f_{l_1} \leq f_{\text{then}} \cdot 2$	$f_{l_2}/4 \leq 8$ $f_{\text{then}} \leq f_{l_2}/1 \cdot 2$ $f_{l_2}/4 \leq f_{\text{then}} \cdot 2$

Fig. 1. Example: Transformation of annotations after loop interchange [Pra10]

5.2 Experimental Evaluation

We evaluated the impact of source-level loop optimization and annotation transformation on the analyzed WCETs of the programs from the WCET benchmark suite from Mälardalen University and the DSPstone benchmark suite. On average, our source-based optimizations succeed in reducing the analyzed WCET by about 13 % on the Mälardalen benchmarks and by about 21 % on DSPstone.

Figure 2 shows results for the standard WCET analysis benchmarks collected by Mälardalen University[12]. In Figure 3 results for the fixed-point version of the DSPstone benchmarks [ŽVSM94] are shown: Using the benchmarks that could be fully analyzed (and automatically annotated at the source code level) by an unassisted TuBound, the diagrams show how the WCET bound is affected by the source-level loop optimizations. Each value is normalized by the WCET bound of the program with just the low-level optimizations applied, marked as the 'original' analyzed WCET in the graphs. The WCET bound calculation was done using the CalcWCET$_{C167}$ back end of TuBound. The source-level loop optimizations use the upper and lower loop bound information found by TuBound. All low-level optimizations performed by the target compiler do not alter the control flow any more. The rightmost bar represents the geometric mean of the scaled execution speed of each of the benchmarks.

The benchmarks show that the potential for optimizations is significant. Unsurprisingly, the implemented optimizations (loop unrolling, fusion, interchange,

[12] `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`

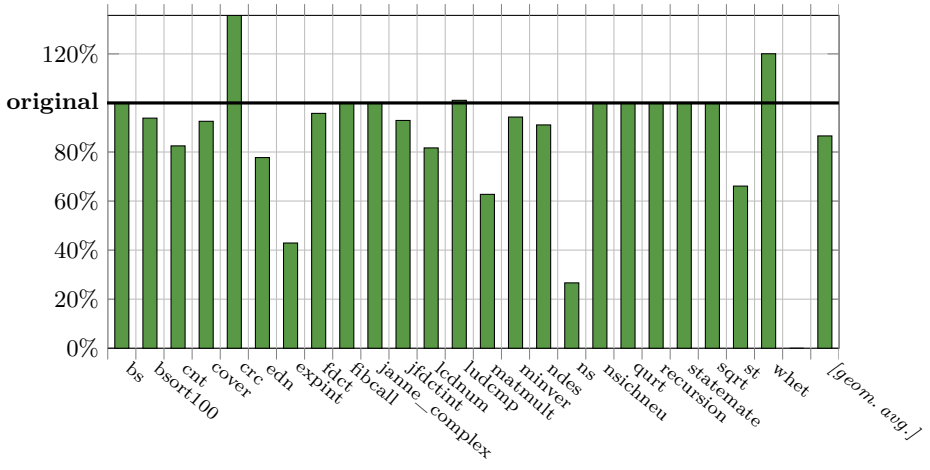


Fig. 2. Tighter analyzed WCETs due to high-level loop optimizations: Mälardalen benchmarks

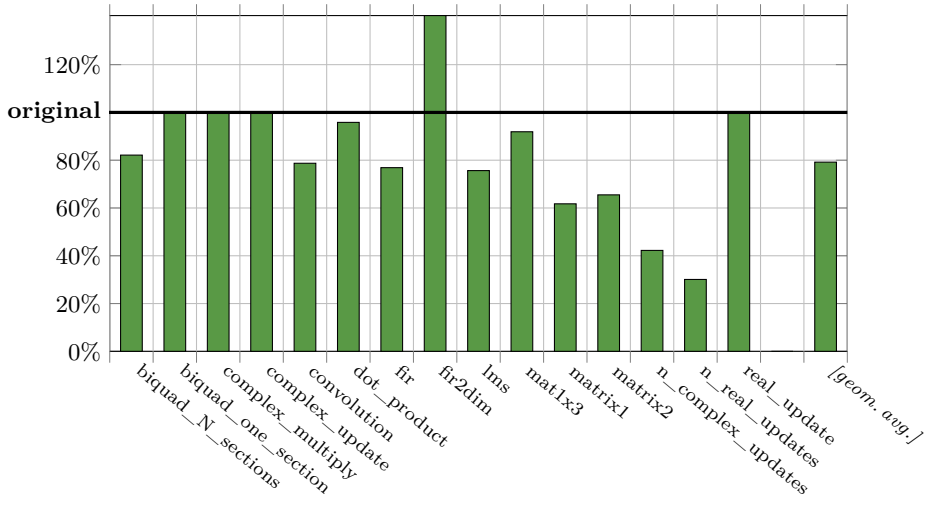


Fig. 3. Tighter analyzed WCETs due to high-level loop optimizations: DSPstone benchmarks

splitting), perform well on DSP kernels (e. g., `matrix2`) and show little to no impact on branch-intensive code (e. g., `nsichneu`) that lacks tight loops with a high trip-count.

Outliers like `crc` show that careful selection of the different optimization phases is very important. This process, however, can be supported by an automatic WCET analysis, which can be used to guide the optimizer by judging the improvement of a program transformation [LM09].

6 Related Work

Wilhelm et al. [WEE+08] present a thorough discussion of the worst-case execution time problem and various tools and methods to approach it. Gustafsson et al. [GLS+08] give a more detailed overview of the ALL-TIMES project and the tools involved. Schordan [Sch08] presents the SATIrE system in more detail and considers some challenges of annotating source code with analysis information.

There does not appear to be much previous work that deals specifically with integration of source code analysis and WCET calculation. This paper summarizes and unifies several threads of work in this area performed at Vienna University of Technology in between 2006 and 2010 within the ALL-TIMES and CoSTA projects. Prantl et al. discuss the TuBound tool in much more detail [PSK08, Pra10], while Barany [Bar09] gives more details on the integration of SATIrE in the ALL-TIMES project.

Schulte [Sch07] as well as Engblom et al. [EEA98] discuss the transformation of flow annotations along with control-flow altering program optimizations.

Herrmann et al. [HBH+07] use a (conceptual) virtual machine as an intermediate step for combining high-level and low-level analysis of programs written in the functional language Hume: For each target machine, timings of abstract machine instructions are derived using aiT. Source-level analysis and knowledge of compiler internals allows their framework to determine the set of abstract machine instructions that would be generated for each input program. Straightforward composition of this information with the low-level timings yields a WCET bound.

7 Conclusions

We have presented our approach to supporting timing analysis on the source code level using the SATIrE analysis framework. Using this framework, we built the TuBound tool for WCET analysis, combining source-level analysis, source-level optimization, and a back-end compiler performing WCET analysis. We have also implemented connections to several other timing analysis tools of various kinds, demonstrating that source-level analysis information can be useful for a wide range of timing-related analyzers that cover different analysis approaches.

Acknowledgements. The authors would like to thank Viktor Pavlu and Alexander Jordan for many helpful comments on earlier versions of this article. We are grateful to the anonymous reviewers for their comments that helped us improve the paper's focus and presentation.

References

[Bar09] Barany, G.: SATIrE within ALL-TIMES: Improving timing technology with source code analysis. In: Kolloquium Programmiersprachen und Grundlagen der Programmierung (KPS 2009), ch. 15, Maria Taferl, Austria, p. 230 (October 2009)

[CC77] Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL 1977: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, pp. 238–252. ACM, New York (1977)

[EEA98] Engblom, J., Ermedahl, A., Altenbernd, P.: Facilitating worst-case execution time analysis for optimized code. In: Proc. 10th Euromicro Real-Time Workshop, Berlin, Germany (June 1998)

[GEL+09] Gustafsson, J., Ermedahl, A., Lisper, B., Sandberg, C., Källberg, L.: ALF – a language for WCET flow analysis. In: Proceedings of the 9th International Workshop on Worst-Case Execution Time Analysis (WCET 2009) (June 2009)

[GESL06] Gustafsson, J., Ermedahl, A., Sandberg, C., Lisper, B.: Automatic derivation of loop bounds and infeasible paths for wcet analysis using abstract execution. In: The 27th IEEE Real-Time Systems Symposium, RTSS 2006 (December 2006)

[GLS+08] Gustafsson, J., Lisper, B., Schordan, M., Ferdinand, C., Jersak, M., Bernat, G.: ALL-TIMES - a European project on integrating timing technology. In: Proc. Third International Symposium on Leveraging Applications of Formal Methods (ISOLA 2008), October 2008, pp. 445–459. Springer, Heidelberg (2008)

[HBH+07] Herrmann, C.A., Bonenfant, A., Hammond, K., Jost, S., Loidl, H.-W., Pointon, R.: Automatic amortised worst-case execution time analysis. In: Proceedings of the 7th International Workshop on Worst-Case Execution Time (WCET) Analysis (2007)

[Kir01] Kirner, R.: User's Manual – WCET-Analysis Framework based on WCETC. Vienna University of Technology, Vienna, Austria, 0.0.3 edition (July 2001), `http://www.vmars.tuwien.ac.at/~raimund/calc_wcet/`

[KPP10] Kirner, R., Puschner, P., Prantl, A.: Transforming flow information during code optimization for timing analysis. Real-Time Systems 45(1-2), 72–105 (2010)

[LLA07] Lattner, C., Lenharth, A., Adve, V.: Making context-sensitive points-to analysis with heap cloning practical for the real world. In: PLDI 2007: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation, pp. 278–289. ACM, New York (2007)

[LM09] Lokuciejewski, P., Marwedel, P.: Combining Worst-Case Timing Models, Loop Unrolling, and Static Loop Analysis for WCET Minimization. In: The 21st Euromicro Conference on Real-Time Systems (ECRTS), Dublin, Ireland, pp. 35–44. IEEE Computer Society, Los Alamitos (July 2009)

[PKK+09] Prantl, A., Knoop, J., Kirner, R., Kadlec, A., Schordan, M.: From trusted annotations to verified knowledge. In: Proceedings of the 9th International Workshop on Worst-Case Execution Time Analysis (WCET 2009), Dublin, Ireland, Österreichische Computer Gesellschaft, pp. 39–49 (June 2009) ISBN: 978-3-85403-252-6

[PKST08] Prantl, A., Knoop, J., Schordan, M., Triska, M.: Constraint solving for high-level WCET analysis. In: The 18th Workshop on Logic-based methods in Programming Environments (WLPE 2008), Udine, Italy, pp. 77–89 (December 2008)

[Pra10] Prantl, A.: High-level compiler support for timing analysis. PhD thesis, Vienna University of Technology (2010)

[PSK08] Prantl, A., Schordan, M., Knoop, J.: TuBound – A Conceptually New Tool for Worst-Case Execution Time Analysis. In: 8th International Workshop on Worst-Case Execution Time Analysis (WCET 2008), pp. 141–148, Prague, Czech Republic, Österreichische Computer Gesellschaft (2008) ISBN: 978-3-85403-237-3.

[Sch07] Schulte, D.: Modellierung und Transformation von Flow Facts in einem WCET-optimierenden Compiler. Master's thesis, Universität Dortmund (2007)

[Sch08] Schordan, M.: Source-to-source analysis with SATIrE - an example revisited. In: Scalable Program Analysis, Dagstuhl, Germany, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany. Dagstuhl Seminar Proceedings, vol. 08161 (2008)

[Ste96] Steensgaard, B.: Points-to analysis in almost linear time. In: POPL 1996: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 32–41. ACM, New York (1996)

[Tar75] Tarjan, R.E.: Efficiency of a good but not linear set union algorithm. J. ACM 22(2), 215–225 (1975)

[WEE+08] Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., Stenström, P.: The worst-case execution-time problem—overview of methods and survey of tools. ACM Trans. Embed. Comput. Syst. 7(3), 1–53 (2008)

[ŽVSM94] Živojnović, V., Velarde, J.M., Schläger, C., Meyr, H.: DSPstone: A DSP-Oriented Benchmarking Methodology. In: Proceedings of the International Conference on Signal Processing and Technology (ICSPAT), Dallas (October 1994)

Practical Experiences of Applying Source-Level WCET Flow Analysis on Industrial Code*

Björn Lisper[1], Andreas Ermedahl[1], Dietmar Schreiner[2], Jens Knoop[2], and Peter Gliwa[3]

[1] School of Innovation, Design, and Engineering, Mälardalen University, SE-721 23 Västerås, Sweden
[2] Institute of Computer Languages, Vienna University of Technology, A-1040 Vienna, Austria
[3] GLIWA GmbH embedded systems, Dollmann Str. 4, D-81541 München, Germany

Abstract. Code-level timing analysis, such as Worst-Case Execution Time (WCET) analysis, takes place at the binary level. However, much information that is important for the analysis, such as constraints on possible program flows, are easier to derive at the source code level since this code contains much more information. Therefore, different source-level analyses can provide valuable support for timing analysis. However, source-level analysis is not always smoothly applicable in industrial projects. In this paper we report on the experiences of applying source-level analysis to industrial code in the ALL-TIMES FP7 project: the promises, the pitfalls, and the workarounds that were developed. We also discuss various approaches to how the difficulties that were encountered can be tackled.

1 Introduction

Today, over 99 percent of all processors are *embedded* in products or systems. Many of these embedded systems are *real-time systems*, i.e., computer systems that must react within a certain time to events in their environment. Failure of such systems to meet their timing constraints could endanger human life and cause substantial economic losses. Thus, improved methods for timing verification are of paramount importance.

The purpose of *timing analysis* is to find out whether the timing constraints of a system are met. It is usually divided into *system-level analysis*, dealing with the combined execution of different programs ("tasks"), and *code-level analysis* dealing with the timing of individual tasks. The most important entity to estimate on code level is the *worst-case execution time* (WCET) of a task. WCET estimates are needed for many system-level analyses. For these to yield reliable results, the WCET estimates must be *safe* meaning that they never will underestimate the true WCETs. Unsafe WCET estimates can also be useful, in situations like early design phases where they can help dimensioning the system.

* This work was supported by the EU FP7 project ALL-TIMES (Integrating European Timing Analysis Technology, grant agreement no. 215068).

T. Margaria and B. Steffen (Eds.): ISoLA 2010, Part II, LNCS 6416, pp. 449–463, 2010.

WCET analysis assumes that the code is running to completion in isolation, without being interrupted. The two main kinds of analysis are *measurements* and *static analysis*. In general, measurements cannot give safe estimates and are thus suitable for less time-critical software. For time-critical software, where the WCET must be safely estimated, static analysis is preferable. *Hybrid analysis* combines elements of static and measurement-based analysis.

