S. Ramesh
P. Sampath
Editors

Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems

Proceedings of the GM R&D Workshop, Bangalore, India, January 2007

Springer

Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems

S. Ramesh · P. Sampath
Editors

Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems

Proceedings of the GM R&D Workshop, Bangalore, India, January 2007

Springer

S. Ramesh
GM R&D, India Science Lab
Bangalore, India

P. Sampath
GM R&D, India Science Lab
Bangalore, India

Library of Congress Control Number: 2007934037

ISBN: 978-1-4020-6253-7 e-ISBN: 978-1-4020-6254-4

Printed on acid-free paper.

9 8 7 6 5 4 3 2 1

springer.com

Preface

This volume brings out the proceedings of the workshop "Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems" conducted by General Motors R&D, India Science Lab, Bangalore. This workshop is the first of its kind to be organised by an automotive Original Equipment Manufacturer (OEM) to bring together the experts in the field of embedded systems development to present state-of-the-art work, and to discuss future strategies for addressing the increasing complexity of embedded control systems. The theme of the workshop is an important focus area for the current and future automotive systems.

Embedded Control Systems are growing in complexity with the increased use of electronics and software in high-integrity applications for automotive and aerospace domains. In these domains, they provide for enhanced safety, automation and comfort. Such embedded control systems are distributed, fault-tolerant, real-time systems with hybrid (discrete and continuous) behaviour. Furthermore, many of the control functions, such as by-wire controls, have stringent performance and high-integrity requirements.

The research community has been addressing these challenges, and over the last few years, several design methodologies and tools for developing distributed embedded control systems have emerged. In spite of these, development of embedded control applications remains a daunting task, requiring a great degree of human skill, expertise, time, and effort. It is imperative to invest significant R&D effort in coming up with methods and tools for future embedded control applications.

We believe that future methodologies will involve three key ingredients: comprehensive model-based development, math-based formal frameworks and component-oriented and product-line based development.

Although model-based development has been adopted in system development, the extent of its usage is rather limited to less complex systems and/or restricted to the design phase of the development cycle. We expect model-based methodologies to permeate every aspect of embedded control systems development from requirements to verification.

- The verification of current day systems, though consuming significant time and effort, continues to be manual and mainly focused on run-time checking or testing. A math-based formal framework will enable powerful static analysis and formal verification techniques that exhaustively analyze the model space for high integrity systems.

- To reduce the cost of development of embedded systems, and to improve reliability, current industrial practice mandates that systems should be developed from an assemblage of standard and reusable off-the-shelf components. OEMs need to conceive suitable component-based architectures that enable precise specification of components, their usage policies, and frameworks for composing components.
- Apart from small-grained component usage, a large-grained product-line approach would also prove to be more cost-effective and efficient in the long run. Correct-by-construction approach to design of integrated systems will help in reducing verification time and improving product quality.

The workshop was held during January 5–6 2007 at the NIAS auditorium, IISc campus, Bangalore, India. It consisted of several invited talks given by leading experts and researchers from academic and industrial organizations. The participants included advanced graduate students, post-graduate students, faculty members from universities, and researchers from industry. The participants came from USA, Europe, Asia, and all parts of India – from Mumbai to Guwahati; Chennai to Delhi.

The workshop covered all areas of embedded systems development and in particular:

- Formal specification and verification of distributed, heterogeneous, embedded systems,
- Formal semantics of modeling languages,
- Model-based specification and testing,
- Formal approach to component-based development,
- Software product line engineering, and
- Automatic code generation for distributed, embedded systems.

Bangalore, S. Ramesh
February 2007 P. Sampath

Acknowledgements

The idea for such a workshop was conceived by Dr. B.G. Prakash, the Director, India Science Lab, GM R&D. This workshop would not have been possible but for his constant support and encouragement throughout. He, along with Mr. N.H. Sathyaraja, Lab Group Manager, India Science Lab, gave us a free hand in deciding the programme of the workshop. Mr. Sathyaraja's meticulous planning at appropriate stages resulted in the successful organization of the workshop. We would also like to thank Dr. Alan Taub, Executive Director, GM R&D, and Dr. Patrick Popp, the former director of ECI Lab, GM R&D, Warren, USA, who welcomed the idea of this workshop and gave us complete support and encouragement.

The high quality of the workshop programme would not have been possible without the speakers and their technical talks. The speakers are successful top-class researchers in the subject matter of the workshop and came from all parts of the world making the conference truly international. In spite of their busy schedule, they attended the workshop and prepared the materials for the proceedings.

We would like to thank Mr. Mark de Jongh, Springer Publication, for his immediate perception of the value and potential of this workshop and for readily agreeing to bring out this volume.

Special thanks go to all the members of the Control Software Engineering Methods and Tools Group at India Science Lab, GM R&D Bangalore who worked hard for many weeks before the workshop to bring out such a fine workshop. It would not have been possible to organize such a high quality workshop without their support.

Finally, we would like to thank the members of the operation team of General Motors Technical Centre India for their support and assistance; and National Institute of Advanced Studies for providing a wonderful ambience for conducting the workshop.

Bangalore,
February 2007

S. Ramesh
P. Sampath

Advisory Committee

B.G. Prakash (GM R&D, Bangalore)
N.H. Sathyaraja (GM R&D, Bangalore)

Programme Committee

S. Ramesh (GM R&D, Bangalore) (Chairman)
Yaron Wolfsthal (IBM Haifa)
Prahladavaradan Sampath (GM R&D, Bangalore)
Shengbing Jiang (GM R&D, Warren)
Deepak D'Souza (IISc, Bangalore)

Organizing Committee

Rajeev A. C. (GM R&D, Bangalore)
Manoj G. Dixit (GM R&D, Bangalore)
Ambar A. Gadkari (GM R&D, Bangalore)
Prasanna Vignesh V. Ganesan (GM R&D, Bangalore)
Suresh J. (GM R&D, Bangalore)
Swarup K. Mohalik (GM R&D, Bangalore)
Manoranjan M. Satpathy (GM R&D, Bangalore)
K. C. Shashidhar (GM R&D, Bangalore)
Anand V. Yeolekar (GM R&D, Bangalore)

Contents

Contributors

Rajeev Alur
University of Pennsylvania

Ansuman Banerjee
Department of Computer Science & Engineering, Indian Institute of Technology, Kharagpur, India-721302
ansuman@cse.iitkgp.ernet.in

Gerard Berry
Esterel Technologies
gerard.berry@esterel-technologies.com
www.esterel-technologies.com

Purandar Bhaduri
Department of Computer Science and Engineering,
Indian Institute of Technology, Guwahati 781039, India
pbhaduri@iitg.ernet.in

Jatin Bharadia
Embedded Real-Time Systems Lab, Indian Institute of Technology, Bombay
jatin@it.iitb.ac.in

Unmesh D. Bordoloi
Department of Computer Science, National University of Singapore
unmeshdu@comp.nus.edu.sg

Partha P. Chakrabarti
Department of Computer Science & Engineering, Indian Institute of Technology, Kharagpur, India-721302
ppchak@cse.iitkgp.ernet.in

Samarjit Chakraborty
Department of Computer Science, National University of Singapore
samarjit@comp.nus.edu.sg

Supratik Chakraborty
Indian Institute of Technology, Bombay, India
supratik@cse.iitb.ac.in

Prakash Chandrasekaran
Chennai Mathematical Institute

Edmund M. Clarke
Computer Science Department, Carnegie Mellon University,
Pittsburgh, PA 15213
emc@cs.cmu.edu

Christopher L. Conway
New York University

Werner Damm
OFFIS, Oldenburg, Germany

Pallab Dasgupta
Department of Computer Science & Engineering, Indian Institute of Technology,
Kharagpur, India-721302
pallab@cse.iitkgp.ernet.in

F. Doucet
Department of Computer Science and Engineering, University of
California at San Diego, CA 92093-0404, USA
fdoucet@ucsd.edu

Stephan Falke
Computer Science Dept., University of New Mexico, Albuquerque, NM, USA
spf@cs.unm.edu

Paolo Giusto
General Motors Research and Development, 30500 Mound Road,
Warren, MI 48090-9055

Ziv Glazberg
IBM, Haifa Research Lab., Mount Carmel, 31905 Haifa, Israel
glazberg@il.ibm.com

R. Gupta
Department of Computer Science and Engineering,
University of California at San Diego, CA 92093-0404, USA
rgupta@ucsd.edu

Andrei Hagiescu
Department of Computer Science, National University of Singapore,
hagiescu@comp.nus.edu.sg

Joseph M. Joy
Microsoft Research India

Sudeep Juvekar
Indian Institute of Technology, Bombay, India
sjuvekar@cse.iitb.ac.in

Varun Kanade
Georgia Institute of Technology, USA
varunk@cc.gatech.edu

Deepak Kapur
Computer Science Dept., University of New Mexico, Albuquerque, NM, USA
kapur@cs.unm.edu

I.H. Krüger
Department of Computer Science and Engineering,
University of California at San Diego, CA 92093-0404, USA
ikrüger@ucsd.edu

Aniket Kulkarni
TRDDC
aniket.kulkarni@tcs.com

Flavio Lerda
Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15213
flerdda@cs.mu.edu

Deepak A. Mathaikutty
Virginia Polytechnic and State University,
damathai@vt.edu

Ravindra Metta
TRDDC,
ravindra.metta@tcs.com

Alexander Metzner
OFFIS, Oldenburg, Germany

Mark Moulin
IBM, Haifa Research Lab., Mount Carmel, 31905 Haifa, Israel
markm@il.ibm.com

Marco Di Natale
General Motors Research and Development, 30500 Mound Road,
Warren, MI 48090-9055

Avigail Orni
IBM, Haifa Research Lab., Mount Carmel, 31905 Haifa, Israel
ornia@il.ibm.com

Paritosh K. Pandya
Tata Institute of Fundamental Research, Colaba, Mumbai, India 400005
pandya@tifr.res.in

Sriram K. Rajamani
Microsoft Research India

Raj Rajkumar
Professor, ECE and CS, Carnegie Mellon University, Pittsburgh, USA
raj@ece.cmu.edu

Krithi Ramamritham
Embedded Real-Time Systems Lab, Indian Institute of Technology, Bombay
krithi@cse.iitb.ac.in

Gurulingesh Raravi
Embedded Real-Time Systems Lab, Indian Institute of Technology, Bombay
guru@it.iitb.ac.in

Sitvanit Ruah
IBM, Haifa Research Lab., Mount Carmel, 31905 Haifa, Israel
sitvanit@il.ibm.com

Natarajan Shankar
Computer Science Laboratory
SRI International, Menlo Park CA 94025 USA
shankar@csl.sri.com

Vipul Shingde
Embedded Real-Time Systems Lab, Indian Institute of Technology, Bombay
vipul@cse.iitb.ac.in

Sandeep K. Shukla
Virginia Polytechnic and State University, shukla@vt.edu

Ulka Shrotri
TRDDC
ulka.s@tcs.com

R.K. Shyamasundar
IBM India Research Lab, Block 1, IIT Delhi, Hauz Khas, New Delhi 110016, India
rshyamas@in.ibm.com

Mahadevan Subramaniam
Computer Science Dept., University of Nebraska at Omaha, Omaha, NE, USA
msubramaniam@mail.unomaha.edu

Syed M. Suhaib
Virginia Polytechnic and State University,
ssuhaib@vt.edu

Jean-Pierre Talpin
IRISA/INRIA
jean-pierre.talpin@irisa.fr

Muralidhar Talupur
Computer Science Department,
Carnegie Mellon University, Pittsburgh, PA 15213
tmurali@cs.cmu.edu

Ankur Taly
Indian Institute of Technology, Bombay, India
ankur123@cse.iitb.ac.in

R. Venkatesh
TRDDC
r.venky@tcs.com

Emmanuel Zarpas
IBM, Haifa Research Lab., Mount Carmel, 31905 Haifa, Israel
zarpas@il.ibm.com

Wei Zheng
University of California at Berkeley,
EECS Department, Berkeley CA 94720

An Abstraction Technique for Real-Time Verification

Edmund M. Clarke, Flavio Lerda, and Muralidhar Talupur

Abstract In real-time systems, correctness depends on the time at which events occur. Examples of real-time systems include timed protocols and many embedded system controllers. Timed automata are an extension of finite-state automata that include real-valued *clock variables* used to measure time. Given a timed automaton, an equivalent finite-state region automaton can be constructed, which guarantees decidability. Timed model checking tools like UPPAL, KRONOS, and RED use specialized data structures to represent the real-valued clock variables. A different approach, called integer-discretization, is to define clock variables that can assume only integer values, but, in general, this does not preserve continuous-time semantics.

This paper describes an implicit representation of the region automaton to which ordinary model checking tools can be applied directly. This approach differs from integer discretization because it is able to handle real-valued clock variables using a finite representation and preserves the continuous-time semantics of timed automata. In this framework, we introduce the GOABSTRACTION, a technique to reduce the size of the state space. Based on a conservative approximation of the region automaton, GOABSTRACTION makes it possible to verify larger systems. In order to make the abstraction precise enough to prove meaningful properties, we introduce auxiliary variables, called Go variables, that limit the drifting of clock variables in the abstract system. The paper includes preliminary experimental results showing the effectiveness of our technique using both symbolic and bounded model checking tools.

Keywords: Abstraction, model checking, real-time systems, timed automata.

1 Overview

Real-time systems are a class of systems whose correctness depends on the time at which events occur. Examples include embedded controllers, time triggered systems, and timed protocols. In fact, most safety critical systems are real-time systems as they require guarantees on the timing of events. For instance, in order for the braking system of a car to be correct, it is not sufficient for the correct output to be produced, but it has to be produced within a given time bound.

S. Ramesh and P. Sampath (eds.), *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*, 1–17.

Model checking is widely used in the semiconductor industry and it has been successful for software. For example, the model checker SLAM [3] is the basis for the *Driver Verifier*, which is currently being distributed by Microsoft as part of their Device Driver Software Development Kit. Moreover, all device drivers must be verified using the Driver Verifier in order to be certified by Microsoft's Windows Hardware Quality Labs. However, finite-state model checking cannot be applied directly to real-time systems because time is modeled as a continuous, real-valued quantity. The success of finite-state model checking has spurred the development of model checking techniques for infinite-state systems, including real-time systems.

Real-time systems are often modeled using *timed automata* [2]. Timed automata are an extension of finite-state automata that include a set of *clock variables* to keep track of time. The transitions of a timed automaton are labeled with *clock constraints* that must hold when a transition is taken, and sets of *clock variables to be reset* after a transition occurs. To specify properties of timed automata, extensions of ordinary temporal logics, e.g., *Timed Computation Tree Logic* (TCTL) [1], have been proposed.

In recent years, there has been extensive work on verification of real-time systems. Alur et al. [1, 2] developed the theoretical foundations for much of the work in this area. They introduced *timed automata*, an extension of finite-state automata that can be used to model real-time systems. They proposed the *region graph construction*, which maps questions about an (infinite-state) real-time system into questions about a corresponding finite-state automaton. Many tools and techniques for real-time verification are based on this work but employ specialized data structures to represent clock variables: Difference Bounded Matrices [7], Region Encoding Diagrams [17], Clock-Restriction Diagrams [20], and Difference Decision Diagrams [13], just to name a few. Henzinger and Kupferman [9] showed how to reduce the problem of checking a timed temporal logic property of a timed automaton to checking an (untimed) temporal logic property of a property of the region automaton. Therefore, from now on, we will only consider untimed temporal logic properties. Other approaches have also been proposed. For instance, extensive work has been done on integer discretization based techniques [4, 5, 10, 11], where real-valued clock variables are replaced by integer-valued variables. While these techniques in general do not preserve the continuous-time semantics of timed-automata, they are sound for a class of real-time systems and properties [10].

In this paper, we explore techniques for the verification of timed automata that are based on the region graph construction but do not use specialized data structures to represent clock variables. Preliminary work in this area is due to Gōllü et al. [8], however, no implementation or experimental results were presented by the original authors. These techniques are usually referred to as *discretization* techniques, but they are radically different from the integer-discretization techniques mentioned above. The main difference is that the former provide a finite (discrete) representation for sets of real-valued clock variables while the latter is able to handle only integer-valued clock variables. In this paper, we describe a new implicit representation of the region automaton. The representation is implicit in that we do not enumerate regions

or transitions explicitly. The resulting system can be verified using existing model checking tools. Our representation of clock regions is similar to the one of Wang [18], however, their approach is based on a specialized data structure for symbolic model checking and it cannot be used with other model checking tools.

The region graph construction [1] is a well known technique for model checking timed automata. A state of a timed automaton is defined as a pair made of a location and a valuation of the clock variables. A clock region is a (possibly infinite) set of clock valuations. The region graph construction defines a bisimulation between the states of a timed automaton and a finite set of clock regions. The result of the construction is a region automaton, a finite-state automaton that is bisimilar to the original timed automaton. The bisimulation defined in [1] preserves temporal logic properties. Verification of a property of a timed automaton is reduced to the verification of the same property on the corresponding region automaton. The region automaton is, by construction, finite, therefore, ordinary model checking techniques can be applied to it. However, since the region automaton can be exponential in the size of the original timed automaton, existing tools like UPPAAL [12], KRONOS [21], and RED [19] treat clock variables differently from discrete state variables and use specialized data structures to represent clock regions.

Since the region automaton is, in the worst case, exponential in the number of clock variables [2], we introduce a new abstraction technique called GOABSTRACTION that addresses this blow up. Approaches that use a representation similar to ours, e.g., [8, 18], do not have a similar abstraction technique.

Abstractions have been widely used in hardware and software model checking to improve the performance of verification. Predicate abstraction has been applied to timed automata by Möller et al. [14] and Sorea [15]. This approach is based on identifying a set of predicates that is sufficient to discriminate between any two clock regions and uses abstraction/refinement to find a minimal subset of these predicates that is sufficient to perform the verification. Tripakis and Yovine [16] define an abstraction that removes the actual value of the delays to obtain a timeless system, which is finite-state. Our approach, instead, is based on merging clock regions that differ only in the ordering of fractional parts.

In the region graph construction, a clock region corresponds to a set of clock valuations that are equivalent according to the bisimulation relation presented in [1]. One of the conditions necessary for two clock valuations v and v' to be equivalent is that the *ordering of the fractional parts* of each pair of clock variables is the same in both valuations. For instance, if the fractional part of clock c_1 is less than the fractional part of clock c_2 in v, the same must hold for v', even if the actual values may differ. This is necessary to precisely compute the successors of a given clock region. However, given n clock variables, there are $n!$ possible orders of their fractional parts, and, in principle, $n!$ different clock regions. This can cause an exponential blow up in the number of states in the region automaton, which can lead to intractability.

Our approach abstracts the relative ordering between the fractional parts. By doing so, we obtain an over-approximation of the behavior of the system, where precise information is lost. Regions that differ only because of the ordering of the fractional

parts of some clock variables are merged into a single abstract clock region. By reducing the number of clock regions, we decrease the number of states in the region automaton and, therefore, we obtain a smaller state space. However, while the abstraction is safe and it is guaranteed to preserve the validity of properties, it may introduce spurious counterexamples.

The abstraction scheme as presented so far is too coarse. The problem is that, as we discard the relative ordering between clock variables, we allow them to drift apart unboundedly. In order to make the abstraction more precise, we introduce auxiliary variables, called Go variables, that keep track of the way clock variables evolve and limit the drifting to at most one time unit.

In our preliminary experiments, we show how GOABSTRACTION is sufficient to prove properties for a real-time protocol, namely Fischer's mutual exclusion protocol, that could not be established with a naive abstraction scheme that did not make use of the Go variables.

The remainder of the paper is organized as follows. Section 2 recalls some useful definitions. Our discretization is presented in Section 3, and GOABSTRACTION is introduced in Section 4. Section 5 contains some preliminary experimental results and Section 6 gives conclusions and directions for future work.

2 Preliminaries

2.1 Timed Automata

Timed automata are a formalism used to model real-time systems. They are an extension of finite-state automata that include a set of real-valued *clock variables* used to measure time. Transitions of a timed automaton are labeled with a *clock constraint* and a set of clock variables known as the *reset set*. A transition can be taken only if the clock constraint associated with it is true in the current state. After a transition is taken, the values of the clock variables in the reset set are set to zero.

Definition 1 (Clock Constraints) *A* clock constraint *is a Boolean combination of equalities and inequalities involving a single clock variable x and an integer constant c (i.e., $x < c$, $x \leq c$, $x = c$, $x \geq c$, and $x > c$).*

The set of all possible clock constraints over a set of clock variables X is denoted by $C(X)$.

Definition 2 (Clock Valuations) *A* clock valuation *over a set of clock variables X is a function $v : X \to \mathbb{R}^+$ that assigns to every clock variable in X a non-negative real value.*

The set of all possible clock valuations over a set of clock variables X is denoted by $V(X)$. Let v_0, called the *zero clock valuation*, be the clock valuation that assigns

the value zero to all clock variables. Given a clock valuation $v \in V(X)$ and a non-negative real value $\delta \in \mathbb{R}^+$, we denote by $v + \delta \in V(X)$ the clock valuation that maps every clock variable $x \in X$ to the value $v(x) + \delta$. Given a clock valuation $v \in V(X)$ and a reset set $\lambda \subseteq X$, we denote by $v[\lambda = 0] \in V(X)$ the clock valuation that maps every clock variable x in λ to zero and every clock variable x not in λ to the same value v does. Given a clock constraint $g \in C(X)$ and a valuation $v \in V(X)$, v satisfies g if and only if the expression obtained by replacing in g every occurrence of a clock variable x with the value $v(x)$ evaluates to true.

Definition 3 (Timed Automaton) *A* timed automaton *is a 5-tuple* $A = (Q, X, q_0, I, T)$ *where* Q *is a finite set of locations;* X *is a finite set of real-valued clock variables;* $q_0 \in Q$ *is an initial location;* $I : Q \to 2^{V(X)}$ *is a location invariant, a function that assigns to every location a set of valid valuations; and* $T \subseteq Q \times C(X) \times 2^X \times Q$ *is a set of discrete transitions, such that* $(q, g, \lambda, q') \in T$ *if and only if there is a discrete transition from location* q *to location* q' *labeled with the clock constraint* g *and the reset set* λ.

The state of a timed automaton A is a pair (q, v) such that $q \in Q$ is a location and $v \in V(X)$ is a clock valuation. Timed automata allow two types of transitions: (i) time transitions, which correspond to the passing of time; and (ii) discrete transitions, which correspond to the discrete transitions of the automaton. A *time transition* is labeled by a positive real value δ and maps state (q, v) into state $(q, v + \delta)$ if for all non-negative real values $\delta' \leq \delta$, $v + \delta'$ belongs to the invariant $I(q)$. A *discrete transition* is labeled by $(q, g, \lambda, q') \in T$ and maps state (q, v) into state $(q', v[\lambda = 0])$ if v satisfies the clock constraint g and $v[\lambda = 0]$ belongs to the invariant $I(q')$.

2.2 Region Graph Construction

A state of a timed automaton is a pair made of a location and a clock valuation. Therefore, the set of possible states is infinite, as the clock variables are assigned values from $\mathbb{R}^+$. Model checking was developed as a technique for automatically verifying properties of finite-state systems. As such, it is not directly applicable to timed automata, since they may have an infinite number of states.

Alur et al. [1] proposed the *region graph construction* as a way to make verification of real-time systems feasible. Given a timed automaton, the region graph construction produces a *region automaton*, a finite-state automaton that is bisimilar to the original timed automaton. Model checking can then be performed on the region automaton, which satisfies the same set of properties as the original timed automaton.

Given a timed automaton A, for each clock variable $x \in X$, let M_x be the largest constant against which x is compared in the clock constraints associated with the discrete transitions of A. We call M_x the *maximum constant value* of clock variable x in the timed automaton A. Let $\lfloor x \rfloor$ be the integer part of clock variable x, and $\langle x \rangle = x - \lfloor x \rfloor$ be its fractional part.

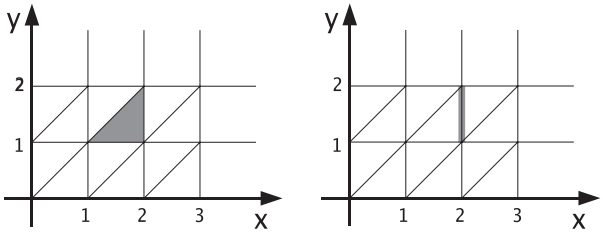


Fig. 1 The shaded area in each diagram represents a set of equivalent clock valuations

Definition 4 (Equivalent Clock Valuations) *Given a set of clock variables X and their maximum constant values M_x, two clock valuations $v_1, v_2 \in V(X)$ are* equivalent, *$v_1 \approx v_2$, if and only if:*

- *For all $x \in X$, either $\lfloor v_1(x) \rfloor = \lfloor v_2(x) \rfloor$ or both $v_1(x)$ and $v_2(x)$ are greater than M_x;*
- *For all $x \in X$ such that $v_1(x) \leq M_x$, $\langle v_1(x) \rangle = 0$ if and only if $\langle v_2(x) \rangle = 0$; and*
- *For all $x, y \in X$ such that $v_1(x) \leq M_x$ and $v_1(y) \leq M_y$, $\langle v_1(x) \rangle \lhd \langle v_1(y) \rangle$ if and only if $\langle v_2(x) \rangle \lhd \langle v_2(y) \rangle$, for $\lhd \in \{<, =, >\}$.*

As an example, consider Figure 1. Each point on one of the two diagrams corresponds to a clock valuation, each shaded area to a set of equivalent clock valuations. The shaded area on the left shows the clock valuations such that $\lfloor x \rfloor$ and $\lfloor y \rfloor$ are equal to 1 and $\langle y \rangle$ is smaller than $\langle x \rangle$. The shaded area on the right represents the clock valuations such that $\lfloor x \rfloor = 2$, $\lfloor y \rfloor = 1$, $\langle x \rangle < \langle y \rangle$, and $\langle x \rangle = 0$.

The first two conditions of Def. 4 guarantee that given two equivalent clock valuations, they satisfy the same set of clock constraints. Given a clock constraint $x \lhd c$, the validity of the constraint can be decided by knowing the integer part of x and whether the fractional part of x is equal to zero.

The third condition is needed to guarantee that, given two equivalent clock valuations, as time passes, they will reach clock valuations that are equivalent. Consider again Figure 1. For all clock valuations represented by the shaded area in the diagram on the left $\langle y \rangle < \langle x \rangle$. As a consequence, as time passes, since both variables are incremented at the same rate, clock variable x will reach the value 2 before clock variable y does. Therefore, the set of clock valuations such that $\lfloor x \rfloor = 2 \wedge \lfloor y \rfloor = 1 \wedge \langle x \rangle < \langle y \rangle \wedge \langle x \rangle = 0$ is reachable. The shaded area in the diagram on the right represents this set of clock valuations. If we did not know the ordering of the fractional parts of x and y, two other sets of equivalent clock valuations would also be reachable, $\lfloor x \rfloor = \lfloor y \rfloor = 2 \wedge \langle x \rangle = \langle y \rangle = 0$ and $\lfloor x \rfloor = 1 \wedge \lfloor y \rfloor = 2 \wedge 0 = \langle y \rangle < \langle x \rangle$.

Definition 5 (Clock Region) *A* clock region *μ is an equivalence class of the relation $\approx$ defined above.*

Let the set of clock regions of the automaton A be denoted by $\Gamma(A)$. $\Gamma(A)$ is finite by construction. Since all valuations in a clock region satisfy the same set of

clock constraints, a region μ satisfies a clock constraint c if and only if every clock valuation $v \in \mu$ satisfies c. Given a clock region μ, we define $\mu' = \mu[\lambda = 0]$ to be the clock region such that, for all clock valuations $v \in \mu$, $v[\lambda = 0]$ belongs to μ'.

Definition 6 (Time Successor) *Given a clock region μ, a clock region $\mu' \neq \mu$ is a* time successor *of μ if and only if there exists a clock valuation $v \in \mu$ and a positive real value δ, such that $v + \delta \in \mu'$ and for all non-negative real values $\delta' < \delta$, $v + \delta'$ belongs either to μ or μ'.*

Notice that each clock region μ has at most one time successor because of the way we defined the equivalence relation $\approx$ on clock valuations.

Definition 7 (Region Automaton) *Given a timed automaton A, the corresponding* region automaton *is a finite-state automaton $R(A) = (S, s_0, R)$ where $S = Q \times \Gamma(A)$ is a finite set of states; $s_0 = (q_0, \mu_0)$ is an initial state, where μ_0 is the clock region containing the zero clock valuation v_0; and $R \subseteq S \times S$ is a finite transition relation such that $((q_1, \mu_1), (q_2, \mu_2))$ belongs to R if and only if either:*

- *$q_1 = q_2$, μ_2 is the time successor of μ_1, and μ_2 satisfies $I(q_1)$; or*
- *there exists a discrete transition (q_1, g, λ, q_2) such that μ_1 satisfies g, $\mu_2 = \mu_1[\lambda = 0]$, and μ_2 satisfies $I(q_2)$.*

The *region automaton* captures the behaviors of the original timed automaton exactly, i.e., they satisfy the same sets of properties.

3 Discretization

In this section, we give a representation of the region automaton. The representation is implicit as, in the model we construct, time transitions are not enumerated explicitly but are represented by two transitions called the *from-integer* and the *to-integer time transitions*.

Given a timed automaton A, let M_x be the *maximum constant values* in A. For each clock variable $x \in X$, let us introduce two shared variables: an integer part variable I_x and a fractional order variable F_x.

The *integer part variable* represents the integer part of a clock variable. For a clock variable x, I_x is equal to $\lfloor x \rfloor$ if $x \leq M_x$ and M_x otherwise. Therefore I_x is an integer ranging between 0 and M_x.

For a given clock valuation, order the clock variables that are smaller or equal to the corresponding maximum constant value according to the values of their fractional parts. The *fractional order variable* represents the position of a clock variable in this order. The fractional order variable of a clock variable $x \leq M_x$ is equal to zero if and only if the fractional part of x is equal to zero. For the variables with the smallest, non-zero fractional part (there may be more than one), the corresponding fractional order variable is set to 1. For the variables with the second smallest,

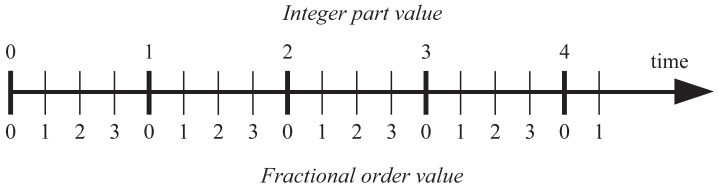


Fig. 2 The possible values of integer part and fractional order variables. The example shows the case of 3 clock variables with the maximum constant value for the variable shown equal to 4. The fractional order values represent only the relative ordering between fractional parts

non-zero fractional part, the corresponding fractional order variable is set to 2, and so on. If $x > M_x$ then the corresponding fractional order variable F_x is set to 1, as the order between fractional parts is not relevant for clock variables larger than their maximum constant value. If two clock variables x and y such that $x \leq M_x$ and $y \leq M_y$ have the same fractional part, their fractional order variables are equal. The fractional order variables are integers ranging between 0 and n, where n is the number of clock variables. The order between fractional parts is maintained by the fractional order variables, i.e., given two clock variables x and y such that $x \leq M_x$ and $y \leq M_y$, $F_x \lhd F_y$ if and only if $\langle x \rangle \lhd \langle y \rangle$, for $\lhd \in \{<, =, >\}$. While clock variable $x \in X$ is a real-valued variable, I_x and F_x are discrete (cf. Figure 2).

Definition 8 (Discrete Clock Valuations) *Given a set of clock variables X, a* discrete clock valuation *is a function v^d that, for each clock variable $x \in X$, assigns to I_x a value from $\{0, \ldots, M_x\}$ and to F_x a value from $\{0, \ldots, n\}$.*

Let $V^d(X)$ be the set of discrete clock valuations defined for a set of clock variables X. Given a clock valuation $v \in V(X)$, the corresponding *discrete clock valuation* v^d assigns values to each integer part and fractional order variables as described above. Given a clock variable $x \in X$, we will denote by $v^d(x)$ the pair $(v^d(I_x), v^d(F_x))$, called the discrete value of x.

Theorem 9 (Equivalence to Regions) *Each* discrete clock valuation *corresponds to a unique clock region and vice-versa, i.e., given two clock valuation v_1 and v_2, v_1 is equivalent to v_2 (Def. 4) if and only if the corresponding discrete clock valuations v_1^d and v_2^d are equal.*

The states of the region automaton can be represented by a pair made of a location and a discrete clock valuation. Now that we have defined discrete clock valuations, a representation for clock regions, and discrete states, we define how transitions between states in the region automaton map to transitions between discrete states in the discrete timed system. The region automaton defines two types of transitions: time transitions and discrete transitions.

Time transitions are represented by two transitions: (i) the *from-integer time transition*, which is taken when one of the clock variables has an integer value; and (ii) the *to-integer time transition*, which leads to a state where one of the clock variables has

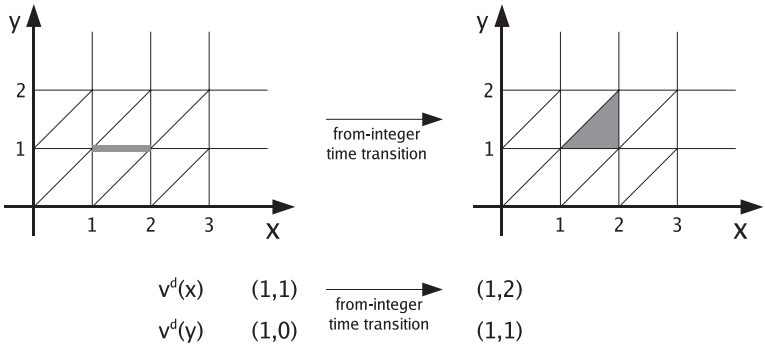


Fig. 3 Evolution of a region and the corresponding discrete valuation due to the *from-integer time transition*. Each shaded area represents the clock valuations belonging to a region

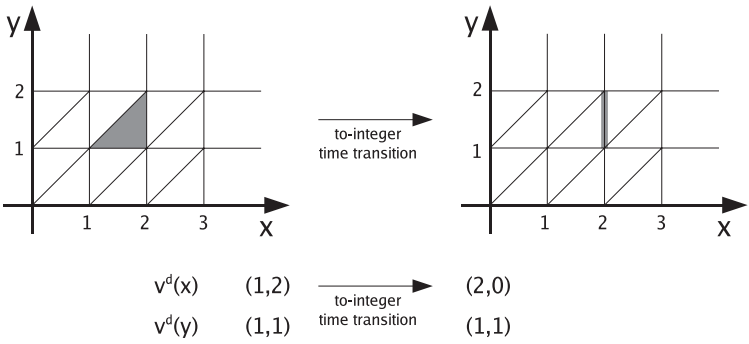


Fig. 4 Evolution of a region and the corresponding discrete valuation due to the *to-integer time transition*. Each shaded area represents the clock valuations belonging to a region

an integer value. Each time transition represents a set of actual transitions. We use two types of time transitions to capture two possible scenarios: (i) the case where at least one clock variable has an integer value (cf. Figure 3); and (ii) the case where none of the clock variables has an integer value (cf. Figure 4). In the figures, the diagrams at the top, represent the clock regions as shaded area as before. The bottom shows a discrete clock valuation by assigning an integer part and a fraction order to each clock variable.

The *from-integer time transition* can be taken only if there exists at least one fractional order variable equal to zero. When this transition is taken, all fractional order variables are incremented by one, while all integer part variables remain unchanged. The example in Figure 3 contains two clock variables x and y. The maximum constant value of x is 3 and the one of y is 2. A point in one of the diagrams at the top of the figure represents a clock valuation, which assigns the corresponding values to x and y. The thin lines split the clock valuations into regions. A shaded area is used to represent a specific clock region. Initially the discrete value of x is (1, 1) and the one of y is (1, 0). The shaded area in the diagram at the top-left of the figure shows the

region corresponding to this discrete state. As time progresses, clock variable y will become greater than 1 before clock variable x reaches 2, and it will have the smallest, non-zero fractional part. Therefore, its discrete value will be (1, 1). At the same time, variable x will still have integer part equal to 1, but its fractional part will become the second smallest one and, therefore, its discrete value will be (1, 2). The shaded area in the diagram at the top-right of the figure shows the region corresponding to the new state.

The *to-integer time transition* can be taken only if none of the fractional order variables is equal to zero. The fractional order variables with the largest value (there might be more than one) are set to zero and the corresponding integer part variables are incremented by one. All other integer part and fractional order variables remain unchanged. The example in Figure 4 contains clock variables x and y as before. Initially the discrete value of x is (1, 2) and the one of y is (1, 1). The shaded area in the diagram at the top-left of the figure shows the region corresponding to this discrete state. As time progresses, variable x will be the first one to reach an integer value, because it has the largest fractional part. Its new value will be (2, 0) and the next value of y will remain (1, 1), since y still has the smallest, non-zero fractional part. The shaded area in the diagram at the top-right of the figure shows the region corresponding to the new state.

The clock variables $x \in X$ such that the integer part variable I_x is equal to M_x and the fractional order variable F_x is greater than zero are treated differently: their integer part and fractional order variables are not updated by the from-integer or the to-integer time transitions. This is because, in the region graph construction, the order between fractional parts is relevant only for those clock variables that are smaller than the corresponding maximum constant value.

Each discrete transition of the region automaton is mapped into a corresponding discrete transition between discrete clock valuations. A discrete clock valuation satisfies a clock constraint g if the corresponding clock region does. Since clock variables are only compared against integer constants, it is possible to determine if a discrete clock valuation satisfies a clock constraint by looking only at the integer part and fractional order variables. After a transition is taken, the clock variables in the reset set λ must be set to zero. If clock variable x belongs to λ, both the corresponding integer part and fractional order variables are set to zero.

Given a timed automaton A, the result of our discretization is the *discrete timed system* A^d, a system made of two asynchronous processes and containing, for each clock variable x, two discrete variables I_x and F_x shared by the two processes. The first process, called the *discrete-transition process* has the same locations and transitions as the original timed automaton, where clock constraints are mapped into expressions over I_x and F_x and reset sets are mapped into resets of these variables, as described above. The second process, called the *time-transition process*, defines the from-integer and to-integer time transitions. The system is modeled using two asynchronous processes: one process defines the time transitions, the other defines the discrete ones. The time transitions can occur at any location of the timed automaton. Having two asynchronous processes allows us to use a smaller representation:

time transitions are defined only once but, by virtue of the asynchronous composition, they can be taken at any location of the timed automaton. The idea of separating discrete transitions and time transition into two asynchronous processes has been used by Lamport [11] in his integer discretization based approach for real-time systems. However, as with other integer discretization techniques, this approach handles only integer-valued clock variables and, therefore, does not capture the continuous time semantics of timed automata.

Theorem 10 (Discrete Equivalence) *Given a timed automaton A, the* discrete timed system A^d *and the region automaton $R(A)$ are equivalent.*

The main advantages of this construction are: (i) the construction is implicit, it does not enumerate the clock regions or the time transitions between them; (ii) the resulting system can be checked using any of the existing model checking tools and therefore exploit the recent advances in this domain; (iii) this approach can easily be extended to the composition of a set of timed automata: since they need to synchronize over the time transitions, we can represent the composition using a *discrete-transition process* for each automaton and a single instance of the *time-transition process*.

4 GoAbstraction

The discretization given in the previous section makes it possible to verify properties of timed automata using standard model checking tools. However, in the worst case, the region automaton can be exponential in the number of clock variables and the largest constant. Therefore, even if our construction does not explicitly enumerate the clock regions, model checking might not terminate because of the size of the state space.

In this section, we introduce a new abstraction technique, called GoAbstraction, which aims at reducing the size of the state space. This is a conservative approximation of the behaviors of the system, i.e., each behavior of the original system is maintained in the abstraction, but it may introduce spurious counterexamples.

In the construction given in Section 4, for each clock variable x, the fractional order variable F_x is used to represent the ordering relation between the fractional parts of the different clock variables. Keeping track of this ordering, however, may lead to a number of different permutations that is exponential in number of clock variables. For some applications, this can cause the verification to be intractable.

We propose an abstraction that discards part of the ordering relation between clock variables. In the previous construction, the fractional order variables ranged between 0 and n, where n is the number of clock variables. In the abstraction, we replace the fractional order variables F_x with *abstract fractional order variables* F_x^α. These variables assume values in the abstract domain $F^\alpha = \{0, \alpha\}$, where 0 represents clock variables whose fractional part is equal to zero, and α represents all other possible fractional order values (cf. Figure 5).

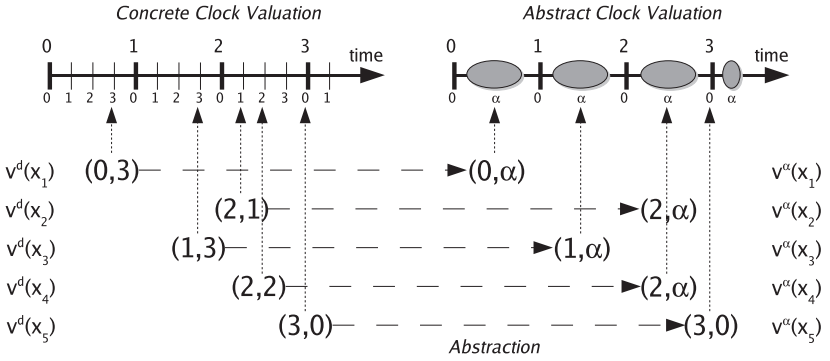


Fig. 5 The mapping between concrete and abstract clock valuations

Definition 11 (Abstract Clock Valuations) *Given a set of clock variables X, an* abstract clock valuation *is a function v^α that, for each clock variable in $x \in X$, assigns to I_x a value from $\{0, \ldots, M_x\}$ and to F_x^α a value from F^α.*

Let $V^\alpha(X)$ be the set of abstract clock valuations defined for a set of clock variables X. Given a clock variable $x \in X$, we will denote by $v^\alpha(x)$ the pair $(v^d(I_x), v^d(F_x^\alpha))$, called the abstract value of x.