Static WCET analysis derives a WCET estimate analysing mathematical models of the code and the hardware. If the models are correct, then a static WCET analysis will always derive a safe WCET estimate. The static analysis is usually divided into three phases: a *flow analysis* where information about the possible program execution paths is derived, a *low-level analysis* where the execution times for atomic parts of the code, like basic blocks, are decided from a performance model of the target architecture, and a final *calculation* where the flow and timing information is combined to derive a WCET estimate.

The flow analysis must handle a possibly very large number of paths. Thus, approximations are needed. If the approximations are safe (representations include at least the possible paths) then a WCET analysis is still safe, although it might become pessimistic. Flow analysis research has mostly focused on *loop bound* analysis [11], since upper bounds on the number of loop iterations must be known in order to derive finite WCET estimates. Automatic methods to find these exist, but often some loop bounds must still be bounded manually. Flow analysis can also identify other types of program flow constraints like *infeasible paths* [10]. constraining the number of times different program parts can be executed together.

To be accurate, flow analysis should be performed on the binary level, for the code actually being executed. However, much information is often missing in this code. Therefore, flow analysis is sometimes performed on source code. The advantage is that there is much more high-level information in the code that can help obtaining a precise flow analysis. On the backside, the program flows in source and binary are sometimes not exactly the same, especially if the compiler applies heavy optimizations. Program flow constraints derived from the source code can then be unsafe for the program flows of the binary. But there are still many situations where this is tolerable, for instance when making rough timing estimates in early design phases. Also, the technique is applicable for safety-critical applications where safety standards demand that compiler optimizations are turned off.

In this paper, we report on some practical experiences of source-level flow analysis in the ALL-TIMES FP7 project. The purpose of this project was to create methodologies and toolchains for combinations of different timing tools and techniques, including source-level flow analysis. The techniques were finally tried out on an industrial project from the automotive area. We ran into a number of problems when attempting to analyse the source code for the project. We describe these problems, and the workarounds that we had to make to be able to proceed. The problems that we encountered seem to be quite general for

source-level analysis of this kind of code, and we discuss a number of possible ways to tackle them.

The rest of this article is organized as follows: in Section 2 we give an account for related work. Section 3 describes the ALL-TIMES project briefly. In Section 4 we describe the target system for which we applied source-level flow analysis. Section 5 describes how the source-level analysis was used, and which aspects were tested. Section 6 describes our analysis tool SWEET, which was used in the case study, and its flow analysis. Section 7 reports on the results and experiences of the case study. Section 8, finally, wraps up and discusses possible directions for future work.

2 Related Work

There have been a number of studies of WCET analysis of industrial code. There are some reports on using commercial, static analysis WCET tools to analyze code for space applications [13,12,17], and in avionics industry [8,20,15]. In [4,18], time-critical parts of the commercial real-time OS Enea OSE were analysed. Experiences from WCET analysis of real-time communication software for automotive was reported in [3]. All these case studies concern analysis of binary code only. A problem detected in several of them is the need to provide manual annotations for program flow properties at binary level. This was often found to be cumbersome and error-prone.

[19] reports on WCET analysis of a number of tasks in the transmission control of an articulated hauler. This analysis was done on binary level. In a follow-up study [1], the program flow analysis was done on source level, and the results passed to the binary level analysis be hand. The experiment was successful in the sense that almost all program flow constraints on the binary level could be automatically derived on source level. Compiler optimizations changing the control structure did not pose any big problems for the translation. What did pose a problem was the fact that the tasks communicate through data stored in intricate data structures. Value annotations had to be provided for some of these values, which proved to be difficult for two reasons: the inability of the annotation language to express accurately the value fields in the data structures, and the difficulty to find proper value ranges for data stored and updated by several tasks. For this case study, there was access to all the source code. The code was strictly ANSI-C, and parsing did not pose any problems.

There are a number of general-purpose academic or commercial tools for source-level static analysis. Some examples are ASTRÉE [6], Coverity [2], Polyspace, and Klocwork. These tools can check for a variety of possible errors such as possible division by zero, array index out of range, or potential memory leaks to name a few. A comparative evaluation of Coverity, Polyspace, and Klocwork is found in [7].

In [2], a number of practical problems for source-level static analysis tools, when applied to industrial code, were reported. Among the experiences reported were the need to handle different environments for program development, problems parsing code accepted by the customers' compilers, and the need to reduce

the ratio of false to true positives. Clearly these experiences apply to supporting source-level analyses for WCET analysis as well, and they are in line with what we report here.

3 The ALL-TIMES Project

ALL-TIMES (Integrating European Timing Analysis Technology)[1] is an FP7 small to-medium focused research project (STREP) with six partners: Mälardalen University (coordinator), Vienna University of Technology, AbsInt Angewandte Informatik GmbH, Gliwa GmbH, Symtavision GmbH, and Rapita Systems Ltd. The partners are either research groups, with academic tools applicable to timing analysis, or vendors of timing analysis tools. The project has brought the following main results:

- integrated methodologies for timing analysis,
- prototypes of integrated tool chains,
- new/improved code and timing analysis tools,
- new tool connections, including open tool interfaces, and
- a validation of the integrated tool chains.

Fig. 1 shows the different tools, and the different tool connections that have been created within the project. The connections enable the integrated tool chains that in turn support the integrated methodologies.

The tools have the following functions: SymTA/S is a system-level timing analysis tool performing tasks like schedulability analysis. aiT is a static WCET analysis tool, and RapiTime is a tool for WCET analysis and worst-case performance profiling which uses a hybrid method. T1 is a tool for timing- measurement, -analysis and -visualisation. SWEET is a static WCET analysis tool: however, only its program flow analysis part has been of concern in ALL-TIMES. SATIrE, finally, is a source-code program analysis and transformation workbench built on top of the Rose compiler framework[2].

With the new tool connections that have been implemented in the project, a number of integrated tool chains can be formed. An example is RapiTime passing estimated WCETs for tasks to SymTA/S through tool connection M. RapiTime, in turn, may have its analysis enhanced by source-level analyses. For instance, SATIrE may pass information about possible function pointer values through tool connection P, and SWEET may provide program flow constraints through tool connection T. (SWEET must then use SATIrE as a frontend using tool connection Q, see below.) In all, this constitutes a tool chain involving four of the tools in the project.

An outspoken goal of ALL-TIMES has been to provide support for timing analysis in early, explorative design phases. Early indications of timing properties can help avoiding costly redesigns in situations where a late timing verification shows that the timing constraints are not met. For this purpose, a version

[1] Website: `www.all-times.org`
[2] `www.rosecompiler.org`

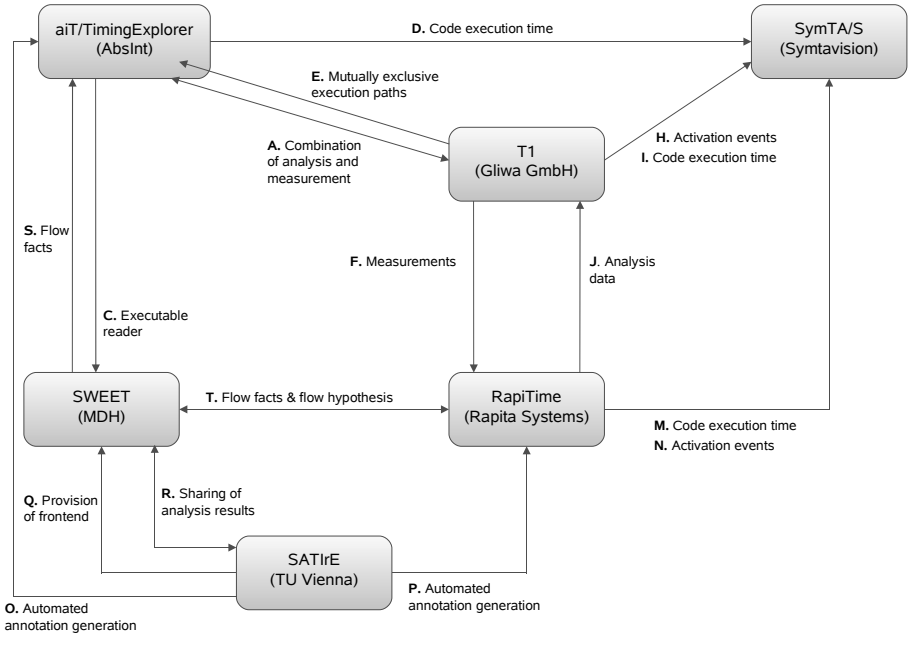


Fig. 1. ALL-TIMES integrations between timing analysis tools

of aiT called "TimingExplorer" was implemented. TimingExplorer sacrifices absolute safety of WCET estimates for analysis speed, and has means to do quick design space exploration varying the hardware configuration. It has the same interfaces as aiT, and can thus take part in various tool chains for providing early timing estimates. For instance, it can use information from source-level analyses performed by SATIrE and SWEET.

Another goal of ALL-TIMES has been to provide support for timing analysis from source-level analyses. Two kinds of program flow information were deemed to be of high interest: information about possible values of function pointers in different program points, and traditional program flow information such as bounds on the number of loop iterations, and infeasible path constraints. These analyses are provided by SATIrE and SWEET. SATIrE uses the C frontend of Rose to parse C source code before analysing it: it can then deliver function pointer sets to aiT and RapiTime using each tool's native format for manual source-level annotations. SATIrE also has an experimental loop bounds analysis. SWEET can compute a number of program flow constraints, ranging from simple loop bounds to complex infeasible path constraints, and can export these to aiT and RapiTime using the same annotation formats as SATIrE. SWEET's flow analysis analyses the ALF code format (see Section 6). SATIrE can convert Rose's internal program representation to ALF through the "melmac" tool, and can thus act as a C frontend to SWEET. Thus, the source-level analysis of SWEET always uses SATIrE as a frontend.

The results of ALL-TIMES were validated using industrial code from the automotive domain, see Section 4. The validation included an estimation of the productivity increase brought by the technologies developed in ALL-TIMES. Two typical timing analysis scenarios were analysed, where for each step the time to perform it was estimated with and without the support of the ALL-TIMES technologies, respectively. The results indicate that the improved timing analysis support brought by ALL-TIMES can have a very significant impact on the efficiency of the timing analysis process.

4 The Target System

The target system that was used for the final validation in ALL-TIMES is an automotive embedded control unit. For this unit, the project could obtain access to the source code. This was crucial for the validation of the source-level tools and -connections. The unit has a Freescale MPC 564 processor. It has no hardware trace facilities, and it uses the ETAS ERCOSEK operating system. The source code consists of ca. 685k lines of code, divided on 1297 C files (.c), 768 C header files (.h), and 28 assembly files (.a). The C files range in size from less than 1 KB to 1997 KB (22910 lines of code). The source code is compiled with the Windriver compiler (Diab Data).

In addition to the source code, the project had access to a test environment including compiler, debugger, and hardware.

5 Source Code Analysis Validation

The source code analyses developed within ALL-TIMES were validated as parts of larger tool chains, involving also the WCET analysis and system-level timing analysis tools in the project (see Section 3). The tool chains were validated using two scenarios, in the following way. The first validation scenario was an "early design exploration" scenario, where TimingExplorer is used to select an appropriately upgraded hardware platform for an existing application. For this purpose, a set of approximate WCETs of the tasks were computed for each investigated hardware configuration, and they were subsequently sent to SymTA/S for a schedulability analysis. Computing bounded WCETs require that all loops in the code are bounded. TimingExplorer does perform an automatic loop bounds analysis: however, this analysis is done on the binary code, and so, for the target code of the validator, TimingExplorer was unable to bound 102 loops. The source-level analysis validation for this scenario was to see how many loops SWEET could additionally bound by a source-level analysis, and pass the results to TimingExplorer to avoid the time-consuming process of manually annotating the loop bounds.

The second scenario was a "late stage verification" scenario, where SATIrE, SWEET, and RapiTime are used to analyse/measure the code and determine the WCETs of the tasks in the target system. RapiTime produces traces that are visualised using one of the so-called "trace-viewers" provided with T1 or

SymTA/S. SymTA/S also performs a scheduling analysis to verify schedulability under worst-case conditions.

Here, it was tested how well SATIrE could determine function pointer sets, which RapiTime needs to cut down the set of possible execution paths when calling a function that is indirectly referenced through such a pointer. SWEET's loop analysis was used in a different way than for the first scenario. RapiTime measures execution times for program fragments, and combines this information into a WCET estimate using program flow constraints. The validator has no hardware tracing facilities, and thus the code has to be instrumented for measuring execution times. The instrumentation has to be sparse since otherwise buffers would overflow and data would be lost. Therefore it is interesting to identify parts of the code with little variability of the execution time, since such parts are prime candidates not to instrument. Loops that always execute the same number of times are likely to have low execution time variability. Since SWEET can compute both upper and lower iteration bounds for loops, it was tested how many loops SWEET could find where these bounds were equal (and thus the loop always executes the same number of times). This information could help reduce a time-consuming inspection of the code to find those parts of the code where instrumentation could be reduced.

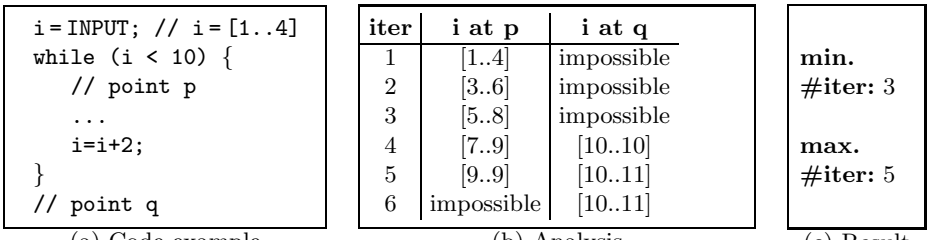
6 SWEET, and Its Flow Analysis

SWEET[3] is a WCET analysis research tool from MDH. It has the standard WCET analysis tool architecture with a flow analysis, a low-level analysis, and a final WCET estimate calculation.

The flow analysis of SWEET is the part currently being actively maintained and developed, and this is the part of SWEET that has been of concern for the ALL-TIMES project. The analysis is able to produce a number of program flow constraints, ranging from simple upper and lower loop iteration bounds to complex infeasible flow constraints. The analysis is context-sensitive as well as *input-sensitive*, the latter meaning that the analysis can take restrictions on input values into account when computing the program flow constraints.

The main analysis method of SWEET is called *abstract execution* (AE) [10]. AE can be seen as a fully context-sensitive abstract interpretation, where each loop iteration (or recursive function call) is analysed separately. It is therefore a "deep" analysis method, based on the semantics of the program. Alternatively, AE can be seen as a kind of symbolic execution where program variables store abstract values, and where operators and functions are given their interpretations in the abstract value domain. Fig. 2 illustrates how AE works for a simple loop. Note the input-sensitivity: the restricting interval for the initial value of `i` will directly influence the resulting iteration bounds for the loop. A simpler, non-input-sensitive analysis would not be able to bound this loop. The current implementation of AE in SWEET uses the abstract interval domain for numerical values, but AE can in principle use any abstract domain.

[3] `www.mrtc.mdh.se/projects/wcet/sweet.html`



(a) Code example (b) Analysis (c) Result

Fig. 2. Example of abstract execution

The computed flow constraints are arithmetic constraints on so-called *execution counters*, virtual variables that count the number of executions in different program points. (In Fig. 2, "**#iter**" is such a counter.) SWEET has a general method to collect information about possible executions into different kinds of program flow constraints. The resulting constraints can be directly used in the final WCET estimate calculation.

The analysis method of SWEET is general and not a priori tied to any language or instruction set. To take advantage of this generality, SWEET's flow analysis analyses the code format ALF [9]. ALF is an intermediate format designed to be able to faithfully represent code on different levels, from source to binary level. It therefore contains high-level concepts, such as functions and function calls, as well as low-level concepts like dynamic jumps. Thus, SWEET can analyse code both on binary and source level provided that a translator into ALF is present. As mentioned, the melmac tool is such a translator that maps the internal format of Rose into ALF, thereby enabling SATIrE to be used as a C frontend to SWEET.

7 Results and Experiences

We now describe our experiences from doing source-level analysis on the validator (automotive ECU) code in the ALL-TIMES project. We discuss the problems encountered for the source-level program flow analysis performed by SWEET. Most of the problems are common to SATIrE's analyses as well, and indeed they should be common to "deep", semantics-based source-level analyses in general.

The steps necessary to perform SWEET's flow analysis are basically the following. This scheme should hold for source-level analysis in general:

1. identify the source files that are needed for the analysis,
2. convert the files to a format suitable for the static analyses to be applied,
3. "link" the converted files to enable inter-procedural program analyses,
4. perform the program analyses, and
5. map the results back to the source code, in a format suitable for other tools.

We now describe our experiences for each step in turn.

7.1 Step 1: Identify Needed Source Files

Academic benchmarks for WCET analysis research tend to consist of small, self-contained programs where each program to analyse resides in a single source file. Not so in real projects: they may consist of thousands of files, where the code that needs to be analysed is scattered over many different files.

The first step is to *identify the entry points for the code to analyse*. For WCET analysis, this typically amounts to finding the start functions for the tasks, or runnables, in the system. If the application uses an operating system, then the tasks can usually be found from some of its configuration files. This is the method that we used. It requires knowledge of how the specific operating system used is configured.

Second, the files containing code that is needed for the analysis are to be identified. This may sound trivial: a brute force approach that should work would seemingly be to include all source files in the project. But this requires knowledge of where the source files are located, which typically has to come from the build tool (which, if unlucky, may be some set of obscure scripts). A complication is that some source files may include assembler, or that they are simply missing (only binaries available): more on the consequences of this in Section 7.4.

A further complication, which we encountered in the project, is *late binding*. Functions with the same name may be defined in different compilation units, and first at linking time it is selected which of these functions to actually call, by including the corresponding object file. Late binding seems to be very common for embedded codes. With late binding, the source code simply does not contain sufficient information which source files to include, information from the build tool is needed.

To summarize: Step 1 in general requires information both from the operating system configuration and the build tool, and a source-level analysis tool must somehow acquire it.

7.2 Step 2: Converting Source Files

For SWEET's flow analysis, the conversion is done in two steps: first, the source files are parsed into the internal format of Rose, and then a translation to ALF is done by the melmac tool.

SWEET needs to generate flow constraints in the annotation formats of aiT and RapiTime. Both these annotation formats relate the flow constraints to locations in the source code. Therefore, in addition to the code conversion, melmac generates a so-called *map file* relating code locations in ALF with the corresponding locations in the original source code.

The two first steps presented problems. The Rose compiler framework uses a commercial C/C++ frontend which is distributed with Rose as a binary, so it cannot be changed. It turned out that the parser could not process some of the target system's source files. We had to rewrite some source files by hand to get them through the parser. Some source files also contained inlined assembler,

which the frontend could not process either. We dealt with this mainly by commenting out the assembler, although this is not a strictly safe solution. Other constructs causing problems were compiler specific pragmas, and preprocessor directives. The problems encountered were very similar to those reported in [2].

It also turned out that Rose itself had problems to handle some features of the target system's C code. We had to rely on the Rose developers to for fixing the problems, which took time.

The morale of this is that an industrial strength source analysis tool, at least for C, must be able to adapt to the variety of C dialects defined by the C compilers that are in use. It is hard to accomplish this without being in full control of the translation chain.

7.3 Step 3: Link the Converted Files

When analysing code that stretches several files or compilation units, a global namespace has to be created where symbols have an unambiguous meaning. This process can be seen as a kind of linking [14], but on source-code level.

Source-level linkers exist. One example is `cilly`, from the CIL framework [5], which merges a number of C file into one large C file. However, `cilly` changes the line numbers of statements in a way that cannot be easily traced. The original line numbers are required for flow constraint annotations in the annotation format for aiT, so if they are lost then such annotations cannot be generated. For this and other reasons, we decided to create or own linker for ALF code instead.

Our ALF linker works similarly to `cilly`, and merges all the ALF files into one big file. However, it also maintains the relation between program code locations in C and ALF files by creating a global map file from the merged ALF files. To differ between local entities with the same name, the ALF linker has to perform some "name mangling" to create unique names.

A problem, similar to the late binding problems described in Section 7.1, was caused by the fact that some header (.h) files contained full function definitions. Since the C preprocessor's semantics of including header files is to copy them verbatim, the result was the creation of several instances of an identical function with the same name in different C files. The C preprocessor would then be used to select which copy to include in the end. However, since the C files were translated into ALF one by one, the result would be several ALF files with identical function declarations. The ALF linker, righteously, did not accept this. Our solution was to change the linker to eliminate all identical copies of a function declaration but one.

7.4 Step 4: Performing the Flow Analysis

The linked ALF file should in principle be ready for analysis. Alas, a successful flow analysis for real industrial codes typically requires additional information not present in the available source code. This information must then somehow be provided, or approximated.

When analysing the possible program flows of a task, typically some bounds must be known for inputs that may affect the program flow (see Fig. 2 for an example). Inputs can be either arguments to the start function of the task, if any, and values of global variables when the start functions begin executing.

Especially important is to restrict the possible values of input pointers. If the abstract value of a pointer is TOP (any address), and an assignment is done using the pointer value as target, then the analysis must assume that the written value may possibly be written to any data address. This typically destroys the information needed to identify any interesting program flow constraints.

SWEET provides value annotations for arguments and variables, which allow the user to provide simple interval bounds for these. Inputs that are not annotated will assume the abstract value TOP (any value is possible). However, the problem remains how to identify the important inputs and find proper value ranges for these.

Since we had little time to make elaborate annotations in the ALL-TIMES project, we would mostly let input values default to TOP. We suspect that this affected the precision of the analysis adversely in many cases.

For systems that allow parallel or pre-emptive execution of tasks accessing shared data areas, the additional complication arises that during the execution of a task, a variable can be reassigned a new value by some other task. Such shared variables must be treated differently in the analysis. If there is no knowledge about the possible preemption points, then the value of such a variable must be considered possible to change, due to the actions of some other task in the system, at any program point. An analysis that does not take this into account is potentially unsafe. As far as we know, all current WCET analysis tools ignore this problem.

C has a keyword "volatile", which is a hint to the compiler that the variable might be changed by some other activity in the system. A variable might be marked volatile for many other reasons, though, and conversely any "non-volatile" global variable could potentially be shared between tasks. Analyses that identify shared data and compute safe value bounds for these are conceivable: however, we had no time to develop such analyses in the project. SWEET currently deals with the problem by forcing volatiles to assume the abstract value TOP throughout the program, but as explained this is potentially unsafe.

The second problem is that vital source code often is missing. For the validator, the big problem turned out to be that important parts of the code came from a subcontractor, and we had no access to its source code. Apparently this is a common situation in the automotive domain, and the trend towards componentising the software and use third-party code seems to become stronger. This means that missing source code will be a significant problem for source-level analysis in the future.

We tackled this problem in two ways. First, we added a mode to SWEET where it will assume a certain default semantics for unknown entities. Unknown variables will be set to TOP, which is a safe overapproximation but may lead to very imprecise results especially of the variable holds pointer values. The default

semantics for unknown functions is that their return value is TOP, and that they do not affect any global variables (no side-effects). The latter assumption is clearly unsafe, but without knowledge of the function a safe analysis will have to assume that it can set any global variable to anything, which will render the resulting analysis extremely imprecise.

Second, we extended the annotation language of SWEET to give it some limited capability to express the semantics of unknown functions. These annotations allow to specify bounds on return values, and for possible side effects on given global variables.

Missing code, and uncertainty about the nature of volatile-declared variables, posed a problem for the source-level validation. As mentioned, TimingExplorer failed to bound 102 loops, located in 50 different C functions. Of these, only 19 functions could be translated into ALF (mostly due to that the source code was missing). 24 of the not bounded loops were present in those functions. Two of these 19 functions refer to unknown entities, and 18 functions refer to volatiles. Of the total number of functions that could be translated into ALF, about 49% contained neither undefined entities nor volatiles (and should thus be possible to analyse with safe results, assuming that no non-volatile globals can be affected from outside the analysed task).

There were 18 start functions for tasks in the system. Of these 10 referred to undefined functions, and 16 to undefined data. Nine contained loops for which we had ALF code, and which TimingExplorer could not bound. For a number of reasons, we could not obtain any bounds for any of the not bounded loops by analysing the start functions. This was due to imprecision from unknown entities (especially unknown pointers), but also presence of volatiles whose safe treatment required them to be set to TOP, and infinite `for(;;);` loops that pose problems for the abstract execution algorithm used by SWEET. Thus, we resorted to analyzing functions deeper in the call tree, containing (possibly indirect) calls to the functons with the interesting loops. This had the advantage of avoiding to analyse much problematic code. On the other hand, calling context information was lost which would decrease the precision. By analysing these functions, we were able to bound three of the previously unbounded loops by a safe analysis (w.r.t. undefined entities and volatiles), and two more by a potentially unsafe analysis making default assumptions about undefined entities, and treating volatiles as "ordinary" variables. We believe that the low number of interesting loops that we were able to bound mainly is due to imprecision from undefined entities, and lack of knowledge about the calling contexts.

7.5 Step 5: Map Results Back to Source Code

The last step is to map flow constraints back to TimingExplorer or RapiTime using their native source-level annotation formats. Also here, we encountered some practical problems. One problem is that some precision was lost in the translation to these formats: for instance, SWEET can generate highly context-sensitive flow constraints which cannot be expressed to the same degree in them. Another issue is that SWEET derives loop bounds as bounds on the number of

times the loop header executes, while both annotations formats use the number of times that the loop body executes. This forced us to make adjustments in the generated loop annotations.

8 Conclusions

We have identified a number of problems with source-level program flow analysis. Some problems are due to important information not being present in the source code, some to weak language standards (allowing different compilers to define different dialects), and some to missing source code. Most problems should be common to deep, semantics based source-level analysis in general. Thus, solutions are important not only for WCET analysis, but also for general source-level analysis.

It is not easy to change industrial practices. However, we can still make a "wish list". One thing on the list would be standardised formats for build processes, which an analysis tool could read to find the relevant source code to analyse. On the task level, clean standards for task communication are very desirable. Metadata should be provided, in standardised formats, that describes the task communication as well as the task entry points. Metadata should also be available to describe possible value restrictions on inputs to the system, like values read from sensors.

It should also be possible for third-party code providers to ship metadata describing the semantics of the delivered code. This metadata could then be used by an analysis tool when the source code is not provided. Examples of useful metadata are input-output relations for functions, possible side effects, and value restrictions for global variables. Function side effects can be expressed in types, and found by a type-based analysis [16].

The natural place to standardise metadata formats is in component-oriented frameworks such as AUTOSAR. However, also without them, there are improvements to be done. One interesting option is to develop mixed source/binary-level analyses, that use the binaries when sources are not available. Using a multilevel language as ALF for the analysis, which allows translations from both source- and binary level code, is then clearly an advantage. Analysis tools also need better annotation languages to let the user provide precise information when needed.

Finally, note that most problems described above will not appear for a binary level analysis of a statically linked executable. All the code will then be available, and all ambiguities will have been resolved by the compiler and linker. This compensates for the disadvantages of binary level analysis to some degree. However, we think that source-level analysis offers distinct advantages warranting efforts to make it applicable to real embedded production codes.