Definition 12 (Abstraction Function) *The* abstraction function $h : V^d(X) \rightarrow V^\alpha(X)$ *maps discrete clock valuations into abstract clock valuations and is defined as:*

$$h(v^d)(V_x) = \begin{cases} v^d(I_x) & \text{if } V_x = I_x \\ 0 & \text{if } V_x = F_x \text{ and } v^d(F_x) = 0 \\ \alpha & \text{if } V_x = F_x \text{ and } v^d(F_x) \neq 0 \end{cases}$$

Given the abstraction function h, it is possible to construct an abstract timed system A^α using a technique called *existential abstraction* [6]. Existential abstraction produces an over-approximation of the concrete system that is guaranteed to preserve universal CTL (∀CTL) properties. The abstract timed system A^α is analogous to the discrete timed system A^d but uses the abstract fractional order variables instead of the (concrete) fractional order ones. Each transition of A^d is mapped into an abstract transition of A^α as described below.

The *from-integer abstract time transition* can be taken only if there exists at least one abstract fractional order variable equal to 0. When this transition is taken, all fractional order variables equal to 0 are set to α.

The *to-integer abstract time transition* can be taken only if all abstract fractional order variables are equal to α. When this transition is taken, any non-empty subset of the fractional order variables can be set to 0 and the corresponding integer part variables are incremented by one. Notice that the transitions represented by the *to-integer abstract time transition* can be non-deterministic.

Each discrete transitions is mapped into an abstract discrete transition. The validity of a constraint can be determined by knowing the value of the integer part variables and whether the abstract fractional order variables are equal to zero. The reset of a clock variable can be done by setting both the integer part and the abstract fractional order variables to zero.

Given a timed automaton A, the result of our abstraction is the *abstract timed system* A^{α}, a system made of two asynchronous processes and containing, for each clock variable x, two discrete variables I_x and F_x^{α}. The first process, called the *abstract discrete-transition process* has the same locations and transitions as the original timed automaton, where clock constraints are mapped into expressions over I_x and F_x^{α} and reset sets are mapped into resets of these variables, as described above. The second process, called the *abstract time-transition process*, defines the from-integer and to-integer abstract time transitions.

Theorem 13 (Abstraction Preservation) *Given a timed automaton A, the* abstract timed system A^{α} *is an over-approximation of the discrete timed system* A^{d}*, i.e., every trace of* A^{d} *corresponds to an equivalent trace of* A^{α}.

The abstraction above, however, is too coarse. Given two clocks that are assigned the same value, it is possible for them to drift apart arbitrarily, i.e., there exists a sequence of abstract time transitions such that the difference between the two clocks grows unboundedly (cf. Figure 6).

This is because, in the abstraction, we discarded the order between the fractional parts and introduced non-determinism in the to-integer abstract time transition. It is possible to increment one of the clock variables multiple times before incrementing the others.

In order to prevent this and obtain a more precise abstraction, for each of the clock variables $x \in X$, we introduce in our model the Boolean variable Go_x. The purpose of this variable is to keep track whether a clock variable has been already incremented.

Initially, all Go_x variables are set to *true*. This means that all variables can be incremented. Once a clock variable x has been incremented, the variable Go_x is set to *false*. This prevents the same clock variable from being incremented again. When all Go_x variables are *false*, i.e., every variable has been incremented once, they are set to *true* simultaneously. Figure 7 illustrates the behavior of the Go_x variables. They guarantee that two clock variables cannot drift apart by more than one time unit, making the abstraction more precise.

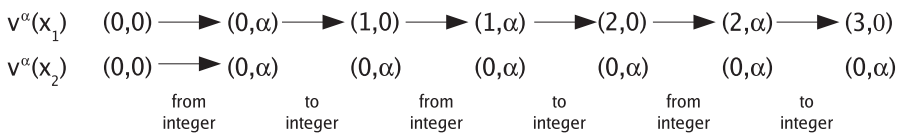


Fig. 6 Given two clock variables initially equal, they can drift apart by means of an appropriate sequence of from-integer and to-integer abstract time transitions

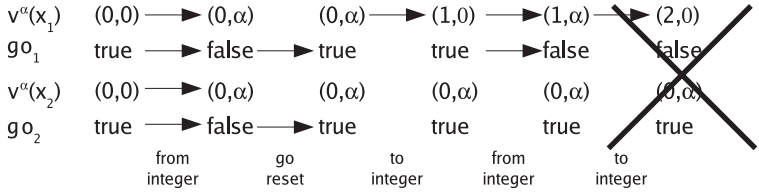


Fig. 7 The Go_x variables prevent clock variables from drifting apart

We can now construct the GOABSTRACTION timed system A^{α}_{Go}, which is obtained by introducing the Go_x variables and updating the abstract transitions.

The *from-integer* GOABSTRACTION *time transition* is analogous to the from-integer abstract time transition, but it also updates the Go_x variables. If for all clock variables x such that $I_x < M_x$ and $F_x = \alpha$ the corresponding variable Go_x is *false*, then the Go_x variables of all clocks are set to *true*; otherwise the Go_x variables of the clocks whose abstract fractional order variable is equal to 0 are set to *false* and all other Go_x variables are left unchanged. The *to-integer* GOABSTRACTION *time transition* is analogous to the to-integer abstract time transition, but only clock variables whose Go_x variable is *true* can be updated by this transition.

Given a timed automaton A, the result of GOABSTRACTION is the GOABSTRACTION *timed system* A^{α}_{Go}, a system made of two asynchronous processes and containing, for each clock variable x, three discrete variables I_x, F^{α}_x, and Go_x. The first process, called the GOABSTRACTION *discrete-transition process* has the same locations and transitions as the original timed automaton, where clock constraints are mapped into expressions over I_x and F^{α}_x and reset sets are mapped into setting I_x and F^{α}_x to zero and Go_x to *true*. The second process, called the GOABSTRACTION *time-transition process*, defines the from-integer and to-integer GOABSTRACTION time transitions, and the GOABSTRACTION reset transition.

Theorem 14 (GOABSTRACTION **Preservation)** *Given a timed automaton A, the* GOABSTRACTION timed system A^{α}_{Go} *is an over-approximation of the discrete timed system* A^d*, i.e., every trace of* A^d *corresponds to an equivalent trace of* A^{α}_{Go}.

Moreover, the GOABSTRACTION timed system is more precise than the abstract time system defined above, that is:

Theorem 15 (Refined Abstraction) *Given a timed automaton A, the* abstract timed system A^{α} *is an over-approximation of the* GOABSTRACTION timed system A^{α}_{Go}*, i.e., every trace of* A^{α}_{Go} *corresponds to an equivalent trace of* A^{α}.

5 Experimental Results

In this section, we give some preliminary experimental results that we obtained by applying GOABSTRACTION to Fischer's mutual exclusion protocol.

This protocol guarantees mutual exclusion by imposing minimum and maximum delays for the execution of some statements. We modeled such delays by means of clock constraints in the timed automaton.

We model checked the protocol using Cadence SMV both as a symbolic model checker and a bounded model checker. The results for symbolic model checking are presented in Table 1. The first column shows the value of the timing parameter k, a parameter of the protocol. The second and third columns report the time required by SMV to perform the verification. The model used for the second column corresponds to the discrete timed system A^d (cf. Section 3) and the one used for the third column corresponds to the GOABSTRACTION timed system A^{α}_{Go} (cf. Section 4). In both cases, SMV was able to verify mutual exclusion, which demonstrates that GOABSTRACTION is precise enough to verify the property. Moreover, by using GOABSTRACTION, we were able to reduce the running time of the model checker by an order of magnitude.

Table 2 shows the results obtained by performing bounded model checking on the same models. Since bounded model checking is mostly aimed at detecting property violations, instead of checking for mutual exclusion, we checked if one of the processes is unable to reach the critical section. Since the protocol is correct, every process is guaranteed to eventually reach the critical section and the model checker reports a counterexample. The first column in the table contains the value of the timing parameter k. The next two columns contain three values: the running time and the depth l at which a valid counterexample was found, and the running time of the verification for depth $l - 1$, at which no error can be found. As it can be seen from the results, GOABSTRACTION has the side effect of reducing the depth at which an error can be detected: this is because all the intermediate steps needed to increment the fractional order variables of the different clocks are removed by the abstraction. Moreover, the running times are reduced again by one order of magnitude.

Table 1 Fischer's protocol with symbolic model checking for 4 nodes

k	Discrete	Go
2	28.1 s	4.8 s
3	82.5 s	22.5 s
4	175.3 s	24.3 s
5	355.6 s	43.6 s
6	728.1 s	48.8 s

Table 2 Bounded model checking applied to Fischer's protocol with 6 nodes

k	Discrete			Go		
2	100 s	$l = 25$	[326 s]	19 s	$l = 13$	[16 s]
3	450 s	$l = 32$	[617 s]	50 s	$l = 16$	[57 s]
4	969 s	$l = 38$	[2500 s]	61 s	$l = 19$	[71 s]
5	1200 s	$l = 46$	[1605 s]	137 s	$l = 22$	[118 s]
6	1800 s	$l = 54$	[3115 s]	316 s	$l = 28$	[347 s]

While these are only preliminary results, they show how, in this case, GOABSTRACTION is precise enough to prove interesting properties, and it is effective in reducing verification time.

6 Conclusions and Future Work

We described an implicit representation of the region automaton that can be used to perform verification of real-time systems using existing state-of-the-art model checking tools. Since the size of the region automaton can be exponential in the number of clock variables, we introduced GOABSTRACTION, a new abstraction technique that, by making use of auxiliary variables, is precise enough to preserve interesting properties of real-time systems. We demonstrated this technique on a typical real-time example.

In our experiments, we manually checked whether a counterexample was spurious. However, this process can be automated, and we would like to do so in future work. While GOABSTRACTION was sufficient for the example we considered, we would like to develop a counterexample-guided abstraction/refinement framework for timed automata based on GOABSTRACTION.

Moreover, we would like to develop additional techniques for the verification of real-time systems based on the representation we presented. Specifically, we would like to develop additional abstractions that can be used to address the verification of real-time properties of large systems.

Acknowledgements This research was sponsored by the National Science Foundation under grant nos. CNS-0411152, CCF-0429120, CCR-0121547, and CCR-0098072, the US Army Research Office under grant no. DAAD19-01-1-0485, the Office of Naval Research under grant no. N00014-01-1-0796, the Defense Advanced Research Projects Agency under subcontract no. SA423679952, the General Motors Corporation, and the Semiconductor Research Corporation. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government, or any other entity.

References

1. Rajeev Alur, Costas Courcoubetis, and David L. Dill. Model-Checking for Real-Time Systems. In *Proc. of the 5th Annual IEEE Symposium on Logic in Computer Science*, 1990.
2. Rajeev Alur and David L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126(2):183–235, April 1994.
3. Thomas Ball and Sriram K. Rajamani. Automatically Validating Temporal Safety Properties of Interfaces. In *Proc. of the 8th International SPIN Workshop*, 2001.
4. Dirk Beyer, Claus Lewerentz, and Andreas Noack. Rabbit: A Tool for BDD-Based Verification of Real-Time Systems. In *Proc. of the 15th International Conference on Computer Aided Verification (CAV)*, 2003.

5. Marius Bozga, Oded Maler, and Stavros Tripakis. Efficient Verification of Timed Automata Using Dense and Discrete Time Semantics. In *Proc. of 10th Conference on Correct Hardware Design and Verification Methods (CHARME)*, 1999.
6. Edmund M. Clarke, Orna Grumberg, and David E. Long. Model Checking and Abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.
7. David Dill. Timing Assumptions and Verification of Finite-Sate Concurrent Systems. In *Proc. of the Workshop on Automatic Verification Methods for Finite State Systems*, 1989.
8. Aleks Gõllü, Anuj Puri, and Pravin Varaiya. Discretization of Timed Automata. In *Proc. of the 33rd IEEE Conference on Decision and Control*, 1994.
9. Thomas A. Henzinger and Orna Kupferman. From Quantity to Quality. In *Proc. of International Workshop on Hybrid and Real-Time Systems (HART)*, 1997.
10. Thomas A. Henzinger, Zohar Manna, and Amir Pnueli. What Good Are Digital Clocks? In *Proc. of the 19th International Colloquium on Automata, Languages and Programming*, 1992.
11. Leslie Lamport. Real-Time Model Checking is Really Simple. In *Proc. of 13th Conference on Correct Hardware Design and Verification Methods (CHARME)*, 2005.
12. Kim G. Larsen, Paul Pettersson, and Wang Yi. Compositional and Symbolic Model-Checking of Real-Time Systems. In *Proc. of the 16th IEEE Real-Time Systems Symposium*, 1995.
13. Jesper Møller, Henrik Hulgaard, and Henrik Reif Andersen. Symbolic model checking of timed guarded commands using difference decision diagrams. *Journal of Login and Algebraic Programming*, 52–53:52–57, July–August 2002.
14. M. Oliver Möller, Harald Rueß, and Maria Sorea. Predicate Abstraction for Dense Real-Time Systems. In *Proc. of the Workshop on Theory and Practice of Timed Systems*, 2002.
15. Maria Sorea. *Verification of Real-Time Systems through Lazy Approximations*. PhD thesis, University of Ulm, Germany, 2004.
16. Stavros Tripakis and Sergio Yovine. Analysis of Timed Systems Using Time-Abstracting Bisimulations. *Formal Methods in System Design*, 18(1):25–68, January 2001.
17. Farn Wang. Efficient Data Structure for Fully Symbolic Verification of Real-Time Software Systems. In *Proc. of the 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2000.
18. Farn Wang. Region Encoding Diagram for Fully Symbolic Verification of Real-Time Systems. In *Proc. of the 20th Annual International Computer Software and Applications Conference*, 2000.
19. Farn Wang. RED: Model-Checker for Timed Automata with Clock-Restriction Diagram. In *Proc. of Workshop on Real-Time Tools*, 2001.
20. Farn Wang. Symbolic Verification of Complex Real-Time Systems with Clock-Restriction Diagram. In *Proc. of the 21st International Conference on Formal Techniques for Networked and Distributed Systems*, 2001.
21. Sergio Yovine. KRONOS: a verification tool for real-time systems. *International Journal on Software Tools for Technology Transfer*, 1(1-2):123–133, December 1997.

SCADE: Synchronous Design and Validation of Embedded Control Software

Gérard Berry

Abstract We describe the SCADE synchronous approach to model-based embedded software design, validation, and implementation for avionics, automotive, railway, and industry applications. SCADE specifications are based on block-diagrams and hierarchical state-machine graphical models with rigorous formal specifications. The SCADE KCG compiler is certified at the highest level of avionics certification, which suppresses the need for generated code unit testing. The SCADE tool has support for visual animation, test-suite coverage analysis, and formal verification. It has gateways to many other tools ranging from system-level specification to performance analysis.

Keywords: model-based development, synchronous languages, safety-critical applications

1 Introduction

We describe the SCADE synchronous methodology and toolset dedicated to model-based embedded software design, validation, and implementation for avionics, automotive, railway, and industry applications. The overall idea is to generate correct-by-construction embeddable implementation from high-level executable formal specifications, increasing software quality while decreasing design and validation costs. Since the specification is executable, it can be thoroughly simulated and verified before embedding. Since the implementation is automatically generated, there are no errors introduced at the implementation phase.

The synchronous methodology is rooted in 25 years of scientific research [3,5,10,11] and 20 years of successful industrial application. It is based upon a conceptual model of embedded computation backed by four strong technical cores: specific high-level rigorous graphical and textual languages, formal semantics, compiling algorithms for correct-by-construction implementation, and formal testing and verification techniques.

S. Ramesh and P. Sampath (eds.), *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*, 19–33.

SCADE evolved from the SAGA tool that was originally developed in 1986 by Schneider Electric [4] for nuclear plant safety systems, as a graphical version of the Lustre synchronous language of Caspi and Halbwachs [10]. It was then developed further to gradually replace the Airbus SAO internal tool for airborne software. It is now used by a large number of avionics, railway, industry, and automotive companies for fly-by-wire, engine control, brake control, safety control, power control, alarm handling, etc. SCADE embodies the KCG compiler from high-level designs to C that is itself certifiable at avionics DO-178B norm highest level A. TÜV certification is also available for automotive.

Source code development is based upon the Scade[1] graphical block-diagram notation familiar to control engineers, complemented by hierarchical Safe State Machines to describe state- or mode-oriented computations. These specification-level notations have precise mathematical semantics. Besides making software development more rigorous, they ease communication between engineers and between suppliers and customers. Functional verification is performed in two ways: conventional simulation techniques enhanced by graphical animation of the design and model coverage analysis, and formal verification of safety properties by model-checking or abstract interpretation. Functional verification is needed only at block-diagram level, since the embeddable C code generated by the certified KCG compiler is automatically correct and qualifiable.

The rest of the paper is organized as follows. Section 2 discusses concurrency and determinism issues for embedded systems and introduces the cycle-based computation model. Section 3 presents the Scade block-diagram and state-machine formalisms; Section 4 discusses the formal synchronous semantics. Section 5 presents the software design and validation flow associated with SCADE, as well as the associated tool ecosystem. We give a brief comparison with conventional OS-based techniques in Section 6. We conclude in Section 7.

2 Concurrency and Determinism of Embedded Software

Embedded software applications are very different from classical IT or networking applications. Instead of dealing with data files and asynchronous interrupts, they deal with the control of physical phenomena through specific sensor-actuator feedback loops and man–machine interfaces. Programs are mostly implementations of control algorithms. This calls for specific description paradigms close to the ones used in control engineering and systems engineering to design such algorithm: block diagrams for continuous control and state machines for discrete control. SCADE is directly based on these design paradigms. A comparison with principle with conventional techniques will be given in Section 6.

[1] We use SCADE for the development environment and Scade for the base specification formalisms.

2.1 The Need for Concurrency

Concurrency is essential for all embedded applications. Control algorithms are most often built by assembling basic elements: samplers, integrators, filters, comparators, state machines, etc. These concurrent elements communicate by exchanging information on a time- or event-trigger basis. Here, concurrency means cooperation. This contrasts with competition-based concurrency found in operating systems or thread-based applications where concurrent processes or threads compete to access and utilize resources.

2.2 The Need for Determinism and Predictability

Functional determinism is a key requirement of most embedded applications. A system is deterministic if it always reacts in the same way to the same inputs occurring with the same timing. On the contrary, a non-deterministic system is allowed to react in different ways to the same inputs, actual reaction depending on internal choices or computation timings. It is obvious that determinism is a must to control a car or a plane: the car should not decide by itself to go right or left. The same applies to man–machine interface or alarm handling.

Of course, determinism may not be a relevant requirement for other application types. An Internet connection naturally behaves in a non-deterministic way, and there is nothing wrong about that. In the same way, a car entertainment system may have some local non-deterministic behavior. But one does not control a car with an Internet-like infrastructure. This is why Scade specifications are deterministic by construction, the SCADE tools preserving determinism all along the specification-to-implementation chain.

On the implementation side, performance predictability is key to ensure functional determinism. In particular, one must ensure that internal computation timings cannot interfere with the functional timings of the control algorithm proper. Predictability is never an easy subject, in particular because of uncertainty due to caching and speculation optimization in recent microprocessors. But, of course, any form of added unessential non-determinism makes it even harder. SCADE achieves predictability by generating simple sequential code from concurrent specification using techniques described below.

2.3 The Cycle-Based Concurrent Computation Model

Cycle-based computation used by SCADE is a Folk model introduced long ago in many industrial designs to deal with embedded computation, but largely ignored by mainstream computer science. It consists of performing a continuous loop of the

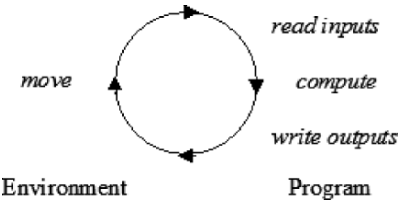


Fig. 1 Cycle-based computation

form pictured in Figure 1. In each cycle of this loop, there is a strict alternation between environment actions and program actions. Once the input sensors are read, the program starts computing the cycle outputs and its own memory state change. During that time, the program is blind to environment changes, ensuring interference-freedom. When the outputs are ready, or at a given time determined by a clock, the output values are fed back to the environment, and the program waits for the start of the next cycle.

Things are very much as in a two-game play: players play in strict alternation and each player does not interfere with the other player's thinking. The cycle-based model can also be viewed as a direct computer implementation of the ubiquitous sampling–actuating model of control engineering and signal processing.

In the implementation, there are several ways to control the cycle: time-triggered computation starts the cycle on a regular basis; polling consists of restarting the cycle as soon as it is over, and event-triggered computation consists of starting the cycle whenever some event occurs. This is application-dependent and will not be detailed further here.

2.4 Synchronous Communication and its Realization by Cycle Fusion

In the cycle-based model, concurrent components communicate by exchanging information during the cycle. An output computed by a component is instantly broadcasted to all concurrent components that want to read it. If one wants to implement delayed communication, one can insert elementary delay components that output at each cycle their input at previous cycle.

Consider the concurrent cyclic components 1 an 2 in Figure 2, which specifies a non-trivial dialogue pattern. The first component reads X and Z and writes Y and T, while the second component reads Y and writes Z. Communication is conceptually instantaneous: within a single cycle, Y is computed by 1 using X and communicated from 1 to 2, which causes Z to be computed by 2 and communicated back to 1. Communication is performed by a logically synchronous chain reaction, all enclosed within a single cycle representing a logical instant.

In a typical implementation, each component generates a straight-line code to execute the actions of its cycle. Communication between concurrent components

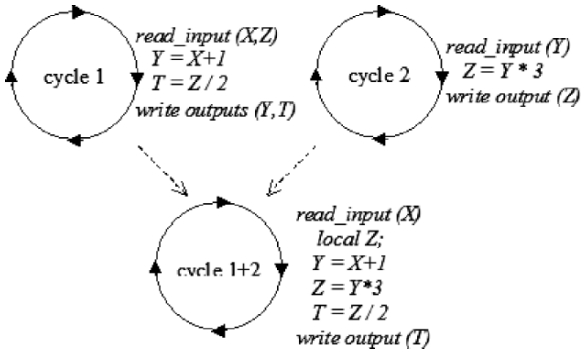


Fig. 2 Cycle fusion

is realized in a very simple way by cycle fusion, see Figure 2: one merges the statements generated by the concurrent blocks into a single straight-line code for the global cycle. Communication between the individual tasks is performed implicitly, by an adequate interleaving of the statements that respects inter-cycle communication dependencies. Notice that there is no overhead for communication, which is implemented using well-controlled shared variables without any context switching. For large hierarchical designs, cycle fusion goes across concurrent blocks and across the hierarchy, building a single sequential code from a network of components. Large-scale cycle fusion is unfeasible by hand but it is a relatively easy task for an automatic code generator. The SCADE compiler fully automates it and guarantees correct access to the shared memory.

Note that cycle fusion can be extended to support full separate compiling of blocks under some output-delay conditions not detailed here.

2.5 Determinism and Predictability of Cycle-based Applications

Determinism is respected by construction, whatever the number of concurrent processes may be. Performance predictability is made relatively simple by cycle fusion, since the generated code is purely sequential and does not imply context switches. It is limited only by the intrinsic non-determinism of modern microprocessors due to cache access and speculative execution.

Notice that the cycle-based computation model carefully distinguishes between logical concurrency and physical concurrency. The application is described in terms of locally cyclic and logically concurrent activities. Such logical concurrency makes the designer's work much easier by breaking complex tasks into simple ones that communicate in a simple way. However, the implementation uses a single process at run-time. The Scade model can be extended to support multi-process execution and physical distribution of multiple processors, see [9], but the SCADE tool does not support this yet.

3 The Scade Formalisms

For cycle-based design, SCADE provides the user with two familiar specification formalisms: block diagrams for continuous control and hierarchical Safe State Machines (SSMs) for discrete control. Both formalisms share the same view of a computation cycle and communicate in the same way.

3.1 Block Diagrams for Continuous Control

By continuous control, we mean sampling sensors at regular time intervals, performing signal processing computations on their values, and outputting values, computed for instance using possibly complex mathematical formulae. Sampled data is continuously subject to the same transformation. In Scade, continuous control is graphically specified using block diagrams such as the one depicted in Figure 3.

Boxes are called nodes. They are concurrent objects that compute outputs as functions of inputs, with possibly internal memory. Arrow between nodes denote communication channels also called flows. They can carry data of any type. All nodes share the same cycle and only communicate through the arrows. In a cycle, communication is conceptually instantaneous: a data element sent by a node reaches its destination in the same cycle. Primitive delay nodes such as the FBY nodes in Figure 3 are available to beak synchrony. At initial cycle, an FBY node ouputs its initial value. Then, at each cycle, it outputs the value of its input at previous cycles. Any loop in the block diagram must contain at least one delay element.

To add some flexibility in functioning modes control, Boolean flows can be used to control the activation of nodes. When a node N is controlled by an activation condition Boolean flow b, N is activated in a cycle only if b is true in the cycle.

Scade blocks are fully hierarchical: blocks at a description level can themselves be composed of smaller blocks interconnected by local flows. In Figure 3, the *ExternalConditions* block is hierarchical, and one can zoom into it with the editor. The same base cycle is shared by all the hierarchical components. A Boolean activation

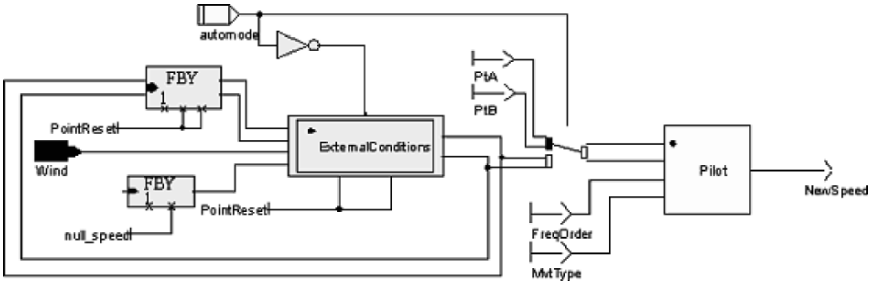


Fig. 3 Scade block diagram

condition for a hierarchical node recursively acts on all its sub-nodes. Scade block hierarchy is purely architectural. At compile-time a hierarchical block occurring in a higher-level block is simply replaced by its contents, conceptually removing its boundaries, and cycle fusion is peformed on the whole flattened result. Therefore, there is no need for complex and often partial hierarchical evaluation rules often found in other hierarchical block diagrams formalisms.

Hierarchy makes it possible to break design complexity using a divide-and-conquer approach and to easily reuse library blocks. There is no need to write complex blocks directly in C or ADA, since defining them hierarchically from smaller blocks is semantically better defined, much more readable, and just as efficient.

3.2 Safe State Machines for Discrete Control

By discrete control, we mean changing behavior according to external events originating either from discrete environment input events or from internal program events, e.g., value threshold detection. Discrete control is where the behavior keeps changing, a characteristics of modal human–machine interface, display control, alarm handling, complex functioning mode handling, or communication protocols.

Manually adding control Boolean flows and operations to block diagrams becomes rapidly messy when discrete control is non-trivial. One must resort to another well-known formalism: state machines. A standard flat state machine is pictured in Figure 4. As for a block diagram, it is composed of boxes, arrows, and names, but with a different meaning: boxes mean states, arrows mean transitions between states, and names denote signals exchanged with the environment. In a transition label I/O, I denotes a trigger signal and O denotes a result signal. If the start state of the transition is active and I occurs, the transition is fired and O is emitted.

Flat state machines have been very extensively studied in the last 50 years, and their theory is well-understood. However, in practice, they are not adequate even for medium-size applications, since their size and complexity tends to explode very

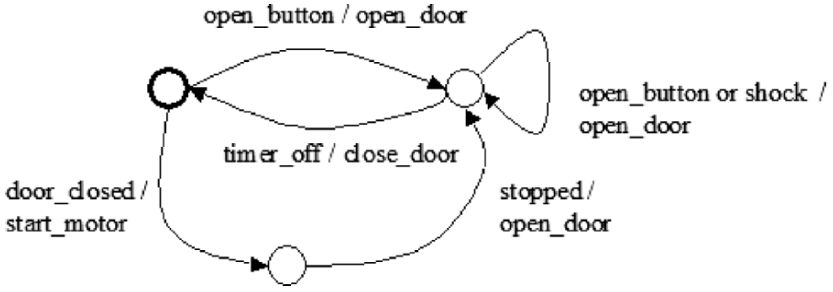


Fig. 4 Standard flat state machine diagram

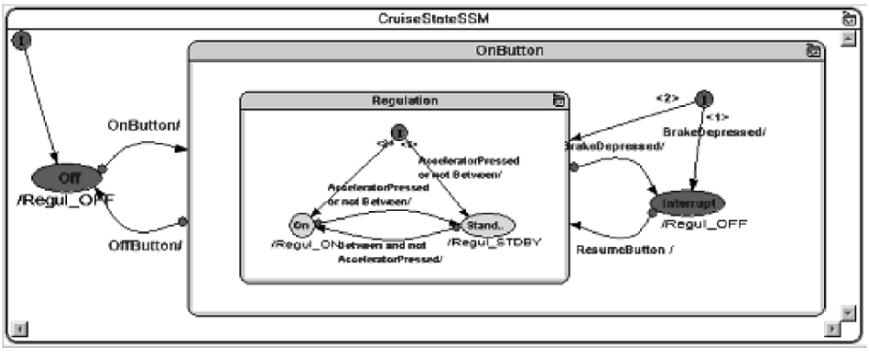


Fig. 5 An SSM hierachical state machine

rapidly. For this reason, richer concept of hierarchical state machines have been introduced, the initial one being Statecharts [12]. The Scade state machines are called Safe State Machines (SSMs), see Figure 5 for an example. These evolved from the Esterel programming language [5] and the SyncCharts synchronous statecharts model [2]. SSMs have been proved to be scalable to large control systems.

SSMs are hierarchical and concurrent. States can be either simple states or macrostates, themselves recursively containing a full SSM or a concurrent product of SSMs. When a macrostate is active, so is the SSMs it contains. When a macrostate is exited by taking a transition out of its boundary, the macrostate is exited and all the active SSMs it contains are preempted whichever state they were in. Concurrent state machines communicate by exchanging signals, which may be scoped to the macrostate that contains them.

The definition of SSMs carefully forbids dubious constructs found in other hierarchical state machine formalisms: transitions crossing macrostate boundaries, transitions that can be taken halfway and then backtracked, etc. These are non-modular, semantically hard to define, very hard to figure out, and therefore inappropriate for safety-critical designs. Their use is usually not recommended by most methodological guidelines anyway.

3.3 Mixed Continuous/Discrete Control

Large applications contain cooperating continuous and discrete control parts. Scade makes it possible to seamlessly couple both data flow and state machine styles. One can include SSMs into block-diagram designs to compute and propagate functioning modes. Then, the discrete signals to which an SSM reacts and which it sends back are simply transformed back-and-forth into Boolean data flows in the block diagram on a per-cycle basis. The computation models are fully compatible.

3.4 Scade 6: Full Integration of Block Diagrams and SSMs

The above desciption is that of Scade version 5. The new Scade 6 formalism currently under development [8] will provide the user with a full interplay between block diagrams and state machines. In Scade 6, a state in a state machine may contain either another state machine or a block diagram. Two block diagrams enclosed in two exclusive states may refer to the same flow, making it possible to implement the mode automata described in [13], where one can switch from a continous control computation to another one for the same flows according to Boolean conditions.

4 Formal Semantics

4.1 The Formal Synchronous Semantics

The formal theory of synchronous concurrency has been developed in the last 25 years. It extends the cycle-based intuitive model into a fully precise synchronous computation model, which gives a strong theoretical basis to the compilation and verification of Scade programs.

We briefly illustrate the synchronous semantics using a very simple example in continuous control. We refer the reader to [5, 10] for the formal development, more examples, and the handling of discrete control. Consider the following specification: given a discrete integer input flow I, output at each step the average A of the values received so far. In basic mathematics, one would use a discrete time index t and write the following system of iterative equations:

$$\begin{aligned} N_0 &= 1 \\ N_{t+1} &= N_t + 1 \\ T_0 &= I_0 \\ T_{t+1} &= (T_t + I_{t+1}) \\ A_t &= T_t / I_t \end{aligned}$$

Such an equation system is good enough for mathematical reasoning, but not for software engineering that requires much more precision. In mathematical notation, one never cares too much about what is allowed or disallowed for indices, because the reader is assumed to be a technically skilled human being. Making A_{t+1} depend on A_{t+2} instead of A_t is a syntactically legal mistake that any reader readily detects. But computers are definitely unskilled and they faithfully reproduce any mistake. Therefore, we need a precise programming formalism in which such mistakes can

be detected and rejected at compile-time. Lustre, the root textual language of Scade, was created for this purpose. In Lustre, the program is written below[2]:

```
node O (I : int) outputs (A : float);
var N : int, T : int;
let
    N = 1->(pre(N)+1);
    T = I -> pre(T) + I;
    A = T / N;
tel;
```

The identifiers I, N, T, and A denote data flows, which are infinite sequence of values. For instance, the single identifier A represents the whole cycle-based infinite sequence of inputs $A_0, A_1, \ldots, A_t, \ldots$, where t denotes the cycle index in the computation, i.e., the logical time. Operators such as addition add sequences componentwise, i.e., in a synchronous way: $A + B$ is $A_0 + B_0, A_1 + B_1, \ldots$ The `pre` delay operator delays a sequence by one cycle: `pre`(A) is the sequence $-, A_0, A_1, \ldots, A_t, \ldots$, where the first element is left uninitialized. The '->' initialization operator returns its left operand at first cycle and its right operand at further cycles. Since it increments its previous value at each cycle, the N symbol denotes the sequence $1, 2, 3, \ldots$, T denotes the accumulated sum of the input values, and A denotes the required sequence of average values. The semantics of Lustre simply defines the sequences corresponding to the variables as the solutions of the system of equations. Here, when seen as a complete flow, N is indeed equal to $1- >(\texttt{pre}(N) + 1)$. The Lustre and Scade formalisms and semantics extends to node activation conditions using a notion of derived clock, see [8, 10].

Using the well-defined Lustre operators, the informal system of equation has been transformed into a fully rigorous program. By construction, there is no way to refer to N_{t+1} instead of N_{t-1}, since there is no operator returning a future value at any given instant.

4.2 Logical vs. Physical Time

In the synchronous approach, one counts logical time only in terms of I/O cycles. Synchrony simply states that events occurring in the program are viewed as logically simultaneous if and only if they occur in the same cycle. One only distinguishes between computations occurring in the same cycle and computations occurring in successive cycles. Therefore, at Scade specification level, the physical time it takes to perform an addition or a division is ignored. This is a basic separation of concerns principle: high-level specifications need not care early on about performance-related issues. However, computing on if physical time is required by the application, one can deal with it using an extra specific extra input.

[2] The equivalent Scade graphical program will not be pictured here. The SCADE compiler would actually translate it into the above Lustre textual form.

5 The SCADE Application Development Flow

5.1 The SCADE Y Development Cycle

The classical software development cycle is called the V cycle. Development flows down from systems requirements to embedded code, the lower tip of the V, while validation flows up from embedded code unitary tests to system-level functional tests. Three steps are particularly difficult and expensive in this cycle: the precise specification of software requirements in a specification language, their precise coding in an executable language, and the low-level testing phase, which is usually the most costly. For embedded software development, the SCADE process and tools help in four ways:

- At the specification level, the transition from mathematical simulation tools to fully precise programs suited for a qualifiable software flow is much more direct than with classical executable language hand-coding, thanks to he native block diagrams and state machine formalisms. This shortens the requirement-to-specification phase.
- Embeddable C code is automatically generated from Scade descriptions by the KCG compiler. For avionics, KCG is qualifiable at highest level A w.r.t. DO-178B guidelines. For automotive, KCG is certified by the TÜV Sud authority at SIL 3 level of the IEC 61508 standard and valid for the development of software up to SIL 4. Because of this, the object code can be considered correct-by-construction w.r.t. the source specification since the code generator itself is qualified with the very same process as the full application. The need for C-level unit testing vanishes.
- Since the Scade model is executable, functional verification can be performed earlier and better. This will be detailed below.
- Because of the intrinsic performance predictability of code generation by cycle-fusion, performance validation is also made easier. Abstract-interpretation based performance evaluation tools are very useful there, see [1].

Altogether, the V cycle is transformed into the Y where the junction between specification and implementation is done at Scade specification level instead of C code level. The thin bottom of the Y represents certified code generation, now certified to be correct.

5.2 Model Validation

Functional validation of an embedded system consist in checking that the system fulfills its requirements. Validation checks can be dynamic or static, as detailed below.

5.3 Dynamic Checks and Coverage Analysis

Dynamic checks consist in test-suite based functional verification. A key issues is to build an appropriate test base, providing a large set of model inputs with minimal redundancy. This very notion is not easy to define. One usually uses various coverages criteria to measure how much a test base stresses a model and how well it describes the possible input cases. SCADE uses an elaborate notion of model coverage, which includes exercising conditional nodes, reaching bounds on operators, etc. (See also the classical MCDC coverage requirements for Boolean expression covering [7].) Generating the test suites can itself be difficult. It can be done either manually or by extracting model boundary inputs from system-level simulations.

5.4 Static Checks

Static compile-time checks consist in basic type-checking augmented by dead code detection and block diagram connections checking: absence of unconnected I/O pins and absence of cyclic data dependencies.

5.5 Formal Verification

Assertion-based verification performs symbolic model-checking[3] to verify the validity of user-provided temporal assertions about program behavior. Assertions can either be derived from application requirements or correspond to self-consistency defensive programming checks developed during the software design phase. They are expressed in the Scade formalisms, technically as Boolean flows that should never become false. Thus, the user does not need to learn specific property-definition languages to use the verifier. Good examples of properties to show by model-checking are *regulation is active if in on state and if speed lies between 30 and 130 km/h* or *the elevator never travels with the door open*. Counter-examples for false properties are automatically generated. Notice that assertion-based verification is now routine in the hardware field. Its extension to cycle-based designs is natural since these are akin to “software circuits”.

Abstract-interpretation based model-checking checks for the absence of run-time arithmetic exceptions or array out-of-bounds access, see [6]. It is automatic and does not require the writing of assertions. It has been used successfully on very large avionics projects.

Formal verification techniques complement human testing abilities very well. In particular, they are very useful in finding nasty bugs that escape conventional testing

[3] SCADE uses the Prover plug-in verification engine from Prover Technologies.

but do show up in production systems. We believe that formal verification engines will continue making constant progress in the future and will become among the most efficient anti-bug weapons.

5.6 The SCADE Automotive Ecosystem

A tool never solves a problem by itself. Therefore, SCADE is coupled with many other tools acting in the systems design or software engineering areas. Designs can be imported from prototypes written in mathematical simulation environments using a semi-automatic importer. UML specifications can be linked to SCADE designs using gateways. Systems requirement are traced in the SCADE design using links with requirement management tools. Documentation is automatically generated from Scade designs.

The C code generated by SCADE for automotive applications is platform independent and MISRA compliant, which is essential for automotive applications. It only uses a small subset of C, with no dynamic memory allocation, no pointer arithmetic, and no loop, callable through a very simple API. for specific execution platforms, SCADE extends the API by providing a customizable wrapping technology that allows a straightforward integration in any target environment: wrapping to OSEK tasks or to popular RTOS tasks are available. Currently, within the AUTOSAR consortium, the generation of AUTOSAR compliant Basic Software Modules and Software Components with SCADE is studied. This includes the compliance with the merging automotive standard ISO 26262.

SCADE embodies other software engineering tools that intend to make the development flow as smooth and safe as possible. The SCADE implementer tool makes it possible to finely control the fixed-point implementation of numerical computations for processors that do not support floating-point. Processor-dependent C compilers are checked to adequately compile the code generated by SCADE using a compiler verification kit that systematically compiles and checks all possible generated C patterns.

6 Comparison with Operating-Systems Based Designs

The other prominent model for embedded control is rooted in computer engineering tradition. It consists of writing sequential tasks for individual computations and using an operating system (OS) to schedule and run the tasks according to various criteria. The advantage is to rely on well-tested and robust off-the-shelf operating systems or language run-times. However, correctness issues become very application-dependent instead of being solved once for all by the programming formalism. The key issues are how the tasks are scheduled and how memory accesses are controlled.

Preemptive dynamic scheduling solves the problem at run-time in a generic way. However, it is a competitive concurrency model where tasks compete for resources (processor cycles, I/O, etc.). This interference-based model introduces a high level of non-determinism and correctness is difficult to ensure. Shared memory accesses have to be controlled by semaphores or similar devices, known to be deadlock-eager and hard to check. Another potentially nasty problem is priority inversion, where a low-priority task permanently takes precedence over high-priority ones. For a specific system, one can show application-level determinism, predictability, and scheduling safety using fancy analysis techniques, but this is never simple.

To improve on this and provide a safer view of dynamic tasking, higher-level rendezvous concurrency primitives have been included conventional languages such as ADA. However, they still rely on OS-like mechanisms and consistency is equally challenging.

Another well-known technique is fixed static scheduling, using algorithms based on task durations, on deadlines, on priorities, etc. Tasks can be either preemptible by other tasks or non-preemptible. This works well for a relatively small number of components, with a reasonable preservation of determinism. However, compared to cycle-based design, static scheduling is more difficult to organize, hard to scale to large applications, and very sensitive to specification changes.

Altogether, the tasking model does not help the programmer in conceptualizing the problem. There is no consistent way to go from a V to a Y cycle.