References

1. Barkah, D., Ermedahl, A., Gustafsson, J., Lisper, B., Sandberg, C.: Evaluation of automatic flow analysis for WCET calculation on industrial real-time system code. In: Proc. 20th Euromicro Conference of Real-Time Systems (July 2008)

2. Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Henri-Gros, C., Kamsky, A., McPeak, S., Engler, D.: A few billion lines of code later: Using static analysis to find bugs in the real world. Comm. ACM 53(2) (February 2010)
3. Byhlin, S., Ermedahl, A., Gustafsson, J., Lisper, B.: Applying static WCET analysis to automotive communication software. In: Proc. 17th Euromicro Conference of Real-Time Systems (ECRTS 2005) (July 2005)
4. Carlsson, M., Engblom, J., Ermedahl, A., Lindblad, J., Lisper, B.: Worst-case execution time analysis of disable interrupt regions in a commercial real-time operating system. In: Pettersson, P., Yi, W. (eds.) Proc. 2nd International Workshop on Real-Time Tools, Copenhagen (August 2002)
5. CIL - infrastructure for C program analysis and transformation (2008), manju.cs.berkeley.edu/cil
6. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The ASTRÉE analyzer. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 21–30. Springer, Heidelberg (2005)
7. Emanuelsson, P., Nilsson, U.: A comparative study of industrial static analysis tools (extended version). Technical report, Linköping University (January 2008), http://www.ep.liu.se/ea/trcis/2008/003/
8. Ferdinand, C., Heckmann, R., Langenbach, M., Martin, F., Schmidt, M., Theiling, H., Thesing, S., Wilhelm, R.: Reliable and precise WCET determination for a real-life processor. In: Henzinger, T.A., Kirsch, C.M. (eds.) EMSOFT 2001. LNCS, vol. 2211, Springer, Heidelberg (2001)
9. Gustafsson, J., Ermedahl, A., Lisper, B., Sandberg, C., Källberg, L.: ALF – a language for WCET flow analysis. In: Proc. 9th International Workshop on Worst-Case Execution Time Analysis (WCET 2009), Dublin, Ireland, pp. 1–11 (June 2009)
10. Gustafsson, J., Ermedahl, A., Sandberg, C., Lisper, B.: Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution. In: Proc. 27th IEEE Real-Time Systems Symposium (RTSS 2006) (December 2006)
11. Healy, C., Sjödin, M., Rustagi, V., Whalley, D., van Engelen, R.: Supporting timing analysis by automatic bounding of loop iterations. Journal of Real-Time Systems 18(2-3), 129–156 (2000)
12. Holsti, N., Långbacka, T., Saarinen, S.: Using a worst-case execution-time tool for real-time verification of the DEBIE software. In: Proc. DASIA 2000 Conference (Data Systems in Aerospace 2000, ESA SP-457) (September 2000)
13. Holsti, N., Långbacka, T., Saarinen, S.: Worst-case execution-time analysis for digital signal processors. In: Proc. EUSIPCO 2000 Conference (X European Signal Processing Conference) (2000)
14. Levine, J.: Linkers and Loaders. Morgan Kaufmann, San Francisco (2000) ISBN 1-55860-496-0
15. Montag, P., Goerzig, S., Levi, P.: Challenges of timing verification tools in the automotive domain. In: Margaria, T., Philippou, A., Steffen, B. (eds.) Proc. 2nd International Symposium on Leveraging Applications of Formal Methods (ISOLA 2006), Paphos, Cyprus (November 2006)
16. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis, 2nd edn. Springer, Heidelberg (2005) ISBN 3-540-65410-0
17. Rodriguez, M., Silva, N., Esteves, J., Henriques, L., Costa, D., Holsti, N., Hjortnaes, K.: Challenges in calculating the WCET of a complex on-board satellite application. In: Proc. 3rd International Workshop on Worst-Case Execution Time Analysis (WCET 2003), Porto (July 2003)

18. Sandell, D., Ermedahl, A., Gustafsson, J., Lisper, B.: Static timing analysis of real-time operating system code. In: Margaria, T., Steffen, B. (eds.) ISoLA 2004. LNCS, vol. 4313, pp. 146–160. Springer, Heidelberg (2006)
19. Sehlberg, D., Ermedahl, A., Gustafsson, J., Lisper, B., Wiegratz, S.: Static WCET analysis of real-time task-oriented code in vehicle control systems. In: Margaria, T., Philippou, A., Steffen, B. (eds.) Proc. 2nd International Symposium on Leveraging Applications of Formal Methods (ISOLA 2006), Paphos, Cyprus (November 2006)
20. Thesing, S., Souyris, J., Heckmann, R., Randimbivololona, F., Langenbach, M., Wilhelm, R., Ferdinand, C.: An abstract interpretation-based timing validation of hard real-time avionics software. In: Proc. of the IEEE Int. Conf. on Dependable Systems and Networks, DSN 2003 (June 2003)

Worst-Case Analysis of Heap Allocations*

Wolfgang Puffitsch[1], Benedikt Huber[1], and Martin Schoeberl[2]

[1] Institute of Computer Engineering, Vienna University of Technology, Austria
wpuffits@mail.tuwien.ac.at, benedikt@vmars.tuwien.ac.at
[2] Dept. of Informatics and Mathematical Modeling, Technical University of Denmark
masca@imm.dtu.dk

Abstract. In object oriented languages, dynamic memory allocation is a fundamental concept. When using such a language in hard real-time systems, it becomes important to bound both the worst-case execution time and the worst-case memory consumption. In this paper, we present an analysis to determine the worst-case heap allocations of tasks. The analysis builds upon techniques that are well established for worst-case execution time analysis. The difference is that the cost function is not the execution time of instructions in clock cycles, but the allocation in bytes. In contrast to worst-case execution time analysis, worst-case heap allocation analysis is not processor dependent. However, the cost function depends on the object layout of the runtime system. The analysis is evaluated with several real-time benchmarks to establish the usefulness of the analysis, and to compare the memory consumption of different object layouts.

Keywords: Worst-Case Analysis, Memory Allocation, Real-Time Java.

1 Introduction

In hard real-time systems, failing to deliver results in time may lead to catastrophic consequences. Deadlines must be met even in worst-case scenarios. As such situations cannot be reliably provoked through measurements, hard real-time systems must be statically analyzed to ensure that all deadlines will be met. Scheduling analysis determines whether all tasks can meet their deadlines. The input for this analysis are the worst-case execution times (WCETs) of the individual tasks and their respective deadlines. However, it is not only important to consider the tasks' timing. An application that runs out of memory cannot deliver its result on time either. Therefore, it is important to bound not only the WCET, but also the worst-case heap allocations (WCHAs). With the known WCHAs, the memory management system, be it scoped memory or a real-time garbage collector (GC), can be correctly dimensioned.

* The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement number 216682 (JEOPARD).

T. Margaria and B. Steffen (Eds.): ISoLA 2010, Part II, LNCS 6416, pp. 464–478, 2010.

Allocations are often "hidden" behind syntactic features or in libraries. In Java, even innocent-looking expressions such as println("info: "+i) allocate several objects. The expression "info: "+i is processed as follows: a StringBuilder object is allocated, which contains an array to store the actual characters. Then, "info: " is appended to that object, potentially allocating a larger character array. The integer i is also appended to that object; converting a number to its decimal representation requires another character array. Finally, the StringBuilder is converted to a String, allocating yet another object. In total, two objects and two to four arrays are allocated. Considering the simplicity of the above expression, manual analysis of realistic programs is clearly not an option.

Manual analysis is also complicated by the fact that the requested amount of memory is not always the same as the allocated amount. In object-oriented languages, objects include some meta-information about the type of an object. Some real-time GCs (RTGCs) split objects, either to avoid [19] or to overcome [5] fragmentation issues. Programmers must have intimate knowledge about the runtime system to find out how much memory is actually allocated.

Knowledge about heap allocations is useful both when using a RTGC and when using scoped memory. Scoped memory was introduced in the real-time specification for Java RTSJ [3] to eliminate the need for garbage collection. As the size of the scoped memory area has to be provided when the area is created, it is important to know how much memory will be allocated in that scoped memory. The analysis helps to size the scoped memory area such that allocation demands can be met even in worst-case scenarios.

RTGCs have gained more acceptance since the RTSJ was formulated, and the use of scoped memory often can be avoided. Hard real-time GCs require knowledge about the application for correct operation. A RTGC must be paced correctly, otherwise it cannot keep up with the allocations from the application. In such a case the system would fail, either because it runs out of memory or because tasks are delayed beyond by the GC. The allocation rate alone is not enough to determine an upper bound for the period of the GC thread [14,15]. However, the allocation rate is a necessary prerequisite for pacing the RTGC.

We propose to use the existing technologies for WCET analysis for the analysis of heap allocations and provide cost formulas for several object layouts. While we focus on Java, we believe that the presented ideas could also be applied to other languages. We evaluate the tightness of our approach by comparing the analysis results to measurements, and identify idioms that introduce pessimism.

The following section provides an overview of work related to this paper. Background on WCET analysis is given in Sec. 3. In Sec. 4, we present the analysis to automatically determine the WCHAs of tasks, which is evaluated in Sec. 5. Section 6 concludes the paper and provides an outlook on future work.

2 Related Work

An early attempt at automated computation of upper bounds for different performance measures was presented by Wegbreit [21]. The analyses are formulated

for Lisp and computation takes place on a symbolic level. Aggregation of worst-case results must be conservative and assume that all program fragments exhibit their worst-case behavior at the same time. In contrast, our analysis uses a more powerful approach based on integer linear programming, which allows expressing cases where the execution of program fragments is not independent.

Unnikrishnan et al. presented analyses for both allocations and live memory [20], which are to some degree similar to the analyses by Wegbreit. However, recursion is handled implicitly rather than through explicit equations. The analyses are formulated for a first-order functional language and assume that the input programs are purely functional. It is not clear whether their findings can be directly applied to imperative languages such as Java.

Albert et al. [1] developed an analysis framework where a sub-set of Java bytecodes is transformed to a rule-based procedural representation. Loops are transformed into recursions, and recurrence equations are generated to characterize the program. The solution to these equations provides parametric bounds for the memory consumption.

The analysis presented by Braberman et al. [4] computes the memory usage of "regions" within a program. The memory consumptions of these regions are combined to derive symbolic bounds on the minimum memory consumption and the total amount of allocated memory.

A type-based heap-space analysis is proposed by Hofmann and Jost [8]. Programs are typed with a special type system that is used to establish bounds on the memory consumption. While this approach seems to work reasonably well for bounds that depend on the *size* of the input, it remains unclear whether bounds that depend on input *values* can be handled efficiently.

Mann et al. [11] developed a data-flow analysis to determine worst-case allocation rates. They use an instruction "window", and determine how much data is potentially allocated within such a window. Clustered allocations, as they are common at the start of a task's period, can lead to considerable pessimism when using an instruction window. Considering the whole execution of a task, as our analysis does, levels out such allocation spikes.

3 WCET Analysis

WCET analysis, similar to many other program analyses, is performed on a program abstraction, the control flow graph (CFG). In a CFG the basic blocks are represented by vertices and the directed edges represent possible control flows. The cost of an edge is set to the maximum time needed to execute the basic block it originates from. WCET analysis needs to find the most expensive execution path between the program's entry and exit node. In order to bound the execution time of a task or method, an upper bound for the number of times loops are executed and for recursion depths has to be known. Additionally, one aims to exclude infeasible paths, which are never taken but are part of the CFG abstraction.

```
for (int i = 2; i <= 10; i++) { // outer loop
  for (int j = i; a[j] < a[j - 1]; j--) {
  // @WCA loop <= 9
  // @WCA loop <= 45 outer
    swap(a,j,j-1);
  }
}
```

Listing 1.1. Loop bound annotation example

A common technique to find the WCET is implicit path enumeration (IPET) [13,10]. The problem of finding the most expensive execution path is transformed to a network flow problem. The variables of the problem correspond to the execution frequency of CFG edges. Unique start and end vertices are created with a single outgoing and a single incoming edge with execution frequency one. For all other vertices, representing basic blocks in the CFG, the flow into the vertex is equal to the flow out of the vertex. Furthermore, linear constraints to bound the maximum number of loop iterations and to exclude infeasible paths are added. Each edge is assigned a constant execution cost. The problem of finding the WCET now amounts to finding the flow with maximal cost. The solution to the resulting integer linear programming (ILP) problem can be found by a standard ILP solver, such as lp_solve.[1]

3.1 Loop and Recursion Bounds

In order to bound the WCET, it is necessary to bound the maximum number of loop iterations and the maximum depths of recursions. Simple loop bounds can be automatically extracted from the program source. For this purpose, a data-flow analysis (DFA) framework providing a loop bound analysis is integrated in our WCET analysis tool [17].

However, if a bound cannot be determined automatically, programmers must provide annotations. An example annotation is shown in Listing 1.1. The annotation @WCA loop<=9 tells the analysis that the loop body is executed at most 9 times whenever the loop is entered. The annotation @WCA loop<=45 outer further restricts the number of times the body is executed by stating that it is executed at most 45 times whenever the *outer* loop is entered. The analysis can therefore compute tight bounds for triangular and other non-rectangular loops.

3.2 Data-Flow Analysis

The data flow analyses are run prior to the WCET calculation, and provide information to deal with dynamic dispatch (receiver type analysis) and cycles in the CFG (loop bound analysis). Both analyses are based on the techniques described in [12], and operate directly on Java bytecode.

[1] http://lpsolve.sourceforge.net/5.5/

Receiver Types. A receiver type analysis computes which types an object may actually have. This is useful to reduce the pessimism that is introduced to the WCET/WCHA analysis by virtual method calls. The term *receiver* refers to the object which *receives* a message through the method call.

Our receiver type analysis take call strings into account and is similar to k-CFA ("k^{th}-order control-flow analysis") [18]; a detailed description of the analysis can be found in [17]. We acknowledge that techniques like the ones described in [2] are more efficient than our approach. However, these techniques trade precision for analysis time; the amount of pessimism introduced by this loss in precision remains to be evaluated.

Loop Bounds. The loop bound analysis is based on an interval analysis that computes an upper bound for the values integer variables may hold. It is augmented with information whether a variable is only incremented or decremented. From the value range of a loop variable and its possible increments/decrements, it can be deduced how often a loop may be executed. As this analysis is not the focus of this paper, please refer to [17] for details of the analysis.

A by-product of the loop bound analysis is that it also computes ranges for array sizes. As the analysis computes ranges for all integer variables, it also computes ranges for the values that are passed to newarray, anewarray, and multianewarray. The only necessary change in the analysis was to keep those ranges available for further processing.

3.3 Execution Time Calculation

In addition to the high-level program analysis described above, the construction of a low-level timing model is necessary before calculating the WCET. The low-level analysis provides a bound on the execution time of basic blocks, and depends on the particular target platform.

Given the results of the low-level and high-level program analysis, an ILP instance is generated, with variables corresponding to the edges in the CFGs. The objective function for the ILP problem is obtained by summing up the cost of all edges. The cost of an edge is the product of the edge's frequency (a variable) and the cost of executing the basic block the edge originates from. Finally, the model is passed to an ILP solver, which calculates the edge's execution frequency on the worst-case path, as well as the WCET itself.

4 Heap Allocation Analysis

The WCHA analysis we propose is based on the WCET analysis described in Sec. 3. Instead of using the execution time as cost function for the analysis, we use the amount of memory a bytecode allocates. Apart from the cost function, the problem to be solved for the WCHA is identical to the problem to be solved for the WCET calculation. This implies that the infrastructure for calculating

the WCET can be reused for calculating the WCHA. All path information obtained from value and loop bound analysis and the information extracted from annotations is available to the allocation analysis as well.

Obviously, most bytecodes do not allocate any memory. The amount of memory a new bytecode allocates is determined by the type it allocates. The size of instances of that type are known at compile time, and can be obtained by adding the sizes of all fields for a given class and its superclasses. When allocating arrays, the allocation size is determined at runtime. The cost functions for bytecodes that allocate data are detailed in Sec. 4.2. All other bytecodes have zero cost.

Note that we do not take into account the effects of garbage collection. Especially for multi-threaded applications, modeling the lifetime of data would require taking into account also the timely behavior of the application and runtime system, which is beyond the capabilities of the proposed technique.

In our analysis, we assume a "closed world", i.e., the application must not change during execution, and features like dynamic class loading are precluded. This assumption is necessary, because we obviously cannot analyse unknown code. However, our tool can handle virtual method calls; the cost of a call is the maximum over all possibly invoked methods.