Of course, cycle-based computation does not rule out the need for basic OS functions. An embedded OS is still needed to perform low-level functions such as communication with sensors and actuators through drivers. But this is far less complicated than full tasking, and the main cycle code remains deterministic and predictable. Notice that a little amount of non-determinism in reading sensor values or driving actuators at cycle boundaries may remain without danger: all robust control algorithms do tolerate slight variations in the actual sampling or actuating timing.

7 Conclusion

We have presented the SCADE methodology and toolset, which are based on the synchronous computation model for embedded control software. SCADE addresses the design flow from precise specification to embedded code generation. It is used in major industrial programs to generate qualifiable implementation code from high-level block diagrams and state machines familiar to control engineers. This implies a dramatic cost reduction in one of the most difficult and error-prone part of systems development cycle. The use of SCADE also makes the upper and lower part of the whole cycle easier: the input formalism is close to classical notations used in high-level modeling, while simplicity of the generated code makes verification and implementation performance analysis simpler. SCADE is widely used for avionics, and is being used for automotive applications such as braking systems, suspension systems, entertainment systems, alarm systems, etc.

The mathematical model of synchronous systems on which SCADE is based is instrumental. It guides the user in the specification process, strongly grounds program semantics, drives compiler development and certification, and makes formal verification of programs possible.

Acknowledgements The author thanks the various people involved in the design, development, and usage of SCADE: P. Caspi and N. Halbwachs who created Lustre, J.-L. Bergerand and E. Pilaud who created SAGA, C. André who created SyncCharts, F.-X. Dormoy who developed SCADE, J.-L. Colaço who is the main Scade semantics and compiler architect, and the whole SCADE team.

References

1. M. Alt, C. Ferdinand, F. Martin, and R. Wilhelm. Cache behavior prediction by abstract interpretation. In *In SAS'96, Static Analysis Symposium, LNCS 1145*, pages 52–66. Springer, 1996.
2. C. André. Representation and analysis of reactive behaviors: A synchronous approach. In *Proc. CESA'96, IEEE-SMC, Lille, France*, 1996.
3. Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, January 2003.
4. J.L. Bergerand and E. Pilaud. Saga: A software development environment for dependability in automatic control. In *Proc. Safecomp'88*. Pergamon Press, 1988.
5. Gérard Berry. The foundations of Esterel. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.
6. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *In PLDI 2003 ACM SIGPLAN SIGSOFT Conference on Programming Language Design and Implementation, San Diego, California, USA*, pages 196–207, 2003.
7. J.J. Chilenski and S.P. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, 9(5):193–200, September 1994.
8. J.-L. Colaço, B. Pagano, and M. Pouzet. A conservative extension of synchronous data-flow with state machines. In *Proc. Emsoft'05, New Jersey, USA*, 2005.
9. A. Girault. A survey of automatic distribution method for synchronous programs. In F. Maraninchi, M. Pouzet, and V. Roy, editors, *International Workshop on Synchronous Languages, Applications and Programs, SLAP'05*, ENTCS, Edinburgh, UK, April 2005. Elsevier Science.
10. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. In *Proceedings of the IEEE*, volume 79(9), pages 1305–1320, 1991.
11. Nicolas Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, 1993.
12. David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8, 1987.
13. F. Maraninchi and Y. Rémond. Mode automata: A new domain-specific construct for the development of safe critical systems. *Science of Computer Programming*, pages 219–254, 2003.

Model-Based Development of Embedded Systems: The SysWeaver Approach

Raj Rajkumar

Abstract Model-based development of embedded real-time systems is aimed at elevating the level of abstraction at which these systems are designed, analyzed, validated, coded, and tested. The use of a coherent multi-dimensional model across all development phases can enable model-based design to generate systems that are correct by construction. However, current code generation capabilities are usually limited to uni-processor targets and to a limited range of operating environments. SysWeaver (previously called "Time Weaver") is a model-based development tool that includes a flexible "syscode" generation scheme for distributed real-time systems that can be easily tailored to a wide range of target platforms. We present our work on creating an interoperable toolchain to automatically generate complete run-time code using models. The toolchain includes a simulation tool (Matlab) and its code generator (Embedded Coder) along with SysWeaver. In this chain, the functional aspects of the system are specified in Simulink, Matlab's modeling language, and translated into a SysWeaver model to be enhanced with timing information, the target hardware model and its communication dependencies. The final run-time code is then generated, automatically integrating the functional code generated with Embedded Coder and SysWeaver's syscode. This syscode includes OS interfacing and network communication code with predictable timing behavior that can be verified at design time. Experiments with multi-node targets with end-to-end timing constraints in an automotive system show that many aspects of syscode and functional code generation can be automated.[1]

Keywords: couplers, embedded, real-time, semantic dimension, semantic separation, software-through-models

1 Introduction

Automated responses to environmental changes are affected by embedded real-time systems, with responses ranging from simple data logging to distributed or hierarchical control. These systems must not only satisfy "functional" properties to be of

[1] This paper is adapted from aspects of work reported in [29–31]. It is supported in part by DARPA, in part by the US Air Force, and in part by Bosch Corporation.

S. Ramesh and P. Sampath (eds.), *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*, 35–46.

benefit to its end-users, but are also subject to requirements such as timeliness, Quality of Service (QoS), and reliability. Examples of these systems span a wide range and include:

(a) cruise control systems that need to sample the vehicle speed at regular intervals;
(b) antilock Braking Systems (ABS) whose components may be replicated to tolerate the failure of one or more of the replicas; and
(c) videoconferencing systems that need to maintain a regular framerate at high data resolution.

Functional requirements are often specified and managed by domain experts. For example, control theorists and engineers are responsible for ensuring that closed-loop systems exhibit stable and responsive behavior. Requirements such as timing, replication, security, and jitter fall into the domain of computer engineers and computer scientists ("systems" folks). We refer to these requirements as "para-functional", since they augment the functional properties expected out of the system. Practical systems must satisfy both functional and para-functional properties. The criticality and cost constraints imposed on para-functional requirements can vary widely, as can the details of specific hardware and software platforms. Traditionally, the design of these systems often make hardwired assumptions about many aspects of the system including communication links, operating systems, programming languages, networking protocols, replication techniques, scheduling policies, and timing relationships. Furthermore, the functional requirements are intertwined into the underlying infrastructure that enables the functional code to execute. Any changes in any elements of the underlying infrastructure can cause major disruption not just to other aspects of the infrastructure but also to the correctness of the functional properties.

We believe that a model-based approach is required to elevate the design and development methodology for embedded real-time systems to a much higher level of abstraction that allows efficient and correct changes to any aspect of the system. What is the correct level of abstraction? We believe that it must be equivalent to the levels at which experts in a particular domain (such as signal processing, control theory, hybrid systems, real-time systems, embedded systems, fault-tolerance, security or safety-criticality) will communicate with one another. For example, control system experts will converse in terms of control laws, order of the system, stability, linear vs. linearized vs. non-linear control, rise-times, settling times, etc. Similarly, signal processing experts will discuss using terms like signal transforms, frequencies used, encoding or modulation schemes, noise levels, and signal strengths. It must be noted that all of these discussions abstract away from specifics of the underlying computing environment. Those are "implementation details" left for the computer experts. In the computing domain, however, real-time system experts exchange notions of resource utilization, hard or soft deadlines, scheduling approaches to be used (e.g. cyclic executives or fixed-priority preemptive scheduling), priority inversion, time resolution, real-time OS to be used, dynamic mode changes, and the (un)predictability of a communication medium. Reliability experts will discuss tradeoffs between active and passive replication approaches, the cost and practicality of using hardware vs. software replication, how to deal with error conditions, etc.

Security experts will explore operational models, applicable attack modes, authentication and authorization schemes to be used, encryption schemes to be adopted, and performance overheads that may be acceptable.

2 Related Work

Several efforts have attempted to address the embedded software development problem and how to ensure high assurance of these systems typically by the use of "correct-by-construction" approaches. We now briefly present some of the key ones and highlight their limitations. Wang and Shin [27] developed an architecture where components are constructed out of functions coordinated by a control logic driver and service protocols. Even though this scheme provides an interesting approach to separate reusable parts, it does not support the need for the independent evolution of para-functional aspects.

Stewart et al. [23] developed an architecture based on port-based objects where the integration of modules is done through state variables. Updates to the state variables are controlled by their plumbing infrastructure. This effort defined an important milestone in the development of software components for embedded systems. However, it hardwires the communication semantics narrowing applicability significantly. In addition, because it has a fixed mapping between components and processes, it hinders reusability across different para-functional requirements. Finally, this framework is tied to a specific inter-module communication mechanism and run-time infrastructure, diminishing its opportunities for reuse across different hardware.

Agha et al. developed an architecture based on active objects called *actors* [2, 18]. Actors interact through events that are buffered at reception and concurrency constraints are delegated to a middleware layer. This architecture is limited to a fixed arrangement of only active components and hence is not appropriate for reuse across different para-functional requirements. In addition, it requires specific middleware, hindering potential optimization across different platforms.

MetaH [10, 26] is an architecture and toolset for developing embedded real-time systems that can generate code for its own specific run-time layer. It also provides a port-based objects model. Ports are used to communicate data in a state variable fashion similar to [23]. Events are separate entities that are used to change the control flow of the execution. The hardware is modeled in a hierarchical fashion and software entities are assigned to processors. The timing model of the final system is verified using rate-monotonic scheduling theory. This model has two important contributions. First, it enables the composition of a software system with an automatic verification of its timing properties. Secondly, it models the hardware platform. However, its software description does not separate functional and para-functional aspects preventing its reuse when para-functional requirements change. In addition, it provides a single inter-component communication mechanism that is not sufficient to address different optimizations. Finally, it relies on a run-time layer that adds complexity to the process scheduling.

Ptolemy [6,13] proposes a formal tool to synthesize embedded software. Its framework enables a hierarchical mixture of models of computation producing as a result a semantic description of the system. This research effort recognizes the need to have different abstractions for the different aspects that embedded system may face. However, it does not separate functional and para-functional aspects, instead it provides fixed mixtures of functional constructs with software system abstractions such as threads. Finally, this effort has not explored code generation techniques as a primary concern in the embedded system development process.

Our framework builds on concepts from previous work on software architecture [22] and, in particular, on efforts for real-time systems such as [5] and [14]. These efforts had studied the software structure and its relationships with para-functional properties.

Our framework is similar in some regards to the decoupling approach proposed by Aspect-Oriented Programming (AOP) [9] and the dimensional arrangement of aspects in Multi-dimensional Separation of Concerns [19, 24]. However, our framework enables hierarchical and reusable compositions along well-defined dimensions that are semantically orthogonal with one another.

Synchronous languages like Esterel [28] make several restrictive assumptions such as actions and messages happen instantaneously, do not deal with tasks with different scheduling policies, and do not help in constructing distributed computing applications. Their primary advantage, however, is that a *single model* written in the Esterel language is used not only to generate the code that runs on the target platform but also to validate the properties of the system. In SysWeaver, we consciously strive to maintain a single model for analysis and code generation, while capturing and analyzing the realities of the deployment platform including execution times, message transmission times, real-time operating systems, middleware, and scheduling policies.

3 The SysWeaver Approach

We have proposed a framework for developing software for embedded realtime systems through the use of models [30–32]. Our general philosophy is summarized next.

The SysWeaver approach is illustrated in Figure 1. Para-functional (sometimes called non-functional) and functional aspects of the system are modeled in SysWeaver and complementary interoperable tools (such as Simulink and StateCharts) respectively. These models are then exported as necessary to analysis, validation and verification tools where detailed **domain-specific analyses** take place. For example, a schedulability analysis verifies whether the system can meet its timing and resource constraints. Control system analysis, hybrid system analysis, model checking of system properties, application of test vectors in the modeling domain, etc. can be performed here. An iterative process may be carried out to

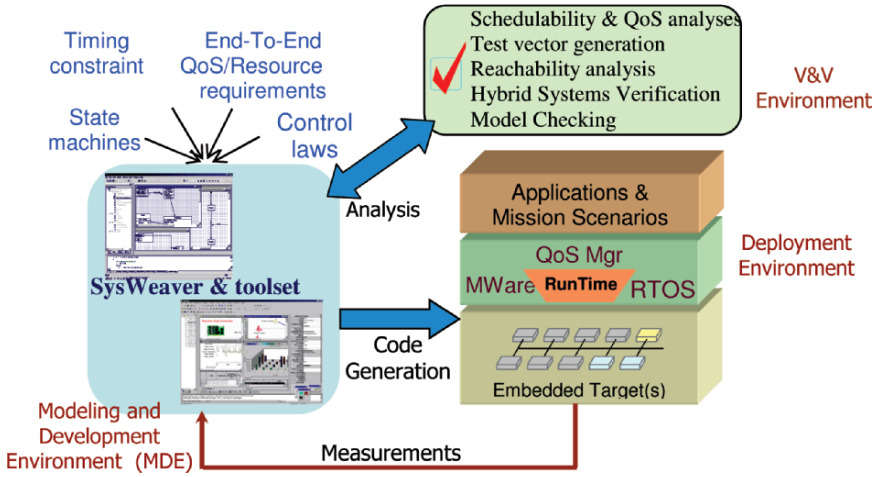


Fig. 1 The Modeling, validation, and code generation methodology using SysWeaver

ensure that the requirements specified by the models are satisfied. This is referred to as a **"Correct by Construction"** approach. Once desired properties (such as rise time, settling time, stability, QoS requirements, schedulability using budgeted values, property verification using model checkers and/or theorem provers) have been satisfied, code can be generated to a target platform in the deployment environment, which is itself defined in terms of hardware elements such as interconnected processors and network/bus links, the real-time operating systems (RTOSs) running on those nodes, the communication protocols and middleware services to be used among those nodes. A desired programming language can also be selected. The approach taken by Model Weaver in generating the *syscode* (also sometimes called "glue code") is summarized in [32].

A functional behavioral tool (such as Simulink and Stateflow) is used to generate the functional behaviors (or can be manually coded) and then merged with the para-functional and deployment-specific code to create the executable images, one for each node in the target environment. Our practical experience in importing Simulink models *automatically*, and merging functional and para-functional code into integral and fully executable images is summarized in [30, 31].

As shown in Figure 2, a **component and model repository** is used to store reusable software components and models for use in subsequent systems. Components can be hierarchical in nature, each potentially comprising of a group of (recursively hierarchical) components. This recursive encapsulation enables us to think of even "**systems as components**", and connections among such system-level components will facilitate the **construction of "systems of systems**". Figure 2 also illustrates the interoperability among tools in the SysWeaver environment.

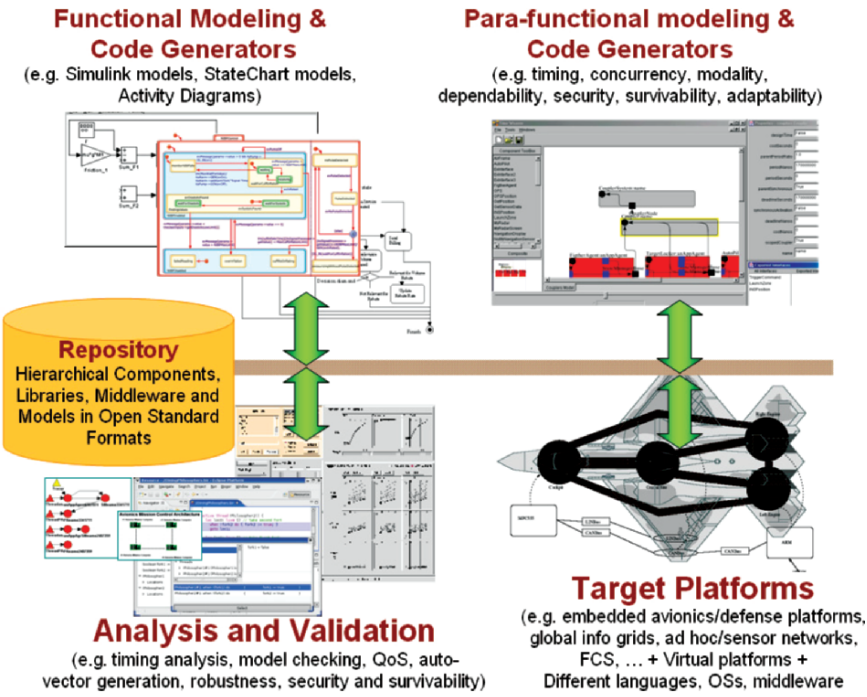


Fig. 2 Interoperability among modeling and analysis tools in the SysWeaver environment, shown with a model and component repository

3.1 Benefits of SysWeaver

Our work is aimed at significantly improving the assurance of embedded software by explicitly capturing both the functional and para-functional aspects of the system within a single coherent (multi-faceted) model, and verifying that they are satisfied. Our goal is not just to detect and correct errors before deploying embedded systems, but to be able to do so at earlier stages of the design and development phase. This would lead to high-assurance but still cost-efficient software.

This positive (and fortunate!) confluence arises from three facts:

(a) capturing high-level models of the system allows system properties to be verified at appropriately higher levels of abstractions;
(b) automated code generators are used to generate the actual code running on the target from these models, brought to play (much like programming languages are *compiled* to the machine language of a target processor); and
(c) modeling different semantic aspects of embedded systems (such as timing, fault tolerance, concurrency, and modalities) separately both allows easier verification and reduces complexity.

Our framework separates para-functional properties into semantic dimensions (e.g. timing, event flow, concurrency, fault-tolerance, deployment) that can be modified independent of one another. The impact of changes in one dimension on the realization of other dimensions are automatically projected and managed. Platform dependencies are also captured separately, enabling a code-generation subsystem to reuse the same components across a wide range of heterogeneous platforms and applications. System components can be recursively composed or decomposed. An analyzable software structure is enforced such that the end-to-end timing behavior of the resulting system can be verified. It is also possible to study how SysWeaver can work with model-checking tools such as Prove It, SMV, Charon and SAL.

3.2 Separation of Concerns using Semantic Dimensions

SysWeaver is designed to meet the goals of building robust and analyzable large-scale real-time systems.

The complexity of large-scale embedded real-time systems is that the system needs to satisfy requirements along multiple dimensions such as logical functionality, timeliness, throughput, fault tolerance, security, etc. We strongly believe that these "concerns" must be modeled and analyzed separately at the higher levels of abstraction, while their interactions are automated and coordinated within a single consistent model. We call these concerns "Semantic dimensions" and argue that they contribute to significant reduction in managing system complexity.

Semantic dimensions are a description of an aspect of a system that can be represented and analyzed independent of another. Examples of these dimensions for embedded real-time systems are: functionality, timing, and fault-tolerance. Changes in one dimension, however, may impact the implementation of another dimension. We project such changes across dimensions as necessary. All such *projections* will be automated and happen without user intervention (Figure 3).

In our framework, we will explicitly identify a set of dimensions that separate and represent distinct aspects of embedded real-time systems software. These dimensions are:

- The **functional dimension** deals with the typical functional transformations often described in domain-specific notation (such as Simulink and StateCharts) or coded in popular languages such as C/C++.
- The **deployment dimension** deals with:
  - the description of the deployment platform (hardware, operating system, middleware, etc.);
  - the definition of the communication mechanisms across deployment entities (processes, processors, networks, etc.);
  - the assignment of components to deployment entities.
- The **timing dimension** deals with the relationship between the periods and deadlines of active components.

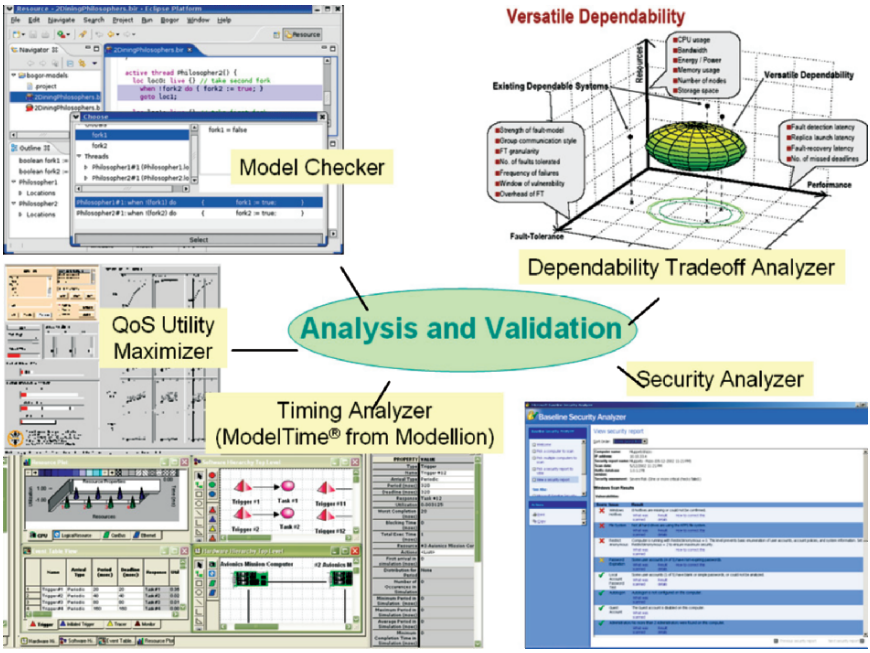


Fig. 3 A wide variety of target platforms using different programming languages, operating systems, middleware and communication protocols will be supported

- The **fault-tolerance dimension** deals with the construction to provide redundant computation.
- The **modality dimension** deals with constructions that enable the system to change its configuration over time to satisfy changing objectives.
- The **concurrency dimension** deals with enabling and synchronizing parallel activities.

3.3 Summary of Approach

Given the large number of potential error sources in embedded real-time systems, a single solution is unlikely to exist. Each of the failure conditions may require the use of a different solution. Some of these solutions may be amenable to formal analysis, others to simulation, and still others only to testing. We therefore recommend a multi-pronged approach that allows the application of a range of solutions within a coherent framework. Our approach consists of:

- A **design-time framework** with a supporting SysWeaver toolset enables functional analysis, timing analysis, fault injection, test vector generation, type matching, formal verification (using model checking) and code generation using a single

representation. Our innovative approach allows the system behavior along different semantic dimensions such as (logical) functionality, timing, fault-tolerance, modality, and concurrency to be captured separately but consistently. Once verified, code to be executed on the target platform can be generated directly from the same specification used for verification. Software modules will be assigned to processors and messages to network links taking into account a host of constraints including execution time and jitter constraints, energy constraints, integrity constraints, fault-tolerance constraints, and sensor/actuator constraints.

- SysWeaver also enables the **reuse of components**, each of which has an extensive list of multi-dimensional attributes along functionality, timeliness properties, target platform dependencies, and physical platform dependencies. A component must satisfy functionality, type signatures, timing, run-time needs, environmental dependencies, and platform assumptions before use. Our component framework will use rich extensions to traditional notions of port-based components. These extensions will include the explicit representation of concurrency and synchronization requirements, communication protocol needs, and customizable state management. Groups of components can be encapsulated recursively, to create sub-systems, systems, and eventually systems of systems. We will also study the use of pre-conditions and post-conditions to characterize components.
- Central to our objective of keeping a faithful model of the run-time image is the ability to generate executable "syscode". By syscode, we mean the code that combines the application code (functional) components together to form the final running system. Our code generation framework is composed of two main parts: an inter-component communication library and the generator of code that performs the inter-component connections and initializes the application. The code generator will also need to understand system calls to be used (for capabilities like task creation, mutex creation, and usage), language dependencies, communication protocol requirements along with the ability to integrate with functional component interfaces, and invoke middleware services. Support for fault-tolerance, security and modal operations must also be supported by the code generator.
- It is important to note that the code generator is *not* a single piece of code, with all generation capabilities encoded in a single subsystem. Components and couplers have a list of property-value pairs, and code generators will be required for special values (which choose a different programming language for example). A high-level code generator framework will traverse the model(s) defined, and appropriate lower-level code generators will be invoked. It is also useful to note that when a new target environment (language, OS, communication protocol) is defined, the code generator for that attribute is written once and can be reused multiple times.
- SysWeaver supports an **assessment framework** with capabilities not unlike those available in requirements tracking. Any mismatches between interfaces, types, platform dependencies, environmental dependencies that components have will

be tracked explicitly within SysWeaver. Formal (or informal) arguments that are used to validate the component (or operation such as type conversion) will be explicitly captured so that they will be forced to be re-considered when *any* aspects around the component change.

3.4 Supported Configurations

SysWeaver has been used to model applications in a wide range of environments including signal processing using software radios, electronic throttle control in automotive systems, distributed fault-tolerant systems, wireless sensor networks, and dynamic QoS adaptation. A variety of target operating systems including (real-time versions of) Linux, (Real-Time) Java virtual machines, uCOS-II, and OSEKWorks is supported. Support for several programming languages including C, C++, Java, XML, and Matlab are included. Communication links like CANbus and multiple communication protocols like TCP and UDP are also supported. SysWeaver can also generate timing models which can be exported to schedulability analysis tools such as TimeWiz. Simulink models can also be imported into the functional dimension of SysWeaver, which can then be enriched along the other semantic dimensions. Furthermore, SysWeaver can invoke Real-Time Workshop from MathWorks to generate code from the Simulink models, and integrate appropriate functional code with the system code generated by SysWeaver.

For configurations, platforms, and languages that are not currently supported, an appropriate back-end generator needs to be added. Once written, it can be used for all future models.

3.5 Summary

In summary, SysWeaver constitutes a comprehensive model-based framework for obtaining high degrees of robustness in embedded real-time systems. We are currently creating a new version of SysWeaver that will both be user-friendly and efficient in terms of performance. Applicability to a wide range of domains is also being studied in concert with multiple targeted users.

References

1. "Adaptive Real-Time Systems for Quality of Service Management", Technical Report, *European Union ARTIST Project*, March 2003. http://www.artist-embedded.org/.
2. G.A. Agha and W. Kim. Actors: A unifying model for parallel and distributed computing. *Journal of Systems Architecture*, 45:1263–1277, 1999.
3. B. Boehm, "Tutorial: Software Risk Management", Washington, DC, IEEE Computer Society Press.

4. Greg Bollela, Ben Brosgol, Peter Dibble, Steve Fur, James Gosling, David Hardin, Mark Turnbull, Rudy Belliardi, Doug Locke, Scott Robbins, Pratik Solanki, and Dionisio de Niz. *The Real-Time Specification for Java*. Addision-Wesley, 2001.
5. Kevin Bradley. *A framework for incorporating real-time analysis into system design processes*. Ph.D. Dissertation, Department of Electrical and Computer Engineering, Carnegie Mellon University, 1998.
6. J. Davis II. et al. *Overview of the Ptolemy project*. Technical Report M99/37, UC Berkeley, Dept EECS, 1999.
7. Dionisio de Niz and Raj Rajkumar. Chocolate: A reservation-based real-time java environment on windows/NT. In *Proceedings of the Real-Time Technology and Applications Symposium*, Washington D.C., May 2000.
8. P.H. Feiler, Bruce Lewis, Steve Vestal, "Improving Predictability in Embedded Real-Time Systems", Technical Report, CMU/SEI-2000-SR-011, Software Engineering Institute, December 2000.
9. C.C. Howell, "Building Dependable Systems: The Power of Negative Thinking", Tutorial given at 2002 International Conference on Dependable Systems & Networks, IEEE Computer Society.
10. J. Krueger, S. Vestal, and B. Lewis. Fitting the pieces together: system/software analysis and code integration using Meta-H. In *IEEE 17th Annual Digital Avionics Systems Conference*, November 1998.
11. K. Lieberherr. Demeter and Aspect-Oriented Programming. In *STJA'97 Conference*, September 1997.
12. Chris Lanfear and Steve Balaccco. The embedded software strategic market intelligence program 2001/2002—volume iv: Embedded operating systems, software development tools, design automation tools, and test automation tools, February 2002.
13. E.A. Lee. What's Ahead for Embedded Software? *Computer*, 1, 2000.
14. Jun Li, *Dependency tracking in real-time fault-tolerant systems*. Ph.D. Dissertation, Department of Electrical and Computer Engineering, Carnegie Mellon University, 2000.
15. Klein M.H. et al. *A Practitioners' Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*. Kluwer Academic Publishers, 1993.
16. Priya Narasimhan. Transparent Fault-Tolerance for CORBA. Technical report, Department of Electrical and Computer Engineering, University of California, Santa Barbara, 1999.
17. Network Reliability Interoperability Council V, Data Reporting and Analysis for Packet Switching, Final Report, Focus Group 2 Subcommittee 2.B2.
18. B. Nielsen and G. Agha. Towards reusable real-time objects. *Annals of Software Engineering*1999.
19. H. Ossher and P. Tarr. *Multi-dimensional separation of concerns in hyperspace*. Technical Report RC21452, IBM T.J. Watson Research Center, April 1999.
20. S.L. Pfleeger and C.C. Howell "The Case For Constructing an Effective and Efficient Assurance Argument Framework".
21. Lui Sha, Ragunathan Rajkumar, and John Lehoczky. Mode change protocols for priority-driven pre-emptive scheduling. In *The Real-Time Systems Symposium*, 1989.
22. Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.
23. D.B. Stewart, R.A.Volpe, and P.K. Khosla. Design of dynamically reconfigurable real-time software using port-based objects. *IEEE Transactions on Software Engineering*, 23, 1997.
24. Peri Tarr, Harold Ossher, William Harrison, and Jr. Stanley M. Sutton. N degrees of separation: multi-dimensional separation of concerns. In *International Conference on Software Engineering*, pages 107–119, 1999.
25. Jim Turley. Embedded Processors, January 2002.
26. Steve Vestal. Mode changes in a real-time architecture description language. In *International Workshop on Configurable Distributed Systems*, March 1994.

27. S. Wang and K.G. Shin. An architecture for embedded software integration using reusable components. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, San Jose, CA, 2000.
28. The Esterel Synchronous Language, Esterel Technologies, www.estereltechnologies.com.
29. Dionisio de Niz and Raj Rajkumar, "Partitioning Bin-Packing Algorithms for Distributed Real-Time Systems", Invited Paper, *International Journal of Embedded Systems*, Special Issue on Design and Verification of Real-Time Embedded Software. 2005.
30. Dionisio de Niz, *Modeling functional and para-functional concerns in embedded real-time systems*, Ph.D. Dissertation, Electrical and Computer Engineering. Carnegie Mellon University, April 2004.
31. Dionisio de Niz and Raj Rajkumar, "Glue Code Generation: Closing the Loophole in Model-based Development", 10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2004), *Workshop on Model-Driven Embedded Systems* (MDES 2004).
32. Dionisio de Niz and Ragunathan Rajkumar, "Time Weaver: A Software-Through-Models Framework For Embedded Real-Time Systems", *Language Compilers and Tools For Embedded Systems* (LCTES 2003).

Verification and Integration of Real-Time Control Software

Rajeev Alur

Abstract Realizing the potential of networked embedded control systems will be predicated upon our ability to produce *embedded software* that can effectively and safely harness the functionality of sensors and processors. Embedded software is different, and more demanding, than the typical programming applications in many ways. Modern programming languages abstract away from real time and resources, and do not provide adequate support for embedded applications. Consequently, current development of embedded software requires significant low-level manual effort for scheduling and component assembly. This is inherently error-prone, time-consuming, and platform-dependent. Consequently, developing novel programming and implementation methodology for synthesizing portable, predictable embedded software is an important challenge for networked control systems. In this abstract, we briefly discuss some of our efforts towards this goal.

Keywords: embedded, control software, formal verification, system integration

1 Formal Verification

To provide assurance guarantees for control software, a formal approach to design is appealing. Model-based design and formal verification have been successful in targeted applications such as microprocessor designs, and we believe that the same success is feasible in the domain of embedded control systems. There are multiple research challenges that need to be addressed to develop the model-based approach. We list some of them along with possible emerging directions:

Modeling The appropriate mathematical model for embedded software systems is *hybrid systems* that combines the traditional state-machine-based models for discrete control with classical differential- and algebraic-equations-based models for continuously evolving physical activities. Starting with *hybrid automata* [3], we have

S. Ramesh and P. Sampath (eds.), *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*, 47–49.

been developing compositional and mathematically rigorous modeling framework for hybrid systems. In recent years, we have developed a hierarchical modeling tool CHARON for modular description of concurrent hybrid systems [1], and have explored applications to automotive controllers, medical devices, and biological systems.

Model Checking Model checking tools can reveal design bugs at early stages by subjecting partial models for compatibility checks against specifications. Impressive progress in symbolic state-space exploration techniques has enhanced the applicability of model checking significantly. This has led to improved reliability for network protocols and device drivers. Extending model checking to allow exploration of state-spaces of hybrid systems has proved to be a challenging problem. Our early work on symbolic analysis of linear hybrid automata led to the tool HYTECH [3]. More recently, we have explored combining state-space exploration using polyhedra with predicate abstraction and counterexample guided abstraction refinement [2].

Software Generation Generating embedded software directly from high-level models, such as hybrid systems, is appealing, but challenging due to the wide gap between the two. In current practice, this gap is bridged with significant manual effort by exploiting the run-time support offered by operating systems for managing tasks and interrupts. A key challenge to systematic software synthesis from hybrid models is to ensure that one can infer properties of the software from the properties of the model, and this problem is receiving increasing attention from researchers [4].

2 System Integration

Contemporary software development emphasizes components with clearly specified APIs. A static API for a software component such as a Java library class consists of all the (public) methods, along with the types of input parameters and returned values, that the component supports. This promotes a clear separation between the specification of the component and its implementation. Such static APIs can be enforced using type systems. But while they are indispensable, these APIs offer only a partial solution to design bug-free systems as they do not capture constraints on resources, real-time guarantees, and other quality-of-service aspects. Consequently, they offer little assistance in "system" integration. This is an important issue not only for being able to derive system-level performance and correctness guarantees, but also for being able to assemble components in a cost effective manner.

Interfaces for Embedded Components The notion of an interface for a control device interacting with its physical environment and other devices must incorporate information about timing delays and continuous parameters such as threshold levels. Capturing the notion of quality-of-service abstractly, and having mechanisms that can enforce the adherence to interfaces as well as check compatibility between

components interfaces, is an emerging and challenging trend in embedded systems research. We believe that using time-triggered allocation can be a basis for analyzable, predictable, and analyzable automata-based interfaces between control designs and software implementations [6].

Quality Metrics for Control Implementations In the context of embedded control systems that interact with an environment, a variety of errors due to quantization, delays, and scheduling policies may generate executable code that does not faithfully implement the model-based design. The performance gap between the model-level semantics of proportional-integral (PI) controllers and their implementation-level semantics can be rigorously quantified if the controller implementation is executed on a predictable time-triggered architecture. Recent work on explicitly computing the impact of the implementation on overall system performance allows us to compare and partially order different implementations with various scheduling or timing characteristics [5].

References

1. R. Alur, T. Dang, J. Esposito, Y. Hur, F. Ivancic, V. Kumar, I. Lee, P. Mishra, G. Pappas, and O. Sokolsky. Hierarchical modeling and analysis of embedded systems. *Proceedings of the IEEE*, 91(1), 2003.
2. R. Alur, T. Dang, and F. Ivancic. Predicate abstraction for reachability analysis of hybrid systems. *ACM Transactions on Embedded Computing Systems*, 5(1):152–199, 2006.
3. R. Alur, T. Henzinger, and P.-H. Ho. Automatic symbolic verification of embedded systems. *IEEE Transactions on Software Engineering*, 22(3):181–201, 1996.
4. R. Alur, F. Ivancic, J. Kim, I. Lee, and O. Sokolsky. Generating embedded software from hierarchical hybrid models. In *Proceedings of the ACM Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 171–182, 2003.
5. T. Nghiem, G. Pappas, A. Girard, and R. Alur. Time-triggered implementations of dynamic controllers. In *Proceedings of the 6th Annual ACM Conference on Embedded Software (EMSOFT)*, pages 2–11, 2006.
6. G. Weiss and R. Alur. Automata based interfaces for control and scheduling. In *Proceedings of the 10th International Workshop on Hybrid Systems: Computation and Control (HSCC 2007)*. Springer, 2007.

Merge Algorithms for Intelligent Vehicles

Gurulingesh Raravi, Vipul Shingde, Krithi Ramamritham, and Jatin Bharadia

Abstract There is an increased concern towards the design and development of computer-controlled automotive applications to improve safety, reduce accidents, increase traffic flow, and enhance comfort for drivers. Automakers are trying to make vehicles more intelligent by embedding processors which can be used to implement Electronic and Control Software (ECS) for taking smart decisions on the road or assisting the driver in doing the same. These ECS applications are high-integrity, distributed and real-time in nature. Inter-Vehicle Communication and Road-Vehicle Communication (IVC/RVC) mechanisms will only add to this intelligence by enabling distributed implementation of these applications. Our work studies one such application, namely Automatic Merge Control System, which ensures safe vehicle maneuver in the region where two roads intersect. We have discussed two approaches for designing this system both aimed at minimizing the *Driving-Time-To-Intersection* (DTTI) of vehicles, subject to certain constraints for ensuring safety. We have (i) formulated this system as an optimization problem which can be solved using standard solvers and (ii) proposed an intuitive approach namely, Head of Lane (HoL) algorithm which incurs less computational overhead compared to optimization formulation. Simulations carried out using Matlab and C++ demonstrate that the proposed approaches ensure safe vehicle maneuvering at intersection regions. In this ongoing work, we are implementing the system on robotic vehicular platforms built in our lab.

Keywords: Automatic merge control, driving-time-to-intersection, area-of-interest, vehicle merge sequence, vehicle interference, continuous vehicle stream.

1 Introduction

It is believed that automation of vehicles will improve safety, reduce accidents, increase traffic flow, and enhance comfort for drivers. It is also believed that automation can relieve drivers from carrying out routine tasks during driving Vahidi and Eskandarian (2003). Automakers are trying to achieve automation by embedding more processors, known as Electronic Control Units (ECUs) and sensors into vehicles which help to enhance their intelligence. This processing power can be utilized effectively to make an automobile behave in a smart way, e.g., by sensing the surrounding environment and performing necessary computations on the captured data

S. Ramesh and P. Sampath (eds.), *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*, 51–65.

either to decide and give commands to carry out the necessary action or to assist the driver in taking decisions. In modern day automobiles, several critical vehicle functions such as vehicle dynamics, stability control and powertrain control, are handled by ECS applications.

Adaptive Cruise Control (ACC) is one such intelligent feature that automatically adjusts vehicle speed to *maintain the safe distance* from the vehicle moving ahead on the same lane (a.k.a. *leading vehicle*). When there is no vehicle ahead, it tries to maintain the safe speed set by the driver. Since ACC is a safety-enhancing feature it also has stringent requirements on the freshness of data items and completion time of the tasks. The design and development of centralized control for ACC with efficient real-time support is discussed in Raravi et al. (2006).

Sophisticated distributed control features having more intelligence and decision making capability like collision-avoidance, lane keeping, and by-wire systems are on the verge of becoming a reality. In all such applications, wireless communication provides the flexibility of having distributed control. A distributed control system brings in more computational capability and information which helps in making automobiles more intelligent. In this paper, we focus on one such distributed control application, namely Automatic Merge Control System which tries to ensure safe vehicle maneuver in a region where n roads intersect. To this end, we have (i) formulated an optimization problem with the objective to minimize the maximum *driving-time-to-intersection (DTTI)* (time taken by vehicles to reach the intersection region) subject to specific safety-related constraints and (ii) proposed *Head of Lane (HoL)* algorithm for achieving the same with less computational overhead compared to optimization formulation.

In this paper, terms *road* and *lane* are used interchangeably. The rest of the paper is organized as follows. Section 2 introduces *Automatic Merge Control System* and describes the problem in detail. The optimization function and constraints are formulated in Section 3. The HoL algorithm is described in Section 4. The results of simulation and Matlab-based evaluations are discussed in Section 6. Section 7 presents the related work followed by conclusions and future work.

2 Automatic Merge Control System

The Automatic Merge Control (AMC) System is a distributed intelligent control system that ensures safe vehicle maneuver at road intersections. The system ensures that no two vehicles coming from different roads collide or interfere at the intersection region. It ensures that the time taken by any two vehicles to reach the intersection region is separated by at least δ (which depends on the length of the intersection region and velocity of vehicles), by giving commands to adapt their velocities appropriately. In other words, it ensures that no two vehicles will be present in the intersection region at any given instant of time. This system involves: (i) determining the *Merge Sequence (MS)*, i.e., order in which vehicles cross the intersection region; (ii) ensuring safety at intersection region; and (iii) achieving an

optimization goal such as minimizing the maximum *DTTI* (time taken by a vehicle to reach the intersection region).

Our goal is to ensure safe vehicle maneuver at intersection regions which involves the above mentioned three subproblems.

We have made the following assumptions while formulating the optimization problem:

- An intelligent (communication + computation) infrastructure node is situated roadside near the intersection region. It performs all computations and determines the commands (acceleration, deceleration) to be given to each vehicle.
- A suitable communication infrastructure exists for vehicles and roadside infrastructure node to communicate with each other.
- Initially, all the vehicles are atleast S distance apart (safety distance) from their respective leading vehicle.
- Each vehicle has an intelligent control application which takes acceleration and time as input and ensures that the vehicle reaches the merge region in that time period by following the given acceleration.
- Only those vehicles which are inside the *Area of interest* (AoI) are part of the system, i.e., their profiles (velocity, acceleration, and distance) will be tracked by roadside infrastructure node and commands can be given to those vehicles to accelerate or decelerate.

3 Specification of the DTTI Optimization Problem

We first take up the simple case of two roads merging and then extend it to more than 2 roads.

3.1 Two-Road Intersection

In this section, we give the formulation of the optimization function subject to constraints ensuring their safety. Consider an intersection of two roads, $Road_1$ and $Road_2$ as shown in Figure 1 where vehicles are represented by points. It is assumed that $Road_i$ contains m_i vehicles where $1 \leq i \leq 2$.

For the rest of this section the range of i and j are given by, $1 \leq i \leq 2$ (represents road index) and $1 \leq j \leq m_i$ (represents vehicle index) unless otherwise specified explicitly. Table 1 describes the notations used in the formulation. These notations will be used throughout the paper.

- **Objective Function:** The objective is to minimize the maximum DTTI (i.e., time taken by the vehicle say x_{imj} to reach the intersection region).

$$Minimize f = \text{MAX}(t_{1m_1}, t_{2m_2})$$

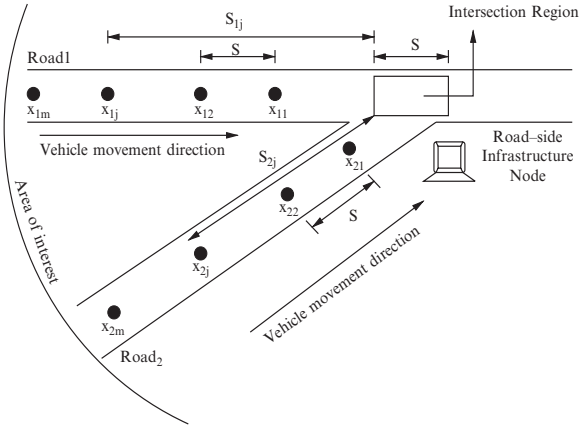


Fig. 1 Automatic Merge Control System

Table 1 Notations used in the formulation

Notation	Description
$Road_i$	represents i*th* road
m_i	number of vehicles in $Road_i$
x_{ij}	j*th* vehicle on $Road_i$
$s_{ij}(t)$	distance of the vehicle x_{ij} from the intersection region at time instant t
u_{ij}	initial velocity of the vehicle x_{ij}
v_{ij}	velocity of the vehicle x_{ij} when it reaches the merge region
t_{ij}	time at which the vehicle x_{ij} reaches the intersection region

This is similar to the *makespan* of a schedule. An alternative is to minimize the average DTTI: $Minimize f = \frac{1}{m_1+m_2} * \left(\sum_{i=1}^{m_1} t_{1i} + \sum_{j=1}^{m_2} t_{2j}\right)$.

- **Precedence Constraint:** This constraint is to ensure that the vehicles within a road reach the intersection region according to the ascending order of their distance from the region, i.e., no vehicle overtakes its leading vehicle:

  For $Road_i, t_{ij} < t_{i(j+1)}$ where $1 \leq j \leq m_i - 1$.
- **Mutual Exclusion Constraint:** This guarantees that no two vehicles are present in the intersection region at any given instant of time. In other words, this condition ensures that before (j+1)*th* vehicle reaches the intersection region, the j*th* vehicle would have traveled through the region.

  For $Road_i, t_{i(j+1)} \geq t_{ij} + \frac{S}{v_{ij}}$, where $1 \leq j \leq m_i - 1$.

  The above condition guarantees that no vehicles from same road will be present in the intersection region. To ensure vehicles from different roads also adhere to this safety criterion we have

$$\forall k, l(|t_{1k} - t_{2l}| \geq \frac{S}{v})$$

where v will take value v_{1k} or v_{2l} depending on whether $t_{1k} < t_{2l}$ or $t_{2l} < t_{1k}$ respectively and k and l represent vehicle index numbers.

- **Safety Constraint:** This constraint ensures that safe distance is always maintained between consecutive vehicles on the same road, before they enter the merge region. Consider two such consecutive vehicles x_{ij} and $x_{i(j+1)}$ on $Road_i$. For safety, the following condition needs to be ensured: $\forall t \in (0, t_{ij})$, $s_{i(j+1)}(t) - s_{ij}(t) > S$.

  Distance between x_{ij} and $x_{i(j+1)}$ is given by:
  $s_{i(j+1)}(t) - s_{ij}(t) = (s_{i(j+1)}(0) - (u_{i(j+1)} * t + *1/2a_{i(j+1)} * t^2)) - (s_{ij}(0) - (u_{ij} * t + *1/2a_{ij} * t^2)) = f(t)$

  Ensuring $f_{\min}(t) > S$ will guarantee safety criteria. On simplification, the following constraint is obtained:
  For $Road_i, \forall j$
  if $(a_{ij} > a_{i(j+1)}$ and $(u_{ij} - u_{i(j+1)})/(a_{i(j+1)} - a_{ij}) < t_{ij})$ **then**
  $s_{i(j+1)}(0) - s_{ij}(0) - L > (u_{ij} - u_{i(j+1)})^2/(2 * (a_{ij} - a_{i(j+1)}))$
  else
  Mutual Exclusion Constraint guarantees that the safety criteria will be satisfied.
- **Lower Bound on Time:** This imposes lower bound on the time taken by any vehicle to reach intersection region with the help of V_{MAX}, maximum velocity any vehicle can attain: For $Road_i, \forall j \;\; t_{ij} \geq \frac{s_{ij}}{V_{\text{MAX}}}$ where s_{ij} is the initial distance from intersection region, i.e., at time instant $t = 0$. Throughout the paper, s_{ij} and $s_{ij}(0)$ are used interchangeably.
- **Equality Constraint on Velocity:** This constraint relates the velocity of vehicle at the intersection region to its initial velocity, the distance traveled and the time taken to do so.
  For $Road_i, \forall j \;\; v_{ij} = \frac{2s_{ij}}{t_{ij}} - u_{ij}$.
- **Other Constraints:** These constraints impose limits on the velocity and acceleration range of vehicles.
  For $Road_i, \forall j \;\; V_{\text{MIN}} \leq v_{ij} \leq V_{\text{MAX}}; A_{\text{MIN}} \leq a_{ij} \leq A_{\text{MAX}}$.

After replacing all v_{ij} in the above set of constraints using the equality constraint on velocity, the system is left with the following design variable(s): t_{ij}.

System Input: $\forall i, j \;\; s_{ij}, u_{ij}, S$, and V_{MAX}.

System output: $\forall i, j \;\; t_{ij}$.

The acceleration or deceleration commands to be given to each vehicle can be computed offline from the output of system using:

$$\forall i, j \;\; a_{ij} = \frac{2 * (s_{ij} - u_{ij} * t_{ij})}{t_{ij}^2}. \tag{1}$$

3.2 n-*Road Intersection*

In this section, we provide the formulation for a case where n roads are intersecting. The formulations in Section 3.1 can be easily extended to suit this scenario.

For the rest of this section the range of i and j are given by, $1 \leq i \leq n$ (represents road index) and $1 \leq j \leq m_i$ (represents vehicle index) unless otherwise specified explicitly. Similarly, range for k and l are given by, $1 \leq k \leq n$ (represents road index) and $1 \leq l \leq m_k$ (represents vehicle index).

- **Objective Function:**
  (1) $Minimize f = \forall i \ \text{MAX}(t_{im_i})$ OR
  (2) $Minimize f = \frac{1}{\sum_{i=1}^{n} m_i} * (\sum_{i=1}^{n} \sum_{j=1}^{m_i} t_{ij})$
- **Precedence Constraint:** $\forall i \ \ t_{ij} < t_{ij+1}$ where $1 \leq j \leq m_i - 1$
- **Mutual Exclusion Constraint:**
  $\forall i, j, k, l \ \ \left|t_{ij} - t_{kl}\right| \geq \frac{S}{v}$ where v will take value v_{ij} or v_{kl} depending on whether $t_{ij} < t_{kl}$ or $t_{kl} < t_{ij}$ respectively
- **Safety Constraint:**
  $\forall i, j$ if $(a_{ij} > a_{i(j+1)}$ and $(u_{ij} - u_{i(j+1)})/(a_{i(j+1)} - a_{ij}) < t_{ij})$ then
  $s_{i(j+1)}(0) - s_{ij}(0) - L > (u_{ij} - u_{i(j+1)})^2/(2 * (a_{ij} - a_{i(j+1)}))$
- **Lower Bound on Time:** $\forall i, j \ \ t_{ij} \geq \frac{s_{ij}}{V_{\text{MAX}}}$
- **Equality Constraint on Velocity:** $\forall i, j \ \ v_{ij} = \frac{2s_{ij}}{t_{ij}} - u_{ij}$
- **Other Constraints:**
  $\forall i, j \ \ V_{\text{MIN}} \leq v_{ij} \leq V_{\text{MAX}}$; $A_{\text{MIN}} \leq a_{ij} \leq A_{\text{MAX}}$

System Input: $\forall i, j \ \ s_{ij}, u_{ij}, S$, and V_{MAX}
System Output: $\forall i, j \ \ t_{ij}$

As can be observed from the above formulation, there is not much difference between our 2-road and n-road formulations.

4 Head of Lane Approach

In this section, we describe another algorithm for determining the merge sequence. We discuss the case of two roads merging at an intersection.[1] This approach is motivated by the way drivers in manually driven vehicles resolve the conflict at intersection region in practice. The drivers who are closest to the merge region on each road decide among themselves the order in which they will pass through the region (based on some criteria, say *First Come, First Serve*).

This algorithm achieves the goal of safe maneuvering by considering the foremost vehicles on each lane for determining the merge sequence. This approach incurs lesser computational overhead compared to optimization formulation and easily

[1] The work of extending this to n-roads merging is currently in progress

maps to the way merging happens in real-world scenario where vehicles are not automated.

4.1 Two-Road Intersection

Consider the scenario depicted in Figure 1, where x_{11} and x_{21} are head vehicles (vehicles nearest to merge region) on $Road_1$ and $Road_2$ respectively whose DTTI are conflicting and hence are the competitors for the same place in MS. The algorithm resolves the conflict among these two vehicles by computing the cost associated with each vehicle (determining this cost is explained in Section 4.3) and adding the one with the lower cost, say x_{21} in the MS. Now, the algorithm considers the head vehicles on each road: x_{11} from $Road_1$ and x_{22} from $Road_2$ (since x_{21} is already included in the MS, x_{22} is the current head vehicle on $Road_2$), resolves conflict, adds the vehicle with minimum cost in MS and so on. This is done iteratively till all the vehicles are merged.

HoL algorithm operates with the same set of constraints formulated in Section 3. The goal of optimization formulation was to achieve minimum average DTTI or maximum throughput. HoL too tries to achieve the same goal by employing *acceleration whenever possible* approach. For example, in the scenario explained above, x_{21} is assigned maximum possible acceleration before inserting it in the merge sequence. A single iteration of the HoL algorithm is discussed in detail below:

1. Let x_{1k} and x_{2l} be the two head vehicles in a particular iteration. In the first iteration the foremost vehicles on each lane (x_{11} and x_{21}), would be the head vehicles.
2. Depending upon the behavior of vehicles in the MS, algorithm determines the future behavior B_{1k}, B_{2l} of the vehicles x_{1k}, x_{2l} respectively. Here future behavior of a vehicle represents its kinematics from current time instant till the vehicle reaches the merge region. While determining these behaviors, the vehicles are accelerated whenever possible while ensuring that all the constraints are met. Note that, B_{1k} and B_{2l} are calculated independent of one another and can conflict during/after merging.
3. Verify whether the behavior B_{1k} and B_{2l} interfere in the merge region (explained in detail in Section 4.2), i.e., whether the vehicles violate the safety criteria in the merge region.
4. If they are not interfering then insert that vehicle in the MS which is reaching the region M first, say x_{1k}. If they are interfering then compute the cost c_1 of the merge sequence (determining cost will be explained in Section 4.3) in which x_{1k} is chosen to be inserted in MS. Similarly compute cost c_2 in which x_{2l} is chosen to be inserted. Compare cost c_1 and c_2 and insert the vehicle with lower cost in the MS.
5. Depending upon which vehicle has been inserted in the MS, say x_{1k}, consider $x_{1(k+1)}$ and x_{2l} as head vehicles for the next iteration. Similarly if x_{2l} is inserted, then consider x_{1k} and $x_{2(l+1)}$ as head vehicles.

4.2 Interference in Merge Region

The head vehicles x_{1k} and x_{2l} from $Road_1$ and $Road_2$ still have the possibility of violating the safety criteria in the merge region even after determining their future behavior B_{1k} and B_{2l} respectively, as the behaviors are computed independent of one another. This violation of safety criteria in the merge region is called *vehicle interference*. The vehicles might be strongly violating the safety criteria, i.e., both the head vehicles might be entering the merge region approximately at same time. In this case, resolving the conflict becomes slightly tricky and the algorithm must choose the vehicle with lower cost. While in another scenario, the vehicles might be violating the safety criteria by a very small amount, i.e., when a vehicle is just about to exit the merge region, another vehicle might enter it. This special case is handled in a similar way as the non-interference one, where the leading head vehicle is inserted in the MS. We differentiate these two cases as described below:

- **Vehicle Interference** ($|t_{1k} - t_{2l}| < \delta$): Determine cost c_1, of the merge sequence in which x_{1k} is chosen to be added first to MS. Similarly determine cost c_2 for adding x_{2l}. Insert that vehicle in the MS which has lower cost associated with it.
- **Non-Interference** ($|t_{1k} - t_{2l}| > \delta$): If $t_{1k} > t_{2l}$ then insert x_{2l} in MS else insert x_{1k} in MS.

The value of δ can be determined using safety distance (S) and velocity of the head vehicles.

4.3 Merge Cost Computation

When two vehicles strongly interfere (in the merge region) and compete for the same place in MS, the HoL approach described above computes the cost of inserting each vehicle in the MS at that particular place and resolves the conflict by choosing the one with lower cost. Here we describe two approaches for determining this cost (associated with a particular vehicle for inserting it in the MS). The first approach has been simulated and work is in progress on simulation of the second approach.

- **Nearest Head:** In case of strong interference, both the head vehicles take almost same time to reach the merge region. It is more reasonable to allow the vehicle which is closer to the merge region to go first as it will have lesser time to adapt to any changes (deceleration). If both the vehicles are equidistant from the merge region, then algorithm randomly chooses one of them.
- **Cascading Effect:** This approach considers the effect on previous vehicles on each road, while computing the cost for resolving the conflict. This effect can be measured in terms of net deceleration introduced, the number of vehicles that are being affected as both give a measure of increase in DTTI of vehicles. Optimal approach would be to consider all possible merge sequences and choose the best

among them. Though this solution is better in terms of optimality, it will be computation intensive, as in this case the total number of merge orders considered will be exponential.

4.4 Pseudo-code

Pseudo-code of HoL algorithm is presented in detail below. All the notations conform to the notation used in Section 3. Functions used in the pseudo-code are explained below:

1. **getBestPossibleBehavior(Profile** P**, Merge Sequence** MS**)**: It takes profile(P) of a vehicle and Merge Sequence(MS) as input and with the help of behavior of vehicles that are in the MS, the function determines (and returns) the best possible future behavior for that vehicle.
2. **computeTimeToReach(Behavior** B**)**: It takes future behavior(B) of a vehicle as input and then computes (and returns) DTTI of that vehicle.
3. **checkStrongInterference(Behavior** B_1**, Behavior** B_2**, safe distance** S**, InterferenceParameter** δ**)**: It takes behavior(B_1 and B_2) of two vehicles and system parameters: S and δ as input and then determines whether these vehicles interfere in the merge region. An appropriate Boolean value is then returned (1—if they interfere, 0—otherwise).

HoL Algorithm Begin

$k = l = 1$;
while ($k <= m_1$ **and** $l <= m_2$){
B_1 = ***getBestPossibleBehavior***(P_{1k}, MS);
B_2 = ***getBestPossibleBehavior***(P_{2l}, MS);
//Where P_{ij} is the current profile(velocity, acceleration and distance
//from region M) of vehicle x_{ij}.
t_1 = ***computeTimeToReach***(B_1);
t_2 = ***computeTimeToReach***(B_2);
StrongInterference = ***checkStrongInterference***(B_1, B_2, S, δ);
if (StrongInterference) **then**{
Determine the cost c_1 and c_2;
if($c_1 < c_2$) **then**
Insert x_{1k} in MS; $k = k + 1$;
else
Insert x_{2l} in MS; $l = l + 1$;
}
else
if($t_2 > t_1$) **then**
Insert x_{1k} in MS; $k = k + 1$;

else
Insert x_{2l} in MS; $l = l + 1$;
}
if($k == m_1 + 1$) **then**
Append remaining vehicles on $Road_1$ to MS
else
Append remaining vehicles on $Road_2$ to MS

HoL Algorithm End

5 Continuous Stream of Vehicles

The optimization formulation and HoL algorithms described above are applicable only for a snapshot of real world scenario. In reality, there is continuous inflow of vehicles in AoI and hence we need to extend these algorithms to deal with it. This involves following issues:

- identifying the snapshot of vehicles to which algorithms will be applied;
- determining how often these snapshots should be captured, i.e., how often the algorithm is run.

In this section, we have described two ways in which our algorithms can be tuned to address these issues.

Sporadic Approach The above algorithms can be run sporadically on vehicle snapshots, i.e., all the vehicles present in AoI. This approach makes a realistic assumption that the minimum time that any vehicle takes to enter the AoI is known. The frequency of execution of this algorithm is driven by following parameters: x, the distance of the closest vehicle outside the AoI and V_{MAX}, maximum velocity any vehicle can attain. Hence, the closest vehicle will take at least x / V_{MAX} time to enter AoI. The sporadicity of this task can be determined by imposing a lower bound on x.

Multi-Zone Approach The drawbacks of sporadic approach is whenever a new vehicle enters the AoI: (i) *computational overhead:* it reconsiders all the vehicles (except those who passed through the merge region) from previous snapshot for determining the solution and (ii) *stability concern:* reconsidering the vehicles which are nearer to merge region might pose a threat to system stability. To overcome these drawbacks, in this approach the AoI region is divided into three zones as shown in the Figure 2. In this approach, the snapshot comprises of all the vehicles present in zone 2. Initially solution is computed for a snapshot. When vehicles from zone 3 enter zone 2 or after time δ, whichever is minimum, the algorithm takes the next snapshot and computes the solution. Note that the vehicles which enter zone 1 are left undisturbed, as these vehicles are very close to the merge region and have very little flexibility to adapt any changes to their profile. The parameter δ can be computed

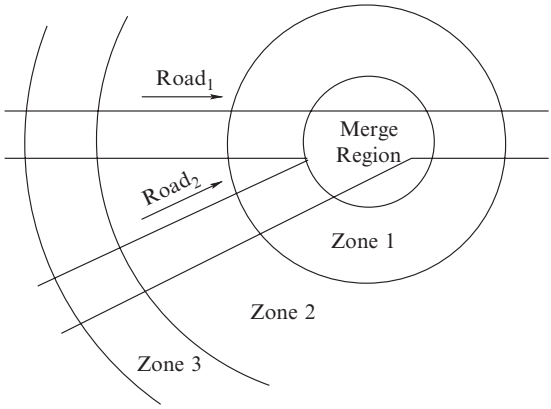


Fig. 2 Region partitioning

using the radii of the zones and $V_{\max}$. Thus solution can be computed sporadically to deal with the continuous stream of vehicles. The work is in progress to formally characterize these zones.

6 Simulations and Observations

This section describes Matlab-based evaluations of optimization formulation and C++ simulation of HoL approach and observations made from these. For simplicity, we considered a 2-road intersection problem where each road is having five vehicles. For modeling optimization formulation in MATLAB, we used Optimization Toolbox (function *fmincon*). Various parameters of the system were set to following values while performing experiments:

$$A_{\max} = 4\,\mathrm{m/s^2}, A_{\min} = -4\,\mathrm{m/s^2},$$
$$V_{\max} = 27\,\mathrm{m/s}, V_{\min} = 0\,\mathrm{m/s}, S = 5\,\mathrm{m}$$

Input The vehicle profiles at time $t = 0$ (which system takes as input) is shown in Table 2. For instance, entries in the first row of the table represent: the initial velocity of vehicles x_{11} and x_{12} are set to 20 m/s and 22 m/s and their distances from the intersection region are set to 55 m and 40 m respectively. The acceleration of all the vehicles are assumed to be zero initially, i.e., the vehicles are moving with uniform velocity u_{ij}.

Output The algorithms came up with the time (i.e., Merge Sequence order) at which each vehicle is allowed to enter the intersection region which is depicted along with acceleration of the vehicles in Table 3.

Table 2 Initial vehicle profiles (i.e., at time t = 0)

Road1				Road2			
Id	u (m/s)	S (m)	Acc (m/s^2)	Id	u (m/s)	S (m)	Acc (m/s^2)
1	20	55	0	1	22	40	0
2	22	62	0	2	20	60	0
3	21	69	0	3	25	73	0
4	25	84	0	4	22	80	0
5	23	91	0	5	21	87	0

Table 3 Simulation results showing the DTTI of all vehicles and the merge sequence

Optimization Formulation				Head of Lane			
Road Id	Veh Id	Time (s)	Acc (m/s^2)	Road Id	Veh Id	Time (s)	Acc (m/s^2)
2	1	1.63	3.06	2	1	1.63	3.06
1	1	2.34	2.99	1	1	2.34	2.99
1	2	2.53	1.98	1	2	2.53	1.98
2	2	2.72	1.54	2	2	2.71	1.55
2	3	2.92	-0.01	2	3	2.92	0.00
1	3	3.12	0.70	1	3	3.12	0.72
1	4	3.34	0.10	1	4	3.33	0.12
2	4	3.54	0.35	2	4	3.53	0.37
1	5	3.75	0.67	1	5	3.75	0.69
2	5	4.10	0.10	2	5	3.94	0.55
		29.99				**29.81**	

As we can see, the average latency obtained using both approaches are quite comparable. Also the merge sequence order was observed to be the same in both the approaches. The results from the optimal approach are slightly inferior than those from the HoL approach. These inconsistencies can be attributed to the fact that function $fmincon$ is a derivative-based search algorithm and it does not guarantee a global optimum (Coleman et al., 1999).

Figures 3 and 4 show the same results when plotted as graphs. The X-axis represents the time and Y-axis indicates the distance of each vehicle from region of intersection (i.e., $-x$: vehicle needs to travel x distance to reach the intersection region, 0: vehicle has reached the region and $+x$: distance covered by vehicle after leaving the region).

It can be observed our model guarantees that when x_{ij} reaches the region of interest the distance between it and vehicle in front of it, say x_{kl}, is at least S. It can also be observed that the curves are quadratic in nature falling in-line with the quadratic equation of motion (see Equation 1). It should be observed that the model shown does not provide the safe distance guarantee after the region of interest. But we can incorporate other region of interests in the model by adding few more constraints in the same model.

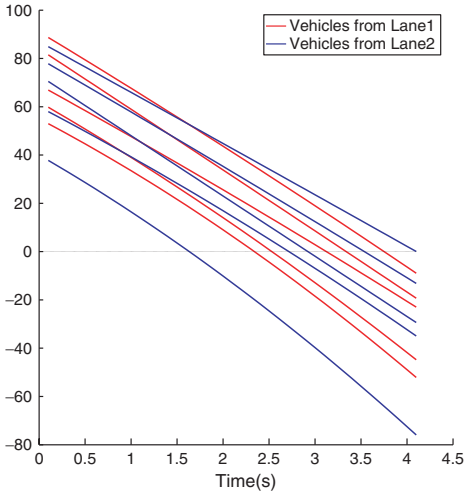


Fig. 3 Optimization formulation results

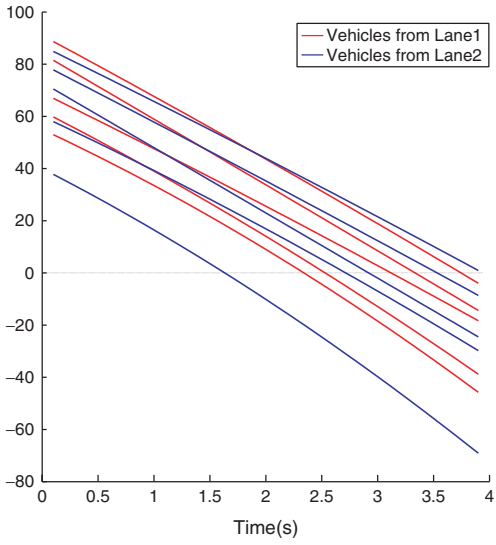


Fig. 4 HoL results

While conducting experiments, it was observed that optimization formulation approach is computationally intensive compared to HoL. This observation can be attributed to the way optimization formulation functions, i.e., it considers several possible combinations by considering all vehicles on each road at a time for determining the merge sequence whereas HoL considers only head of lane vehicles at a time for determining the same.

7 Related Work

The merge control application with inter-vehicle communication is also studied in Uno et al. (1999). It uses the concept of *virtual vehicle* that is used to map vehicles on one lane onto the other lane (assuming a 2-lane merge) for ensuring safe distance criteria. But the algorithm for determining the merge order of vehicles is not provided. The intersection region is divided into multiple zones in Bruns and Munch (2006) where initially computed suboptimal velocity profiles of vehicles gets refined to optimal profiles as the vehicles approach the zone nearer to intersection region. The approach requires more processing power in every vehicle compared to ours since each vehicle computes the merge order. The communication overhead is also more since every vehicle communicates with all the nearby vehicles about its profile. Also, we believe our formulation is simple to understand and implement.

In Dresner and Stone (2005), a reservation based multi-agent (reservation manager and driver agent) approach is proposed for designing the intersection control system. The driver agents "call ahead" to the intersection manager and request space-time in the intersection. The intersection manager then determines whether or not these requests can be met. If the request is met then the driver agent records the parameters of the request (the reservation) and attempts to meet them, else it sends the request again by adapting vehicle's velocity. This work comes close to ours. We believe the main drawback of this approach is the process of repeated requests by the driver agent when its initial request is not met. The intersection manager should be more smart to make use of all the vehicles' information available with it and suggest or block an alternate space-time in the intersection instead of rejecting the request and wait for that driver agent to make another request.

8 Conclusions and Further Work

In this paper, we presented Automatic Merge Control System that ensures safe vehicle maneuver at road intersections. We formulated this as an optimization problem with constraints to guarantee safety. It is shown with the help of MATLAB Optimization Toolbox that the existing constraint solvers can be used to determine the solution. We also presented HoL approach which is less computationally intensive and whose performance (merge sequence, DTTI of vehicles) is comparable to that of optimization approach. The observations from simulations carried out confirm these things.

We are working on several possible extensions to the research described here. First, extending the HoL approach for n-road intersection scenario considering the effect on previous vehicles while determining the cost associated with each vehicles (cascading effect described in the paper). Second, decentralizing the proposed approaches in which vehicles communicate with each other and resolve any conflicts among

themselves without the help of any centralized controller. The real-time communication protocols for the decentralized approach are being studied. Third, fine-tune our approaches to be able to consider several factors driven by real-world constraints such as giving preference to vehicles on a particular road, particular vehicles (say ambulance), angle of intersection of roads, etc. Fourth, augmenting the existing mechanisms to deal with a mix of automated vehicles and human driven vehicles. Lastly, provide real-time support for the system and demonstrate the concepts on robotic vehicular platforms built in our lab.

References

Bruns, Tornsten and Munch, Eckehard (2006). Intersection management as self-organisation of mechatronic systems. In *Proceedings of the 6th International Heinz Nixdorf Symposium on New Trends in Parallel and Distributed Computing*, Paderborn, Germany.

Coleman, Thomas, Branch, Mary Ann, and Grace, Andrew (1999). Optimization toolbox for use with matlab user's guide version 2 January 1999, 3rd printing (For Release 11).

Dresner, Kurt and Stone, Peter (2005). Multiagent traffic management: An improved intersection control mechanism. In Dignum, Frank, Dignum, Virginia, Koenig, Sven, Kraus, Sarit, Singh, Munindar P., and Wooldridge, Michael, editors, *The Fourth International Joint Conference on Autonomous Agents and Multiagent Systems*, New York, NY. ACM Press.

Raravi, Gurulingesh, Sharma, Neera, Ramamritham, Krithi, and Malewar, Sachitanand (2006). Efficient real-time support for automotive applications: A case study. In *Proceedings of the 12th IEEE International Conference on RTCSA*, pages 335–341, Sydney, Australia.

Uno, A. Sakaguchi, T. and Tsugawa, S. (1999). A merging control algorithm based on inter-vehicle communication. In *IEEE International Conference on Intelligent Transportation Systems*, pages 783–787, Tokyo, Japan.

Vahidi, Ardalan and Eskandarian, Azim (2003). Research advances in intelligent collision avoidance and adaptive cruise control. *IEEE Transactions on Intelligent Transportation Systems,* 4: 143–153.

All Those Duration Calculi: An Integrated Approach

Paritosh K. Pandya*

Abstract Duration Calculi are a family of real-time logics incorporating the measurement of duration of a proposition in an observation interval. The original Duration Calculus (DC) was defined over continuous timed behaviours. But variants of DC with different notions of time such as sampled time or discrete time have been investigated and used. Yet another variation is whether the time is taken to be weakly or strictly monotonic. The applicability, expressiveness and decidability of these Duration Calculi vary based upon the underlying nature of time.

In this paper, we propose a generic Duration Calculus $GWDC[M]$ which integrates various Duration Calculi. It has behaviours with continuous, weakly monotonic time but the logic is parameterised by the set of observable time intervals within each behaviour. By suitably choosing the parameter M, we show that the different Duration Calculi can all be obtained as $GWDC[M]$. Such a common framework allows investigation of relationships and translations between various Duration Calculi. We provide an overview of the sampling and digitization techniques for abstracting the undecidable continuous timed logics into decidable discrete timed logics.

Keywords: real-time logics, duration calculus, continuous time, discrete time

1 Introduction

Timed behaviours capture how the system state evolves with time. Temporal logics specify properties of such behaviours. Real-time logics specify quantitative timing properties of timed behaviours. Duration Calculus (DC) is one such logic [22]. It is an interval temporal logic incorporating the measurement of accumulated duration for which a proposition holds in a time interval. Duration Calculus constitutes a convenient and highly expressive notation for capturing and reasoning about real-time requirements.

Duration Calculus has found use as a formal notation for expressing requirements over real-time systems. Many other real-time logics as well as practical notations such as message sequence charts and timing diagrams can be formalised within it.

*This work was partially supported by the General Motors sponsored project "Advanced Research on Formal Analysis of Hybrid Systems".

S. Ramesh and P. Sampath (eds.), *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*, 67–81.

Duration Calculus has also been used in giving compositional semantics and proof systems for concurrent and real-time programming notations [16,18,20] and in specifying scheduling and architectural constraints on such programs [20]. But its high expressive power has also made the validity of Duration Calculus undecidable in general and hard to check in practice. Availability of effective automatic validity and model checking tools for Duration Calculus has been a long-standing quest. Various decidable subsets of *DC* have been considered [3, 5, 15, 24], but these have not yet found way into credible tools. One practical approach has been to combine a restricted fragment of *DC* with CSP and Object-Z [9].

Timed logics can make use of various notions of time: continuous, sampled (with precise clocks) or discrete (with finite precision clocks). Continuous time, where observable propositions are Boolean functions of real-valued time (also called signals), corresponds most naturally to our intuitive notion of timed behaviour. A finite variability condition postulates that only finitely many state changes can occur in finite time. Discrete time, where the set of time points is natural numbered can be appropriate when describing clocked systems such as synchronous circuits or synchronous programs [18]. There are other intermediate notions such as timed words [1] which take a sampled view of timed behaviour. The behaviour is given as a sequence of states where each state has a real-valued time stamp. Yet another variation in the nature of time is whether time is strictly monotonic or weakly monotonic. Weakly monotonic time allows several state changes to take place at the same time point. Timed behaviours with weakly monotonic time have found use in modelling concurrency under interleaving [1, 16] and true synchrony hypothesis [2].

The original Duration Calculus *(DC)* [22] made use of strictly monotonic, continuous time but variants of this logic with other notions of time have also been defined. A discrete time version of *DC* called *QDDC* was shown to be decidable using a finite automata theoretic decision procedure [13]. A validity and model checking tool called DCVALID has been built for this logic [12–14]. Pandya [15] proposed a sampled time version of *DC*, called Interval Duration Logic (IDL). It was argued that this logic, although undecidable in general, is more amenable to partial but automatic validity checking techniques [4, 19]. Recently, a sampling approach which allows abstracting continuous time Duration Calculus into the sampled time logic IDL has been proposed [17]. A Duration Calculus with weakly monotonic time, *WDC*, has also been formulated [16] and shown to be useful in giving compositional semantics and proof system for concurrent, real-time programs under interleaving and true synchrony hypothesis.

All these variations in the nature of time have led to a rather large number of Duration Calculi which have all been studied separately in an ad hoc fashion. The applicability, expressive power and decidability of these logics vary greatly depending upon the notion of time used. Unfortunately, there seems little understanding of the relationships between these logics.

In this paper we propose a generic duration logic called $GWDC[M]$ which uses continuous and weakly monotonic time for recording behaviours. But the behaviour is parameterised by a set of admissible observation intervals. By suitably restricting the observable intervals in the behaviours of M, we show that the plethora of Duration Calculi with different notions of time can all be systematically obtained. Such

a common formulation also allows us to consider the relationships between various logics. Thus, some logics are shown to be special cases of others as characterised by suitable axioms. This also allows us to develop generic techniques and theories applicable to families of Duration Calculi.

Having a common notion of continuous behaviour and allowing variation only in the admissible set of observable intervals also permits us to cleanly formulate morphisms between behaviours over different time structures and to investigate when these morphisms preserve the logical formulae. Such an approach enables us to effectively approximate the properties written in undecidable continuous time logics by formulae of decidable discrete time logics. These approximations are based on the notions of *sampling* [17] and *digitization* [4]. Sampling abstracts the dense set of observable points on real line by a discrete but representative set of sampling points. Since all measurements have to be approximated to such sampling points, a sampling error in measurements must be allowed. Digitization replaces the exact real-valued positioning of sampling points with finite precision approximate positioning. This introduces quantization errors. Formulae must be relaxed to allow such errors. These sampling and digitization approximations have been built into tools and the experimental results show that they constitute a partial but practically usable method for validity and model checking of many formulae of interest [17, 19].

The rest of the paper is organised as follows. Section 2 introduces $GWDC[M]$, the generic Duration Calculus with weakly monotonic time. Various Duration Calculi are defined as special cases of this in Section 3. The Section 4 gives an overview of the automatic validity checking of various Duration Calculi, especially focusing on the abstraction techniques from continuous timed logics to discrete timed logics using sampling and digitization. The paper ends with a discussion section.

2 Generalised Weakly Monotonic Duration Calculus

Duration Calculus is a real-time logic which was originally defined for continuous time finitely variable behaviours [22]. The time was strictly monotonic, i.e. the system could be only in one state at each real numbered time point. This was extended to *WDC* with weakly monotonic time which allowed multiple discrete state changes at the same time point [16]. Variants of *DC* and *WDC* having other forms of time such as sampled time or discrete time have also been investigated [14, 15]. Recently, Pandya et al. [17] proposed a generic Duration Calculus, *GDC*, which integrated some these variants but the weak monotonicity of time was not handled.

In this section, we present a logic *GWDC*. It has continuous time behaviours with weakly monotonic time. Moreover the behaviours are parameterised by a set of admissible observation intervals I. This allows us to give a uniform treatment of a variety of Duration Calculi which can all be obtained by suitably choosing I.

Behaviours Let $(\Re^0, <)$ be the set of non-negative real-numbers with usual order. We will use $t, t_1, \ldots$ to range over reals. Let *Pvar* be the set of observable

propositions. Consider $F \subseteq \Re^0$ which has either the form $[t]$ denoting a singleton set, or the form $[t_1, t_2)$ with $t_1 < t_2$ denoting a non-singular convex set of reals which is half open. Here t_2 can also be ∞. Let $\nwarrow F$ and $\nearrow F$ denote the left and right end-points (limits) of F. We call such a subset a phase. Let a *frame* $\mathbb{F}$ be a finite or infinite sequence of adjacent phases which partition $\Re^0$. Formally, $\mathbb{F} = (F_1, F_2, \ldots)$ such that $\nearrow F_i = \nwarrow F_{i+1}$ and $(\cup_i F_i) = \Re^0$. Also, if F_k is the last element of $\mathbb{F}$ then $\nearrow F_k = \infty$. Let $dom(\mathbb{F})$ denote the set of indices (positions) of $\mathbb{F}$. For example $\mathbb{F} = [0, 1.2)[1.2, 3.4)[3.4][3.4, 6)[6, \infty)$ is a frame which partitions the reals into five phases. Hence, $dom(\mathbb{F}) = \{1, \ldots, 5\}$. Notice that in $\mathbb{F}$ two phase changes take place at the time point 3.4 which belongs to multiple phases.

The set of points in a frame $\mathbb{F}$ is given by $Points(\mathbb{F}) \stackrel{\text{def}}{=} \{(t, i) \mid t \in F_i\}$. It is easy to see that under the point-wise ordering $(t_1, i_1) \leq (t_2, i_2) \iff (t_1 \leq t_2) \wedge (i_1 \leq i_2)$, this set is linearly ordered. We shall use b, e, z to range over points.

A behaviour θ over a $\mathbb{F}$ has the form $\theta \in dom(\mathbb{F}) \rightarrow (Pvar \rightarrow \{0, 1\})$. Thus, a behaviour assigns to each phase F_i of $\mathbb{F}$ a state giving the truth values of the propositions. Such a behaviour encodes a finitely variable evolution of system state with time, where only finitely many state changes take place in a finite time interval. However, we do allow multiple state changes to take place at the same time. This is analogous to the timed state sequences model with super-dense time (see [7]). We shall denote the set of all behaviours over a frame $\mathbb{F}$ by $BEH(\mathbb{F})$.

Example 1 A behaviour $(\mathbb{F}, \theta)$ is given below.

$$\begin{array}{rcl} \mathbb{F} & = & [0,1.5)\ [1.5,2.4)\ [2.4]\ [2.4]\ [2.4,3)\ [3,4.3)\ [4.3,\infty) \\ \theta & = & \neg P,\quad P,\qquad \neg P,\ P,\quad \neg P,\quad P,\qquad \neg P \end{array}$$

Duration Calculi are interval temporal logics with measurements over time intervals. Given $C \subseteq Points(\mathbb{F})$ let $Intv(C) = \{[b, e] \mid b, e \in C,\ b \leq e\}$. Note that these include point intervals of the form $[b, b]$. The subset of extended intervals is given by $Intv^+(C) = \{[b, e] \mid b, e \in C,\ b < e\}$. An interval specifies the amount of time it spans and also the number of phase changes. For example, in the behaviour of Example 1, the interval $[(2.3, 2), (5.3, 7)]$ denotes an interval spanning time length 3 and having 5 phase changes. Another interval with zero duration but one phase change is $[(2.4, 3), (2.4, 4)]$. Note that $Intv(Points(\mathbb{F}))$ denotes the set of all intervals.

The measurement terms mt of logic $GWDC$ have the form $\int P$ or ℓ. The measurement term ℓ denotes the time length of an interval $[b, e]$. The measurement term $\int P$ denotes the accumulated duration for which P is true in θ in an interval $[b, e]$. The value of mt is denoted by $Eval(mt)(\theta, [b, e])$. We omit this obvious definition (see [16]).

Syntax of *GWDC* Let P range over $Prop$, c over natural numbers, mt over measurement terms and and D over $GWDC$ formulae. Let $op \in \{<, \leq, =, \geq, >\}$. Let $\top$ denote the formula "true". $GWDC$ is given by the abstract syntax:

$$\top \mid \lceil\ \rceil \mid \lceil P \rceil^0 \mid \lceil\lceil P \rceil\ \mid mt\ op\ c \mid D_1 \frown D_2 \mid D_1 \wedge D_2 \mid \neg D_1$$

If the atomic formula $\lceil P \rceil^0$ (called now P) is disallowed as a sub-formula but its special case $\lceil\ \rceil$ is allowed, we get the syntactic subset $GWDC_{nl}$. If both $\lceil P \rceil^0$ and $\lceil\ \rceil$ are disallowed, we get the syntactic subset $GWDC_{pl}$, the so called pointless logic.

Semantics For a given behaviour $(\mathbb{F}, \theta)$, the semantics of formulae is parameterised by $\mathbb{I} \subseteq Intv(Points(\mathbb{F}))$, where $\mathbb{I}$ is called the set of admissible intervals. In this paper, we shall make the assumption that $\mathbb{I} = Intv(C)$ or $\mathbb{I} = Intv^+(C)$ for some $C \subseteq Points(\mathbb{F})$.

Let the triple $(\mathbb{I}, \mathbb{F}, \theta)$ be called a segmented behaviour or *s-behaviour*. Let M be a specified set of s-behaviours. We parameterise the semantics of logic $GWDC$ by M and denote this by $GWDC[M]$. A quadruple $\mathbb{I}, \mathbb{F}, \theta, [b, e]$ where $(I, \mathbb{F}, \theta) \in M$ and $[b, e] \in \mathbb{I}$ is called an *M-model*.

For $D \in GWDC[M]$ let $\mathbb{I}, \mathbb{F}, \theta, [b, e] \models D$ denote that formula D evaluates to true in M-model $\mathbb{I}, \mathbb{F}, \theta, [b, e]$. Omitting the usual Boolean cases, this is inductively defined below. For a proposition P and a point $b \in Points(\mathbb{F})$, let $\theta, b \models P$ denote that the proposition P has value 1 at point b in behaviour θ. We omit this straightforward definition.

$$\mathbb{I}, \mathbb{F}, \theta, [b, e] \models \lceil P \rceil^0 \text{ iff } b = e \text{ and } \theta, b \models P$$

$$\mathbb{I}, \mathbb{F}, \theta, [b, e] \models \lceil\ \rceil \text{ iff } b = e$$

$$\mathbb{I}, \mathbb{F}, \theta, [b, e] \models \lceil\lceil P \rceil\rceil \text{ iff } b < e \text{ and for all } b' : b \leq b' < e.\ \theta, b \models P$$

$$\mathbb{I}, \mathbb{F}, \theta, [b, e] \models mt\ op\ c \text{ iff } Eval(mt)(\theta, [b, e])\ op\ c$$

$$\mathbb{I}, \mathbb{F}, \theta, [b, e] \models D_1 \frown D_2 \text{ iff for some } z : b \leq z \leq e.$$
$$[b, z] \in \mathbb{I} \text{ and } [z, e] \in \mathbb{I} \text{ and } \mathbb{I}, \mathbb{F}, \theta, [b, z] \models D_1 \text{ and } \mathbb{I}, \mathbb{F}, \theta, [z, e] \models D_2$$

Note that in the definition of $\frown$, an interval $[b, e] \in \mathbb{I}$ must be chopped into *admissible* sub-intervals $[b, z], [z, e] \in \mathbb{I}$.