4.1 Array Size Bounds

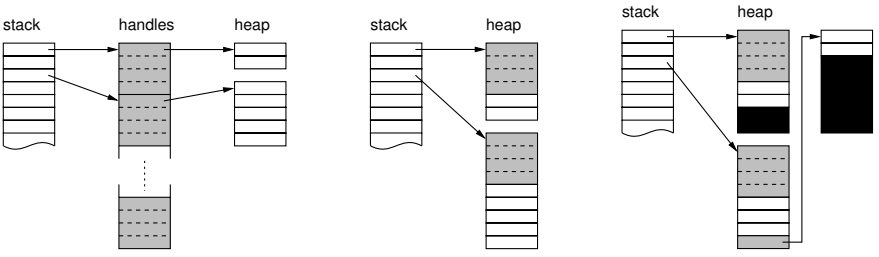
To bound the maximum size of allocated arrays, the DFA has been extended to provide array sizes at the allocation site. During the loop bound analysis, ranges for all integer values have to be computed. As this also includes values on the stack, the analysis has been extended to record bounds for array allocations. When encountering a newarray or anewarray instruction, a mapping between the allocation site and the value range for the array size is added to an allocation bound table. For multianewarray instructions, the analysis must record multiple ranges, to bound every level of the multidimensional array. When computing the cost of an array allocation, the appropriate mapping is retrieved from the allocation bound table.

If the analysis does not succeed in computing a bound, annotations have to be provided by the programmer. An example for an array size annotation would be `arr = new int[n]; // @WCA size<=100`, where the array is annotated to occupy no more than 100 elements.

4.2 Object Layouts

When analyzing the WCET it is also necessary to include low-level details of the processor in the analysis. Some of the low-level architecture details, such as caches and branch predictors, are irrelevant for the WCHA analysis. However, the analysis must take into account the object layout of the underlying JVM.

Figure 1 shows object layout variants that are used in JVMs with RTGCs. White cells contain user data, while gray cells contain meta-information required by the runtime system. Black cells denote memory which cannot be used due to the object layout. The black cells are the result of internal fragmentation.



(a) Handles (b) Headers (c) Fixed Blocks

Fig. 1. Comparison of object layouts

In a RTGC, it is necessary to either defragment memory (by relocating objects) or to avoid unbounded fragmentation. In a handle-based layout (Fig. 1(a)), a handle in a separate handle area points to the actual object. The handle area does not need compaction, because the fixed length of the handles eliminates fragmentation. Object relocation is simple, because only the indirection pointer in the handle has to be updated. While the cost for an object is similar to a header-based layout, it has to be taken into account that also the number of allocated objects has to be bounded in order to fit the handle area.

A layout that incorporates header data (Fig. 1(b)) into the object eliminates the indirection. However, when relocating objects, forwarding pointers have to be followed. On average, such a layout speeds up object accesses, but in the worst case, access times are similar to a handle-based layout due to the indirection through the forwarding pointer during object relocation. Such a layout also complicates defragmentation, because all references in the objects and on the thread stacks have to be updated to point to the new object location.

When using fixed block sizes for all allocations, external fragmentation can be eliminated, albeit at the expense of internal fragmentation. Such a layout is shown in Fig. 1(c). Objects that are too large to fit into a single block are organized as linked list. Arrays are organized as a tree to achieve logarithmic costs for accesses. Individual accesses may be more expensive than with the other object layouts. However, when considering the whole system, this is alleviated by the fact that no defragmentation is necessary.

4.3 Cost Functions

Different object layouts lead to different cost functions for the heap allocation analysis. Object fields can be allocated packed or at word boundaries. Object meta-data (e.g., object type, array size, ...) can be organized differently to optimize for size or for speed. Furthermore, large objects can be split into constant sized blocks to avoid heap compaction or to make object relocation interruptible.

In the following, $\mathcal{F}(o)$ are the fields of an object o, and $\mathcal{F}_k(o)$ is the k-th field of object o (with indices starting at 1). With $s(f)$ we denote the size of a field f in bytes, and with $a(f)$ the required alignment for the field f. For the total

memory usage of an object o we write $mu(o)$. To handle alignment requirements, we define $P(n, m)$ such that it pads the address n to a multiple of m.

$$P(n, m) = \left\lceil \frac{n}{m} \right\rceil m$$

$S(n, f)$ returns the memory usage of an object after adding a field f to that object at relative address n. For example, $S(3, f)$ would evaluate to 8 for a 4-byte field f that requires alignment to 4-byte boundaries.

$$S(n, f) = P(n, a(f)) + s(f)$$

Handle-Based Layout. In a handle-based layout, header data is stored at the handle site, while the payload is located in the remaining heap space. Together, the handle and the payload must fit the total available memory. Furthermore, it is necessary that the handle area is large enough for the handles, and the rest of the heap is large enough for the object data.

The memory usage of an object can be computed with the following formulas:

$$\begin{aligned} mu_h(o) &= s(handle) \\ mu_f^0(o) &= 0 \\ mu_f^k(o) &= S(mu_f^{k-1}(o), \mathcal{F}_k(o)) && k > 0 \\ mu_f(o) &= P(mu_f^{|\mathcal{F}(o)|}(o), A_{field}) \\ mu(o) &= m_h(o) + mu_f(o) \end{aligned}$$

We assume that handles are always aligned, and that any required padding or unused fields are part of $s(handle)$. Furthermore, the formulas assume that object fields start at an address that never requires padding. The maximum alignment for object fields is denoted by A_{field}. By padding the end of the user data to such an alignment boundary, we ensure that the next object starts at this boundary and its first field indeed does not require padding.

The equations for arrays are the same as for objects, except that arrays use an *array_handle* instead of a *handle*. The handle types can differ, because the *array_handle* must accommodate the size of the array and type information may be treated differently.

When being interested in the overall memory consumption, $mu(o)$ is the appropriate cost function. When considering the handle area and the remaining heap space separately, $mu_h(o)$ and $mu_f(o)$ provide the respective cost functions.

Layout with Header Data. Header data is located in the same place as the payload. The memory usage can be computed as follows:

$$\begin{aligned} mu^0(o) &= s(header) \\ mu^k(o) &= S(mu^{k-1}(o), \mathcal{F}_k(o)) && k > 0 \\ mu(o) &= P(mu^{|\mathcal{F}(o)|}(o), A_{header}) \end{aligned}$$

Again, we assume that objects start appropriately aligned and add padding at the end of the object as required. The equations for arrays are the same, except that an *array_header* is used instead of *header*.

Fixed-Block Layout. In the fixed-block object layout, the header data and the start of the object are in the same block. If the header data and the payload exceed the size of a single block, the object is split across several blocks. The link to the next block may be located at the start or the end of a block, which leads to slightly different equations for the memory usage computation. With B we denote the size of a block; we assume that fields never require padding at the beginning of a block.

Next Pointer at End of Block. If the pointer to the next block is located at the end of a block, it must be checked whether the field and the *next* pointer fit the current block when adding a field to an oject. The function S_{next} returns a value greater than B if these two fields do not fit the current block.

$$S_{next}(n, f) = S(S(n \bmod B, f), next)$$

The function S_{block} uses S_{next} to determine the actual memory usage when adding field f at position n.

$$S_{block}(n, f) = \begin{cases} S(n, f) & \text{if } S_{next}(n, f) \leq B \\ P(n, B) + S(0, f) & \text{if } S_{next}(n, f) > B \end{cases}$$

Next Pointer at Beginning of Block. If the *next* pointer is located at the beginning of a block, S_{next} and S_{block} have a slightly different definition:

$$S_{next}(n, f) = S(n \bmod B, f)$$
$$S_{block}(n, f) = \begin{cases} S(n, f) & \text{if } S_{next}(n, f) \leq B \\ P(n, B) + S(s(next), f) & \text{if } S_{next}(n, f) > B \end{cases}$$

In this flavor of the fixed-block object layout, the *next* pointer must taken into account for all blocks. This is especially true for the first block; the *next* pointer for this block is considered as part of the object header.

Memory Usage of Objects. The memory usage for objects can be computed with the following formulas:

$$\begin{aligned} mu^0(o) &= s(header) \\ mu^k(o) &= S_{block}(mu^{k-1}(o), \mathcal{F}_k(o)) && k > 0 \\ mu(o) &= P(mu^{|\mathcal{F}(o)|}(o), B) \end{aligned}$$

The formulas are the same for both flavors of the fixed-block object layout; the placement of the *next* pointer is already taken into account by S_{block}.

Memory Usage of Arrays. Arrays have a special header, that includes the size of the array and depth of the tree representation. As all fields are the same size and have the same padding requirements, we do not need to use a recursive definition for the memory consumption. $L(a)$ captures how many array fields fit into a single block. $N(a)$ is the number of blocks the array elements occupy. M is the number of *next* pointers within an inner node of the tree representation.

$$L(a) = \left\lfloor \frac{B}{s(\mathcal{F}_1(a))} \right\rfloor \qquad N(a) = \left\lceil \frac{|\mathcal{F}(a)|}{L(a)} \right\rceil \qquad M = \left\lfloor \frac{B}{s(next)} \right\rfloor$$

$$depth(a) = \lceil \log_M(N(a)) \rceil$$

$$mu^0(a) = s(array\_header)$$

$$mu(a) = \begin{cases} P(mu^0(a), B) & \text{if } |\mathcal{F}(a)| = 0 \\ P(mu^0(a), B) + \sum_{i=0}^{depth(a)} B \left\lceil \frac{N(a)}{M^i} \right\rceil & \text{if } |f(a)| > 0 \end{cases}$$

5 Evaluation

To evaluate the heap allocation analysis, three benchmarks and two applications are analyzed and the memory consumption is compared to measurements of the five applications on the target JVM.Measurements cannot reliably capture the the worst-case behavior; comparing the analysis results with measurements only hints at bounds that might be overly pessimistic.

In order to compare our work with existing memory allocation analyses, we use the JOlden benchmark suite [6], which was also used in [4,1]. We use the subset of the benchmarks that does not require recursion and hence can be analysed by our tool. The benchmarks were modified such that they do not get their parameters via the command line arguments. We cannot compute a worst-case bound for unknown input values and hence initialize the appropriate variables internally. Where the DFA could not find loop bounds, we provided manual annotations.

The first application we evaluate is based on the demo application presented in [5].[2] It emulates a multi-threaded financial transaction system, which must react to market changes within a bounded amount of time. For the evaluation, we chose the methods MarketManager.onMessage() and OrderManager.checkForTrade(), both of which perform core functionality of the respective thread.

The application was adapted in three ways: First, the execution model of our execution platform is closer to the thread model of safety-critical Java [7], than the thread model of the RTSJ. The thread management therefore had to be reorganized. Second, our platform does not support the libraries for receiving and transmitting messages. This part had to be rewritten such that messages are read from and sent to standard in- and output. Third, we used string buffers without automatic resizing where suitable. The reasoning behind this is discussed in Sec. 5.3.

[2] We thank Eric Bruno and Greg Bollella for open-sourcing this demo application. It is available at `http://www.ericbruno.com`

Table 1. Analysis and measurement results

Benchmark	Method	Allocated Objects		Allocated Words	
		Analysed	Measured	Analysed	Measured
MST	MST.main()	242	221	501	459
Em3d	Em3d.main()	814	805	11627	7298
BH	Tree.createTestData()	464	464	1954	1954
Trading	MarketManager.onMessage()	8	8	4104	2004
Trading	OrderManager.checkForTrade()	35	29	7876	741
CD_x	Main.run()	197936	22907	590150	73841

The second application used for evaluation was extracted from the CD_x benchmark for RTSJ [9]. The source code of the original benchmark is freely available.[3] We analyze the real-time thread responsible for collision detection of airplanes.The benchmark in its original form is unsuitable for WCET analysis, as it makes heavy use of hash tables, which have poor worst-case performance. For heap allocation analysis on the other hand, the benchmark is both challenging and, with a few modifications, within the capabilities of our tool.

We first adopted the benchmark to meet the requirements of our target platform. The recursive voxel intersection procedure, which needs large amounts of stack space, was replaced by an efficient iterative version. The number of planes and other constants were reduced to meet the memory restrictions of our embedded system. For the analysis it was necessary to annotate some loops that use iterator objects, as these are beyond the capabilities of our data flow analysis.

5.1 Results

The analysis results for the six benchmark methods is shown in Tab. 1. In order to keep the pessimism within reasonable bounds, we used a modified version of the Java development kit (JDK) suitable for real-time applications. It requires to pass a maximum capacity for lists and maps in the constructor, and prohibits on-demand reallocations, which cannot be handled by the analysis (see Sec. 5.3 for a discussion of the respective issues). Furthermore, the results for the trading engine application were obtained under the assumption that input messages are at most 1024 characters long. This is enforced by the input routines, but not a constraint that is visible at the application level. For unbounded input messages, the memory consumption of the trading engine would not be boundable.

Table 1 compares the results of the analysis with the results of a measurement; the numbers in the last two columns of this table refer to the raw amount of memory allocated by the application, excluding object meta-data. Results that take into account the overhead of the object layout are discussed in the following section.

[3] http://adam.lille.inria.fr/soleil/rcd/

The figures in Tab. 1 show that the analysis yields relatively tight results for some benchmarks, while introducing a considerable pessimism for others. One reason for this pessimism is the fact that the measurement is not guaranteed to actually trigger the worst case. Some of the pessimism is however introduced by the analysis itself.

The three benchmarks from the JOlden benchmark suite are relatively simple and their execution is independent from input data. The analysis can therefore find reasonably tight bounds. The figures for the object count are tighter than the figures for the memory consumption because array sizes are overestimated at a few occasions. The pessimism for the object count is similar to the pessimism reported in [4] for these benchmarks.

The analysis results for the MarketManager.onMessage() method are off by a factor of around two. The method parses the input string for a name and a price and updates the market price of a traded item accordingly. Although the measurement was performed with a message that was designed to trigger as much memory allocation as possible, the analysis fails to find tight bounds on the string variables and assumes that all strings are 1024 characters long.

OrderManager.checkForTrade() shows considerably more pessimism. This is mainly caused by conversions from numbers to strings. Within these conversions, the analysis is not able to bound the length of the result string and assumes that such strings are 1024 characters in size. It is notable that the number of objects is overestimated by only about 20%, but the number of allocated words by about an order of magnitude.

The heap allocations reported for the collision detector thread of the CD_x benchmark are relatively high. The main reason for the overestimation is that it is difficult to find tight bounds for all collection sizes and loops in the benchmark, and that our tool does not yet support context dependent manual annotations. On the other hand, the benchmark showed that the analysis scales up for larger programs, and helped us to identify many problematic language constructs which complicate the analysis.

5.2 JVM Comparison

To compare the effects of different object layouts, we variated the cost function for the WCHA analysis as described in Sec. 4.2. The results are shown in Tab. 2. We assume 4 words for header data, and a blocks size of 8 words. We also include the number of allocated objects in Tab. 2, as this number is crucial to correctly dimension the handle area for a handle-based object layout.