Derived Operators

- $\Diamond D \stackrel{\text{def}}{=} true \frown D \frown true$ holds provided D holds for some admissible sub-interval.
- $\Box D \stackrel{\text{def}}{=} \neg\Diamond\neg D$ holds provided D holds for all admissible sub-intervals.
- Let $ext \stackrel{\text{def}}{=} \neg\lceil\ \rceil$. Define $Unit \stackrel{\text{def}}{=} ext \wedge \neg(ext \frown ext)$. Formula $Unit$ holds for admissible extended intervals which cannot be chopped further into smaller admissible intervals.

Prefix Validity Let $\overline{0}$ denote the initial time point 0 in the initial phase 1. A prefix model of $D \in GDC[M]$ is an M-model of the form $\mathbb{I}, \mathbb{F}, \theta, [\overline{0}, e]$ such that $\mathbb{I}, \mathbb{F}, \theta, [\overline{0}, e] \models D$. Thus, in prefix models the interval begins at the initial point. Formula D is prefix satisfiable if there is a prefix model making it true. Finally, $D \in GDC[M]$ is prefix-valid denoted $\models D$ iff $\mathbb{I}, \mathbb{F}, \theta, [\overline{0}, e] \models D$ for all prefix M-models $\mathbb{I}, \mathbb{F}, \theta, [\overline{0}, e]$.

3 A Variety of Duration Calculi

Different Duration Calculi available in the literature can be defined as special cases of $GWDC[M]$ by appropriately choosing the set of s-behaviours M, and by syntactically restricting the constructs available in the logic. In some cases, we shall also give axioms characterising a sub-logic L_2 of L_1.

Definition 1 *Let $L_1 = GWDC[M_1]$ and $L_2 = GWDC[M_2]$ with $M_2 \subseteq M_1$.*

- *We say that D is an axiom of L_2 provided for all $\rho \in M_2$ we have $\rho \models D$.*
- *We say that formula D axiomatizes L_2 within L_1 provided for all $\rho \in M_1$, we have $\rho \models D$ iff $\rho \in M_2$.*

Continuous Time Duration Calculus with Weakly Monotonic Time (WDC) This logic was investigated by Pandya and Hung [16] as a variant of Duration Calculus with weakly monotonic time. This variant was found useful and necessary for modelling behaviour of concurrent programs working under synchrony hypothesis [2]. Henzinger [7] argues that weakly monotonic time is necessary when considering interleaved models of concurrency in real-time setting. In the timing analysis of circuits with wire and gate delays, the analysis is simplified by lumping delays only at certain gates/wires and the remaining gates work under the synchrony hypothesis. Modelling such situations also requires logic with weakly monotonic time such as WDC. Validity of WDC is is undecidable in general.

Let $M_{wdc} \stackrel{\text{def}}{=} \{Intv(Points(\mathbb{F}))\} \times \{\mathbb{F}\} \times BEH(\mathbb{F})$, i.e. models where the set of admissible intervals is fixed as $\mathbb{I} = Intv(Points(\mathbb{F}))$, the set of all intervals. Logic WDC can be defined as $GWDC[M_{wdc}]$. Because of this, we shall abbreviate $Intv(Points(\mathbb{F})), \mathbb{F}, \theta, [b,e] \models D$ in logic WDC by $\mathbb{F}, \theta, [b,e] \models_{wdc} D$. The syntax of WDC is identical to the syntax of $GWDC$ in the last section and we do not repeat it again. The following formula is an axiom of WDC.

$$\Box(\ell \geq 1 \Rightarrow \ell = 1 \frown \top) \tag{1}$$

Interval Duration Logic with Weakly Monotonic Time (WIDL) This logic was proposed by Pandya [15] as a variant of DC with sampled time. It was argued that $WIDL$ is more amenable to validity checking. While the validity of $WIDL$ is undecidable in general, several effective techniques and tools have been developed as partial methods for validity and model checking of $WIDL$. These include Bounded Model Checking [19] as well as reduction to the decidable Discrete-time Duration Calculus using digitization [4, 19]. Logics with sampled time are often called pointwise logics in literature.

Given a frame $\mathbb{F}$, let $Beg(\mathbb{F}) \subseteq Points(\mathbb{F})$ be the set of beginning points of all phases in $\mathbb{F}$. Let $S(\mathbb{F}) \subseteq Points(\mathbb{F})$ be such that $Beg(\mathbb{F}) \subseteq S(\mathbb{F})$ and $S(\mathbb{F})$ is countably infinite and time divergent. Thus $S(\mathbb{F})$ represents a countably infinite set of sampling points which includes all change points between phases. Such an $S(\mathbb{F})$ is called *adequate*.

Example 2 Consider the behaviour $(\mathbb{F}, \theta)$ in Example 1, an adequate set of sampled points is as follows:

$$\begin{aligned} Beg(\mathbb{F}) &= \{(0,1),(1.5,2),(2.4,3),(2.4,4),(2.4,5),(3,6),(4.3,7)\} \\ S_1(\mathbb{F}) &= \{(0,1),(1.1,1),(1.5,2),(2.2,2),(2.4,3),(2.4,4),(2.4,5), \\ &\quad (3,6),(3.3,6),(4.3,7)\} \cup \{(4.4,7),(5.5,7),(6.6,7),\ldots\} \end{aligned}$$

Define $M_{widl} = \{(Intv(S(\mathbb{F})), \mathbb{F}, \theta) \mid \theta \in BEH(\mathbb{F}) \text{ and } S(\mathbb{F}) \text{ is adequate}\}$. Then, logic $WIDL$ can be defined as $GWDC[M_{widl}]$. The syntax of $WIDL$ is the same as the syntax of *GWDC* and we do not repeat it again. The following formula is an axiom of *WIDL*:

$$\Box(ext \Rightarrow (Unit^\frown\top) \wedge (\top^\frown Unit)) \tag{2}$$

It should be noted that the original *WIDL* [15] was formulated using timed state sequences as models. Here, we reformulate this as continuous behaviour with admissible intervals spanning the sampling points. It can be shown that the two formulations are equivalent.

Well Sampled Interval Duration Logic (WSWIDL) This is a special case of *WIDL* where the continuous time behaviour is sampled at the beginning of every phase and at every integer valued point. Moreover the behaviour is also 1-oversampled by including the midpoint between every consecutive pair of above sampling points. This provides a faithful method of sampling continuous behaviours.

Formally, given a behaviour $(\mathbb{F}, \theta)$ let $Beg(\mathbb{F})$ be the set of beginning points of phases in $\mathbb{F}$ as in case of *WIDL*. Let $\mathbb{N}$ be the set of natural numbers. Let $Int(\mathbb{F}) = \{(t,i) \in Points(\mathbb{F}) \mid t \in \mathbb{N}\}$ be the set of integer valued points. Let $BI(\mathbb{F}) = Beg(\mathbb{F}) \cup Int(\mathbb{F})$. Let $Mid(\mathbb{F}) = \{((t_1+t_2)/2, i) \mid (t_1,i),(t_2,j)$ are consecutive points in $BI(\mathbb{F})\}$. Define $WS(\mathbb{F}) = BI(\mathbb{F}) \cup Mid(\mathbb{F})$. The set $WS(\mathbb{F})$ is called the set of *well-sampling points with 1-oversampling*. Here, 1-oversampling refers to the fact that we add one additional point between every pair of consecutive elements of $BI(\mathbb{F})$. Note that $WS(\mathbb{F})$ is uniquely determined by $\mathbb{F}$.

Example 3 For the behaviour $(\mathbb{F}, \theta)$ of Examples 1 and 2, we have

$$\begin{aligned} Beg(\mathbb{F}) &= \{(0,1),(1.5,2),(2.4,3),(2.4,4),(2.4,5),(3,6),(4.3,7)\} \\ Int(\mathbb{F}) &= \{(1,1),(2,2),(3,6),(4,6),(5,7),(6,7),(8,7),\ldots\} \\ Mid(\mathbb{F}) &= \{(0.5,1),(1.25,1),(1.75,2),(2.2,2),(2.7,5),(3.5,6), \\ &\quad (4.15,6),(4.65,7),(5.5,7),(6.5,7),(7.5,7),(8.5,7),\ldots\} \\ WS(\mathbb{F}) &= Beg(\mathbb{F}) \cup Int(\mathbb{F}) \cup Mid(\mathbb{F}) \end{aligned}$$

Let $M_{wswidl} = \{(Intv(WS(\mathbb{F})), \mathbb{F}, \theta)\}$ consisting of the set of s-behaviours where admissible intervals span exactly the set of well sampled points. Such models provide one way of canonically representing the continuous behaviour by sampling. Then, logic *WSWIDL* can be defined as $GWDC[M_{wswidl}]$. The syntax of *WSWIDL* is same as that of *GWDC* and we do not repeat it. We will abbreviate the *WSWIDL* satisfaction $Intv(WS(\mathbb{F})), \mathbb{F}, \theta, [b,e] \models D$ by $\mathbb{F}, \theta, [b,e] \models_{wswidl} D$.

The formula of Equation 2 is also valid for *WSWIDL* models. The following formula is an axiom the *WSWIDL*:

$$\Box(Unit^\frown Unit \;\Rightarrow\; \ell \leq 1) \tag{3}$$

Integer time IDL with Weakly Monotonic Time (ZWIDL) This is the discrete time variant of logic *WIDL*. In *ZWIDL* behaviours each phase change happens at an integer valued time point. Moreover, the set of sampling points is also integer valued. Thus, $M_{zwidl} = \{(Intv(S(\mathbb{F})), \mathbb{F}, \theta) \in M_{widl} \mid S(\mathbb{F}) \subseteq Int(\mathbb{F})\}$. Syntax of *ZWIDL* is identical to *WIDL*. It has been shown that validity of *ZWIDL* is decidable [4].

Discrete Duration Calculus with Weakly Monotonic Time (WDDC) This is a special case of *ZWIDL* where sampling points are exactly the integer valued points. Let $M_{wddc} = \{(Intv(S(\mathbb{F})), \mathbb{F}, \theta) \in M_{zwidl} \mid S(\mathbb{F}) = Int(\mathbb{F})\}$. Note that $Intv(S(\mathbb{F}))$ is uniquely determined by $\mathbb{F}$.

Example 4 Let $\mathbb{F} = [0, 1)[1][1][1, \infty)$. Then, the set of admissible intervals is precisely the intervals between the points $Int(\mathbb{F})$ where

$$Int(\mathbb{F}) = \{(0, 1), (1, 2), (1, 3), (1, 4), (2, 4), (3, 4), (4, 4), \ldots\}.$$

The syntax of *WDDC* is identical with the syntax of *GWDC*. We shall abbreviate $Intv(Int(\mathbb{F})), \mathbb{F}, \theta, [b, e] \models D$ in *WDDC* by $\mathbb{F}, \theta, [b, e] \models_{wddc} D$. The following formula axiomatizes *WDDC* within logic *ZWIDL*:

$$\Box(Unit \;\Rightarrow\; (\ell = 0 \vee \ell = 1)) \tag{4}$$

3.1 Special Sub-classes of Logics

We now give standard ways of constructing variants such as logics with strictly monotonic time, or logics without point intervals.

1. *Strictly Monotonic Sub-logic*: A strictly monotonic frame $\mathbb{F}$ is such that all its phases are non-singular. Thus, in such frames at most one state change can happen at a time point. Let $SFRAM$ denote the set of s-behaviours with strictly monotonic frames. We shall define the strict subset of a class of s-behaviours M by $M^{strict} = M \cap SFRAM$. Given a logic $L = GWDC[M]$, we denote by $L^{strict} = GWDC[M \cap SFRAM]$.
   The following formula axiomatises logic L^{strict} within logic L:
   $$\Box(ext \;\Rightarrow\; \ell > 0) \tag{5}$$
2. *Pointless Sub-logic*: Given a set of intervals $\mathbb{I}$ let $\mathbb{I}_{pl} \subset \mathbb{I}$ be the set of all non-point intervals in $\mathbb{I}$. Given a model $\rho = (\mathbb{I}, \mathbb{F}, \theta)$ let $\rho_{pl} = (\mathbb{I}_{pl}, \mathbb{F}, \theta)$. For a set of s-behaviours M let $M_{pl} = \{\rho_{pl} \mid \rho \in M\}$. Given a logic $L = GWDC[M]$, we denote by $L_{pl} = GWDC_{pl}[M_{pl}]$. Note that the syntax for L_{pl} is restricted to $GWDC_{pl}$.

3. *Nowless Sub-logic*: The "now P" formula $\lceil P \rceil^0$ was not included in the original Duration Calculus. It first appeared within the Mean Value Calculus of Zhou and Li [23]. We now define the nowless fragment of a duration logic. Given a logic $L = GWDC[M]$ let $L_{nl} = GWDC_{nl}[M]$. Note that L_{nl} uses the syntax $GWDC_{nl}$.

Using above constructions, we can obtain several known logics in the Duration Calculus family.

$$\begin{aligned} DC &= WDC_{nl}^{strict} \\ WSIDL &= WSWIDL^{strict} \\ PLDC &= DC_{pl} \\ DDC &= WDDC^{strict} \end{aligned}$$

4 Validity Checking of Duration Calculi

Zhou et al. [21] showed that the validity of Duration Calculus (DC) is undecidable. To show this, they encoded runs of 2-counter machine T by a formula $D(T)$ such that $D(T)$ is satisfiable if and only if T has a halting computation. This encoding requires that (an arbitrarily complex) configuration of 2-counter machine be represented by state changes within an interval of length $\ell = 1$. This is possible due to the density of real numbers. The same technique can be used to to show the following.

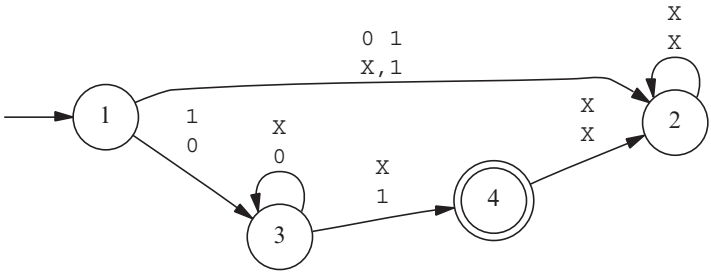
Proposition 1 *Satisfiability (validity) of logics WDC, $WIDL$, $WSWIDL$ is undecidable. If L is any of the above logics, then the satisfiability of their strict fragments L^{strict} (e.g. WDC^{strict}) is also undecidable. The satisfiability of the pointless versions L_{pl} as well as nowless versions L_{nl} is also undecidable. In particular, the satisfiability of DC, WDC, $WIDL$ and $WSIDL$ are all undecidable.*

Using an automata theoretic technique, Pandya [13] showed that the validity of logic DDC and its extension with state quantification $QDDC$ is decidable. $QDDC$ models can also be encoded as finite sequences of states. Let $pvar(D)$ be the finite set of propositional variables occurring free within a $QDDC$ formula D. Let $VAL(Pvar) = Pvar \to \{0, 1\}$ be the set of valuations over $Pvar$.

Theorem 1 *For every QDDC formula D, we can effectively construct a finite state automaton $A(D)$ over the alphabet $VAL(pvar(D))$ such that for all $\sigma \in VAL(pvar(D))^*$, we have $\sigma \models D$* **iff** *$\sigma \in L(A(D))$.* □

We refer the reader to [13] for a proof of this theorem. The formula automaton construction has been implemented into a tool called DCVALID [12–14]. The tool DCVALID is built on top MONA [6] which is a sophisticated BDD-based implementation of an automata theoretic decision procedure for monadic second order logic over finite words. Several optimizations to improve the performance of formula automaton construction are used in DCVALID [11].

Example 5 Consider the $QDDC$ formula $\lceil P \rceil^0 \frown [\![\neg Q]\!] \frown \lceil Q \rceil^0$. The automaton corresponding to this formula is given below. Each edge is labelled with a column vector giving the truth values of variables P, Q. Also, letter X is used to denote either 0 or 1. Note that the automaton is minimal, deterministic and total. □



Chakravorty and Pandya [4] showed that the validity checking of $ZWIDL$ can be reduced to the validity checking of logic $QDDC$.

Theorem 2 *There is a linear time translation $\alpha_{zwidl2qddc} : ZWIDL \rightarrow QDDC$ such that $\models_{zwidl} D$* **iff** *$\models_{qddc} \alpha_{zwidl2qddc}(D)$). Hence, the validity of ZWIDL is decidable.* □

We refer the reader to the original paper [4] for details. This reduction has been implemented into a tool *ZWIDL2QDDC* [19]. Note that the logic $WDDC$ is the sub-logic of $ZWIDL$ satisfying the axiom (4). Hence, we can also decide the validity of $WDDC$ formulae.

Digitization Now, we discuss validity checking of some useful but undecidable logics. While the validity of logics $WIDL$, $WSWIDL$ and DC is undecidable in general, there are partial techniques for solving their validity problem.

The following theorem states that using digitization [8] we can approximate the $WIDL$ formulae by $ZWIDL$ formulae while preserving either their validity or counter examples. Note that digitization replaces the precise times of phase boundaries with approximate (integral valued) times recorded by digital clocks. This introduces quantization errors in measurements. Hence, in the translation, the formulae have to be "relaxed" to allow for such errors.

Theorem 3 *We can define linear time computable functions $\alpha^{+}_{widl2zwidl}$ and $\alpha^{-}_{widl2zwidl}$ of type $WIDL \rightarrow ZWIDL$ such that for any $D \in WIDL$,*

$$
\begin{array}{lcl}
- \models_{zwidl} \alpha^{+}_{widl2zwidl}(D) & \Rightarrow & \models_{WIDL} D \\
- \not\models_{zwidl} \alpha^{-}_{widl2zwidl}(D) & \Rightarrow & \not\models_{WIDL} D.
\end{array}
$$
□

For the details and the proof of this theorem we refer the reader to the original paper [4]. Note that as *WSWIDL* is a sub-logic of *WIDL*. We can use a similar translations to approximate *WSWIDL* formulae by *ZWIDL* formulae (see [17]).

Bounded Validity Checking Let $Unit^k$ abbreviate the formula $Unit \frown \ldots \frown Unit$ with $Unit$ occurring k times. A *WSWIDL* formula D is said to be k-satisfiable provided there exists a prefix model $\mathbb{I}, \mathbb{F}, \theta, [\overline{0}, e]$ such that $\mathbb{I}, \mathbb{F}, \theta, [\overline{0}, e] \models D \wedge Unit^k$. It is easy to see that if $\neg D$ is k-satisfiable, then $\not\models D$. Thus, checking for k-satisfiability of $\neg D$ gives a partial technique for finding counter examples to the validity of D. This is called bounded validity checking.

A $LINSAT$ formula consists of a Boolean combination of linear constraints over reals. There are several effective SMT solvers which can determine whether a *LINSAT* formula ϕ is satisfiable.

Theorem 4 *There exists a polynomial time computable translation $\beta : \mathbb{N} \to WIDL \to LINSAT$ such that D is k-satisfiable if and only if $\beta(k)(D)$ is satisfiable.* □

We refer the reader to the original paper [19] for more details and the proof. We can use similar technique for bounded validity checking of $WSWIDL$.

Sampling Recall that $WSIDL$ is the strict fragment of $WSWIDL$. The following theorem states that we can approximate DC formulae by $WSIDL$ formulae using sampling while preserving either their validity or counter examples. In sampling, every real-valued point is approximated by a nearby sampled point. This introduces sampling errors in measurements. Hence, in the translation the formulae have to be "relaxed" to allow such errors.

Theorem 5 *We can define exponential time computable functions $\alpha^+_{dc2wsidl} : DC \to WSIDL$ and $\alpha^-_{dc2wsidl} : DC \to WSIDL$ such that for any $D \in DC$,*

$$\begin{aligned} &- \models_{wsidl} \alpha^+_{dc2wsidl}(D) \;\Rightarrow\; \models_{dc} D \\ &- \not\models_{wsidl} \alpha^-_{dc2wsidl}(D) \;\Rightarrow\; \not\models_{dc} D. \end{aligned}$$ □

For the details and the proof of this theorem we refer the reader to the original paper [17]. The reduction first requires a model preserving transformation of DC formula into $PLDC$ formula. The $PLDC$ formula can then be approximated by $WSIDL$ formula which is preserved under sampling abstraction of an interval.

Experimental Results Together, the theorems of this section provide a partial but practical technique for validity checking of formulae of logics DC, $WIDL$ and $WSWIDL$. For example, Theorems 5, 3 and 2 imply that applying the abstraction $ST(D) \stackrel{\text{def}}{=} \alpha_{zwidl2qddc}(\alpha^+_{widl2zwidl}(\alpha^+_{dc2wsidl}(D)))$ provides a strong translation of a DC formula D into a $QDDC$ formula such that the validity of the resulting $QDDC$ formula guarantees the validity of original DC formula D. Similarly, applying the weak abstraction $WT(D) \stackrel{\text{def}}{=} \alpha_{zwidl2qddc}(\alpha^-_{widl2zwidl}(\alpha^-_{dc2wsidl}(D)))$ generates a $QDDC$ formula whose counter example gives (with suitable translation) a counter example for the original formula D. These translations $ST(D)$ and $WT(D)$ have been implemented. The following well known example from Duration Calculus can be used to illustrate the usage of our tools.

Example 6 (Gas burner) Consider the following specification of a gas burner in Duration Calculus, DC. The requirement $Concl$ states that within any observation interval of at most $winlen$ seconds the accumulated duration of leakage of gas must be no more than $leakbound$ seconds. To achieve this, the following design decisions are made. Let $Des1$ state that the gas must not leak for more than $maxleak$ seconds at a stretch. Let Des_2 state that between any two leakages there must be at least $minsep$ seconds.

$$
\begin{aligned}
Des_1 &\stackrel{\text{def}}{=} \Box(\lceil\!\lceil Leak \rceil\!\rceil \Rightarrow \ell \leq maxleak) \\
Des2 &\stackrel{\text{def}}{=} \Box(\lceil\!\lceil Leak \rceil\!\rceil \frown \lceil\!\lceil \neg Leak \rceil\!\rceil \frown \lceil\!\lceil Leak \rceil\!\rceil \Rightarrow \ell > minsep) \\
Concl &\stackrel{\text{def}}{=} \Box(\ell \leq winlen \Rightarrow \textstyle\int Leak \leq leakbound) \\
G(maxleak, minsep, winlen, leakbound) &\stackrel{\text{def}}{=} Des1 \wedge Des2 \Rightarrow Concl
\end{aligned}
$$

The correctness of gas burner is established by showing that for the given values of the parameters $maxleak$, $minsep$, $winlen$ and $leakbound$, we have that $\models G(maxleak, minsep, winlen, leakbound)$. □

The following experimental results taken from [17] and [19] give some indication of the applicability of these techniques.

1. The gas burner problem, formulated in Example 6, requires checking validity of the DC formula $G(maxleak, minsep, winlen, leakbound)$ for given values of the parameters. This can be checked by making strong and weak translations to decidable logic $QDDC$ as stated before. In our experiments, the instance $G(4, 8, 30, 18)$ was shown to be valid by applying the strong translation ST into $QDDC$ and checking the validity of resulting $QDDC$ formula using the tool DCVALID. This verification took 0.3 seconds for translation and 2.91 seconds for the validity checking. The instance $G(20, 40, 120, 50)$ was also shown valid with translation time 0.3 seconds and validity checking time 148 seconds. The instance $G(20, 40, 200, 75)$ was shown to be invalid as follows. Its strong translation ST into $QDDC$ took time 0.3 seconds and the DCVALID tool required about 34 minutes to show that the resulting $QDDC$ formula was invalid. This does not guarantee that the formula G itself is invalid. To confirm this, we carried out the weak translation WT of the formula G into $QDDC$ in time 0.3 seconds. The resulting $QDDC$ formula was be shown to be invalid by DCVALID tool in about 7 minutes. The tool also gave a counter example for the original formula. We refer the reader to the original paper [17] for more exhaustive experimental results.
2. Several examples such as the MINEPUMP control and the LIFT control from Duration Calculus literature have been re-formulated in Logic $WIDL$. The validity/counter examples for various instances of these problems could be found automatically by reduction to $QDDC$ using digitization as in Theorems 3 and 2. However, for some choices of the parameters, the method did not succeed in a reasonable time. See [19] for details.
3. For some of the instances of the MINEPUMP and LIFT problems with large constants, the digitization technique above failed to give results in reasonable time.

But using bounded validity checking as in Theorem 4, we could find counter examples to many these formulae with small values of k up to 20. See [19] for the details and for a comparison between the digitization and the bounded validity checking approaches.

5 Discussion

There exist a wide variety of Duration Calculi with different notions of time in literature. These include continuous time logic *DC*; sampled time logics IDL and *WSIDL*; and discrete time logics *ZIDL* and *DDC*. Moreover, their variants with weakly monotonic time such as *WDC*, *WIDL*, *WSWIDL*, *ZWIDL* and *WDDC* have also be defined. In this paper we have integrated all these logics into a common framework. We have presented a generic logic called Generalised Weakly Monotonic Duration Calculus $GWDC[M]$. Its behaviours are recorded using continuous and weakly monotonic time, but each behaviour in M is accompanied by a set of admissible observation intervals. By suitably restricting the observation intervals in the behaviours of M we can obtain all the different varieties of Duration Calculi listed above.

Having an integrated formulation is a powerful theoretical tool to investigate the role of time structures as a parameter in controlling the expressiveness and decidability of Duration Calculi. Moreover, using the common framework, we can better discover the interrelationships between these logics. For example, we can define an interesting new sub-logic of *WIDL* called *FWIDL* as follows: Given a time frame $\mathbb{F}$ let the set of sampling points $S(\mathbb{F}) = Beg(\mathbb{F})$ be exactly the set of beginning points of phases. Let M_{fwidl} be the subset of M_{widl} having this set sampling points for each behaviour. The observation intervals in M_{fwidl} span only the complete phases. One important and unfinished part of the current development is to fully work out the characterising axioms for various logics.

Using the common framework, we can also study morphisms between behaviours with different time structures and whether these morphisms preserve formulae. Abstraction of continuous time *DC* into sampled time *WSIDL* (Theorem 5) and abstraction of sampled time *WIDL* into discrete time *ZWIDL* (Theorem 3) are examples of this. Similar such abstractions need to be worked out for other logics. An abstraction from WDC to $WSWIDL$ is a subject of our current further study.

Having such a variety of Duration Calculi raises questions about their relevance and use. For example, we can ask the following interesting question: which Duration Calculus should be used for specifying properties of standard models such as timed automata? The natural choices are logics WDC, $WSWIDL$ and $WIDL$. However, there are important differences between them. For example, the *DC* formula $\ell = 3 \wedge \lceil\!\lceil P \rceil\!\rceil$ states that P holds invariantly for 3 time units. The *DC* formula $(\ell = 1 \wedge \lceil\!\lceil P \rceil\!\rceil)^\frown(\ell = 2 \wedge \lceil\!\lceil P \rceil\!\rceil)$ states that P holds invariantly for 1 time unit and this followed by P holding invariantly for 2 more time units. Although intuitively, the two

properties are the same, unfortunately the two formulae are not equivalent in logics $WIDL$ or $WSIDL$ as intermediate sampling point at time 1 may not be available. With this in mind, Hirschfeld and Rabinovich [10] have argued that continuous time logics should be preferred for real-time requirements. On the other hand, sampled time logics are closer to automata theoretic models and they have better decidability properties.

Duration Calculus (DC) has been used for specifying requirements over real-time systems. The Gas burner problem (Example 6) illustrates this. Notations such as sequence diagrams and timing diagrams can be easily translated into Duration Calculus.

Compositional semantics of several reactive and real-time languages have been formulated in various Duration Calculi [16, 18, 20]. Moreover, scheduling and architectural constraints over such programs have also been specified using DC [20]. The availability of the Duration construct $\int P$ is particularly relevant in this.

Formulation of compositional semantics of concurrent programs in presence of interleaving and true concurrency seems to require weakly monotonic time [1, 16]. A compositional semantics of Timed CSP under the synchrony hypothesis was given [16] using the logic $\mu QWDC$, which is WDC extended with fixed point operators as well as state quantification. In an interesting formulation, the compositional semantics of synchronous programming language Esterel has been given using $\mu QDDC$, which is DDC extended with fixed point operator and quantification over propositional variables [18]. This semantics does not deal with the causality issue and it only specifies the observable external behaviour of Esterel programs. Analysis of causality would require a more detailed semantics with weakly monotonic time as in [16].

For specifying the behaviour of clocked systems, logics $ZWIDL$ and $WDDC$ or their strict time versions $ZIDL$ and DDC seem appropriate. In particular, we are investigating the use of $ZWIDL$ in specification of time triggered architectures. Some other potential uses of Duration Calculi under study include the specification of timing behaviour of web services, and specification and synthesis of run-time monitors from logical specification.

Acknowledgements The author thanks Kamal Lodaya and Swarup Mohalik for their helpful comments.

References

1. R. Alur and D.L. Dill, Automata for Modeling Real-time Systems, *Proc. of 17th ICALP* LNCS 443, (1990) Springer-Verlag, pp 332–335.
2. G. Berry, *The Constructive Semantics of Esterel* (1999).
3. A. Bouajjani, Y. Lakhnech and R. Robbana, From Duration Calculus to Linear Hybrid Automata, *Proc. of 7th CAV*, LNCS 939 (1995), Springer-Verlag, pp 196–210.
4. G. Chakravorty and P.K. Pandya, Digitizing Interval Duration Logic, *Proc. of 15th CAV*, LNCS 2725 (2003), Springer-Verlag, pp 167–179.
5. M. Fränzle, Model-Checking Dense-Time Duration Calculus, in M.R. Hansen (ed.), *Duration Calculus: A Logical Approach to Real-Time Systems Workshop, Proc. of ESSLLI X* (1998).

6. J.G. Henriksen, J. Jensen, M. Jorgensen, N. Klarlund, B. Paige, T. Rauhe and A. Sandholm, Mona: Monadic Second-Order Logic in Practice, *Proc. of TACAS'95*, LNCS 1019 (1996), Springer-Verlag.
7. T.A. Henzinger, Its About Time: Real-time Logics Reviewed, *Proc. of 9th CONCUR 1998*, LNCS 1466 (1998), Springer-Verlag.
8. T.A. Henzinger, Z. Manna and A. Pnueli, What Good are Digital Clocks? *Proc. of 19th ICALP*, LNCS 623 (1992), Springer-Verlag, pp. 545–558.
9. J. Hoenicke and E.R. Olderog, CSP-OZ-DC: A Combination of Specification Techniques for Processes, Data and Time, *Nordic J. of Computing*, 9(4) (2002), pp 301–334.
10. Y. Hirshfeld and A. Rabinovich, Logics for Real-time: Decidability and Complexity, *Fundamenta Informaticae*, 62(1) (2004), pp 1–28.
11. S.N. Krishna and P.K. Pandya, Modal Strength Reduction in QDDC, *Proc. of 25th FST & TCS*, LNCS 3821 (2005), Springer-Verlag, pp 444–456.
12. P.K. Pandya, DCVALID User Manual, Tata Institute of Fundamental Research, Bombay (1997). (Available in revised version at http://www.tcs.tifr.res.in/∼pandya/dcvalid.html)
13. P.K. Pandya, Specifying and Deciding Quantified Discrete-time Duration Calculus Formulae using DCVALID: An Automata Theoretic Approach, in *Proc. of RTTOOLS'2001* (2001).
14. P.K. Pandya, Model Checking CTL*[DC], *Proc. of 7th TACAS*, LNCS 2031 (2001), Springer-Verlag, pp 559–573.
15. P.K. Pandya, Interval Duration Logic: Expressiveness and Decidability, *Proc. of TPTS*, ENTCS 65(6) (2002), Elsevier Science B.V.
16. P. Pandya and D.V. Hung, Duration Calculus of Weakly Monotonic Time, *Proc. of FTRTFT*, LNCS 1486 (1998), Springer-Verlag.
17. P.K. Pandya, S.N. Krishna and K. Loya, On Sampling Abstraction of Continuous Time Logic with Durations, *Proc. of TACAS 2007*, LNCS 4424 (2007), Springer-Verlag, pp 246–260.
18. P.K. Pandya, Y.S. Ramakrishna and R.K. Shyamasundar. A Compositional Semantics of Esterel in Duration Calculus. In *Proc. Second AMAST workshop on Real-time Systems: Models and Proofs*, Bordeux, June (1995).
19. B. Sharma, P.K. Pandya and S. Chakraborty, Bounded Validity Checking of Interval Duration Logic, *Proc. of 11th TACAS*, LNCS 3440 (2005), Springer-Verlag, pp 301–316.
20. Zhou Chaochen, M.R. Hansen, A.P. Ravn and H. Rischel, Duration Specification for Shared Processors, *Proc. of FTRTFT'92*, LNCS 571 (1992), Springer-Verlag.
21. Zhou Chaochen, M.R. Hansen and P. Sestoft: Decidability and Undecidability Results for Duration Calculus, *Proc. of STACS'93*, Würzburg (1993).
22. Zhou Chaochen, C.A.R. Hoare and A.P. Ravn, A Calculus of Durations, *Info. Proc. Letters*, 40(5) (1991).
23. Zhou Chaochen and Li Xiaoshan, A mean value calculus of durations, in *A classical mind: essays in honour of C.A.R. Hoare*, Prentice Hall International (1994), pp 431–451.
24. Zhou Chaochen, Zhang Jingzhong, Yang Lu and Li Xiaoshan, Linear Duration Invariants. In *Proc. of 3rd FTRTFT*, LNCS 863 (1994), Springer Verlag, pp 86–109.

Adding Time to Scenarios*

Prakash Chandrasekaran and Madhavan Mukund

Abstract Message Sequence Charts (MSCs) are used to specify the behaviour of communicating systems through scenarios. Though timing constraints are natural for describing the behaviour of real-life protocols, the basic MSC notation has no mechanism to specify such constraints. We propose a notation for specifying collections of timed scenarios and describe a framework for automatic verification of scenario-based properties for communicating finite-state machines equipped with local clocks.

Keywords: MSC, timed scenarios, verification, communicating finite-state machies

1 Introduction

In a distributed system, several agents interact with each other to generate a global behaviour. The interaction between these agents is usually described in terms of scenarios, using message sequence charts (MSCs) [9].

We extend scenarios to incorporate timing constraints, yielding timed MSC templates. These templates are built from fixed underlying MSCs by associating a lower and upper bound on the time interval between certain pairs of events. Timed MSC templates are a natural and useful extension of the untimed notation for scenarios, because protocol specifications typically include timing requirements for message exchanges, as well as descriptions of how to recover from timeouts.

We propose a simple specification language based on guarded commands, along the lines of Promela [8], for generating collections of timed MSC templates. The semantics of this language is given in terms of a version of HMSCs (high-level MSCs) [7], with annotations attached to edges rather than nodes.

Our aim is to verify properties of timed systems with respect to timed MSC template specifications. Our basic system model consists of communicating finite-state machines equipped with local clocks. Clock constraints are used to guard transitions and specify location invariants, as in other models of timed automata [3]. Just as the runs of timed automata can be described in terms of timed words, the interactions exhibited by communicating finite-state machines with clocks can be described using timed MSCs.

*Partially supported by *Timed-DISCOVERI*, a project under the Indo-French Networking Programme.

S. Ramesh and P. Sampath (eds.), *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*, 83–97.

Specifications in terms of scenarios give rise to several natural verification problems. At preliminary stages of system design, scenario specifications are typically incomplete and can be classified into two categories, positive and negative. Positive scenarios are those that the system is designed to execute—for instance, these may describe a handshaking protocol to set up a reliable communication channel between two hosts on a network. Negative scenarios indicate undesirable behaviours, such as a situation when both hosts independently initiate a handshake, leading to a collision. This leads to the following verification problem: given a distributed system and a positive (or negative) scenario, does the system exhibit (or avoid) the scenario?

In general, a timed MSC template is compatible with infinitely many timed MSCs. This makes the scenario matching problem more complicated than in the untimed case, where a single scenario describes exactly one pattern of interaction. In our setting, the scenario matching problem amounts to checking whether the intersection of two collections of timed MSCs is nonempty.

As the design of a system evolves, the interpretation of a scenario-based specification also changes. The specification is now typically seen as an exhaustive description of how the system should behave. Universality then becomes an important condition to check—does the implementation exhibit a representative behaviour consistent with each of the timed templates in the specification? Once again, the complication is that each timed template in the specification is compatible with an infinite set of timed behaviours. Moreover, we also have an infinite set of timed templates to verify.

We propose an approach to tackle these verification problems using the model checking tool UPPAAL, which is designed to verify properties of timed systems. This paper extends the work reported in [4], where we only consider finite sets of timed templates. Since the basic system model of UPPAAL uses synchronous handshakes, rather than message-passing, we need to encode message-passing channels by creating special processes to model buffers. Exploiting the handshake mechanism in UPPAAL, we can synchronize the system with the specification. This allows us to transform our verification questions into properties for UPPAAL to verify on the composite system.

In the untimed setting, efficient algorithms for the scenario matching problem have been identified in [11]. An approach to solve this problem using the model checker SPIN was proposed in [6].

Adding timing constraints to individual scenarios has been proposed in [1], where an algorithm is given to check whether such a set of timing constraints in consistent. At the level of sets of scenarios, the *live sequence chart (LSC)* formalism [5] allows adding interval constraints similar to those we consider. One important difference is that the semantics of LSCs assumes synchronous composition of scenarios—all processes are assumed to move together from one scenario to the next. We retain the usual asynchronous semantics for MSC composition, which is more natural from the point of view of implementations.

The paper is organized as follows. In the next two sections, we formally define timed MSCs and timed message-passing automata. In Section 4, we propose a new notation for specifying timed scenarios. In the next section, we describe some verification problems for scenario based specifications. In Section 6, we describe our

approach to address verification problems for scenario-based specifications using UPPAAL. We conclude with a brief discussion.

2 Timed MSCs

2.1 Message Sequence Charts

Let $\mathcal{P} = \{p, q, r, \ldots\}$ be a finite set of processes (agents) that communicate with each other through messages via reliable FIFO channels using a finite set of message types $\mathcal{M}$. For $p \in \mathcal{P}$, let $\Delta_p = \{p!q(m), p?q(m) \mid p \neq q \in \mathcal{P}, m \in \mathcal{M}\}$ be the set of communication actions in which p participates. The action $p!q(m)$ is read as *p sends the message m to q* and the action $p?q(m)$ is read as *p receives the message m from q*. The set of actions that p performs is given by $\Sigma_p = \Delta_p \cup \{i_p\}$, where i_p is a local action of p. We will use local actions to describe timeouts. We set $\Sigma = \bigcup_{p \in \mathcal{P}} \Sigma_p$. We also denote the set of *channels* by $Ch = \{(p, q) \mid p \neq q\}$.

Labelled Posets A Σ-labelled poset is a structure $M = (E, \leq, \lambda)$ where $(E, \leq)$ is a poset and $\lambda : E \to \Sigma$ is a labelling function. For $e \in E$, let $\downarrow e = \{e' \mid e' \leq e\}$.

For $p \in \mathcal{P}$ and $a \in \Sigma$, we set $E_p = \{e \mid \lambda(e) \in \Sigma_p\}$ and $E_a = \{e \mid \lambda(e) = a\}$, respectively. For each $(p, q) \in Ch$, we define the relation $<_{pq}$ as follows:

$$e <_{pq} e' \iff \lambda(e) = p!q(m),\ \lambda(e') = q?p(m) \text{ and } |\downarrow e \cap E_{p!q(m)}| = |\downarrow e' \cap E_{q?p(m)}|$$

The relation $e <_{pq} e'$ says that channels are FIFO with respect to each message—if $e <_{pq} e'$, the message m read by q at e' is the one sent by p at e.

Finally, for each $p \in \mathcal{P}$, we define the relation $\leq_{pp} = (E_p \times E_p) \cap \leq$, with $<_{pp}$ standing for the largest irreflexive subset of $\leq_{pp}$.

Definition 1 *An MSC (over $\mathcal{P}$) is a* finite *Σ-labelled poset $M = (E, \leq, \lambda)$ that satisfies the following conditions:*

(i) Each relation $\leq_{pp}$ is a linear order.
(ii) If $p \neq q$ then for each $m \in \mathcal{M}$, $|E_{p!q(m)}| = |E_{q?p(m)}|$.
(iii) If $e <_{pq} e'$, then $|\downarrow e \cap \left(\bigcup_{m \in \mathcal{M}} E_{p!q(m)}\right)| = |\downarrow e' \cap \left(\bigcup_{m \in \mathcal{M}} E_{q?p(m)}\right)|$.
(iv) The partial order $\leq$ is the reflexive, transitive closure of the relation $\bigcup_{p,q \in \mathcal{P}} <_{pq}$.

The second condition ensures that every message sent along a channel is received. The third condition says that every channel is FIFO.

In diagrams, the events of an MSC are presented in *visual order*. The events of each process are arranged in a vertical line and messages are displayed as horizontal or downward-sloping directed edges. Figure 1 shows an example with three processes $\{p, q, r\}$ and seven events $\{e_1, e_1', e_1'', e_2, e_2', e_3, e_3'\}$ corresponding to three messages—m_1 from p to q, m_2 from q to r and m_3 from p to r—and one local event on p, e_1'.

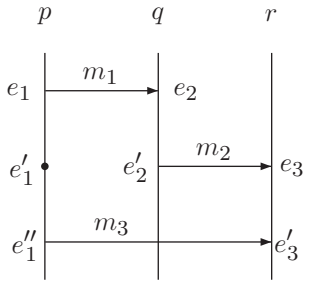


Fig. 1 An MSC over $\{p, q, r\}$

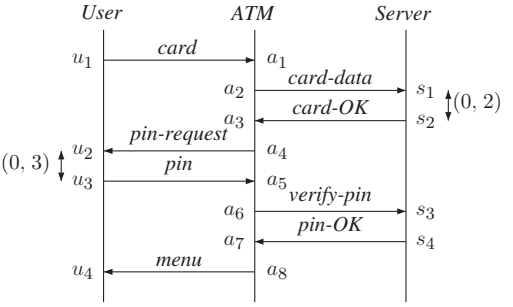


Fig. 2 A timed MSC template describing interaction with an ATM

For an MSC $M = (E, \leq, \lambda)$, we let $\text{lin}(M) = \{\lambda(\pi) \mid \pi \text{ is a linearization of } (E, \leq)\}$. For instance, $p!q(m_1)\ q?p(m_1)\ q!r(m_2)\ i_p\ p!r(m_3)\ r?q(m_2)\ r?p(m_3)$ is one linearization of the MSC in Figure 1.

2.2 Timed MSC Templates

A timed MSC template is an MSC annotated with time intervals between pairs of events along a process line. For instance, consider the interaction between a user, an ATM and a server depicted in Figure 2. This MSC has sixteen events generated by eight messages. The events u_2 and u_3 are linked by a time interval $(0, 2)$, as are the events s_2 and s_3. These time intervals represent constraints on the delay between the occurrences of the events. Thus, this template specifies that the server is expected to respond to a request to authenticate an ATM card within 2 units of time. Similarly, a user has to type in his PIN within 3 units of time of the ATM requesting the PIN.

Figure 3 shows an alternative scenario in which the user does not supply the PIN within the specified time limit, leading to the ATM rejecting the card. Notice that the timeout event is modelled as a local event on the ATM process.

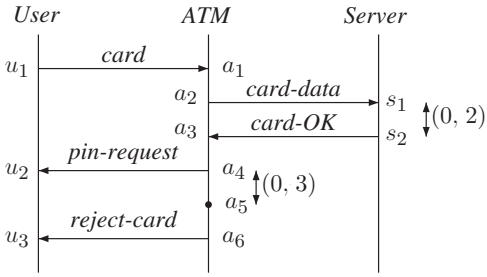


Fig. 3 The user's PIN message times out

We assume that time intervals are bounded by natural numbers. A pair of time points (m, n), $m, n \in \mathbb{N}$, $m \leq n$, denotes the time interval $\{x \in \mathbb{R} \mid m \leq x \leq n\}$.[1]

Definition 2 *Let* $M = (E, \leq, \lambda)$ *be an MSC. An* interval constraint *is a tuple* $\langle (e_1, e_2), (t_1, t_2) \rangle$*, where:*

- $e_1, e_2 \in E$ *with* $e_1 \leq_{pp} e_2$ *or* $e_1 <_{pq} e_2$ *for some* $p, q \in \mathcal{P}$.
- $t_1, t_2 \in \mathbb{N}$ *with* $t_1 \leq t_2$.

The restriction on the relationship between e_1 and e_2 ensures that an interval constraint is either local to a process or describes the delay in transmitting a single message.

Definition 3 *A* timed MSC template *is pair* $\mathcal{T} = (M, \mathcal{I})$ *where* $M = (E, \leq, \lambda)$ *is an MSC and* $\mathcal{I} \subseteq (E \times E) \times (\mathbb{N} \times \mathbb{N})$ *is a set of interval constraints.*

2.3 Timed MSCs

In a timed MSC, events are explicitly time-stamped so that the ordering on the time-stamps respects the partial order on the events.

Definition 4 *A* timed MSC *is pair* (M, τ) *where* $M = (E, \leq, \lambda)$ *is an MSC and* $\tau : E \to \mathbb{R}_{\geq 0}$ *assigns a nonnegative time-stamp to each event, such that for all* $e_1, e_2 \in E$*, if* $e_1 \leq e_2$ *then* $\tau(e_1) \leq \tau(e_2)$.

A timed MSC satisfies a timed MSC template if the time-stamps assigned to events respect the interval constraints specified in the template.

Definition 5 *Let* $M = (E, \leq, \lambda)$ *be an MSC,* $\mathcal{T} = (M, \mathcal{I})$ *a timed template and* $M_\tau = (M, \tau)$ *a timed MSC.* M_τ *is said to* satisfy $\mathcal{T}$ *if the following holds:*

$$\text{For each } \langle (e_1, e_2), (t_1, t_2) \rangle \in \mathcal{I}, t_1 \leq \tau(e_2) - \tau(e_1) \leq t_2.$$

[1] For simplicity, we restrict ourselves to closed timed intervals in this paper. We can easily generalize our approach to include open and half-open time intervals.

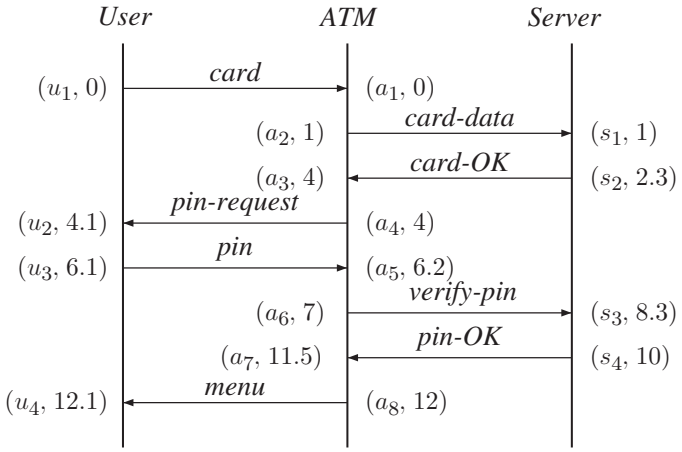


Fig. 4 A timed MSC instance describing interaction with an ATM.

Definition 6 *Let $\mathcal{T}$ be a timed MSC template. We denote by $L(\mathcal{T})$ the set of timed MSCs that satisfy $\mathcal{T}$.*

Figure 4 shows a timed MSC that satisfies the template in Figure 2.

Let $M_\tau = (M, \tau)$ be a timed MSC, where $M = (E, \leq, \lambda)$, and let $\pi = e_0 e_1 \ldots e_m$ be a linearization of $(E, \leq)$. By labelling each event with its time-stamp, this linearization gives rise to a timed linearization $(e_0, \tau(e_0))(e_1, \tau(e_1)) \cdots (e_n, \tau(e_n))$. As is the case with untimed MSCs, under the FIFO assumption for channels, a timed MSC can be faithfully reconstructed from any one of its timed linearizations.

3 Timed Message-Passing Automata

Message-passing automata are a natural machine model for generating MSCs. We extend the definition used in [7] to include local clocks on each process and time-bounds on the channels.

Definition 7 *Let $\mathcal{C}$ denote a finite-set of real-valued variables called* clocks*. A* clock constraint *is a conjunctive formula of the form $x \sim n$ or $x - y \sim n$ for $x, y \in \mathcal{C}$, $n \in \mathbb{N}$ and $\sim \in \{\leq, <, =, >, \geq\}$. Let $\Phi(\mathcal{C})$ denote the set of clock constraints over the set of clocks $\mathcal{C}$.*

Clock constraints will be used as guards and location invariants in timed message-passing automata.

Definition 8 *A* clock assignment *for a set of clocks $\mathcal{C}$ is a function $\upsilon : \mathcal{C} \to \mathbb{R}_{\geq 0}$ that assigns a nonnegative real value to each clock in $\mathcal{C}$.*

A clock assignment υ is said to satisfy *a clock constraint φ if φ evaluates to true when we substitute for each clock c mentioned in φ the corresponding value $\upsilon(c)$.*

Definition 9 *A* timed message-passing automaton (timed MPA) *over* Σ *is a structure* $\mathcal{A} = (\{\mathcal{A}_p\}_{p\in\mathcal{P}}, \Sigma, \mathcal{B})$. *Each component* $\mathcal{A}_p$ *is of the form* $(S_p, S_{in}^p, \mathcal{C}_p \rightarrow_p, I_p)$, *where:*

- S_p *is a finite set of p-local states.*
- $S_{in}^p \subseteq S_p$, *is a set of initial states for p.*
- $\mathcal{C}_p$ *is a set of local clocks for p.*
- $\rightarrow_p \subseteq S_p \times \Phi(\mathcal{C}_p) \times \Sigma_p \times 2^{\mathcal{C}_p} \times S_p$ *is the p-local transition relation.*
- $I_p : S \rightarrow \Phi(\mathcal{C}_p)$ *assigns an* invariant *to each state.*

The function $\mathcal{B} : (\mathcal{P} \times \mathcal{P}) \rightarrow (\mathbb{N} \times \mathbb{N})$ *associates with each channel a lower and an upper bound on the transmission time of messages on that channel.*

The local transition relation $\rightarrow_p$ specifies how the process p changes state when it performs internal events or sends and receives messages.

A transition of the form (s, φ, a, X, s') says that in state s, p can perform the action a and move to state s'. This transition is *guarded* by the clock constraint φ—the transition is enabled only when the current values of all the clocks satisfy φ. The set X specifies the clocks whose values are reset to 0 when this transition is taken. If a is of the form i_p, this transition corresponds to performing a local event on p. If $a = p!q(m)$, then this transition involves sending a message m from p to q. Symmetrically, if $a = p?q(m)$, then this transition involves p receiving a message m from q.

A process can remain in a state s only if the current values of all the clocks satisfy the invariant $I(s)$. To make our model amenable for automated verification, we restrict location invariants to constraints that are downward closed—that is, constraints of the form $x \leq n$ or $x < n$, where x is a clock and $n \in \mathbb{N}$.

As is customary with timed automata, we allow timed MPA to perform two types of moves: moves where the automaton does not change state and time elapses, and moves where some local component p changes state instantaneously as permitted by $\rightarrow_p$.

A global state of $\mathcal{A}$ is an element of $\prod_{p\in\mathcal{P}} S_p$. For a global state $\overline{s}$, $\overline{s}_p$ denotes the pth component of $\overline{s}$. A *configuration* is a triple $(\overline{s}, \chi, \upsilon)$ where $\overline{s}$ is a global state, $\chi : Ch \rightarrow \mathcal{M}^*$ is the *channel state* describing the message queue in each channel c and $\upsilon : \mathcal{C} \rightarrow \mathbb{R}_{\geq 0}$ is a clock assignment, where $\mathcal{C} = \bigcup_{p\in\mathcal{P}} \mathcal{C}_p$. An *initial configuration* of $\mathcal{A}$ is of the form $(\overline{s}_{in}, \chi_\varepsilon, \upsilon_0)$ where $\overline{s}_{in} \in \prod_{p\in\mathcal{P}} S_{in}^p$, $\chi_\varepsilon(c)$ is the empty string ε for every channel c and $\upsilon_0(x) = 0$ for every $x \in \mathcal{C}$.

The set of reachable configurations of $\mathcal{A}$, $Conf_\mathcal{A}$, is defined inductively in the usual way, together with a transition relation $\Longrightarrow \subseteq Conf_\mathcal{A} \times (\Sigma \cup \mathbb{R}_{\geq 0}) \times Conf_\mathcal{A}$. A move labelled by $d \in \mathbb{R}_{\geq 0}$ is a time elapsing move. All clocks advance by d, but the local states of processes and the channel contents remain unchanged. A move labelled by $a \in \Sigma$ is a local transition taken by one of processes. For each process p, the local state of p determines the set of moves available for p in the current configuration. If $a = i_p$, only the state of p changes and the rest of the configuration is unchanged. If $a = p!q(m)$, the message m is appended to the channel (p, q). An action of the

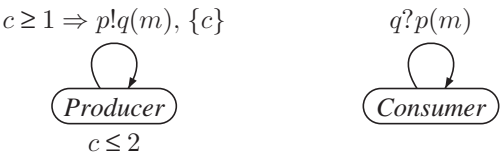


Fig. 5 A timed MPA: producer–consumer

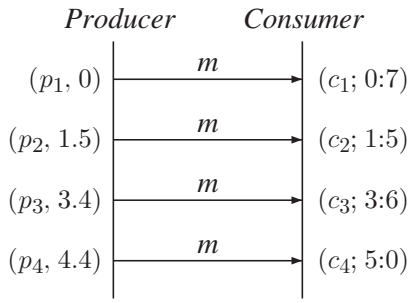


Fig. 6 A timed MSC generated by the producer–consumer system

form $p?q(m)$ is enabled only if m is currently at the head of the channel (q, p). For a more formal definition of the global transition relation, see [4].

Let $\text{prf}(\sigma)$ denote the set of prefixes of a timed word $\sigma = (a_1, t_1)(a_2, t_2)\ldots(a_k, t_k) \in (\Sigma \times \mathbb{R}_{\geq 0})^*$. A run of $\mathcal{A}$ over σ is a map $\rho : \text{prf}(\sigma) \to \mathit{Conf}_{\mathcal{A}}$ where $\rho(\varepsilon)$ is assigned an initial configuration $(\overline{s}_{in}, \chi_\varepsilon, v_0)$ and for each $\sigma' \cdot (a_i, t_i) \in \text{prf}(\sigma)$, $\rho(\sigma') \xRightarrow{d_i} \xRightarrow{a_i} \rho(\sigma' \cdot (a_i, t_i))$ with $t_i = t_{i-1} + d_i$ and $t_0 = 0$.

The run ρ is *complete* if $\rho(\sigma) = (s, \chi_\varepsilon, v)$ is a configuration in which all channels are empty. When a run on σ is complete, σ is a timed linearization of a timed MSC. We define $L(\mathcal{A}) = \{\sigma \mid \mathcal{A} \text{ has a complete run over } \sigma\}$. $L(\mathcal{A})$ corresponds to the set of timed linearizations of a collection of timed MSCs.

Figure 5 is a simple example of a timed MPA. Here, the traditional producer–consumer system is augmented with a clock c in the producer process. The constraint $c \geq 1$ on the transition ensures that each new message is generated by the producer at least one unit of time after the previous one. The location invariant $c \leq 2$ forces the producer to generate a new message no later than two units of time after the previous one. The consumer has no timing constraints. Figure 6 shows a typical timed MSC generated by this timed MPA.

4 Specifying Timed Scenarios

The standard method to describe multiple communication scenarios is to use High-Level Message Sequence Charts (HMSCs). An HMSC is a finite directed graph with designated initial and terminal vertices. Each vertex in an HMSC is labelled by an

MSC. The edges represent the natural operation of MSC concatenation. The collection of MSCs represented by an HMSC consists of all those MSCs obtained by tracing a path in the HMSC from an initial vertex to a terminal vertex, concatenating the MSCs that are encountered along the path.

In an HMSC, MSCs are concatenated asynchronously. This corresponds to gluing together the process lines of consecutive MSCs. The implication is that the boundaries between the individual MSCs along a path disappear, as a result of which some processes could move ahead of others. If the asynchrony between processes is bounded, all channels remain universally bounded and the specification is globally finite-state. Unfortunately, it is undecidable in general whether an HMSC specification satisfies this property, though sufficient structural conditions are known [7].

We propose a guarded command language inspired by Promela [8] to describe families of timed scenarios generated from basic timed templates. The basic building blocks of the language are finite timed MSC templates, as defined in Section 2.2. Statements are combined using sequential composition (`;`), nondeterministic guarded choice (`if...fi`) and nondeterministic guarded looping (`do...od`). We allow statements to be labelled, and permit labelled breaks from within loops as well as explicit gotos.

Rather than providing a precise grammar describing the syntax, we explain the notation through an example. Continuing with our ATM example, suppose the ATM is programmed to ask for the user's PIN after he has inserted the card. If the user does not enter his PIN within a specified time limit, the ATM repeats the request. At some point, nondeterministically, the ATM can also decide to reject the card. Once the user does respond, there is a possibility that the PIN is wrong. If so, the ATM swallows the card. If the PIN is correct, the user may ask for his balance or may try to make a withdrawal. These scenarios can be combined in our notation as follows, where some of the basic timed templates used in the specification are shown in Figure 7.

```
L0:: Initiate;
L1:: do
     [] NoPin
     [] NoPin; RejectCard; goto L0
     [] SwallowCard; goto L0
     [] OKPin ; break L1
     od;
     if
     [] BalanceEnquiry; goto L0
     [] WithdrawCash; if
                      [] InsufficientFunds; goto L0
                      [] DispenseCash; goto L0
                      fi
     fi
```

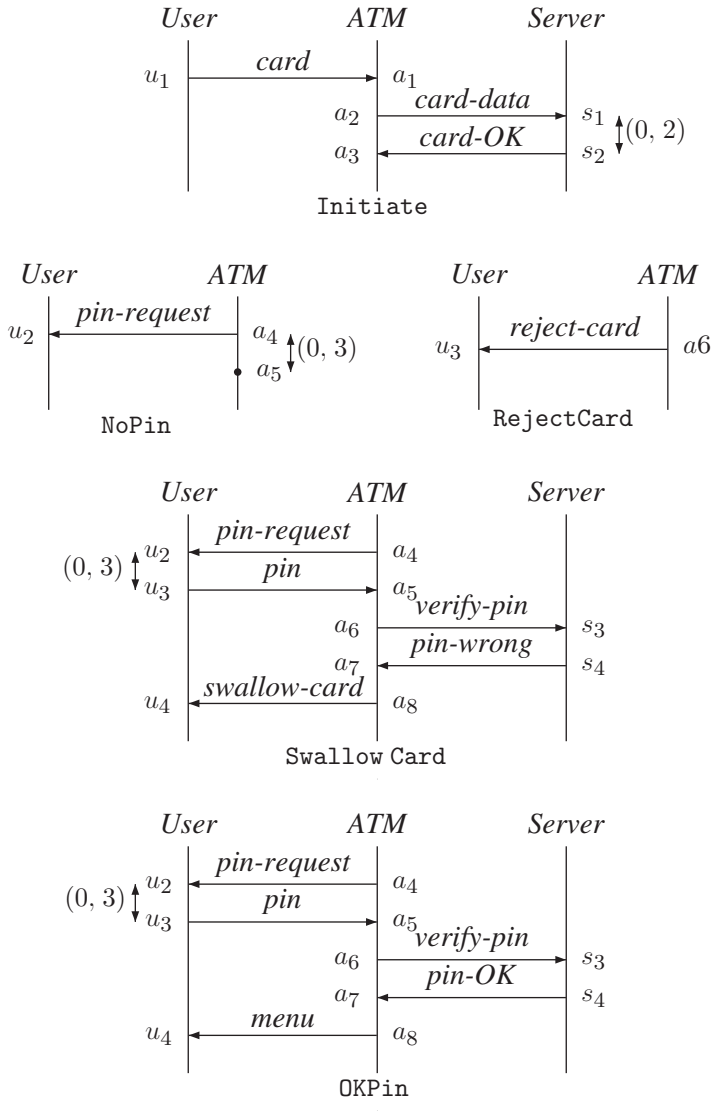


Fig. 7 Some basic timed MSC templates used in the sample specification

It is not difficult to see that our textual notation can be translated into a graphical HMSC-like notation, provided we annotate edges in the HMSC, rather than nodes, by basic timed MSC templates. Figure 8 shows the HMSC corresponding to the current example.

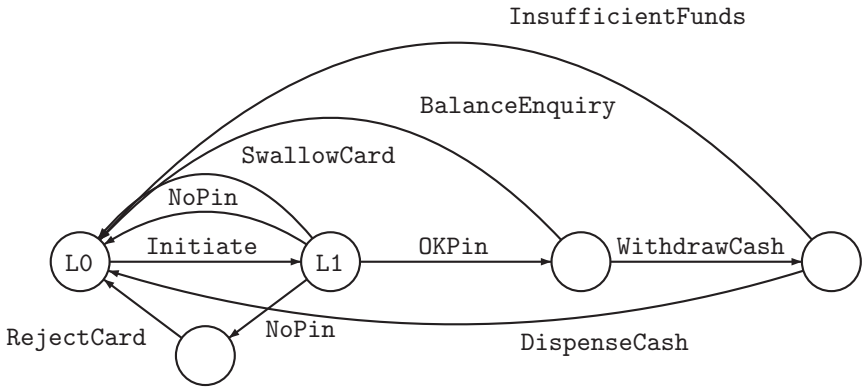


Fig. 8 HMSC corresponding to the sample specification

5 Verification Questions for Timed Scenarios

In the setting of timed MSC templates and timed MPAs, there are multiple verification questions that one can address. We focus on two of them here.

5.1 Scenario Matching

Given a timed MSC template $\mathcal{T}$ and a timed MPA $\mathcal{A}$, we ask whether $\mathcal{A}$ exhibits any timed scenario that is consistent with $\mathcal{T}$. In other words, we would like to check that $L(\mathcal{T}) \cap L(\mathcal{A})$ is nonempty. This question is natural in the early stages of a specification, when scenarios are not expected to exhaustively describe the system's behaviour.

Sometimes, it is fruitful to describe forbidden scenarios as timed templates. Let $\mathcal{T}$ be such a *negative* template. We then want to check that a timed MPA $\mathcal{A}$ does *not* exhibit a timed scenario consistent with $\mathcal{T}$. In other words, we would like $L(\mathcal{T}) \cap L(\mathcal{A})$ to be empty.

The scenario matching problem for timed MSCs is more complicated than the same problem for untimed MSCs in one obvious way. Even though a timed template is defined with respect to a single underlying MSC, the set of timed MSCs that satisfy a given template is in general infinite. Thus, even with a single template, the matching problem comes down to one of comparing infinite collections of (timed) MSCs.

5.2 Universality

As the specification evolves, it is expected that it more exactly describes the desired behaviour. In an untimed setting, it would be natural at this stage to demand that the

behaviour of the implementation match the specification upto, say, language equivalence. However, in a timed setting, we may have a specification with generous time constraints to be compared with an implementation that is more restrictive. Hence, the natural analogue of language equivalence is to ask whether for every timed MSC template in the specification, there is at least one timed behaviour in the implementation that is consistent with the template. We refer to this problem as *universality*.

6 Using UPPAAL for Scenario Verification

In [4], we present a framework for verifying properties of timed scenarios using UPPAAL, a model checker for timed systems [2]. The framework in [4] is designed to deal with finite sets of timed MSC templates, which can essentially be handled one at a time. Here, we extend this framework to tackle with specifications that encompass a possibly infinite set of scenarios.

UPPAAL supports the analysis of networks of timed automata for timing properties. Unfortunately, UPPAAL does not have a direct way of modelling asynchronous communication. We can simulate asynchronous communication by creating explicit buffer processes. Moreover, we can exploit the synchronous communication paradigm built-in to UPPAAL to synchronize the system with the specification at each communication action. This allows the system to evolve only along trajectories that are consistent with the specification, thus automatically restricting the behaviours of the composite system to those that are of interest.

6.1 Modelling Channels in UPPAAL

Since UPPAAL has no notion of buffered communication, we construct an explicit buffer process for each channel between processes. Message passing is simulated by a combination of shared memory and binary synchronization. Let p and q be processes and let c be the channel between p and q. We create a separate process c which maintains, internally, an array of messages M_{pq} whose size corresponds to the capacity of c. This array is used by c as a circular buffer to store the state of the channel. The process c maintains two pointers into the array: the next free slot into which p can write and the slot at the head of the queue from which q will next read a message.

The channel c shares two variables s_{pc} and r_{cq} with p and q, respectively. These are used to transfer information about the actual message between the processes and the channel. The channel c also uses two special actions a_{pc} and a_{cq} to synchronize with p and q, respectively. These synchronizations represent the actual insertions and deletions of messages into and from the channel.

When p sends a message m to q, it sets the shared variable s_{pc} to m and synchronizes with c on a_{pc}. When c synchronizes with p, it copies the message from s_{pc} into

the array slot that currently corresponds to the end of the queue and then moves the free slot pointer to the next position in the array.

Symmetrically, when q wants to read a message m from p, it sets the shared variable r_{cq} to m and then synchronizes with c on action a_{cq}. In c, this synchronization is guarded by conditions that check that there is at least one message in the queue and that the message at the head of the queue matches the one q is looking for, as recorded in the shared variable r_{cq}.

6.2 Modelling Channel Delays

In an MPA, clocks are local and must be associated with a fixed process. However, UPPAAL permits global clocks. To faithfully model channel delays, we associate an array of clocks with each channel, one for each position in the queue. With universally bounded channels, we can always assign a fresh clock from this array to each new message sent on a channel that is initialized when the message is sent. The receive action for this message is guarded by clock constraints corresponding to the time bounds associated with the channel.

6.3 Modelling Timed MSC Specifications in UPPAAL

To verify a timed MSC specification, the first step is to convert the specification into a timed MPA, preserving the language of timed MSCs of the specification.

For a single timed MSC template, the communication structure of the MPA is fixed and can be computed easily, using the FIFO property of channels. We introduce a new local clock for each local timing constraint and add clock constraints using these clocks to guard the actions of the MPA so that it respects the timed template.

Since we can interpret a general timed MSC specification as an HMSC in which edges are labelled by basic timed MSC templates, we construct an MPA for each basic timed MSC template and connect these up using internal actions and dummy states to reflect the overall structure of the corresponding HMSC.

The usual difficulty with this construction is to ensure that all processes follow consistent paths in the HMSC. For this, we add a monitor process that tracks the path followed by each process in the system. We have to ensure that the information maintained by the monitor process is bounded. As we have observed earlier, with asynchronous concatenation, some processes may be arbitrarily far ahead of others and the overall behaviour may be non-regular. We can ensure regularity by imposing structural restrictions on the HMSC [7]. Instead, we impose a bound on the number of live instances of each basic timed MSC template in the system. This allows us to perform a form of bounded model checking for arbitrary timed template specifications.

6.4 Scenario Matching

We can now augment the system description in UPPAAL so that the evolution of the system to be verified is controlled by the external template specification. Recall that each action corresponding to sending or receiving a message by a local process is broken up into two steps in the UPPAAL implementation, one which sets the value of a shared variable s_{pc} and another which communicates with the buffer process via a shared action a_{pc}. We extend this sequence to a third action, b_{pc}, by which the system synchronizes with the specification. A move of the form $s \stackrel{p!q(m)}{\Longrightarrow} s'$ in the original timed MPA now breaks up, in the UPPAAL implementation, into a sequence of three moves $s \stackrel{s_{pc}=m}{\Longrightarrow} s_1 \stackrel{a_{pc}}{\Longrightarrow} s_2 \stackrel{b_{pc}}{\Longrightarrow} s'$. The third action, b_{pc} synchronizes with the corresponding process p in the timed MPA derived from the timed template that is being verified. Thus, the system can progress via this action only if it is consistent with the constraints specified by the template.

Symmetrically, for a receive action of the form $s \stackrel{p?q(m)}{\Longrightarrow} s'$, the UPPAAL implementation executes a sequence of the form $s \stackrel{r_{pc}=m}{\Longrightarrow} s_1 \stackrel{\bar{a}_{cp}}{\Longrightarrow} s_2 \stackrel{\bar{b}_{cp}}{\Longrightarrow} s'$, where, by convention, an action a synchronizes with a matching action $\bar{a}$.

By construction, it now follows that the timed MSCs executed by the composite system are those which are consistent with both the timed template and with the underlying timed MPA being modelled in UPPAAL. Thus, we have restricted the behaviour of the system to $L(\mathcal{T}) \cap L(\mathcal{A})$, for a given timed template $\mathcal{T}$ and a given timed MPA $\mathcal{A}$. From this, it is a simple matter of invoking the UPPAAL model checker to verify whether this set of behaviours is empty and whether all behaviours in this set satisfy a given property. This answers the scenario verification problems posed in the previous section.

6.5 Universality

Recall that universality is the property that the implementation exhibits at least one timed behaviour consistent with each timed template generated by the specification. In general, we do not know how to solve this problem. Instead, we address a weaker version that we call *coverage*.

We assume that the user provides a (finite) set of paths through the specification that he would like to see exhibited in the implementation. In particular, we can always ensure that we cover all the edges in the HMSC through such a collection of paths. In UPPAAL, we can verify reachability properties written in CTL. This includes formulas that assert that there exists a path along which a sequence of state properties holds. By adding state labels to the UPPAAL implementation, we can mark when a basic timed MSC template is executed by the composite system obtained by synchronizing the specification with the implementation. Each path to be covered can then be

described using an appropriate CTL formula of the form permitted by UPPAAL. The overall problem then reduces to verifying a finite conjunction of such CTL formulas.

7 Discussion

Adding time to specifications of distributed systems appears to be a problem of both practical and theoretical interest.

Augmenting scenarios with timing constraints allows us to specify and verify, more accurately, the interactions associated with typical protocol specifications. Timing constraints give rise to new variants of verification questions, some of which we do not know how to tackle, such as universality.

Global time indirectly synchronizes processes, leading to undecidability—for instance, boundedness of channels is undecidable even if we have only local clocks [10]. It would be interesting to explore whether it is possible to relax the correlation the time across components without completely decoupling all clocks and yet obtain some positive results.

Another interesting theoretical question is to explore the relationship between automata, logic and languages in a setting that incorporates both distribution and time. A first step in this direction is the work reported in [12].

References

1. R. Alur, G. Holzmann and D. Peled: An analyzer for message sequence charts. *Software Concepts and Tools*, **17(2)** (1996) 70–77.
2. G. Behrmann, A. Davida and K.G. Larsen: A Tutorial on Uppaal, *Proc. SFM 2004*, LNCS **3185**, Springer-Verlag (2004) 200–236.
3. J. Bengtsson and Wang Yi: Timed Automata: Semantics, Algorithms and Tools, *Lectures on Concurrency and Petri Nets 2003*, LNCS **3098**, Springer-Verlag (2003) 87–124.
4. P. Chandrasekaran and M. Mukund: Matching Scenarios with Timing Constraints, *Proc. FORMATS 2006*, Springer LNCS 4202 (2006) 98–112.
5. W. Damm and D. Harel: LSCs: Breathing life into message sequence charts. *Formal Methods in System Design* **19(1)** (2001) 45–80.
6. D. de Souza and M. Mukund: Checking Consistency of SDL+MSC Specifications, *Proc. SPIN Workshop 2003*, LNCS **2648**, Springer-Verlag (2003) 151–165.
7. J.G. Henriksen, M. Mukund, K. Narayan Kumar, M. Sohoni and P.S. Thiagarajan: A theory of regular MSC languages. *Inf. Comp.*, **202(1)** (2005) 1–38.
8. G.J. Holzmann: The model checker SPIN, *IEEE Trans. on Software Engineering*, **23**, 5 (1997) 279–295.
9. ITU-T Recommendation Z.120: *Message Sequence Chart (MSC)*. ITU, Geneva (1999).
10. P. Krcal and Wang Yi: Communicating Timed Automata: The More Synchronous, the More Difficult to Verify, *CAV 2006*, LNCS, Springer-Verlag (2006), 249–262.
11. A. Muscholl, D. Peled, and Z. Su: Deciding Properties for Message Sequence Charts. *Proc. FOSSACS'98*, LNCS **1378**, Springer-Verlag (1998) 226–242.
12. Akshay Sunderaraman, *Formal Specification and Verification of Timed Communicating Systems*, Master's thesis, LSV, ENS Cachan (2006). Available at `http://www.lsv. ens-cachan.fr/Publis/PAPERS/PDF/Akshay-M2.pdf`

Using System-Level Timing Analysis for the Evaluation and Synthesis of Automotive Architectures

Marco Di Natale, Wei Zheng, and Paolo Giusto

Abstract Emerging technologies allow the implementation of advanced car features enhancing the safety and the comfort of the driver. These complex functions are distributed among several ECUs, implemented by multiple tasks executed on each processor, and are characterized by non-functional requirements, including timing constraints. The design of the physical architecture and the placement of tasks and messages must be performed in accordance with the constraints and optimizing the performance of the functions. We show how schedulability analysis can be used in the development of complex automotive systems to find the architectures that can best support the target application in a what-if iterative process, and we address the opportunities for the synthesis of architecture configurations. A case study of an experimental vehicle shows the applicability of the approach.

Keywords: Automotive systems, real-time computing, timing analysis, schedulability.

1 Introduction

Past work in electronics/controls/software-based (ECS) vehicle architectures and function development has been fundamentally component or sub-system focused. Recently, however, there has been a clear shift towards the networking of control modules within application domains (e.g. Power train) as well as across domains. This shift has been driven by a large increase in the number of horizontally integrated safety–critical functions (e.g. adaptive cruise control, brake or steer-by-wire), with increasing vehicle control authority, requiring a large number of smart sensors and actuators and often characterized by real-time requirements.

A novel system design methodology and tool support is needed to assist in the design, evaluation, and development of architectures, with quantitative metrics and with early error detection and assessment of the trade-offs. Such a methodology is based on the concept of virtual platforms and enables late-binding design decisions

S. Ramesh and P. Sampath (eds.), *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*, 99–113.

and early verification of them as opposed to early-binding decisions with late verification. This paper focuses on a system-level methodology for quantitative architecture evaluation based on timing constraints and metrics. A case study vehicle is used to illustrate the use of timing analysis for the evaluation and the synthesis of solutions.

1.1 Background

Our approach follows the Platform Based Design methodology [1], where the orthogonalization of concerns between levels of abstractions enables re-use and re-deployment of higher level artifacts onto lower ones.

Because of space constraints, it is impossible to summarize all the research works on models and methods for system level design and analysis. Among them, the Ptolemy [2] and Metropolis [3] frameworks from the University of California, and the GME [4] from Vanderbilt provide support for the integration of heterogeneous models of computations or the creation of domain-specific metamodels. A description of a system-level design flow based on synchronous languages can be found in [5] and SysML [6] from the OMG is a representative of a standardization effort. Other examples specific to the automotive domain are the SymTAS framework for system-level timing analysis [7] and many other commercial tools, including Simulink [8] from Mathworks and ASCET-SD [9] from ETAS.

The focus of this research work is the use of the schedulability analysis theory to evaluate latencies in distributed architectures supporting (hard) real-time applications and the possibility of using optimization techniques for synthesizing at least part of the task and message configuration. Because of resource efficiency, many automotive controls are designed based on run-time priority-based scheduling of tasks and messages [10]. Examples of standards supporting this scheduling model are the OSEK operating system standard [11] and the CAN [12] bus arbitration model.

At the interface between any two resource domains, and very often also at the interface between two abstraction layers (such as, for example, the application and the middleware layers), different interaction models may be implemented. The simplest is the purely periodic activation model, where all interacting tasks and messages are activated periodically and communicate by means of asynchronous buffers implementing a freshest value semantics. Another possible activation model is the data-driven activation [13], where task executions and message transmissions are triggered, respectively, by the arrival of the input data and by the availability of the signal data.

The two models of periodic activation with asynchronous communication and data-driven activation are reconciled by a conceptual framework for the analysis of distributed chains of computations, based on network calculus [14] and its application for evaluating the propagation of event models [7]. In [15] this model is used for distributed schedulability analysis, where the system can be described as an arbitrary mix of data-driven and periodic models. Other works, including [16]

and [17], focused on providing optimal-size lock-free and wait-free communication mechanisms that ensure deterministic delays in the implementation of models integrating both event and time triggered subsystems. However, even if these works provide analysis procedures with increasing quality, the synthesis problem is scantly analyzed: one approach is provided by [18], where genetic algorithms are used for optimizing priority and period assignments with respect to end-to-end deadlines and jitter.

2 A Methodology for Architecture Exploration

We propose a new methodology for architecture exploration based upon an *Iterative Process* in which alternatives are produced, evaluated, and scored according to a set of *Constraints* and *Metrics*.

Complex embedded systems are characterized by safety and timing requirements and metrics, including the verification of deadlines and the evaluation of latencies and jitter. Other non-functional metrics include power and especially cost, with its related secondary metrics including reusability, flexibility, and extensibility. Collectively, these properties cannot be assessed based on an abstract model of the system functions alone, but they depend upon the execution architecture and on the allocation of the functions on the underlying physical architecture. In this work, we focus on the formal evaluation of the timing behavior at the earliest possible stages in the design flow, estimating latencies and extensibility by providing a measure of the available processor and communication time for new functions and messages in product derivatives.

2.1 Functions, Architectures, and Platforms Models

This section briefly introduces the meta-modeling entities and rules that are used to define the models at the three levels of the Platform based design methodology [1] (see Figure 1).

2.1.1 Functional Models

The starting point for the definition of a car electronic/software system is the specification of the set of features, that is, high level system capabilities (e.g. Cruise Control), that the system is expected to provide. Functional models are created from the decomposition of the feature in a hierarchical network of component blocks encapsulating a behavior within a provided and required interface, expressed by a set of ports. The system view abstracts from the details of the functional behavior and models only the interface and the communication and synchronization semantics.

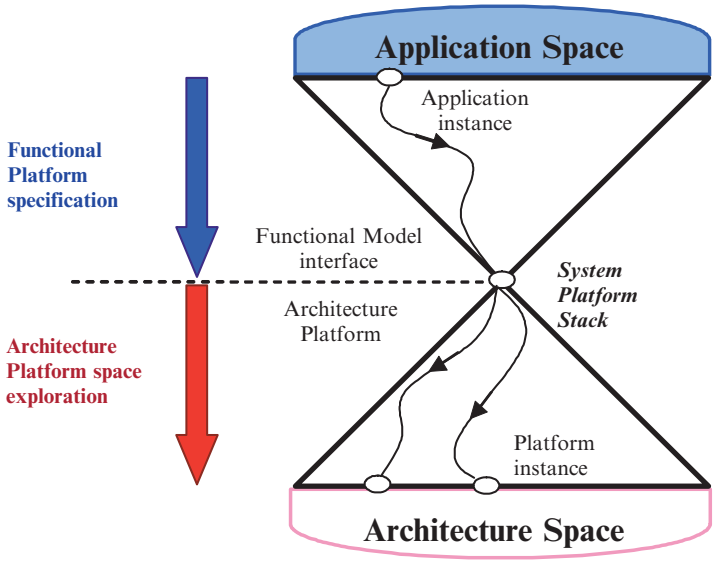


Fig. 1 Platform-based design in architecture exploration

A function label F_i is associated to each block, which computes a set of output values o_i based on a set of inputs i_i and its internal state S_i at the time the block is activated $o_i = F_i(i_i, S_i)$. The nature of the function F_i and its internal state are not relevant for the purpose of our analysis. Functional blocks are either triggered by a clock event or by the arrival of all or some combination of its input signals. If periodic, the function is activated at time instants $a_{i,k}$ with $k = 0 \cdots \infty$ and $a_{i,k+1} - a_{i,k} = T_k$. Each activation instant triggers a function instance $f_{i,k}$. Functional blocks conceptually execute in zero time. The model also allows the definition of precedence constraints and of a set of triggering events or signals activating the model functions. Finally, timing requirements are specified, including end-to-end deadlines on the computation paths, maximum jitter on activation/output signals, and input coherency bounds. The dataflow model is further characterized by the communication semantics. Signal links variables can be *sticky* variables with overwriting (1-place buffer), meaning that the variable preserves the last value written into it and the value can be overwritten by a new output, or they can represent a queue of tokens produced by the sender block and consumed by the receiver.

2.1.1 Architecture Models

The architecture model is hierarchical and captures the topology of the car network, including the communication busses (CAN), the ECU architecture, and the management policies (e.g. priority-driven scheduling) that control the shared resources.

The latter are usually provided by an RTOS, with its scheduling policy, and by the MAC layer and the scheduling policy of the physical communication links.

2.1.1 Mapping and System Platform Model

The system platform model is where the resource requirements for the execution of the functional blocks are expressed. At this level, the units of computation are the tasks or threads executing concurrently in response to environment stimuli, or prompted by an internal clock. Tasks cooperate by exchanging messages and synchronization or activation signals. Tasks and messages contend for use of the execution and communication resources.

The mapping phase consists of allocating each functional block to a task and each communication signal variable to a virtual communication object. If more than one functional block is mapped to a task, the order of the execution must be provided during the mapping phase. As a result of the mapping of the platform model into the execution architecture, the entities in the functional models are characterized with timing execution information derived by worst (best) case execution time analysis. Given a deployment, it is possible to determine which signals are local and which are received/sent over the bus and packed into messages. Each communication signal variable is mapped to a communication resource of the implementation, that is, a message, or a task private variable or a protected shared variable. Each message, in turn, is mapped to a serial data link.

The mapping of the task and message model into the corresponding architecture model and the selection of resource management policies allows the subsequent validation of the system against non-functional constraints. When considering the definition of the functional model and the mapping stage, the opportunities for design synthesis by means of optimization techniques include: the definition of the activation periods of the functions, hence, of the tasks and messages implementing their computation and communication. The synthesis of the communication and activation models, the definition of the placement of the tasks and finally, the definition of the priorities of tasks and messages (encoded into the CAN identifiers for the latter). In this work, we summarize the results of our study on the synthesis of the activation models. The interested reader may refer to [19] for further details.

3 Task and Message Model

Our model for the distributed end-to-end real-time computations is a *dataflow* of tasks, represented with a *Directed Acyclic Graph*. The *model* is a tuple $\{\mathcal{V}, \mathcal{E}, \mathcal{R}\}$, where $\mathcal{V}$ is the set of vertices, $\mathcal{E}$ the set of edges, and $\mathcal{R}$ is the set of shared resources supporting the execution of the tasks (ECU) and the transmission of messages (bus).

$\mathcal{V} = \{o_1, \ldots, o_n\}$ is the set of objects (tasks and messages) implementing the computation and communication of the system. Each object o_i has maximum time

requirement C_i and a resource R_{o_i} that it needs to execute or for its transmission. All objects are scheduled according to their priority. π_i is the priority of o_i and object indexes are assigned by decreasing priority levels. r_i is the worst case response time of o_i, representing the largest time interval from its activation to its completion in case it is a task or its arrival at the destination in case it is a message. w_i is defined as the worst case time spent from the instant the job is released with maximum jitter J_i to its completion or arrival. An object o_i has one *input port* and one or more *output ports*. Input and output ports are used to exchange data and optionally *activation signals* or *events*. At the end of its execution or transmission, an object delivers its results (task) or its data content (message) on all output ports. Each object runs at a base period T_i. It reads its inputs at the time it starts executing, if it is a task, or it samples the incoming signal values and it is enqueued at the activation time in case it is a message.