As it is the case on our evaluation platform, the Java Optimized Processor (JOP) [16], fields are always stored at word boundaries and do not require any further padding. Due to the simple model for alignment requirements, a handle-based layout and a header-based layout consume the same amount of memory. The heap has to be large enough to fit the number of words given in the "Handles/Headers" column. Similarly, the total memory consumption of a fixed-block layout is provided in the "Blocks" column.

Table 2. Analysis results for different object layouts

Benchmark	Method	Objects	Allocated Words		
			Raw	Handles/Headers	Blocks
MST	MST.main()	242	501	1469	2040
Em3d	Em3d.main()	814	11627	14883	22776
BH	Tree.createTestData()	464	1954	3810	5768
Trading	MarketManager.onMessage()	8	4104	4136	4768
Trading	OrderManager.checkForTrade()	35	7876	8016	9528
CD_x	Main.run()	197936	590150	1381894	1915336

For the handle-based layout, not only the total amount of memory must fit the heap, but also the handle area has to be dimensioned correctly. The number in the "Object Count" column times the handle size has to fit the handle area, and the rest of the heap has to be large enough to fit the number of words given in the "Raw" column.

When relating the total amount of allocated memory to the number of allocated objects, the trading engine benchmarks differ considerably from the other benchmarks. While the former allocate a few relatively large objects (mostly arrays), the latter allocate many small objects. The overhead for the header data is therefore considerably higher for these benchmarks than for the trading engine benchmarks.

Using a fixed-block layout increases the memory consumption by 15 to around 50%, when comparing it to a simple header layout. A GC that uses such a layout must be considerably more efficient in other areas to make up for this increased memory consumption.

5.3 Programming Style

During the evaluation of the analysis, we encountered several times that relatively simple operations resulted in seemingly excessive memory allocations. A closer look at these operations revealed that this was due to the automatic resizing of data structures. Except for a few special cases, the analysis assumed that such a resizing would always occur. For example, when appending characters to a StringBuffer, the analysis assumed that the array to hold the actual characters would be resized for each invocation of append(). Converting a float to a String was reported to allocate several megabytes of memory, instead of a just few dozen words. Similar effects were observed for other common data structures of the Java library, such as ArrayLists.

Such situations can be circumvented in two ways. On the one hand, some data structures exhibit more analysis-friendly behavior than others. For example, adding an element to a LinkedList requires only the allocation of a single list element. The drawback of this solution is that such data structures do not always have the desired performance characteristics. On the other hand, it is

sometimes possible to size the data structure upon allocation such that no resizing is necessary. However, this solution requires programmers to correctly predict the sizes of data structures. We believe that further research is necessary to find a suitable tradeoff between analyzable memory allocation and ease of use for the respective data structures.

6 Conclusion

Bounds for the WCHAs of real-time tasks are needed to correctly size scoped memories or to calculate the maximum period of the GC task. We have adapted technologies from the WCET analysis field to analyze the heap allocations of tasks. Instructions that allocate memory get a cost equivalent to the the size of the allocated data structure. All other instructions have zero cost. Analyzing the program with those costs gives the maximum memory allocation for a task instead of its maximum execution time.

We have shown that our analysis can find reasonably tight bounds for moderately complex programs. However, more realistic Java programs that are not explicitly designed for real-time systems are hard to analyze and result in considerable pessimism for the WCHA bounds. As future work we plan to investigate the right Java based programming style for real-time applications. Furthermore, we will investigate better analyzable replacements for library elements of the JDK.

References

1. Albert, E., Genaim, S., Gómez-Zamalloa Gil, M.: Live heap space analysis for languages with garbage collection. In: ISMM 2009: Proceedings of the 2009 international symposium on Memory management, pp. 129–138. ACM, New York (2009)
2. Bacon, D.F., Sweeney, P.F.: Fast static analysis of C++ virtual function calls. In: OOPSLA 1996: Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pp. 324–341. ACM, New York (1996)
3. Bollella, G., Gosling, J., Brosgol, B., Dibble, P., Furr, S., Turnbull, M.: The Real-Time Specification for Java. Java Series. Addison-Wesley, Reading (2000)
4. Braberman, V., Fernández, F., Garbervetsky, D., Yovine, S.: Parametric prediction of heap memory requirements. In: ISMM 2008: Proceedings of the 7th international symposium on Memory management, pp. 141–150. ACM, New York (2008)
5. Bruno, E.J., Bollella, G.: Real-Time Java Programming: With Java RTS. Prentice Hall PTR, Upper Saddle River (2009)
6. Cahoon, B., McKinley, K.S.: Data flow analysis for software prefetching linked data structures in java. In: PACT 2001: Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques. pp. 280–291. IEEE Computer Society, Washington (2001)
7. Henties, T., Hunt, J.J., Locke, D., Nilsen, K., Schoeberl, M., Vitek, J.: Java for safety-critical applications. In: 2nd International Workshop on the Certification of Safety-Critical Software Controlled Systems (SafeCert 2009). York, United Kingdom (March 2009)

8. Hofmann, M., Jost, S.: Type-based amortised heap-space analysis. In: Sestoft, P. (ed.) ESOP 2006. LNCS, vol. 3924, pp. 22–37. Springer, Heidelberg (2006)
9. Kalibera, T., Hagelberg, J., Pizlo, F., Plsek, A., Titzer, B., Vitek, J.: Cdx: a family of real-time java benchmarks. In: JTRES 2009: Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems, pp. 41–50. ACM, New York (2009)
10. Li, Y.T.S., Malik, S.: Performance analysis of embedded software using implicit path enumeration. In: LCTES 1995: Proceedings of the ACM SIGPLAN 1995 workshop on languages, compilers, & tools for real-time systems, pp. 88–98. ACM Press, New York (1995)
11. Mann, T., Deters, M., LeGrand, R., Cytron, R.K.: Static determination of allocation rates to support real-time garbage collection. SIGPLAN Not. 40(7), 193–202 (2005)
12. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer, New York (1999)
13. Puschner, P., Schedl, A.: Computing maximum task execution times – a graph-based approach. Journal of Real-Time Systems 13(1), 67–91 (1997)
14. Robertz, S.G., Henriksson, R.: Time-triggered garbage collection: robust and adaptive real-time GC scheduling for embedded systems. In: LCTES 2003: Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems, pp. 93–102. ACM Press, New York (2003)
15. Schoeberl, M.: Real-time garbage collection for Java. In: Proceedings of the 9th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC 2006), pp. 424–432. IEEE, Gyeongju (April 2006)
16. Schoeberl, M.: A Java processor architecture for embedded real-time systems. Journal of Systems Architecture 54(1–2), 265–286 (2008)
17. Schoeberl, M., Puffitsch, W., Pedersen, R.U., Huber, B.: Worst-case execution time analysis for a Java processor. Software: Practice and Experience 40(6), 507–542 (2010)
18. Shivers, O.: The semantics of scheme control-flow analysis. In: PEPM 1991: Proceedings of the 1991 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation, pp. 190–198. ACM, New York (1991)
19. Siebert, F.: Hard Realtime Garbage Collection in Modern Object Oriented Programming Languages. aicas Books (2002) ISBN: 3-8311-3893-1
20. Unnikrishnan, L., Stoller, S.D., Liu, Y.A.: Automatic accurate stack space and heap space analysis for high-level languages. Tech. Rep. 538, Indiana University (April 2000)
21. Wegbreit, B.: Mechanical program analysis. Commun. ACM 18(9), 528–539 (1975)

Partial Flow Analysis with oRange

Marianne de Michiel, Armelle Bonenfant, Clément Ballabriga, and Hugues Cassé

Université de Toulouse, Institut de Recherche en Informatique de Toulouse (IRIT)
{michiel,bonenfant,ballabri,casse}@irit.fr

Abstract. In order to ensure that timing constrains are met for a Real-Time Systems, a bound of the Worst-Case Execution Time (WCET) of each part of the system must be known. Current WCET computation methods are applied on whole programs which means that all the source code should be available. However, more and more, embedded software uses COTS (Components ...), often afforded only as a binary code. Partialisation is a way to solve this problem.

In general, static WCET analysis uses upper bound on the number of loop iterations. oRange is our method and its associated tool which provide mainly loop bound values or equations and other flow facts information. In this article, we present how we can do partial flow analysis with oRange in order to obtain component partial results. These partial results can be used, in order to compute the flow analysis in the context of a full application. Additionally, we show that the partial analysis enables us to reduce the analysis time while introducing very little pessimism.

1 Introduction

Critical hard real-time systems are composed of tasks which must imperatively finish before their deadline. Static WCET analysis is performed by a timing analysis tool which needs loops upper bounds. Such bounds may be given by manual annotations of programs or automatic evaluation. All feasible paths through the program have to be studied in order to extract some flow information which is used to statically bound the number of times loops are iterated.

Several approaches have been proposed in Flow Analysis about loop bounds [2,3,7,8,9,10,11,13].

The current WCET computation methods are designed to be used on a whole program. However, there are some drawbacks to this approach. First, the analyses used for WCET computation usually run in exponential time with respect to the program size. Second, when the program to analyze depends on external components (e.g. Components Off The Shelf (COTS) or libraries) whose sources are not available, the lack of information about the components prevents the WCET computation if information from the user is requested (and not provided). Partial analysis becomes then crucial to improve the time needed to compute WCET and more important the computation of the WCET itself.

T. Margaria and B. Steffen (Eds.): ISoLA 2010, Part II, LNCS 6416, pp. 479–482, 2010.

This article presents how partial analysis is possible with oRange, our tool which calculates upper bounds of loops in C programs. In section 2, we present the partial analysis . Section 3 compares classical and partial analysis provided by oRange. Finally, Section 4 gives our conclusions.

2 Partial Analysis

oRange[12] combines a) loop bound expression building of C program as in [7] with b) abstract interpretation as in many previous works [9,10,11,13]. It is used by OTAWA[1] [1] in order to obtain loop bounds expressions *total* and *maxi*. For a loop L_i, $total_i$ represents the total number of iterations in the overall execution of the program and $maxi_i$ represents the maximum number of iterations for each loop startup.

2.1 Description

A partial result: When partialising a function, we build what we call a *partial result*. It is a pair made of a tree representing loop and function calls included in the partialised function and an abstract store[2] which contains the assignments of the non local variables (global, static variables and parameters being modified by the function). The tree is a parametric flow fact, it is instantiated with the call context in the full analysis. The abstract store is used to evaluate the impact of the function on the rest of the application. These two results are called summaries of the function.

Usage of a partial result: A partial result can be used to build either a partial result of another function, or a complete analysis. The pair of the partial result is combined with the caller context: the abstract store representing the caller context is instantiated with the tree and composed with the abstract store of the partial result.

Partial analysis and pessimism: The number of times we cannot determine if a branch is executed or not is higher in partial analysis because we know less of the context. Thus, we have to take into account the two possible branches and usually add indeterministic results. This leads to pessimism, but is only effective if the loop iteration number depends on at least a variable in an alternative branch.

2.2 Automatization

It is possible to choose manually functions to be partialised. We can also do an automatic partialisation. There are two options: a pessimistic option which

[1] oRange is integrated in the OTAWA tool chain dedicated to WCET computation.
[2] A set which maps each variable of the execution state to an expression which can depend on the input of the instruction (i.e. preceding instructions).

improves time computation by choosing to partialise large functions. The second option partialises only non-pessimistic functions. In order to automatize, we first select functions regarding the options. Potentially partialised functions are then classified according to their nesting level.

We define an internal weight which depends on the number of internal function calls, the weight of the function called, the number of assignments in loops, the assignments of in and out parameters. We then obtain a total weight depending on the imbrication level, the frequency of call of the function, the number of nested loops in the function... During partialisation, the total weight of a function can be re-evaluated.

Depending on its internal, total weight and the option chosen, functions are potentially partialisable.

Levels are determined by the imbrication level of inside function calls where functions are either partialisable themselves or not.

3 Results

In table 1 we present a comparison between classical full and partial analysis on the debie application (WCET'07 challenge[5]) and its functions/sub functions. Experiment has been done on a Core2 Duo Processor T7200 2GHz.

Table 1. Classical versus Partial Analysis

			Classical			Partial		
Program	L_C	L_N	% of B	time	C	% of B	time	partialisation
class	2	2	100%	15.885s	6	100%	14.529s	none
hw_if	3	3	66%	16.037s	16	66%	14.481s	none
measure	12	6	25%	26.394s	124	25%	25.062s	none
tc_hand	9	4	78%	6m42.573s	127	78%	1mn24.861s	2(2 levels)
harness	2379	43		>11h	50171	87%	46mn50.068s	55 (8 levels)
telem	4	4	50%	15.981s	9	50%	14.677s	none
health	85	11	86%	92m12.706s	1344	86%	28.014s	11(8 levels)
debie	2390	1		>11h	50269	88%	47mn3.728s	56 (11 levels)

L_C: number of loop calls from any file
L_N: number of loops directly into the file
B: number of loops bounded
$\%B$: % of B according to L_C
C: number of function calls

Thanks to partialisation, there is a real gain of time computation. Fortunately, the number of loops bounded is identical or greater.. In term of pessimism, tc_hand and health could have obtain less accurate loop bounds because of partialisation, but in these cases loop bounds do not depend on alternative branches for these cases (see 2.1). In most cases, we think partialisation introduces pessimism.

4 Conclusion

oRange, which computes flow facts information (especially loop bounds), can be used to obtain partial results of functions and can use them. The main achievement of our approach is that partial evaluation may be processed for a function independently of any call, partial results are then combined with each call context in a complete evaluation. Experimental results show that the computing time to get the flow facts can be greatly improved. oRange automatic analysis supports options allowing pessimism reduction.

One of the main usage of partialisation is to obtain COTS summaries and to be able to use them without analyzing the entire library as we have done in [6]. This study has been partially funded by the European Community under the Merasa [4] project.