$\mathcal{E} = \{l_1, \ldots, l_m\}$ is the set of *links*. A link $l_i = (o_h, o_k)$ connects the output port of object o_h (the source) to the input port of object o_k (the sink). Alternatively, a link may be labeled with the indexes of the source and destination task as in $l_{h,k} = (o_h, o_k)$. One object can be the source or sink of many links. A link l_i may carry the activation signal produced when the source object completes its execution or transmission and instantaneously received on the input port of the sink. However, a different communication and synchronization model is possible, where the sink is activated by a periodic timer and, when it executes, reads the latest value that was transmitted over the link. In the following, the source and the sink of link l_i will also be denoted by $\mathrm{src}(l_i)$ and $\mathrm{snk}(l_i)$, respectively.

When an object is activated by the completion of a predecessor we define an *event-driven* activation model. If an object is activated by a single completion event, then the only condition is that its period must be an integer multiple of the predecessor object period. In this case, the activation semantics is of one every k signals. We define a less restrictive activation semantics by allowing an object to be activated by multiple completion events. In this case, the activation is of type AND, meaning that all the predecessor objects on the selected links must be completed in order for the object to be activated. The only allowed case for multiple activation events from multiple incoming links is when the links are connected to predecessor objects having periods that are integer dividers of the target object period, have a unique common predecessor, and are scheduled on the same resource. In this case, we define a set of link groups $\mathcal{G} = \{lg_1, \ldots, lg_k\}$ where each link group $lg_i = \{l_{i0}, \ldots, l_{iki}\}$ has the following properties, $snk(l_{ij}) = snk(l_{il})$ and $\mathrm{R}(\mathrm{src}(l_{ij})) = \mathrm{R}(\mathrm{src}(l_{il}))$ for any link pair $l_{ij}, l_{il} \in lg_i$. If $\tau_{j1} = \mathrm{src}(l_{ij})$ and $\tau_{j2} = snk(l_{ij})$ then $kT_{j1} = T_{j2}$ for some integer k. Finally, $\forall\, lg_i$, $\exists!\, o_p$ such that $\forall\, l_{j,k} \in lg_i$ there exists a link $l_{p,j} \in \mathcal{E}$ and there is no other incoming link to o_j. If all the links in a group carry an activation signal, then the source objects must be activated at the same time or they must all be activated by a completion event. These last conditions do not apply to singleton groups.

$G(o_k)$ is the set of link groups that are incoming to o_k. An object can be activated by a periodic trigger, by a signal coming from a single predecessor object or by the AND composition of signals coming from a single link group. In this last case, the

object is actually activated by the completion of the lowest priority object o_r in the group lg_i, which is called *group representative* $o_r = \text{rep}(lg_i)$.

$\mathcal{R} = \{R_1, \ldots, R_z\}$ is the set of logical resources that can be used by the objects to perform their computations. Resources are either ECUs or buses and are scheduled with a priority-based scheduler.

An *external event* e_i results from the execution of an external virtual object with no input links, representing one entity in the environment. External events can be *periodic* with period T_i and jitter J_i, or sporadic with a minimum interarrival time, equally denoted by T_i. An *output object* o_j is a special functional block with no output link. It represents a consumption by the environment of the data produced by the system and sent to some actuator. For our purposes, an output is merely a stub where execution ends.

A *path* $P(o_i, o_j)$ or $P(i, j)$ is a *functional chain* from o_i to o_j, that is, an ordered sequence $P = [l_1, \ldots, l_n]$ of links that, starting from $o_i = \text{src}(l_1)$, reaches $o_j = \text{snk}(l_n)$ crossing $n + 1$ objects such that $\text{snk}(l_k) = src(l_{k+1})$. A path represents one end-to-end execution of the system, from the triggering of the external event to the generation of the output. The *path deadline* for $P_{i,j}$, denoted by $d_{i,j}$, is the end-to-end constraint for the computation performed in the path.

When task and messages are activated periodically and communicate on a freshest value semantics, the end-to-end latency $L_{i,j}$ associated to a path $P_{i,j}$ is defined as the largest possible time interval that is required for the change of the input at one end of the chain to be propagated to the last task at the other end of the chain, whatever is the state of the tasks in the path and regardless of the fact that some intermediate result may be overwritten.

We assume in this paper that *the application can tolerate the semantic variation when changing from one synchronization model to the other*. In many control applications, the nondeterminism in time introduced by the periodic activation model and the jitter introduced by the event-driven activation can both be tolerated within acceptable ranges.

3.1 Periodic Activation Model

In the periodic activation model (top of Figure 2), the release jitter is zero and the worst case end-to-end latency is computed by adding the worst case response times and the periods of all the objects in the path ($r_k = w_k$).

$$L_{i,j} = \sum_{k:o_k \in P(i,j)} T_k + r_k$$

Given that it is always $r_k = w_k + J_k$, the formula also applies in the case the destination offset is activated with jitter J_k.

Due to unsynchronized timers, in the worst case, the external event arrives right after the completion of the first instance of task o_2 with minimum (negligible)

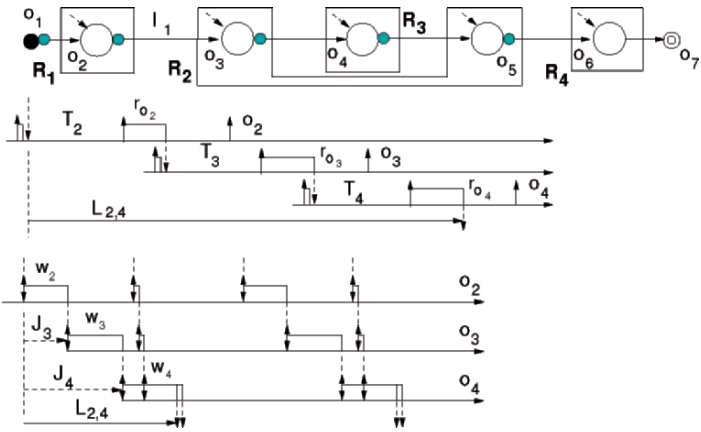


Fig. 2 Periodic and data-driven activation models

response time. The event data will be read by the task on its next instance and the result will be produced after its worst case response time, that is, $T_2 + r_2$ time units after the arrival of the external event. The same reasoning applies to the execution of the following objects.

When communicating tasks run on the same ECU, the relative phase/offsets between periodic tasks can be controlled and sampling delays can be minimized. When the tasks have the same period, the reader task can be activated with a constant relative offset equal to the worst case response time r_w of the writer task and zero sampling delays. Similarly, there is no sampling delay when the tasks have harmonic periods and the reader has the shorter period, i.e. $T_w = nT_r$.

3.2 Data-Driven Activation Model

In the data driven activation model (bottom part of Figure 2), if we assume the same activation period for all the nodes that are activated in a computation chain, then for all the intermediate neighboring nodes $o_i \rightarrow o_j$ it is clearly $r_i = J_j$. The worst case end-to-end latency can be computed for each path by adding the worst case queuing and execution/transmission times of all the objects in the path.

$$L_{i,j} = \sum_{k:o_k \in P(i,j)} w_k$$

3.3 Processor Scheduling

The scheduling paradigms for the tasks is the preemptive fixed-priority based scheduling, implemented by most commercial operating systems, including the

OSEK RTOS standard for automotive. The worst case response time for a periodic task ti, activated with maximum jitter J_i is:

$$\begin{gathered} w_i(q) = (q+1)\,C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i(q)+J_j}{T_j} \right\rceil C_j \\ r_i = \max_q \{J_i + w_i(q) - qT_i\} \end{gathered} \tag{4}$$

For all $q = 0, \cdots, q^*$, until $r_i(q) \leq T_i$.

Where $j \in hp(i)$ refers to the set of tasks such that $\pi_j \geq \pi_i$ and $R_{oi} = R_{oj}$. A lower bound on the worst case response time can be obtained by restricting the computation to the first task instance in the busy period ($q = 0$). This bound is tight in case $r_i \leq T_i$.

$$w_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i + J_j}{T_j} \right\rceil C_j \qquad r_i = J_i + w_i$$

Linear upper and lower bounds for the solution to the previous fixed point equation can be obtained and, if $u_i = C_i/T_i$, a linear combination of the upper and lower bounds, with coefficient α, yields the following:

$$\begin{gathered} \tilde{r}_i(\alpha) = \alpha r_i^{\uparrow} + (1-\alpha) r_i^{\downarrow} \\ \tilde{r}_i(\alpha) = J_i + \frac{C_i + \alpha \sum\limits_{j \in hp(i)} C_j + \sum\limits_{j \in hp(i)} J_j u_j}{1 - \sum\limits_{j \in hp(i)} u_j} \end{gathered} \tag{5}$$

3.4 Bus Scheduling

In this paper, we assume that *message objects are transmitted over CAN buses*. The evaluation of the worst-case latencies for the messages follows the same rules for the worst-case response time of the tasks, with the exception that an additional blocking term B_i must be included in the formula in order to account for the non preemptability of CAN frames, and that the transmission time of the message cannot be preempted [20].

The blocking term B_i for a generic message o_i can be computed as the largest worst-case transmission time of any frame having a priority lower than π_i and sharing the same bus resource ($wq_i > 0$ is the queuing delay part of w_i, without the transmission time).

$$\begin{gathered} wq_i(q) = B_i + qC_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i(q)+J_j}{T_j} \right\rceil C_j \qquad (w_i(q) > 0) \\ w_i = \max_q \{C_i + wq_i(q) - qT_i\} \\ r_i = w_i + J_i \end{gathered} \tag{6}$$

for all $q = 0, \cdots, q^*$, until $r_i(q) \leq T_i$.

A lower bound on w_i and r_i can be computed by only considering the first instance ($q = 0$) and, similar to processor scheduling, the response times of messages can be approximated by linear functions of the jitter variables.

$$\tilde{r}_i(\alpha) = J_i + C_i \frac{B_i + \alpha \sum_{j \in hp(i)} C_j + \sum_{j \in hp(i)} J_j u_j}{1 - \sum_{j \in hp(i)} u_j} \tag{7}$$

The activation on event model brings the potential for a substantial reduction of the end-to-end latencies. However, it must be carefully used, since the increased jitter in the activation of high priority tasks and messages causes bursts of load requests on the ECUs and the buses, possibly resulting in large worst case response times for low priority tasks and messages.

4 Synthesis of the Activation Model

A mixed integer linear programming formulation can be used to find a solution with respect to the deadline constraints on the paths. In addition to r_i, J_i, w_i, $L_{s,t}$ we define $y_{h,k}$ as

$$y_{h,k} = \begin{cases} 1 & \text{if the activation of } o_k \text{ is event-driven by } o_h \\ 0 & \text{otherwise} \end{cases}$$

The ***feasibility constraints*** are defined according to the rules for computing the jitter, the response times and the latencies at all nodes in the graph. Consider a scheduled object o_k with multiple incoming link groups. We are only interested in those groups (links) that can possibly carry an activation signal (for all the other links $l_{j,k}$ it is clearly $y_{j,k} = 0$). All the links in one group must assume the same activation model.

$$y_{r,k} = y_{s,k} \tag{8}$$

for all the pairs $l_{r,k}, l_{s,k}$ belonging to the same group lg_h.

If o_k has more than one incoming link group, only one of the group representatives can provide its activation signal. For each object o_k it must be

$$\sum_{\mathrm{lg}_h \in G(o_k)} y_{r,k} \leq 1 \quad \text{where } o_r = rep(\mathrm{lg}_h)$$

If all group links have a periodic activation (all $y_{r,k} = 0$) then o_k is activated periodically and $J_k = 0$. Otherwise, J_k will be equal to the response time of the representative object in the group from which the activation signal is received. The two alternative ways of computing J_k can be encoded in a pair of constraint sets leveraging the big M formulation in use in integer linear programming (depending on the value of $y_{r,k}$).

$$\begin{gathered} J_k \leq \sum_{\lg_h \in G(o_k)} y_{r,k} \times M \quad \text{where } o_r = rep(\lg_h) \\ 0 \leq J_k \end{gathered} \tag{9}$$

If all $y_{r,k} = 0$, then $J_k = 0$. If $y_{r,k} = 1$ for one of the incoming link groups, then J_k is equal to the worst-case response time r_r of the predecessor object o_r that is the representative of the activating group.

$$\begin{array}{ll} J_k \leq r_r + (1 - y_{r,k}) \times M & \text{where } o_r = rep(\lg_h) \\ r_r - (1 - y_{r,k}) \times M \leq J_k & \text{where } o_r = rep(\lg_h) \end{array} \tag{10}$$

The worst-case response time r_h for object o_h can be computed as

$$r_h = w_h + J_h \tag{12}$$

Because of the non-linearity and even non-convexity of the fixed point formula that provides the exact value of w_h, a linear combination with coefficient $\alpha \in [0,1]$ of the linear upper and lower bounds (5,7) is used.

$$w_h = C_h + \sum_{o_k \in hp(h)} \left(\frac{w_h + J_k}{T_k} + \alpha \right) C_k \tag{13}$$

If o_h is a task and a similar formulation from (7) is used in case o_h is a message, where α is chosen as to minimize the following mean square fit function, computed for all $y = 0$ and assuming α goes not depend significantly on the value of the y variables.

$$\sum_{P_r \in P} (\alpha L^{\uparrow}_{P_r} + (1 - \alpha) L^{\downarrow}_{P_r} - L_{P_r})^2 \tag{15}$$

where $L^{\uparrow}_{Pr}$ and $L^{\downarrow}_{Pr}$ are the latencies computed on the path P_r using the upper and the lower linear bound, respectively. Finally, a variable $z_{i,j}$ is defined for each link $l_{i,j}$ to express the link contribution to the end-to-end latencies of the paths containing it. $z_{i,j}$ is equal to w_j if the link $l_{i,j}$ carries an activation event, otherwise, $z_{i,j} = w_j + J_j + T_j$.

$$\begin{gathered} w_j \leq z_{i,j} \\ z_{i,j} \leq w_j + (1 - y_{i,j}) \times M \\ z_{i,j} \leq w_j + J_j + T_j \\ w_j + J_j + T_j - y_{i,j} \times M \leq z_{i,j} \end{gathered} \tag{16}$$

The end-to-end latency $L_{s,t}$ associated with path $P_{s,t}$ is computed as

$$L_{s,t} = \sum_{l_{u,v} \in P_{s,t}} z_{u,v} \tag{17}$$

and should not exceed its deadline.

$$L_{s,t} \leq d_{s,t}. \tag{18}$$

In addition to get a feasible solution, which satisfies the deadline constraints, we can get the optimal solution with respect to different ***objective functions***. For example, the minimization of the number of event buffers

$$\text{maximize} \sum_{\lg_h \in G} y_{j,k} \quad \text{where} o_j = rep(\lg_h)$$

Other possible cost functions are the sum of the end-to-end latencies, or the weighted sum of the positive differences between the end-to-end latencies and the corresponding deadline over all the paths in the system.

$$\sum_{p_r \in P} L_{p_r} \qquad \sum_{p_r \in P} \gamma_{p_r} \max(L_{p_r} - d_{p_r}, 0) \tag{19}$$

5 The Case Study Vehicle

In this section we illustrate the methodology and the synthesis procedure on a case study of an experimental vehicle.

The vehicle supports advanced distributed functions with end-to-end computations collecting data from 360° sensors to the actuators, consisting of the throttle, brake and steering subsystems and of advanced HMI (Human–Machine Interface) devices. To give an idea of the complexity, the functional model resulting from the decomposition of the vehicle features results in a quite complex network of functional subsystems, with more than 1000 signals exchanged among functional blocks.

Ten pairs of endpoints have been identified in the graph as sources and destinations of computation paths with deadlines. Among them, one of the main concerns is the timing performance of features using the vision system to control/augment the steering command. An analysis of the graph found more than 200 paths between these 10 pairs of nodes and deadlines ranging from 100 to 300 ms have been defined for them.

However, it is worth saying that in the case study, the deadlines should not be considered as safety–critical hard constraints, but rather as very desirable performance targets.

5.1 Architecture Selection

Using the iterative analysis methodology and the associated tools, we analyzed several architectures for the support of the case study features. The outcome of the analysis guided the selection of the options.

The candidate architectures have been defined by trying different bus and ECU topologies and by changing the task placement and the definition of the messages, the message and task activation models and, to a limited extent, the task and message periods. For each architecture option, the end-to-end latencies have been computed

Table 1 Latency results for the case study

Feature	Opt1	Opt2	Opt3	Opt4	Opt5	Opt6	Opt7
feature1	2,43	0,89	1,74	1,74	1,74	1,30	0,73
feature2	3,94	1,09	1,82	1,82	1,82	1,82	0,80
feature3	2,26	2,38	2,38	2,40	1,77	1,77	1,12
feature4	1,72	1,60	1,60	1,63	1,63	1,63	1,02
feature5	2,31	2,38	2,38	2,43	1,80	1,80	1,07

according to the communication and activation models of the system, using the previously described worst case response time analysis techniques.

Table 1 summarizes the latencies computed for five features with end-to-end timing requirements when considering seven possible architecture alternatives. For IP protection, feature names have been omitted and all latencies are normalized with respect to the deadlines, assumed equal to 1.00. For each architecture option, the table shows the worst case latencies. The analysis helped in assessing the trade-offs of the possible options, by looking at the latency of the end-to-end computation paths associated to the features and the ECU and bus utilizations, which give a measure of the expected extensibility with respect to future functions.

In the first option, for one of the features, labeled as *feature1* in the table, the end-to-end path spans five ECUs and four buses with the associated delays caused by the middleware level polling tasks and by the message latencies. The corresponding worst case latency is evaluated at about 2.5 times the requirement. The result is quite far from the deadline constraint, but sampling is found to be a large part of the latency. To mitigate the effects of sampling, mappings yielding a lower number of bus hops have been explored, and several messages have been defined to be sent on event. The last option shows the possible trade-offs in the architecture selection. The end-to-end path now only spans three busses and allows for a much shorter latency (see table above). On the other hand, more features have been mapped into the ECU at the steering actuator, increasing its utilization with concerns for future extensibility.

5.2 Optimization of the Activation Modes

The ILP optimization approach has been applied to one of the architecture configurations considered for the vehicle. The optimization method was applied after the actual architecture selection stage and did not affect its design, but the architecture configuration was nevertheless used to validate the assumptions made in the development of the MILP formulation and to prove its effectiveness.

The architecture considered for the optimization consists of 38 nodes connected by 6 CAN buses, with speeds from 25 to 500 kb/s. A total number of 100 tasks are executed on the ECU nodes, supporting from 1 to 22 tasks each, and 322 messages are exchanged over the six buses, with a minimum and maximum number of messages

of, respectively, 32 and 145 for each group. The number of links in the dataflow graph is 507. Bus utilizations are between 30% and 50% and CPU utilizations are estimated between 5% and 60%.

If all tasks and messages are activated periodically, the end-to-end latencies largely exceed (in the worst case) the desired deadlines. For example, a worst-case latency of 577 ms was found for paths with deadline 300, 255.5 for paths with deadline 200, and 145.38 for paths with deadline 100. Of the 507 links, 313 are subject to optimization, including link groups. The sum of the end-to-end latencies was used as the metric function. The problem encodes results in 1673 variables, 313 of which are binary, and 3989 linear constraints. The time required to solve the problem was always close to 0.25 seconds (1.4 GHz PC).

In its original formulation, the problem does not have a solution, because of a path in which most of the links are constrained to be periodic. After reducing the period of one of the messages in the path from 100 to 50 ms, the problem admits a solution. After the first optimization round, the end-to-end latencies were much closer to the desired deadlines, but still not feasible for 12 of the 148 paths. It was necessary to change the period of one more task (from 12.5 to 10) and one more message (from 100 to 80), making it shorter so that an event driven activation could be defined on the corresponding incoming and outgoing links.

After another optimization round, all the latencies became lower than the deadlines, with the largest value of 242 for paths with deadline 300, 145.5 for paths with deadline 200, and 95.4 for paths with deadline 100.

The final result of the optimization was the definition of 116 links and 3 groups (141 total links) to carry an event-driven activation signal.

The value of α changed from 0.465, at the start, when all $y = 0$, to 0.459 for the final solution. When repeating the optimization procedure with the new value of α, the same result was obtained, therefore supporting the validity of our linear approximation assumption.

6 Conclusions

We present a design methodology based on the use of timing analysis for the evaluation of architecture configurations and an optimization algorithm that leverages the trade-offs between the purely periodic and the data-driven activation models to meet the latency requirements of distributed vehicle functions. We demonstrate its application to the analysis and optimization on a complex automotive architecture. In the future, besides the assignment of priorities to tasks and messages, we plan to explore the problem of finding the optimal placement of the tasks on the ECUs.

One of the biggest problems in the application of the methodology is the lack of detailed data in the early phases of design – for example, the need for an estimate of the worst case execution of tasks, often unknown until late in the design process.

Sensitivity analysis can help in this case, by assessing the effect of uncertainty in the input data with respect to the latency results.

The authors would like to thank Jeff Hoorn from GM Engineering and Sri Kanajan from GM Research.

References

1. Sangiovanni Vincentelli A. Defining Platform-based Design. EEDesign of EETimes, February 2002.
2. Lee E.A., Overview of the Ptolemy Project, Technical Memorandum UCB/ERL M03/25, July 2, 2003, University of California, Berkeley, CA, 94720, USA.
3. Balarin F., Hsieh H., Lavagno L., Passerone C., Sangiovanni-Vincentelli A., and Watanabe Y., Metropolis: An Integrated Environment for Electronic System Design, IEEE Computer, April 2003.
4. Akos Ledeczi et al. The Generic Modeling Environment, Workshop on Intelligent Signal Processing, Budapest, Hungary, May 17, 2001.
5. G. Berry, M. Kishinevsky, S. Singh, System Level Design and Verification Using a Synchronous Language, Tutorial ICCAD'03, San Jose.
6. The OMG Systems Modeling Language web page, http://www.omgsysml.org/
7. A. Hamann, R. Henia, M. Jerzak, R. Racu, K. Richter, and R. Ernst, SymTA/S symbolic timing analysis for systems, available at http://www.symta.org, 2004.
8. The Mathworks Simulink and StateFlow. web page: http://www.mathworks.com.
9. ASCET-SD web page, available at http://en.etasgroup.com/products/ascet/
10. M. G. Harbour, M. Klein, and J. Lehoczky. Timing analysis for fixed-priority scheduling of hard real-time systems. IEEE Transactions on Software Engineering, 20(1), January 1994.
11. OSEK. Osek os version 2.2.3 specification. http://www.osek-vdx.org, 2006.
12. R. Bosch. Can specification, version 2.0. Stuttgart, 1991.
13. K. W. Tindell. Holistic schedulability analysis for distributed hard real-time systems. Tech. Report YCS197, Computer Science Dept., University of York, 1993.
14. J. Y. L. Boudec and P. Thiran, Network calculus – a theory of deterministic queuing systems for the internet, in LNCS 2050, Springer, 2001.
15. S. Chakraborty and L. Thiele, A new task model for streaming applications and its schedulability analysis, in IEEE DATE Conference, Munich, March 2005.
16. S. Tripakis, C. Sofronis, N. Scaife, and P. Caspi, Semantics-preserving and memory-efficient implementation of inter-task communication on static-priority or edf schedulers, 5th ACM EMSOFT Conference, 2005.
17. M. Baleani, A. Ferrari, L. Mangeruca, and A.S. Vincentelli, Efficient embedded software design with synchronous models, 5th ACM EMSOFT Conference, 2005.
18. R. Racu, M. Jersak, and R. Ernst. Applying sensitivity analysis in real-time distributed systems. In Proceedings of the 11th RTAS Conference, pages 160–169, San Francisco (CA), U.S.A., March 2005.
19. Wei Zheng, Marco Di Natale, Claudio Pinello, Paolo Giusto, Alberto L. Sangiovanni-Vincentelli: Synthesis of task and message activation models in real-time distributed automotive systems. DATE 2007, Paris, April 2007, pages 93–98.
20. K. Tindell, A. Burns, and A.J. Wellings. Calculating controller area network (can) message response times. Control Eng. Practice, 3(8):1163–1169, 1995.

Verifiable Design of Asynchronous Software

Prakash Chandrasekaran, Christopher L. Conway, Joseph M. Joy, and Sriram K. Rajamani

Abstract Existing tools for static analysis of programs are able to analyze sequential programs, for properties that do not involve reasoning about the heap. Asynchronous software does not obey either of these requirements: (1) control flow is not sequential, (2) operations need to store state in the heap. We propose a programming language, CLARITY, which enables verifiable design of asynchronous software.

Keywords: asynchronous software, static analysis, programming languages

Over the past decade, we have witnessed increasing interest in static analysis tools that "inspect" programs at compile time [1–5]. These tools are able to both find errors, and prove that certain kinds of errors cannot occur at runtime. With the exception of a few efforts, most of these tools deal with existing programs written in existing programming languages. These tools can perform scalable whole-program inter-procedural analysis for sequential programs on properties that do not involve reasoning about the heap, such as locking discipline errors and use-after-free errors involving pointers from the stack. Once an object is put in a heap data structure, such as a linked list or queue, these techniques lose precision and become ineffective.

Event-driven programs are non-sequential, asynchronous, and maintain state in the heap for most operations. Thus, most current static analysis tools can check only limited properties of such programs. An enormous amount of research effort has gone into improving the precision and scalability of static analysis for concurrent programs and heap data, but the performance of these analyses continues to be a significant problem. We propose changing the statement of the problem: *Can we write event-driven programs differently, so that they become more analyzable?*

We introduce a programming language, CLARITY, which enables analyzable design of asynchronous components. The essence of the CLARITY programming style is the separation of computation and coordination: we define a set of high-level coordination primitives, or *coords*, which encapsulate common interactions between asynchronous components; logical operations are defined sequentially, using coords and event-based communication to indicate synchronization requirements. Each coord has a protocol declaration defining the correct usage of its coordination interface. A CLARITY thread using the coord must follow the protocol along all code paths.

S. Ramesh and P. Sampath (eds.), *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*, 115–116.

Several design decisions make CLARITY programs easier to analyze: code annotations delegate protocol obligations to exactly one thread at each asynchronous function call, making the behavior of a thread with respect to each coord effectively sequential. Using the coord protocol, a CLARITY program can be analyzed using simple compositional reasoning: first, we can check that the program follows the protocol, using a purely sequential analysis; then, assuming the program follows the protocol, we can verify that the implementation of the coord does not have deadlocks or assertion violations.

References

1. T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN 01: SPIN Workshop*, volume 2057 of *LNCS*. Springer-Verlag, 2001.
2. S. Chaki, E. M. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. *IEEE Transactions on Software Engineering*, 30(6):388–402, 2004.
3. M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *PLDI*, pages 57–69. ACM, 2002.
4. D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI*. Usenix Association, 2000.
5. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, pages 58–70. ACM, 2002.

Approximate Symbolic Reachability of Networks of Transition Systems

Sudeep Juvekar, Ankur Taly, Varun Kanade, and Supratik Chakraborty

Abstract Symbolic reachability analysis of networks of state transition systems present special optimization opportunities that are not always available in monolithic state transition systems. These optimizations can potentially allow scaling of reachability analysis to much larger networks than can be handled using existing techniques. In this paper, we discuss a set of techniques for efficient approximate reachability analysis of large networks of small state transition systems with *local interactions*, and analyse their relative precision and performance in a BDD-based tool. We use overlapping projections to represent the state space, and discuss optimizations that significantly limit the set of variables in the support set of BDDs that must be manipulated to compute the image of each projection due to a transition of the system. The ideas presented in this paper have been implemented in a BDD-based symbolic reachability analyser built using the public-domain symbolic model checking framework of NuSMV. We report experimental results on a set of benchmarks that demonstrate the effectiveness of our approach over existing techniques using overlapping projections.

Keywords: state transition systems, symbolic reachability, approximation

1 Introduction

Large and complex systems are often built by interconnecting small and simple components. A large class of such systems can be behaviorally modeled as networks of interacting state transition systems, where each individual state transition system, or *component*, has a simple transition structure and involves only a few state variables. Examples of such systems abound in practice, e.g. circuits obtained by interconnecting logic gates and flip-flops, distributed control systems with interacting sensors, controllers and actuators, a collection of devices communicating through a shared bus and arbiter, etc. The reachable state space of such a system can be computed by starting from a specified initial state of all components and by non-deterministically choosing and atomically executing an enabled state transition from the individual components. This produces a change of state of one or more components, and hence

S. Ramesh and P. Sampath (eds.), *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*, 117–136.

of the overall system. Interactions between components can be modeled by sharing state variables and by executing synchronized transitions between components. The above process can then be repeated until all reachable states of the system are explored. If there are k components in the system, and if Σ_i denotes the set of state variables of the i^{th} component, the state variables of the overall system is given by $\Sigma = \cup_{i=1}^{k} \Sigma_i$. Even when $|\Sigma_i|$ is small for each i, $|\Sigma|$ can be large for large values of k. Since the complexity of reachability analysis grows exponentially with the number of state variables, reachability analysis of a large network of components is computationally far more difficult than searching the state spaces of individual components separately.

The additional computational effort needed to search the state space of a large network of components is primarily for reasoning about interactions between components. Interestingly, however, in a large class of practical systems, components primarily interact locally with a few other components in their "neighbourhood". More formally, state transitions of one component affect the state variables of only a few other components. This is not surprising since systems are often designed in a modular and compositional way, where individual components are required to interface and interact locally with a few other components in their spatial neighbourhood. While a few components may interact globally with other components, even these global interactions can often be modeled by synchronized local interactions, as we will see later in Section 2. Thus, interactions between components in a large class of practical systems are largely local in nature. This presents significant opportunities for optimization when performing reachability analysis. In this paper, we exploit these opportunities to design highly scalable Binary Decision Diagrams (BDD)-based [1] techniques for efficiently searching state spaces of large networks of state transition systems with local interactions.

If the behaviour of each component in a network is independent of that of others, the reachable state space of the overall system can be obtained by computing the Cartesian product of the reachable state spaces of individual components. Interactions between components however render such an analysis highly conservative in practice. Traditional symbolic reachability analysers [4] therefore require the transition relations of individual components to be combined into a single system-wide transition relation involving all state variables in Σ. Since representing and manipulating BDDs with thousands of variables in the support set is computationally prohibitive even with state-of-the-art public-domain BDD packages like CUDD [8], BDD-based tools that work with system-wide transition relations do not scale well to large networks. To address this problem, earlier researchers have considered using partitions [3] and overlapping projections [6] of state variables. Govindaraju's approach [6] based on the *multiple constrain* operator is currently among the best BDD-based approaches for computing over-approximations of the reachable state space using *overlapping projections*. Yet other approaches [2, 7] have attempted to characterize BDDs on-the-fly during reachability analysis, and use appropriate approximations to achieve a trade-off between efficiency and accuracy. While these techniques have been used to efficiently compute good over-approximations of reachable state spaces of some large systems, they still require operations (e.g. the multiple

constrain operation in [6]) on BDDs that have almost all variables in Σ in the support set. This makes it impractical to use these techniques for networks of transition systems with thousands of state variables. This difficulty was also observed during our experiments, where the technique of Govindaraju [6] could not compute an over-approximation of the reachable discrete-timed state space of an interconnection of timed logic gates having 6242 state variables in 60 minutes on a moderately powerful computing platform. In this work, we wish to address such scalability issues by designing approximate BDD-based reachability analysis techniques that scale to very large networks without compromising much on accuracy.

Existing techniques, including that of Govindaraju [6], use information about locality of interactions between different components to choose a good set of overlapping projections, but not to optimize the reachability analysis per se. In this paper, we wish to go a step further and exploit locality of interactions to optimize BDD-based reachability analysis using overlapping projections. While the method of Govindaraju provably gives the best over-approximation of the reachable state space using overlapping projections, we show that by exploiting locality of interactions during image computation, we can gain significantly in efficiency without suffering much precision-wise. Significantly, our technique permits scaling the analysis to networks of transition systems with variable counts at least an order of magnitude higher than those analysable using Govindaraju's technique.

The remainder of the paper is organized as follows. Section 2 describes networks of state transition systems and presents a set of techniques to optimize reachability analysis of large networks using locality of interactions. Section 3 discusses experiments for evaluating the optimized analysis techniques, and compares the performance and accuracy of these techniques with each other as well as with Govindaraju's technique. Finally, Section 4 concludes the paper.

2 Networks of State Transition Systems

We represent a state transition system B as a 4-tuple (Σ, Q, I, T), where Σ is a finite set of state variables, Q is the set of all states, $I : Q \rightarrow \{\mathsf{True}, \mathsf{False}\}$ is an initial state predicate, and $T : Q \times Q \rightarrow \{\mathsf{True}, \mathsf{False}\}$ is a transition relation predicate. Each state variable $s \in \Sigma$ has an associated *finite* domain $\mathcal{D}_s$, and Q is the Cartesian product of the finite domains corresponding to all variables in Σ. When describing a state transition, we use unprimed letters to refer to values of variables in the present state, and the corresponding primed letters to refer to their values in the next state. Let Σ' denote the set $\{s' \mid s \in \Sigma\}$. Thus, I is a predicate with free variables Σ, while T is a predicate with free variables $\Sigma \cup \Sigma'$. We will henceforth refer to these predicates as $I(\Sigma)$ and $T(\Sigma, \Sigma')$, respectively.

A *network* of state transition systems is a collection of state transition systems (with possibly overlapping sets of state variables), and a specification of synchronized transitions between them. Let $\mathcal{B} = \{B_1, \ldots B_m\}$ be a set of state transition systems that interact to form a network $\mathcal{N}$. Each B_i is a 4-tuple $(\Sigma_i, Q_i, I_i, T_i)$

and is called a *component* of $\mathcal{N}$. Components B_i and B_j are said to execute state transitions *synchronously* iff every state transition of B_i occurs simultaneously with a state transition of B_j. Thus, state transitions of B_i and B_j cannot be interleaved. We specify synchronization between components by an undirected graph $\mathcal{H} = (\mathcal{B}, E_{\mathcal{H}})$, where $\mathcal{B}$ is the set of components and $(B_i, B_j) \in E_{\mathcal{H}}$ iff components B_i and B_j execute state transitions *synchronously*. It is easy to see that the binary relation on components defined by synchronous execution of transitions is an equivalence relation. Hence, the graph $\mathcal{H}$ consists of a set of disconnected cliques. If a clique consists of only a single component, we say that the corresponding component executes state transitions *asynchronously* with other components. Indeed, its state transitions can be interleaved arbitrarily with those of other components. The network $\mathcal{N}$ is formally defined by the tuple $(\mathcal{B}, \mathcal{H})$.

Given a network $\mathcal{N} = (\mathcal{B}, \mathcal{H})$, the overall state transition system corresponding to the network is given by $B_{\mathcal{N}} = (\Sigma_{\mathcal{N}}, Q_{\mathcal{N}}, I_{\mathcal{N}}, T_{\mathcal{N}})$, where $\Sigma_{\mathcal{N}} = \cup_{i=1}^{m} \Sigma_i$, $Q_{\mathcal{N}}$ is the Cartesian product of the finite domains corresponding to variables in $\Sigma_{\mathcal{N}}$, and $I_{\mathcal{N}}(\Sigma_{\mathcal{N}}) = \bigwedge_{i=1}^{m} I_i(\Sigma_i)$. In order to determine $T_{\mathcal{N}}(\Sigma_{\mathcal{N}}, \Sigma'_{\mathcal{N}})$, we need to model the effect of synchronous transitions of each clique in $\mathcal{H}$. Let $C = \{B_i, \ldots, B_k\}$ be a clique in $\mathcal{H}$. Let $\Sigma_C = \Sigma_i \cup \cdots \Sigma_k$ and $\overline{\Sigma_C} = \Sigma_{\mathcal{N}} \setminus \Sigma_C$. We will henceforth use this notation to denote the set of variables corresponding to a collection of components, and the complement of a set of variables, respectively. We say that the network changes state from q to q' due to a synchronous transition of components in C iff (q, q') satisfies the predicate $\bigwedge_{B_i \in C} T_i(\Sigma_i, \Sigma_i') \wedge \bigwedge_{s \in \overline{\Sigma_C}} (s \leftrightarrow s')$. Let this predicate be called $\Upsilon_C(\Sigma_{\mathcal{N}}, \Sigma'_{\mathcal{N}})$. If $Cliques(\mathcal{H})$ denotes the set of cliques of $\mathcal{H}$, the transition predicate of the overall state transition system $B_{\mathcal{N}}$ is given by $T_{\mathcal{N}}(\Sigma_{\mathcal{N}}, \Sigma'_{\mathcal{N}}) = \bigvee_{C \in Cliques(\mathcal{H})} \Upsilon_C(\Sigma_{\mathcal{N}}, \Sigma'_{\mathcal{N}})$. For clarity of notation, we will henceforth omit the subscript $\mathcal{N}$ from Σ, Q, I and T whenever the network $\mathcal{N}$ is clear from the context.

2.1 Reachability Analysis of Networks of Transition Systems

Let $N2P$[1] be a predicate transformer that transforms a predicate $R(\Sigma, \Sigma')$ by replacing every occurrence of s' in R with the corresponding $s \in \Sigma$, for every $s' \in \Sigma'$. Formally,

$$N2P\left(R(\Sigma, \Sigma')\right) = \exists \Sigma' \left(R(\Sigma, \Sigma') \wedge \bigwedge_{s \in \Sigma} (s \leftrightarrow s') \right) \tag{1}$$

Now consider a predicate $R(\Sigma, \Sigma')$ that has at most one of s and s' (but not both) as free variable, for every $s \in \Sigma$. Such a predicate can be written as $R(\Sigma_*, \overline{\Sigma_*}')$, where Σ_* is the set of all variables $s \in \Sigma$ that appear as free variables of R, and $\overline{\Sigma_*} = \Sigma \setminus \Sigma_*$. The effect of transforming R by $N2P$ is given by $N2P(R(\Sigma_*, \overline{\Sigma_*}')) =$

[1] $N2P$ stands for "next-to-present".

$\exists \Sigma' \left(R(\Sigma_*, \overline{\Sigma_*}') \wedge \bigwedge_{s \in \Sigma}(s \leftrightarrow s')\right) = R(\Sigma_*, \overline{\Sigma_*})$. Effectively, $N2P$ renames a subset of free variables of $R(\Sigma_*, \overline{\Sigma_*}')$. For predicates on Boolean variables, such renaming can be efficiently performed in BDD packages like CUDD. For example, if the BDD for $R(\Sigma_*, \overline{\Sigma_*}')$ is given, the `bdd_permute` operation in CUDD achieves the effect of renaming variables. In the following discussion, whenever we apply $N2P$ to a predicate, the property that at most one of s and s' occurs as free variable, holds for the predicate. Therefore, assuming that we are using a BDD package like CUDD that allows efficient renaming of variables, $N2P$ can be considered an efficiently computable operation.

Let $R(\Sigma)$ be the characteristic predicate of a set of states. Henceforth, we will refer to sets of states and their characteristic predicates interchangeably. The *image* of the set $R(\Sigma)$ under $T(\Sigma, \Sigma')$, denoted $Im(R(\Sigma), T(\Sigma, \Sigma'))$, can be obtained as $N2P(\exists \Sigma\, (R(\Sigma) \wedge T(\Sigma, \Sigma')))$. Given a network $\mathcal{N} = (\mathcal{B}, \mathcal{H})$ of state transition systems, the set of reachable states of $B_{\mathcal{N}}$ can be obtained by initializing the reachable set with the initial set of states, and by repeatedly computing the image of the current reachable set under $T(\Sigma, \Sigma')$ until no further new states are obtained. Since $T(\Sigma, \Sigma')$ is a disjunction of $\Upsilon_C(\Sigma, \Sigma')$ for $C \in Cliques(\mathcal{H})$, computing the image of a set of states $R(\Sigma)$ under $T(\Sigma, \Sigma')$ is equivalent to computing the image of $R(\Sigma)$ under each $\Upsilon_C(\Sigma, \Sigma')$ individually and then taking the union of the resulting sets of states. Note that in general, each $\Upsilon_C(\Sigma, \Sigma')$ has all variables in $\Sigma \cup \Sigma'$ as free variables. Since $|\Sigma|$ can indeed be large (several thousand variables) for a large network, computing the image of a set $R(\Sigma)$ under $\Upsilon_C(\Sigma, \Sigma')$ as $N2P(\exists \Sigma\, (R(\Sigma) \wedge \Upsilon_C(\Sigma, \Sigma')))$ does not scale well in BDD-based tools. However, there is an obvious optimization that can be done. On closer examination of the structure of $\Upsilon_C(\Sigma, \Sigma')$, i.e. $\bigwedge_{B_i \in C} T_i(\Sigma_i, \Sigma_i') \wedge \bigwedge_{s \in \overline{\Sigma_C}}(s \leftrightarrow s')$, we find that the values of all state variables in $\overline{\Sigma_C}$ are preserved in the next state. Since $\Sigma = \Sigma_C \cup \overline{\Sigma_C}$, we can write $R(\Sigma)$ as $R(\Sigma_C, \overline{\Sigma_C})$. Therefore, the image expression $N2P(\exists \Sigma\, (\bigwedge_{B_i \in C} T_i(\Sigma_i, \Sigma_i') \wedge \bigwedge_{s \in \overline{\Sigma_C}} (s \leftrightarrow s') \wedge R(\Sigma_C, \overline{\Sigma_C})))$ can be simplified to $N2P(\exists \Sigma_C\, (\bigwedge_{B_i \in C} T_i(\Sigma_i, \Sigma_i') \wedge R(\Sigma_C, \overline{\Sigma_C}')))$. Using the definition of $N2P$ from equation (1), and noting that the quantification inside $N2P$ eliminates only variables in Σ_C, which is mutually disjoint with $\overline{\Sigma_C}$, we obtain the following equivalent expression for the image:

$$Im\left(R(\Sigma), \Upsilon_C(\Sigma, \Sigma')\right) = N2P\left(\exists \Sigma_C \left(\bigwedge_{B_i \in C} T_i(\Sigma_i, \Sigma_i') \wedge R(\Sigma_C, \overline{\Sigma_C})\right)\right) \quad (2)$$

Notice that the quantification in the final expression is over Σ_C which is potentially much smaller than Σ. Similarly, we have eliminated the potentially large conjunction $\bigwedge_{s \in \overline{\Sigma_C}}(s \leftrightarrow s')$ from the image computation step. In the following discussion, we will refer to this optimization as *"reducing non-transition variables"*. Unfortunately, even with this optimization, $R(\Sigma_C, \overline{\Sigma_C})$ involves all variables in Σ, and hence the scalability problem continues to exist.