References

1. Otawa, http://www.otawa.fr
2. Bound-t tool (2005), http://www.tidorum.fi/bound-t/
3. ait tool (2007), http://www.absint.com
4. Merasa (2007), http://ginkgo.informatik.uni-augsburg.de/merasa-web/
5. Wcet project (2007), http://www.mrtc.mdh.se/projects/wcet/
6. Ballabriga, C., Cassé, H., De Michiel, M.: A generic framework for blackbox components in wcet computation. In: 9th Intl. Workshop on Worst-Case Execution Time Analysis, WCET 2009, Dublin, Ireland (2009)
7. Coffman, J., Healy, C.A., Mueller, F., Whalley, D.B.: Generalizing parametric timing analysis. In: Pande, S., Li, Z. (eds.) LCTES, pp. 152–154. ACM, New York (2007)
8. Cullmann, C., Martin, F.: Data-flow based detection of loop bounds. In: 7th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis, Pisa, Italy (2007)
9. Ermedahl, A., Sandberg, C., Gustafsson, J., Bygde, S., Lisper, B.: Loop bound analysis based on a combination of program slicing, abstract interpretation, and invariant analysis. In: 7th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis, Pisa, Italy (2007)
10. Kirner, M.: Automatic loop bound analysis of programs written in c. Master's thesis, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 3/3/182-1, 1040 Vienna, Austria (2006)
11. Lokuciejewski, P., Cordes, D., Falk, H., Marwedel, P.: A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models. In: Cgo 2009: Proceedings of the 7th International Symposium on Code Generation and Optimization, Washington, DC, USA (2009)
12. De Michiel, M., Bonenfant, A., Cassé, H., Sainrat, P.: Static loop bound analysis of c programs based on flow analysis and abstract interpretation. In: RTCSA, pp. 161–166. IEEE Computer Society, Los Alamitos (2008)
13. Prantl, A., Knoop, J., Kirner, R., Kadlec, A., Schordan, M.: From trusted annotations to verified knowledge. In: Holsti, N. (ed.) WCET. Dagstuhl Seminar Proceedings, vol. 09004, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany (2009)

Towards an Evaluation Infrastructure for Automotive Multicore Real-Time Operating Systems

Jörn Schneider and Christian Eltges

Dept. of Computer Science
Trier University of Applied Sciences
Trier, Germany
{j.schneider,c.eltges}@fh-trier.de

Abstract. The automotive industry is on the road to multicore and already included supporting features in their AUTOSAR standard, yet they could not decide for a multicore resource locking protocol. It is crucial for the future acceptance and possibilities of multicore systems to allow for informed decisions on this topic, as it immediately impacts the inter-core communication performance and thereby the value-cost ratio of such systems. We present the design of a real-time operating system simulator that allows to evaluate the different multicore synchronisation mechanisms of the real-time research community regarding their fitness for automotive hard real-time applications. You can reuse the key design idea of this simulator for any simulation based tool for the early timing evaluation of different real-time mechanisms, e. g. scheduling algorithms.

1 Problem

A challenging application area for multicore systems are hard real-time systems in cars, e. g. electronic control units for airbags, electronic stability control, and driver assistance systems. The automotive industry recently released the specification of a first multiprocessor real-time operating system (RTOS) as part of the AUTOSAR standard [1]. The concept specifies a partitioned system with tasks and interrupt service routines statically mapped to cores. Each core runs a fixed priority scheduler for its particular task set and tasks can be activated across cores. Transferring data is based on sharing memory between cores.

But how can this be done without deadlocks, priority inversion and unbounded remote blocking? Traditionally, resource locking protocols are used in real-time systems to achieve this. In the automotive domain the immediate ceiling priority protocol (referred to as OSEK Ceiling Priority Protocol) is readily available in any OSEK or AUTOSAR compliant RTOS. Yet, this works for uniprocessor systems only.

The responsible AUTOSAR subcommittee, which was then headed by the first author of this paper, initially planned to introduce a multicore resource locking protocol for this release of the operating system specification. However, it turned

T. Margaria and B. Steffen (Eds.): ISoLA 2010, Part II, LNCS 6416, pp. 483–486, 2010.

out that this goal was too ambitious given the short deadline for AUTOSAR release 4.0. The requirements document [2] still reflects the ambitious goal, yet the specification itself clearly changed in this regard after the responsible person for the topic changed. AUTOSAR release 4.0 lacks a multiprocessor synchronization mechanism that is suitable for hard real-time systems — only spin locks are supported.

One reason for this shortcoming is that it is unclear which resource locking approach is most suited to fulfil the requirements of the industrial practice. Clearly, the performance impact plays a central role here. Investigating this impact requires to look at two aspects. First, the blocking behaviour and second the implementation overhead. The former one depends on the resource locking characteristics of the application in combination with the particular resource locking protocol. The quantity of the latter additionally depends on the chosen realisation of the resource locking protocol within the RTOS. Naturally, the timing aspects of the chosen protocol might seriously impact the communication efficiency. Because the main value of multicore systems for the automotive domain lies in its better cost-performance ratio, the timing behavior of the cross-core communication is crucial for the adoption of the new hardware concepts.

The automotive industry cannot afford to go through the painful process of implementing different resource locking protocols, applying them in various products, and making enough experience to differentiate good from bad approaches. We designed a dedicated simulator to address this problem.

2 The Proposed Simulator

The key design idea that distinguishes our simulator from similar approaches is to consequently separate the two concerns *simulated functionality* and *simulated time*. In other words, if you specify a new mechanism to be simulated, e. g. a resource locking concept, you have to specify the algorithm and its temporal behaviour separately. You might think that this is an unnecessary and cumbersome complication. But it even reduces your workload, if you use the simulator for the intended purpose.

When considering the basic functionality of resource locking protocols (or real-time operating systems as a whole) it becomes quite evident that there are a lot of fundamental operations that need to be performed regardless of the specific version. For instance, when deciding which task is to run next, it is in most cases necessary to pick the task with the highest priority. Various implementations of the operation *identify highest priority task* are possible, yet the result is always the same. However, the timing might be completely different. When you use our simulator you can simply take a provided library function *getHighestPriorityJob* and specify the timing behaviour of the particular implementation you have in mind without even implementing it (see Listing 1.3 and 1.4 below for examples). To compare two RTOS A and B with compatible APIs but different implementations you just bind the corresponding timing functions to the API functions for each simulation run.

We believe that the idea to separate *simulated functionality* and *simulated timing* can be widely reused in simulation based tools for the early timing evaluation of different real-time mechanisms, e. g. scheduling algorithms. As we conjecture this makes such tools much more usable for different research groups to compare their approaches on equal terms.

Our simulator allows to define tasks in an abstract language (instead of C code). The only things to be specified in this language for a given task are the scheduling relevant calls to API-functions and sections of the task that, from a scheduling point of view, just consume time. An example that shows this idea can be seen in Listing 1.1. It defines a task that executes some calculations that do not influence the scheduling (denoted by the *time x;* expressions) and locks a resource RES_1 for 5000 processor cycles.

Listing 1.1. Definition of a task

```
Task t1 = do {
    time 10000;
    GetResource RES_1;
    time 5000;
    ReleaseResource RES_1;
    time 3000;
    TerminateTask;
}
```

Internals of API-calls can be specified with the help of *basic functions*. These are the minimal building blocks of the simulator language. Listing 1.2 gives an example API-call implementation using basic functions, like *getHighestPriorityJob* and *setState*.

Listing 1.2. Definition of an API-call

```
TerminateTask = do {
    setState currentJob SUSPENDED;
    j <- getHighestPriorityJob;
    dispatch j;
}
```

To compare different protocols, the execution times of the basic functions are specified via timing functions for each protocol. When the simulator executes a basic function, it calculates the number of cycles via the corresponding timing function.

This makes it possible to have different timing functions that simulate different implementations, without reimplementing the basic functions. For example, consider the *getHighestPriorityJob* function. The priority queue could be implemented as an unsorted list or as a sorted list. In the first case, getting the highest priority job would take O(n) steps, in the second case it would take O(1) steps. Examples are given in Listing 1.3 and Listing 1.4.

Listing 1.3. Timing function for linear runtime

```
linearTimeOfGetHighestPriorityJob
    return 10 + 5 * length(readyQueue)
```

Listing 1.4. Timing function for constant runtime

```
constTimeOfGetHighestPriorityJob
    return 10
```

Note that timing functions can use the complete state of the simulated code to derive the proper execution time for each calling context. This feature is used in Listing 1.3 to consider the current length of the ready queue.

3 Related Work

A multitude of simulators for different purposes were implemented by research groups or companies. Naturally we did not investigate all of them and the ones we investigated were usually different in many important aspects, like timing granularity and so on. None of them seems to follow our concept of consequently separating the two concerns *simulated timing* and *simulated functionality*. At least one simulator nevertheless should be mentioned here. RTSSim [3] is closely comparable with our work, it has an even broader scope regarding its intended usage. The key differences are that we use an abstract language instead of C, that we support multicore, and that our approach consequently separates the two concerns *timing* and *functionality*.

4 Conclusion

We presented the design of a novel simulator for the specific purpose of evaluating multicore resource locking protocols for their fitness to be used by industry in automotive electronic control units. The key idea to separate functional realisation from simulation time is one that, as we believe, is well suited to be used in many simulation approaches in the field of timing analysis. We hope that the final simulator as well as the simple idea of separating function and timing will be (re-)used in the research community.

References

1. AUTOSAR release 4.0 — Specification of Multi-Core OS Architecture (December 2009), http://www.autosar.org/download/R4.0/AUTOSAR_SWS_MultiCoreOS.pdf
2. AUTOSAR release 4.0 — Requirements on Multi-Core OS Architecture (November 2009), http://www.autosar.org/download/R4.0/AUTOSAR_SRS_MultiCoreOS.pdf
3. Kraft, J.: RTSSim - a simulation framework for complex embedded systems. Technical Report, Mälardalen University (March 2009), http://www.mrtc.mdh.se/publications/1629.pdf

Context-Sensitivity in IPET for Measurement-Based Timing Analysis*

Michael Zolda[1], Sven Bünte[1], and Raimund Kirner[2]

[1] Institute of Computer Engineering
Vienna University of Technology, Austria
{michaelz,sven}@vmars.tuwien.ac.at
[2] Department of Computer Science
University of Hertfordshire, Hatfield, United Kingdom
r.kirner@herts.ac.uk

Abstract. The *Implicit Path Enumeration Technique* (IPET) has become widely accepted as a powerful technique to compute upper bounds on the Worst-Case Execution Time (WCET) of time-critical software components. While the technique works fine whenever fixed execution times can be assumed for the atomic program parts, standard IPET does not consider the context-dependence of execution times. As a result, the obtained WCET bounds can often be overly pessimistic.

The issue of context-dependence has previously been addressed in the field of static timing analysis, where context-dependent execution times of program parts can be extracted from a hardware model. In the case of measurement-based execution time analysis, however, contexts must be derived from timed execution traces.

In the present extended abstract we present an overview of our work on the automatic detection and exploitation of context dependencies from timed execution traces.

1 Introduction

The well-known IPET approach [3,2] provides a scheme for formulating the problem of determining a WCET estimate of a software component as an integer linear programming (ILP) [1] problem. Working on the level of the *control flow graph* (CFG), the method introduces variables for the execution count of each *block*, as well as for each control flow edge between blocks. These variables are subject to linear constraints that can exclude some (but not all) infeasible control flow through the CFG. Assuming the availability of a fixed local upper WCET bound for each individual block, the determination of an upper bound of the

* The research leading to these results has received funding from the IST FP-7 research project "Asynchronous and Dynamic Virtualization through performance ANalysis to support Concurrency Engineering (ADVANCE)" and the Austrian Science Fund (Fonds zur Förderung der wissenschaftlichen Forschung) within the research project "Formal Timing Analysis Suite of Real-Time Systems" (FORTAS-RT) under contract P19230-N13.

T. Margaria and B. Steffen (Eds.): ISoLA 2010, Part II, LNCS 6416, pp. 487–490, 2010.

global WCET is reduced to the problem of maximizing the cost-weighted sum of execution counts over all blocks.

When we want to use IPET in practice, two important problems emerge. Both of them can be traced back to the fundamental assumption of a fixed local upper WCET bound for each individual block. They are:

1. How to determine the required local WCET bounds on a real processor that contains highly unpredictable hardware components, like caches, pipelines, branch predictors, out-of-order execution, etc.?
2. How to overcome the high pessimism introduced by using a single local WCET bound for each block, even for those blocks that show a broad spectrum of different execution times, the maximum of which possibly occurring only in very special situations?

As the creation of accurate, precise, and effective formal analyses of the behavior of modern microprocessors has become a highly complex and time-consuming task, *measurement-based timing analysis* (MBTA) has been proposed as a quick, effective, and easily deployable complementary approach.

MBTA allows for the derivation of empirical local WCET estimates from an elaborate choice of representative execution traces. Once such local WCET estimates have been determined for each block, they can be directly used as block costs in an IPET problem.

2 Context-Dependent Execution Times

We are currently working on a method for extending IPET to distinguish between different block execution times, based on empirically determined correlations with the blocks execution history and future.

Figure 1 illustrates the settings of our approach from a bird's eye view.

Our rationale is that the execution time of a block can show both, backward and forward dependencies, with respect to the execution traces.

Backward dependency is the more prominent case, where the execution time of a block depends on the concrete execution history. This is easily exemplified by the distinction of execution times of a block in the presence of a cold vs. a warm instruction cache: In a simple setting, a certain block might be absent from the instruction cache during the first iteration of a loop, but present during all subsequent iterations. A distinction of execution times of the block can then be based on whether the loop body is entered via the back edge, or from outside.

Our archetype for a forward dependency concerns the execution time of conditional jumps: The execution time of such a jump can depend on whether the jump condition is true or false. Because the jump condition controls the subsequent flow of control, we observe an apparent dependency of a block's execution time on its execution future (a circumstance that might seem counterintuitive at first thought).

It is infeasible and impractical to consider all possible dependencies of the execution time of a block on its concrete execution traces. Also, we have to

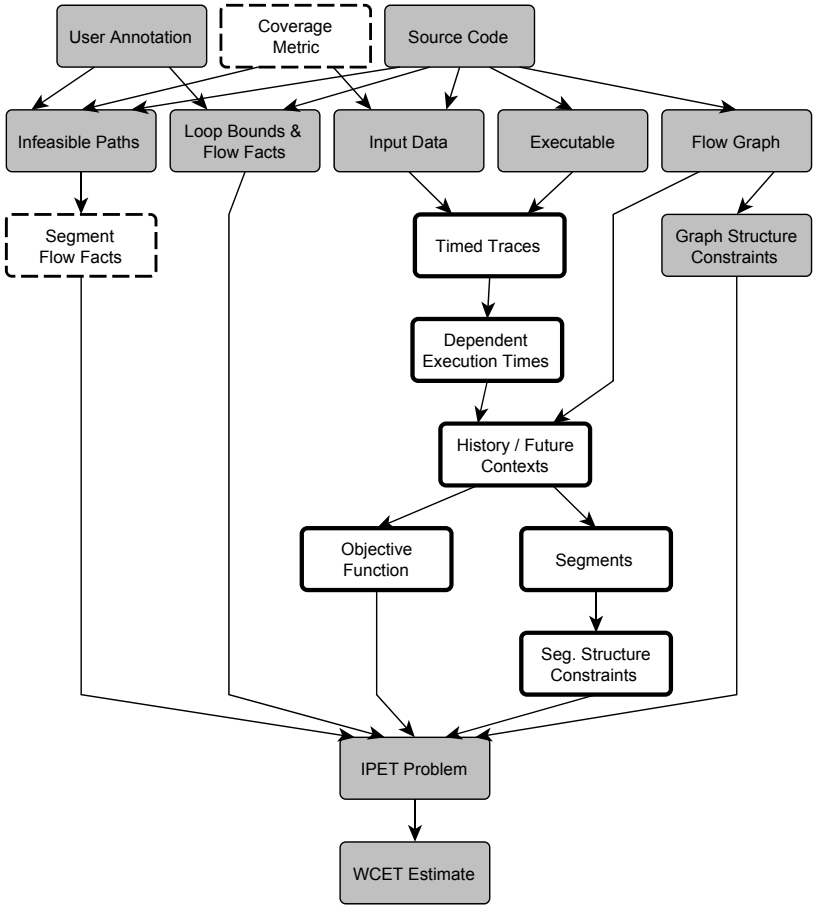


Fig. 1. Pieces of information used in the generation of a context-sensitive IPET problem. The highlighted pieces of information in the center represent our present state of research: *Timed traces* are obtained from runs of the software executable on the target hardware and *dependent block execution times* are extracted. Considering the possible flows in the software *flow graph*, suitable *history / future contexts* are derived that separate the different execution times of each block. These contexts can be easily mapped to graph *segments* (structural clusters of paths). It is then easy to derive *segment structure constraints* that model the control flow through each segment. Also, a new *objective function* is derived that consists of the cost-weighted segment execution frequencies. Adding the usual *graph structure constraints*, which can be derived automatically from the flow graph, the necessary *loop bounds*, and possibly additional *flow facts*, the context-sensitive *IPET problem* is derived.

consider a method to integrate such a distinction into IPET. In our approach, we therefore consider a subset of dependencies that we consider particularly interesting and apt to allow a reduction of the pessimism introduced by IPET.

3 Evaluation

To assess our approach, we used the following experimental setup: We analyzed 1000 traces obtained from running a slightly modified version of the *bsort100* benchmark from the Mälardalen WCET suite. The modifications of the benchmark consisted of reduction of the input array to 15 elements and code reformating for technical reasons. The program was compiled using *GCC* for the *TriCore 1796* processor without optimization. Our second benchmark was a core routine of an elevator control application. To generate the input data we used a mixed approach of systematic block coverage via model checking and a pseudo random data generation. The traces were recorded using a *Lauterbach Power Trace* device.

To estimate the overestimation introduced by IPET, we used the difference between the IPET result and the longest observed end-to-end execution time over all generated traces. Comparing the results of our context-sensitive approach with those of standard IPET, the experiments showed a marginal improvement for the (tiny) *bsort100* benchmark. For the second benchmark, the overestimation was reduced by 8%.

4 Conclusion and Outlook

We have presented our approach towards using context-dependent execution time measurements to reduce the pessimism in IPET, introducing history / future sensitivity. We have also presented first results that show that our approach can in fact help to reduce IPET pessimism in a experimental setting. The details of the approach shall be presented in a full paper.

To improve the effectiveness of our approach, we are currently working on the following two aspects[1]:

Firstly, the separation of contexts by history / future relies on the availability of a suitable set of timed execution traces. To this end, we are currently working on suitable coverage metrics and input-data generation methods.

Secondly, to exploit the full potential of context separation, it will be necessary to derive additional flow facts that restrict the possible combinations of contexts. With respect to this aspect, we are currently pursuing a method to extract such constraints from control flow paths that are known to be infeasible.

References

1. Chvátal, V.: Linear programming. W.H. Freeman, New York (1983)
2. Li, Y.T.S., Malik, S.: Performance Analysis of Embedded Software Using Implicit Path Enumeration. In: DAC 1995: Proceedings of the 32nd annual ACM/IEEE Design Automation Conference, pp. 456–461. ACM, New York (1995)
3. Puschner, P.P., Schedl, A.V.: Computing maximum task execution times - a graph-based approach. Real-Time Systems 13(1), 67–91 (1997)

[1] In Figure 1, the corresponding pieces of information are marked by a dashed borders.

On the Role of Non-functional Properties in Compiler Verification

Jens Knoop[1] and Wolf Zimmermann[2]

[1] TU Wien, Institut für Computersprachen,
A-1040 Wien, Austria
knoop@complang.tuwien.ac.at
[2] Martin-Luther Universität Halle-Wittenberg, Institut für Informatik,
D-06099 Halle/Saale, Germany
zimmer@informatik.uni-halle.de

Abstract. Works on compiler verification rarely consider non-functional properties such as time and memory consumption. This article shows that there are situations where the functional correctness of a compiler depends on non-functional properties; non-functional properties that are imposed by the target architecture, not the application. We demonstrate that this demands for an extended new notion of compilation correctness.

1 Motivation

Functional compilation correctness is informally most commonly considered semantics preservation between source and target program up to deviations due to resource limitations of actual hardware. This notion of correctness has been formalized and made precise in projects on compiler verification such as ProCoS [4] and Verifix [3]. Along these formalizations, compilation correctness essentially boils down towards an appropriate simulation relation between the (binary) target program and the (high-level language) source program up to possible resource limit violations. While intuitive, this notion of compilation correctness does not consider compiler-influenced non-functional properties such as performance and resource utilization, especially time and memory consumption. In fact, the classical notion of compiler correctness is blind wrt non-functional properties.

In this article, we argue that this classical notion of compiler correctness is often too close. We demonstrate this with a practically relevant example: The compilation for *programmable logic controllers* (*PLC*). PLCs follow the control-loop paradigm, i.e., PLCs execute programs iteratively within cycles of a predefined fixed time length. Program portions whose execution has not not been completed at the end of a cycle are skipped. This introduces timing constraints that are not imposed by the application but by the PLC hardware, which makes the straightforward establishing of a simulation relation as required by the classical notion of compilation correctness insufficient. Functional compiler correctness depends in this scenario on adherence to non-functional properties imposed by the target architecture making them first-class citizens for compiler correctness. In this

T. Margaria and B. Steffen (Eds.): ISoLA 2010, Part II, LNCS 6416, pp. 491–495, 2010.

article, we demonstrate this in detail. It shows the need for an enhanced and extended notion of compiler correctness that in addition to functional properties also takes non-functional properties into account.

2 Classical Compiler Correctness

The notion of compiler or translation correctness is often defined as refinement of programming language constructs. In particular each state transition defined by a source language concept (e.g. a conditional statement) must be implemented in exactly the same way by the target machine. Although this is not important for the purpose of this article, it is worth noting that this may forbid some global or interprocedural optimizations. The detailed definition of the notion of compiler correctness took in *Verifix* longer than expected as extensively discussed in [3]. Nowadays, the notion of compiler correctness is based on operational semantics of the programming language and the target languages, respectively.

For this article, we assume that an operational semantics is given by a state transition system. A *state transition system* is a triple $(Q, I, \rightarrow)$ where Q is the (possibly infinite) set of states, $I \subseteq Q$ is the set of initial states and $\rightarrow \subseteq Q \times Q$ is the state transition relation. An *operational semantics* of a programming language L assigns to each program $\pi \in L$ a state transition system $[\![\pi]\!]$. An execution of π is a finite or infinite sequence of states $\langle q_i | i \in \{j | 0 \leq j < k\}\rangle$, $k \in \mathbb{N} \cup \{\infty\}$ such that $q_{i-1} \rightarrow q_i$ for $1 \leq i < k$ and $q_0 \in I$.

From the viewpoint of a compiler user, only the input/output relation of the program is of interest. Each program has such an interaction with an environment which we call *observable behaviour*. The *observable behaviour* of a program consists only of the observable states and state transitions between them induced by the more fine semantics. Compiler users usually only require that the target program preserves the observable behaviour of the source program.

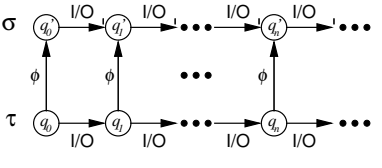


Fig. 1. Preservation of Observable Behaviour

Since usually machine resources are limited while it is easy to write e.g. Java programs that would consume more than 10TByte memory, the target programs may exceptionally stop because of memory overflow. In *Verifix*, we therefore came up with the following notion of correctness: Let τ be a program of the target language with the observable behaviour $(I, Q, \rightarrow)$ and σ be a program of the source language with observable behaviour $(I', Q', \rightarrow')$. *τ preserves the observable behaviour* of *σ up to resource limitations* iff there is a relation $\phi \subseteq Q \times Q'$ such that for any finite or infinite sequence $q_0 \rightarrow q_1 \rightarrow \cdots$ of τ with $q_0 \in I, q_1, q_2, \ldots \in Q$ there is a finite or infinite sequence of states $q_0' \rightarrow q_1' \rightarrow \cdots$ of σ with $q_0' \in I'$ and $q_i \phi q_i'$ for all i except possibly for the last state (if the sequence of observable states of τ is finite). This means that τ halts with violation of resource limitations (cf. Fig. 1). The preservation of observable behaviour up to

resource limitations is transitive and therefore can be applied stepwise for the different phases in a compiler.

Next we apply this classical notion of compiler correctness to PLCs and discuss immediate consequences. Surprisingly, the traditional way to look at embedded systems does not work, i.e., to first ensure functional correctness and to then consider non-functional constraints such as timing conditions. For PLCs it turns out that compiler correctness must include timing constraints from the very beginning as they are imposed by the hardware, not the application.

3 Compiler Correctness for PLCs

Programmable Logic Controllers (abbr. PLCs) are the dominating devices in today's industrial automation systems. The programming of PLCs follows the International Standard IEC 61131-3 that specifies the programming languages. They are rather popular among engineers because of their predictable timing behaviour. Fig. 2 shows the behaviour of a PLC. A PLC-program is iterated infinitely often and implements the control-loop paradigm. At the beginning of each iteration an input phase reads sensor data into the memory of the PLC. Then the PLC-program is executed, and at the end of the iteration an output phase writes some memory cells into actors.

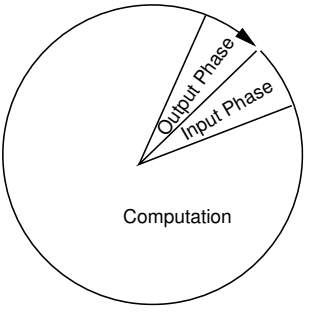


Fig. 2. Execution of programs in PLCs

At first glance, there seems nothing special in proving compiler correctness if the target is a PLC. However, a special feature of PLCs is that the execution of the PLC-program is stopped if a certain time limit (specified by the hardware) has been reached and the next iteration starts again the program.

This behaviour of skipping the execution of program portions on reaching the cycle deadline has implications on the correctness of compilers (as e.g. compilers for *Structured Text* into *Instruction List*).

For compilers for PLCs we thus need an additional requirement for the notion of correctness: Let t be the cycle time, τ be a PLC-program of the target language with the observable behaviour $(I, Q, \rightarrow)$ and σ be a PLC-program of the source language with observable behaviour $(I', Q', \rightarrow')$ such that states contain time and state transitions an increment of time which allows to define the WCET in terms of elapsed time for possible sequences of state transitions. τ *preserves the observable behaviour* of σ *up to resource limitations* iff

i. the worst-case execution time of τ is at most t and

ii. there is a relation $\phi \subseteq Q \times Q'$ such that for any finite or infinite sequence $q_0 \rightarrow q_1 \rightarrow \cdots$ of τ with $q_0 \in I, q_1, q_2, \ldots \in Q$ there is a finite or infinite sequence of states $q'_0 \rightarrow q'_1 \rightarrow \cdots$ of σ with $q'_0 \in I'$ and $q_i \phi q'_i$ for all i except possibly for the last state (if the sequence of observable states of τ is finite).

Hence, the notion of compiler correctness for compilers for PLCs depends on the non-functional property *execution time*. This situation is different to the general situation in embedded systems when general purpose processors are used. The functional correctness of a compiler does not depend on the execution time, only the correctness of the application may depend on execution times. The reason for this difference is that in PLCs the functional behaviour of the target machine depends on the execution time while this is not the case for general purpose processors.

4 Related Work and Conclusions

Correctness of compilers was first considered by McCarthy and Painter [6]. They discussed the compilation of arithmetic expressions. There are a number of works using denotational semantics, e.g. [8,9]. Other works use the approach of refining language constructs, e. g. [4,7,10], or structural operational semantics, e.g. [1]. More recent works [11,5] use a similar notion of correctness as described in this article (although most of them do not consider resource limitations). Glesner et al. considered correct compilers for embedded systems [2]. However, we are not aware of works that consider correct compilation for PLCs as target systems.

In this article we have shown that for PLCs the notion of compiler correctness cannot be based just on a simple simulation relation because the iteration in cycles stops the execution of the program when the cycle time is reached. This shows that non-functional properties can play a surprising and essential role in compiler verification. They are not only needed e.g. to ensure real-time constraints but also for verifying compilers. The classical notion of compiler correctness is insufficient to capture this, and we are not aware of any work that considers non-functional properties quantitatively and integrates them in a verified (optimizing) compiler. In particular, the rapidly developing fields of embedded software demand for such work. Similarities of PLCs and synchronous languages such as Esterel and Lustre need to be further investigated.

References

1. Diehl, S.: Semantics-Directed Generation of Compilers and Abstract Machines. PhD thesis, Universität Saarbrücken (1996)
2. Glesner, S., Geiß, R., Bösler, B.: Verified code generation for embedded systems. In: 1st Workshop on Compiler Optimization meets Compiler Verification COCV 2002. Electronic Notes in Theoretical Computer Science, vol. 65 (2002)
3. Goos, G., Zimmermann, W.: Verification of compilers. In: Olderog, E.-R., Steffen, B. (eds.) Correct System Design. LNCS, vol. 1710, pp. 201–230. Springer, Heidelberg (1999)
4. Hoare, C.A.R., Jifeng, H., Sampaio, A.: Normal Form Approach to Compiler Design. Acta Informatica 30, 701–739 (1993)
5. Leroy, X.: Formal verification of a realistic compiler. Communications of the ACM 52(7) (2009)

6. McCarthy, J., Painter, J.A.: Correctness of a compiler for arithmetical expressions. In: Proceedings of a Symposium in Applied Mathematics, vol. 19. AMS, Providence (1967)
7. Müller-Olm, M.: Modular Compiler Verification: A Refinement-Algebraic Approach Advocating Stepwise Abstraction. LNCS, vol. 1283. Springer, Heidelberg (1997)
8. Palsberg, J.: An automatically generated and provably correct compiler for a subset of Ada. In: IEEE International Conference on Computer Languages (1992)
9. Polak, W.: Compiler Specification and Verification. LNCS, vol. 124. Springer, Heidelberg (1981)
10. Stärk, R., Schmid, J., Börger, E.: Java and the Java Virtual Machine. Springer, Heidelberg (2001)
11. Zuck, L., Pnueli, A., Fang, Y., Goldberg, B.: Voc: A methodology for the translation validation of optimizing compilers. Journal of Universal Computer Science 9(3), 223–247 (2003)

Author Index