To address this problem, we propose to exploit the fact that in a large class of practical systems, interactions between components are local in nature. Thus, state

transitions of a component B_i change the state variables of only a small number of other components. Given a network $\mathcal{N} = (\mathcal{B}, \mathcal{H})$, we can capture this locality of interactions by an undirected graph $\mathcal{G} = (\mathcal{B}, E_G)$, where $\mathcal{B}$ is the set of components and $(B_i, B_j) \in E_G$ iff components B_i and B_j share some state variables, i.e. $\Sigma_i \cap \Sigma_j \neq \emptyset$. For every component B_i, we can then define its k-neighbourhood to be the set of all components that have a path of length at most k from B_i in $\mathcal{G}$. We denote this set as $B_i^{(k)}$. Formally, $B_i^{(0)} = \{B_i\}$ and $B_i^{(k)} = B_i \cup \{B_j^{(k-1)} \mid (B_i, B_j) \in E_G\}$ for all $i \geq 1$. A state transition of component B_i potentially changes some state variables of all components in $B_i^{(1)}$, but does not affect any state variable of any other component. Consequently, if C is a clique in the graph $\mathcal{H}$ representing synchronization between components, when computing the image of a set of states under the synchronized transition $\Upsilon_C(\Sigma, \Sigma')$, it is meaningful to update state variables of only components in $B_i^{(1)}$, where $B_i \in C$. This suggests that instead of considering state predicates on the entire set Σ of variables, it would be beneficial to consider state predicates on appropriately chosen subsets of Σ. In other words, reachability analysis using overlapping projections of states makes sense when analysing large networks with local interactions. While the idea of using overlapping projections for approximate reachability was explored in detail by Govindaraju [6], his work considered reachability analysis of large sequential circuits instead of networks of small state transition systems. Consequently, they were unable to exploit locality of interactions to optimize the updation of various projections when a state transition happens. The primary contribution of this paper is to show that locality of interactions can indeed be exploited to significantly optimize updation of projections, enabling the design of BDD-based reachability analysers that scale to much larger networks than those that can be handled by earlier techniques.

Let $\Pi_1, \ldots, \Pi_p$ be subsets of state variables on which we choose to project the states of the overall system. Since we do not wish to ignore any state variable, we require that $\bigcup_{i=1}^{p} \Pi_p = \Sigma$. As in Govindaraju's approach, this gives rise to p projections, say $R_1(\Pi_1), \ldots, R_p(\Pi_p)$, of the set of reachable states. The collection of projections can be viewed as an abstraction of the actual reachable state set. The conjunction $\bigwedge_{i=1}^{p} R_i(\Pi_i)$ is the corresponding concretization that gives the best overapproximation of the reachable set from a given set of projections. However, computing the conjunction is computationally expensive in BDD-based tools since this involves all variables in Σ. Therefore, following Govindaraju's approach, we initialize the projections $R_1, \ldots, R_p$ with projections of the initial set of states on the sets of variables $\Pi_1, \ldots, \Pi_p$, and update these projections each time the system executes a state transition. As discussed earlier, every transition of the network $\mathcal{N} = (\mathcal{B}, \mathcal{H})$ is a state transition of some clique $C \in \mathcal{H}$. Since a transition of C potentially changes all variables in Σ_C, every projection R_i such that $\Pi_i \cap \Sigma_C \neq \emptyset$ must be updated whenever a transition of C is taken. Conversely, all projections R_j such that $\Pi_j \cap \Sigma_C = \emptyset$ need not be updated when C transitions, since there is no component in C whose transitions change the values of variables in Π_j. Thus, by appropriately choosing $\Pi_1, \ldots, \Pi_p$, it is possible to optimize the updation of projections every time a transition corresponding to a clique C is taken.

A transition of a clique C is basically a set of simultaneous transitions of all its components. Since the transition of an individual component B_i depends solely on the values of variables in Σ_i and potentially changes the values of only these variables, a good choice of $\Pi_1, \ldots, \Pi_p$ is one where for each component B_i, there is at least one Π_j such that $\Sigma_i \subseteq \Pi_j$. Intuitively, such a choice would allow us to compute the effect of a transition of component B_i on the projection R_j with a high degree of accuracy. If $R_1(\Pi_1), \ldots, R_p(\Pi_p)$ represent projections of the set of reachable states seen thus far, the image of $R_j(\Pi_j)$ under a synchronous transition of components in C can be computed as $\exists\overline{\Pi_j}\,(N2P(\exists\Sigma\ \Upsilon_C(\Sigma, \Sigma')\wedge \bigwedge_{i=1}^{p} R_i(\Pi_i)))$. Expanding $\Upsilon_C(\Sigma, \Sigma')$, we get

$$\exists\overline{\Pi_j}\left(N2P\left(\exists\Sigma \bigwedge_{B_i\in C} T_i(\Sigma_i, \Sigma_i') \wedge \bigwedge_{s\in\overline{\Sigma_C}} (s \leftrightarrow s') \wedge \bigwedge_{i=1}^{p} R_i(\Pi_i)\right)\right) \quad (3)$$

Coudert and Madre [5] have shown that the basic image computation in the above expression can be also be done using the constrain operator as follows:

$$\exists\overline{\Pi_j}\left(N2P\left(\exists\Sigma\left(\bigwedge_{B_i\in C} T_i(\Sigma_i, \Sigma_i') \wedge \bigwedge_{s\in\overline{\Sigma_C}} (s \leftrightarrow s')\right) \downarrow \bigwedge_{i=1}^{p} R_i(\Pi_i)\right)\right) \quad (4)$$

Since computing $\bigwedge_{i=1}^{p} R_i(\Pi_i)$ potentially involves all variables in Σ and is computationally expensive in BDD-based tools, Govindaraju proposed using a multiple constrain operator. This operator takes a Boolean predicate F and constrains it with a vector of predicates $\langle C_1, \ldots, C_m\rangle$ iteratively, instead of conjoining C_1 through C_m first and then constraining F with the conjunction. Using this operator, the expression for the image can be written as

$$\exists\overline{\Pi_j}\left(N2P\left(\exists\Sigma\left(\bigwedge_{B_i\in C} T_i(\Sigma_i, \Sigma_i') \wedge \bigwedge_{s\in\overline{\Sigma_C}} (s \leftrightarrow s')\right) \downarrow_m \langle R_i(\Pi_i)\rangle\right)\right) \quad (5)$$

In the above expression, $\downarrow_m$ denotes the multiple constrain operator and $\langle R_i(\Pi_i)\rangle$ denotes the vector $\langle R_1(\Pi_1), \ldots, R_p(\Pi_p)\rangle$ of all projections. Govindaraju showed experimentally that for a large class of sequential circuits, this significantly enhanced the efficiency of image computation compared to applying the single constrain operator as in expression (3). In the following discussion, we will refer to the technique of computing the image using the multiple constrain operator as the *"complete multiple constrain"* method.

Although updation using multiple constrain is more efficient that updation using single constrain, the computation of $\bigwedge_{B_i\in C} T_i(\Sigma_i, \Sigma_i') \wedge \bigwedge_{s\in\overline{\Sigma_C}}(s \leftrightarrow s')$ still involves all variables of Σ and can be computationally expensive. One might wonder if we can reduce non-transition variables, as discussed earlier, to simplify $\bigwedge_{B_i\in C} T_i(\Sigma_i, \Sigma_i') \wedge \bigwedge_{s\in\overline{\Sigma_C}}(s \leftrightarrow s')$ to $\bigwedge_{B_i\in C} T_i(\Sigma_i, \Sigma_i')$, and then apply the multiple constrain (or even Coudert and Madre's constrain) operator to the

simplified predicate. Unfortunately, this is not possible in general. Indeed, reducing non-transition variables simplifies expression (3) to

$$\exists \overline{\Pi_j} \left(N2P \left(\exists \Sigma_C \bigwedge_{B_i \in C} T_i(\Sigma_i, \Sigma_i') \wedge \bigwedge_{i=1}^{p} R_i(\Pi_i) \right) \right) \tag{6}$$

While this expression is simpler than expression (3), we cannot directly apply Coudert and Madre's constrain operator and equate $(\exists \Sigma_C \bigwedge_{B_i \in C} T_i(\Sigma_i, \Sigma_i') \wedge \bigwedge_{i=1}^{p} R_i(\Pi_i))$ to $(\exists \Sigma_C (\bigwedge_{B_i \in C} T_i(\Sigma_i, \Sigma_i')) \downarrow \bigwedge_{i=1}^{p} R_i(\Pi_i))$. This is because the quantification in expression (6) is done over a subset of the variables in $\bigwedge_{i=1}^{p} R_i(\Pi_i)$. We will henceforth refer to the method of computing image using expression (6) as the *"reduced conjunction"* method. Note, however, that in expression (6), we must still compute $\bigwedge_{i=1}^{p} R_i(\Pi_i)$, which involves all variables in Σ. Hence the reduced conjunction method can be computationally expensive in BDD-based tools.

Fortunately, the reduced conjunction method can be improved further using locality of interactions. To exploit locality, we consider as many subsets of state variables as there are components in the network. For each component B_i, we choose a subset Π_i that includes state variables of all components in its $k-$ neighbourhood. Recall that when a component transitions, it affects the variables of only those components that are in its $1-$ neighbourhood. Thus $k = 1$ is a good initial choice of neighbourhood for choosing subsets of state variables. As k increases, the subsets increase in size, and so does their overlap. Of course, if one subset is completely contained in another subset (as can happen for large values of k), only the larger subset is retained. As an extreme case, if d is the diameter of the graph $\mathcal{G} = (\mathcal{B}, E_G)$, and if we choose a neighbourhood of d, we obtain a single subset of state variables, $\Pi_1 = \Sigma$.

As the number of variables in each subset increases and as their overlap increases, the accuracy of computing projections of reachable states using expressions (3), (4), (5) or (6) is expected to increase. This is because large subsets with large overlaps can track correlations between state variables better than small subsets with small overlaps. However, having large subsets also entails increased computational effort in manipulating BDDs with large support sets when computing images of projections. By choosing an intermediate value of k, it is possible to ensure that the number of variables in each subset remains small, while there is adequate overlap between the subsets as well. While the best value of k might be domain dependent, in general, our experiments with discrete-timed circuits indicate that using small values like 1 or 2 often strikes a good balance between accuracy of image computations and computational efficiency.

2.2 Exploiting Locality to Optimize Image Computation

Let us now examine if computing the image of projection $R_j(\Pi_j)$ under a synchronous transition of components in a clique C can be simplified using locality of interactions. Expression (6) tells us how to compute the image. To simplify this

expression, we first identify the set of transitioning components in C and the set of projections in $\{R_1, \ldots, R_p\}$ that are intuitively "important" for computing the image of projection R_j. Expression (6) is then simplified by replacing transition relations of all other components and characteristic predicates of all other projections by True. This leads to an over-approximation of the image of projection $R_j(\Pi_j)$ due to a synchronous transition of components in C. We argue below that that this strategy allows us to scale reachability analysis to very large networks with local interactions, while remaining fairly accurate.

When a synchronous transition of components in clique C occurs, only those components that share state variable(s) with Π_j determine the image of projection $R_j(\Pi_j)$. Let the subset of components in C that share state variable(s) with Π_j be called D, and let Σ_D be the union of state variables of all components in D. Transitions of components in $C \setminus D$ neither read nor modify variables of Π_j, and hence do not affect the image of projection $R_j(\Pi_j)$ *directly*. Intuitively, the set of transitioning components D are "important" for determining the image of projection $R_j(\Pi_j)$ under a synchronous transition of components in C. Similarly, when a component $B_i \in D$ transitions, only those projections that share state variable(s) with B_i potentially constrain the transitions of B_i. Let the set of projections that share state variable(s) with at least one $B_i \in D$ be called $P1$, and let Σ_{P1} denote the union of all Π_k's, where projection R_k is in $P1$. Projections not in $P1$ cannot *directly* constrain transitions of B_i, since transitions of B_i are not guarded by conditions on state variables of these projections. Intuitively, projections in $P1$ are "important" for determining the synchronous transitions of components in D. If $|\Sigma_{P1}|$ is large, the set $P1$ can be further pruned by considering only those projections that share state variable(s) with Π_j and also with some $B_i \in D$. Let this set of projections be called $P2$ and let Σ_{P2} denote the corresponding set of state variables. Intuitively, projections in $P2$ not only directly constrain the transitions of components in D, but also allow projection $R_j(\Pi_j)$ to constrain transitions of components in D indirectly. Such an indirect constraining happens when the conjunction of $R_j(\Pi_j)$ and projections in $P2$ constrains transitions of components in D that would not have been constrained by $R_j(\Pi_j)$ alone. Hence, projections in $P2$ are very "important" for computing the image of $R_j(\Pi_j)$ under a synchronous transition of components in D. In the following discussion, we will use P to denote the set of "important" projections chosen for simplifying expression (6). While either $P1$ or $P2$ could be chosen for P, we have chosen $P2$ for our experiments since $|\Sigma_{P2}| \leq |\Sigma_{P1}|$.

We can now simplify expression (6) by over-approximating the transition relations of all components in $C \setminus D$ by True, and by over-approximating the conjunction of projections not in P by True. The simplified expression for the image of projection $R_j(\Pi_j)$ under a synchronous transition of components in C is then given by

$$\exists \overline{\Pi_j} \left(N2P \left(\exists \Sigma_D \bigwedge_{B_i \in D} T_i(\Sigma_i, \Sigma_i') \wedge \bigwedge_{\Pi_i \in P} R_i(\Pi_i) \right) \right) \tag{7}$$

Due to locality of interactions, the set P is much smaller than the entire set of projections, and similarly, D is a small subset of C, in general. This leads to significant gains in efficiency compared to computing the image of projection $R_j(\Pi_j)$ according to expression (6). One might suspect that this gain in efficiency is achieved at the cost of a significant loss of accuracy. However, as demonstrated by our experiments, the loss in accuracy due to these simplifications is not large, and the accuracy-efficiency trade-off is on the favourable side. We will call this technique that exploits locality of interactions and computes the image of projection $R_j(\Pi_j)$ according to expression (7) as *"partial reduced conjunction"*.

It is important to note that the proposed simplifications lead to over-approximations in the image of projection $R_j(\Pi_j)$ in the worst case. This happens when the complete conjunction $\bigwedge_{B_i \in C} T_i(\Sigma_i, \Sigma_i') \wedge \bigwedge_{i=1}^{p} R_i(\Pi_i)$ forbids a state transition by evaluating to False for a specific pair of present and next states, whereas the partial conjunction $\bigwedge_{B_i \in D} T_i(\Sigma_i, \Sigma_i') \wedge \bigwedge_{\Pi_i \in P} R_i(\Pi_i)$ evaluates to True for the same choice of present and next states.

Similar to the approach of Govindaraju, one might also consider optimizing the evaluation of expression (7) by using the multiple constrain operator. Let Σ_{PD} denote $\Sigma_P \cup \Sigma_D$ and $\Sigma_{P \setminus D}$ denote $\Sigma_P \setminus \Sigma_D$. The expression for the image of $R_j(\Pi_j)$ using the multiple constrain operator is then given by

$$\exists \overline{\Pi_j} \left(N2P \left(\exists \Sigma_{PD} \left(\bigwedge_{B_i \in D} T_i(\Sigma_i, \Sigma_i') \wedge \bigwedge_{s \in \Sigma_{P \setminus D}} (s \leftrightarrow s') \right) \downarrow_m \langle R_i(\Pi_i) \rangle_{\boldsymbol{P}} \right) \right) \tag{8}$$

In the above expression, $\downarrow_m$ denotes the multiple constrain operator and $\langle R_i(\Pi_i) \rangle_{\boldsymbol{P}}$ is a vector of projections belonging to the set P. Note that in order to apply the multiple constrain operator, we had to introduce the subexpression $\bigwedge_{s \in \Sigma_P \setminus \Sigma_D} (s \leftrightarrow s')$, which translates to increased computational cost in evaluating expression (8). As we will see in Section 3, experiments on a set of benchmarks indicate that this technique performs marginally worse than the partial reduced conjunction method. This shows that the benefits of the multiple constrain operator are more than offset by the additional computational cost of evaluating the subexpression $\bigwedge_{s \in \Sigma_P \setminus \Sigma_D} (s \leftrightarrow s')$. This technique of computing the image of projection $R_j(\Pi_j)$ using expression (8) will be called *"partial multiple constrain"* in our future discussion.

The set $\overline{\Pi_j}$ of variables that is finally quantified (corresponding to the leftmost existential quantifier) in expression (7) can be written as the union of $\overline{\Pi_j} \cap \overline{\Sigma_D}$ and $\overline{\Pi_j} \cap \Sigma_D$. Since the quantifier inside the $N2P$ operator eliminates only variables in Σ_D, all free variables of $\bigwedge_{B_i \in D} T_i(\Sigma_i, \Sigma_i') \wedge \bigwedge_{\Pi_i \in P} R_i(\Pi_i)$ that are in $\overline{\Pi_j} \cap \overline{\Sigma_D}$ are unaffected by this quantification. These variables, being present state variables, are not renamed by $N2P$ as well. Therefore, it is possible to push $\exists \overline{\Pi_j} \cap \overline{\Sigma_D}$ inside the $N2P$ and $\exists \Sigma_D$ operators in expression (7) to obtain the semantically equivalent expression

$$\exists(\overline{\Pi_j} \cap \Sigma_D)\left(N2P\left(\exists\Sigma_D\left(\bigwedge_{B_i\in D} T_i(\Sigma_i, \Sigma_i') \wedge \left(\exists(\overline{\Pi_j}\cap\overline{\Sigma_D})\bigwedge_{\Pi_i\in P} R_i(\Pi_i)\right)\right)\right)\right) \tag{9}$$

In the above expression, the argument of $N2P$ is an expression whose free variables can be partitioned into the sets $(\Pi_j \cap \Sigma_D)'$, $(\overline{\Pi_j} \cap \Sigma_D)'$ and $(\Pi_j \cap \overline{\Sigma_D})$. Let these three mutually disjoint sets of variables be called Γ_1', Γ_2' and Γ_3, respectively, and let $\xi(\Gamma_1', \Gamma_2', \Gamma_3)$ denote the argument of $N2P$ in expression (9). Using the definition of $N2P$ from equation (1) and the notation introduced above, expression (9) can now be written as $\exists\Gamma_2\,\xi(\Gamma_1, \Gamma_2, \Gamma_3)$. Unfortunately, it is not straightforward to obtain $\xi(\Gamma_1, \Gamma_2, \Gamma_3)$. Specifically, we note from expression (9) that $\xi(\Gamma_1', \Gamma_2', \Gamma_3)$ is obtained by existentially quantifying variables in $\Gamma_1 \cup \Gamma_2$ from an expression, say ζ, with free variables $\Gamma_1 \cup \Gamma_2 \cup \Gamma_3 \cup \Gamma_1' \cup \Gamma_2'$. Therefore, simply substituting Γ_1 for Γ_1' and Γ_2 for Γ_2' in ζ and quantifying out $\Gamma_1 \cup \Gamma_2$ will not give $\xi(\Gamma_1, \Gamma_2, \Gamma_3)$. To overcome this problem, we use the fact that renaming bound variables does not change a quantified expression. Hence, the image expression $\exists\Gamma_2\,\xi(\Gamma_1, \Gamma_2, \Gamma_3)$ is equivalent to $\exists\Gamma_2'\,\xi(\Gamma_1, \Gamma_2', \Gamma_3)$, which in turn, is equivalent to $N2P(\exists\Gamma_2'\,\xi(\Gamma_1', \Gamma_2', \Gamma_3))$. Recalling that $\xi(\Gamma_1', \Gamma_2', \Gamma_3)$ is the argument of $N2P$ in expression (9), the following image expression, semantically equivalent to expression (9) but with different quantifier ordering, is obtained:

$$N2P\left(\exists\Sigma_D\left(\left(\exists(\overline{\Pi_j} \cap \Sigma_D)' \bigwedge_{B_i\in D} T_i(\Sigma_i, \Sigma_i')\right) \wedge \left(\exists(\overline{\Pi_j} \cap \overline{\Sigma_D})\bigwedge_{\Pi_i\in P} R_i(\Pi_i)\right)\right)\right) \tag{10}$$

The evaluation of expression (10) can be further simplified, albeit with further loss of accuracy, by pushing the quantification of variables in $\overline{\Pi_j}\cap\overline{\Sigma_D}$ inside the conjunction $\bigwedge_{\Pi_i\in P} R_i(\Pi_i)$. Effectively, this amounts to over-approximating the existential projection of a conjunction by the conjunction of existential projections. As a first approximation, we observe that expression (10) computes the (approximate) image of projection $R_j(\Pi_j)$ due to a transition of components in clique C. Consequently, the most important variables in this expression are those in $\Sigma_D \cup \Pi_j$. Hence, the quantification of all other variables, i.e. variables in $\overline{\Pi_j} \cap \overline{\Sigma_D}$, may be pushed inside the conjunction to optimize the computation of expression (10). This gives rise to the optimized image expression

$$N2P\left(\exists\Sigma_D\left(\left(\exists(\overline{\Pi_j} \cap \Sigma_D)' \bigwedge_{B_i\in D} T_i(\Sigma_i, \Sigma_i')\right) \wedge \bigwedge_{\Pi_i\in P} \exists(\overline{\Pi_j} \cap \overline{\Sigma_D}) R_i(\Pi_i)\right)\right) \tag{11}$$

We will call image computation using expression (11) as *"partial approximate quantification"*. Note that in computing the image according to expression (11), the maximum number of variables involved in a BDD operation is $\max_{\Pi_i\in P}(|\Pi_i|) + 2.|\Sigma_D|$.

This is far smaller than the number of variables involved in operations required for computing images by the earlier expressions. This contributes to the efficiency of using expression (11), which is also corroborated by our experiments. The loss of accuracy when computing the image according to expression (11) vis-à-vis when computing using expressions (4) or (5) or (6) stems from two sources: (i) approximating conjunctions by partial conjunctions, as in expression (7), and (ii) pushing quantifications inside conjunctions, as in expression (11). However, in both cases, the approximation is done with projections, components or variables that are intuitively less "important" in determining the image of projection $R_j(\Pi_j)$ under a transition of clique C. The variables, projections and transitions that are more "important" are not approximated as far as possible. Locality of interactions allows us to restrict this set of "important" variables, projections and transitions to a small set even in very large networks. This explains why our experiments indicate a high degree of efficiency, and a fair degree of accuracy when using expression (11).

It is easy to see from the nature of our approximations that in terms of accuracy of results, the ordering of the different methods is "complete multiple constrain" $\geq$ "partial reduced conjunction" $=$ "partial multiple constrain" $\geq$ "partial approximate quantification". The degradation in accuracy from "complete multiple constrain" to "partial reduced conjunction" is due to the over-approximation of several "unimportant" transition relations and projections. The further degradation in accuracy in "partial approximate quantification" is attributable to the over-approximation of a projection of conjunctions by the conjunction of projections. Experimental results however show that the degradation in accuracy is not large for a range of timed circuit benchmarks. In terms of the number of variables involved in BDD operations when computing the images of a projection under a synchronized transition of components, it is clear from expressions (5), (8), (7) and (11) that the ordering of the different methods is "complete multiple constrain" $\geq$ "partial multiple constrain" $\geq$ "partial reduced conjunction" $\geq$ "partial approximate quantification". The variable count involved in BDD operations gives a rough indication of the relative computational time and memory requirements of BDD-based tools implementing these techniques. Our experimental results corroborate that this order is by and large correct.

2.3 Scalability Issues

The technique of "partial approximate quantification" described above offers unique advantages in scaling our BDD-based approach to very large networks. We argue below that the maximum number of variables involved in any BDD operation during image computation using this technique is independent of the size of the transition system. Instead, it depends only on (i) the number of variables in each component, (ii) the degree of the graph $\mathcal{G} = (\mathcal{B}, E_G)$ that represents sharing of variables between components of the network, and (iii) the neighbourhood k used to determine projections. Thus, by carefully dumping BDDs to and from disk, it is never necessary to

represent or manipulate BDDs with very large support sets in main memory. This can enable our BDD-based reachability analysis to scale up to very large networks with local interactions.

As has been described in Section 2.2, the number of variables involved in any image computation step of "partial approximate quantification" is bounded above by $\max_{\Pi_i \in P}(|\Pi_i|) + 2.|\Sigma_D|$. Here, $\max_{\Pi_i \in P}(|\Pi_i|)$ denotes the maximum number of variables in any projection, and depends on (i) the maximum degree of a node in the graph $\mathcal{G} = (\mathcal{B}, E_G)$, (ii) the neighbourhood k used to compute the projections, and (iii) the number of state variables in each component. The quantity $|\Sigma_D|$ depends on (i) the maximum number of components in a clique C that share variable(s) with the projection R_j being updated, and (ii) the number of state variables in a component. Given a network of transition systems, the maximum degree of a node in the graph $\mathcal{G}$ and the neighbourhood k used to compute projections uniquely define an upper bound on the maximum number of components in a clique C that potentially share variables with R_j. Interestingly, neither the degree of $\mathcal{G}$, nor the neighbourhood k, nor the maximum number of variables in a component depend on the total number of components in the network. Thus, if the maximum degree of $\mathcal{G}$ and the number of variables in each component is bounded by a small number, and if we choose a neighbourhood k for defining projections such that $\max_{\Pi_i \in P}(|\Pi_i|) + 2.|\Sigma_D|$ is within the maximum variable count that a BDD-package can efficiently handle, it is possible to scale the analysis to very large networks. Of course, the number of projections and transition relations of components will increase with the size of the network. Hence the total number of BDDs that need to be stored will be large for large networks. However, as argued above, only a few of these, with a bounded number of variables in their support set, are required for image computation at any point of time. Hence by effectively dumping BDDs not currently needed to the disk and by re-loading them when needed, we envisage the possibility of BDD-based reachability analysis tools for arbitrarily large networks with local interactions. Such tools may require significant computational time for exploring the reachable state space of large networks, but will never run out of main memory due to BDD size explosion.

3 Experimental Results

In order to evaluate the effectiveness of our approach, we have implemented the strategies described in the previous section in the public domain reachability analysis engine `NuSMV` [4]. We have modified the reachability routine of `NuSMV` to perform reachability analysis on overlapping projections using asynchronous and synchronous transitions. We have used the resulting tool to explore the entire reachable space of a set of benchmarks, and report the reachable projections using our approach as well as using the multiple constrain approach of Govindaraju [6]. Our experimental results support our hypothesis that reachability analysis of large networks can be significantly optimized by exploiting locality of interactions.

Our benchmark suite consists of gate-level circuits with discrete-time delays. The motivation behind choosing these examples comes from their popularity in the domain of timed systems analysis and also their ease of scalability. Some of our examples consist of small combinational circuits, consisting of tens of gates, used in [9]. We also perform experiments on standard benchmark circuits from the `ISCAS-85` suite, which consists of larger combinational circuits.

3.1 Modeling of Circuits

Every gate used in our experiments implements a combinational logic function and has an *inertial delay* and *bi-bounded pure delay*. The behaviour of a gate is modeled by having the following three parts: (i) a *Boolean logic block* that sets the Boolean value of the output to a function of the Boolean values of the inputs, (ii) the output of the logic block is fed to an *inertial delay element*, and (iii) the output of inertial delay element is fed to a *bi-bounded pure delay element*. We model an inertial delay element having delay D by a timed automaton as shown in Figure 1(a). Similarly, a bi-bounded pure delay element with lower and upper bounds l and u is modeled by a timed automaton as shown in Figure 1(b). In these figures, `Zero` and `One` refer to stable states, where the Boolean values of *in* and *out* are the same.

Given the interconnection of gates representing a circuit, we compose the state transition behaviour of the logic block, inertial delay element and bi-bounded delay element of each gate to form a network of timed automata. To simplify the model, we assume that D, l and u are identical for all gates. To ensure that every pure delay element causes its output to change once between two consecutive changes at its input, we also assume that $u < D$. In fact, for all our experiments, the inertial delay (D) is set to 3, and the lower (l) and upper (u) bounds of bi-bounded pure delay elements are set to 1 and 2, respectively. When the output of a gate feeds the input

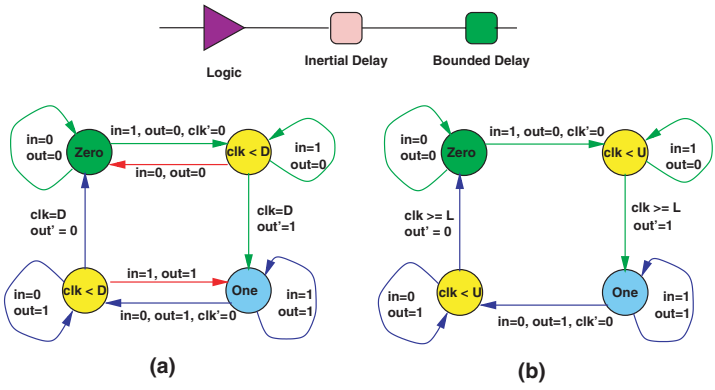


Fig. 1 (a) Inertial delay model; (b) bi-bounded pure delay model

of another gate, we ensure during composition that the corresponding output and input transitions occur simultaneously. Time is assumed to flow synchronously for all gates. For our experiments, the circuit inputs are modeled as signals that non-deterministically change their Boolean values after a predefined delay $\Delta_{in} = 4$. We assume that time is discrete for all our experiments.

For an n-gate circuit modeled as above, the network of timed automata has two types of transitions: discrete (non-time-elapse) transitions for each logic function, inertial and bi-bounded delay element which execute asynchronously, and a global transition for synchronous advancement of time of all clocks. The global timed transition is modeled as the synchronous transition of a group of transition systems, where each transition system models advancement of time for one gate. It is easy to see that both the discrete and timed transitions corresponding to each gate affect the state variables of only those gates that are in its immediate fanout or fanin. By restricting the fanin and fanout of each gate in the circuit to a small number, we can therefore ensure that the locality of interactions is small and independent of the circuit size.

3.2 Comparing Different Techniques

We now present and compare experimental results obtained by application of the techniques referred to as "partial reduced conjunction", "partial approximate quantification", "partial multiple constrain" and "complete multiple constrain" in Section 2.1. We will call these techniques S_0, S_1, S_2 and S_3, respectively in the subsequent discussion. We compare the relative performance of these techniques in terms of BDD sizes, time taken and accuracy. All our experiments were performed on a 3 GHz Intel Pentium 686 processor with 1 GB of main memory, and running Fedora Core Linux 3.4.3-6.fc3.

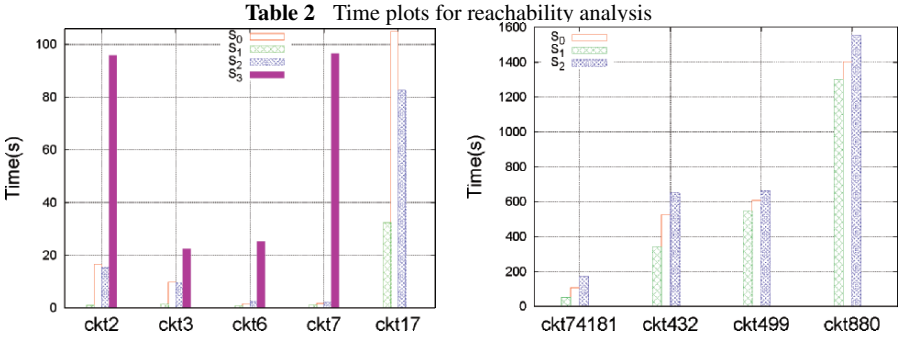
In order to see how the various techniques scale up, we converted all large circuits ($>$70 gates) to functionally equivalent circuits in which the fanout and fanin of each gate is bounded above by 3. As discussed in Section 2.3, this bounds the maximum degree of any node in the graph $\mathcal{G}$ and allows the "partial approximate quantification" technique to scale to large circuits. To reduce the fanout of a gate to 3, we used a linear chain of buffers at the output of the gate. To reduce the fanin of a gate, we used a tree of 2 input gates which combine to give the same Boolean function.

The specification for each circuit used in our experiments is given in Table 1. In this table, column *Ckt* gives the name of the circuit, and column *Gates* gives the number of gates, which is also the total number of discrete transitions for the circuit. Column *Additional gates* gives the number of additional gates that were added to obtain a functionally equivalent circuit with all gates having bounded fanin and bounded fanout, as discussed above. Columns *0-nbd, 1-nbd, 2-nbd* give the statistics for projections using neighbourhoods of 0, 1, and 2, respectively. Sub-column *Proj* gives the number of distinct projections obtained for each value of neighbourhood. A projection whose support set is a subset of the support set of another projection is discarded.

Table 1 Circuit characteristics

Ckt	Gates	Additional Gates	0-nbd		1-nbd		2-nbd		Variables
			Proj	Max	Proj	Max	Proj	Max	
2	10	0	17	38	13	78	9	123	274
3	11	0	19	59	17	115	13	143	306
6	21	0	29	59	23	115	18	157	626
7	31	0	39	59	33	115	29	151	946
17	11	0	17	54	11	131	8	138	274
74181	202	130	390	109	378	813	363	1782	6242
432	564	368	1092	108	1055	748	1013	1960	17574
499	724	481	1407	108	1345	711	1284	1864	22514
880	1182	739	2304	109	2191	811	2045	1883	36866

Table 2 Time plots for reachability analysis



Sub-column *Max* gives the maximum number of variables involved during any image computation. Finally, column *Variables* gives the total number of state variables of the network of transition systems. Note that this number is much larger than the total number of gates in the circuit, since encoding the behaviour of each gate requires several state variables. The plots in Table 2 show the time taken for searching the discrete-timed reachable state space of each circuit using the techniques S_0, S_1, S_2 and S_3 referred to above. The results for small circuits and large circuits are given in two separate plots. The results presented in Table 2 were obtained with projections computed using a neighbourhood of 1. The maximum number of state variables of an individual component (discrete or timed transition system of a gate) is 28 in our experiments. In the plots of Table 2, the absence of a bar for a particular technique and circuit implies that for that circuit, reachability analysis using the specific technique did not terminate within 1 hour. We note that bars corresponding to "complete multiple constrain" are completely absent in the plots for all large circuits.

The plots in Table 3 show on a $\log_2$ scale the peak live BDD node counts attained during reachability analysis using various techniques on the same circuits and using the same projections as in Table 2.

Table 3 Peak live BDD nodes for reachability analysis

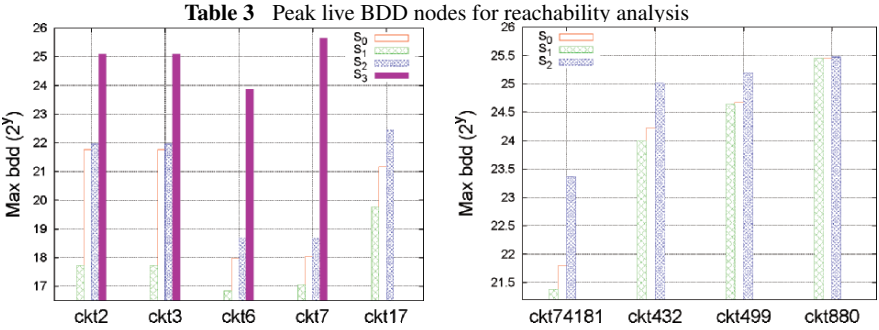


Table 4 Accuracy comparison for reachability analysis

Ckt	S_1/S_0			S_2/S_0			S_3/S_0		
	Avg	Max	Min	Avg	Max	Min	Avg	Max	Min
ckt2	1	1	1	1	1	1	0.940254	1	0.72333
ckt3	1.00273	1.04646	1	1	1	1	0.721266	1	0.265875
ckt6	1	1	1	1	1	1	0.956729	1	0.834483
ckt7	1	1	1	1	1	1	0.969841	1	0.834483
ckt74181	1	1	1	1	1	1			
ckt432	1.00072	1.4	1	1	1	1			
ckt499	1	1	1	1	1	1			
ckt880	1	1	1	1	1	1			

In order to compare the accuracy of the different techniques, we consider each circuit, and compute the ratio of the sizes of the projections of the reachable states using the different techniques. Ideally, we would have liked to compare the total count of reachable states for each circuit using various techniques. However, this requires conjoining the reachable state predicates for all the projections, and then counting the number of satisfying assignments for the resulting conjunction. Unfortunately, for large circuits, computing the conjunction of all projections involves constructing a BDD with almost all state variables of the entire network in its support set. This leads to a BDD size blowup and prevents us from computing the overall reachable state set. For each circuit, we therefore compute the ratios of sizes of projections of reachable sets using various techniques, and report the minimum, maximum and average values of these ratios considered over all projections. These ratios are summarized in Table 4. As discussed in Section 2.1, S_3 corresponding to "complete multiple constrain" gives the smallest sets of states for the projections, while "partial multiple constrain" (S_2) and "partial approximate quantification" (S_1) are comparable in accuracy to "partial reduced conjunction" (S_0). For the larger circuits, the "complete multiple constrain" technique does not terminate in 1 hour and hence the corresponding accuracy figures are not available. Note that the worst-case accuracy of technique S_0 (relative to technique S_3) is on ckt3, where it computes nearly 4 times

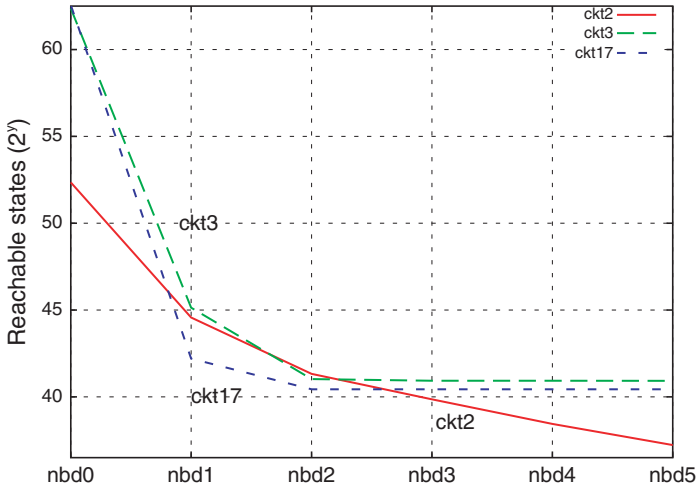


Fig. 2 Variation in over-approximation of reachable states with neighbourhood

the reachable states of a projection compared to what is computed by technique S_0. However, on an average, S_3 computes projections of reachable state sets that are 1.4 times larger than the projections computed by technique S_0 on ckt3.

In order to study the effect of increasing neighbourhoods when determining projections on the accuracy of reachability analysis, we carried out a set of experiments with small circuits in which we increased the neighbourhood from 0 to 5. The experiments were performed only on small circuits, since as explained above, it is difficult to conjoin the BDDs of all projections to obtain the complete set of reachable states for large circuits. Figure 2 shows a plot of the number of states in the over-approximation of the reachable set against the neighbourhood. The over-approximated reachable states in these plots were obtained using the technique S_0. The name of the circuit corresponding to each curve is written on top of the curve.

Increasing the neighbourhood when choosing projections results in larger projections with larger overlaps between them. The increased overlaps better facilitate tracking correlations between state variables, and leads to increased accuracy. This manifests itself as a monotone decrease in the size of the over-approximated reachable state set with increasing neighbourhood, as seen in Figure 2.

4 Discussion and Conclusion

From the bar charts in the previous section, we find that for small circuits, technique S_3 takes much more CPU time as compared to other techniques, and for larger circuits it does not terminate within 1 hour. This behaviour is expected as technique

S_3 requires operating with BDDs that have the complete set of state variables in the support set for every image computation step. Indeed, the total count of such variables (ranging from 6000 to 30000 for our experimental circuits) is far beyond the reach of standard BDD packages like CUDD. In contrast, the other techniques require operating with BDDs that have much smaller support sets. This translates to reduced computation times and lower peak BDD sizes, when using these techniques. In addition, the accuracy of S_0 and S_1 do not suffer much compared to S_3, as measured in those cases where S_3 terminated. This gives concrete evidence in support of our claim that locality of interactions can be exploited to build efficient and scalable reachability analysers, without compromising much on accuracy.

We also observe that for all circuits, technique S_1 gives the best results in terms of CPU time and peak BDD sizes. This can be qualitatively explained by the fact that the image computation step with this technique involves conjunction of BDDs with small support sets (10's of variables), followed by quantification of small sets of variables. In addition, technique S_1 retains the projections and transitions that are most "important" for computing the image of a projection under a transition, while ignoring the effects of other projections and transitions that do not directly affect the image of the projection under consideration. Since the accuracy obtained with this technique is fairly good, we conclude that exploiting locality of interactions in networks of transition systems has the potential to buy us a lot of efficiency without compromising much on accuracy.

Although in Section 2.1 we theoretically argued about the scalability of technique S_1 ("partial approximate quantification"), we were unable to analyse circuits with more than 1200 gates in the current set of experiments. With a fanin and fanout of 2 for each gate, even such large circuits are amenable to analysis using technique S_1, if BDDs can be dynamically swapped to and from disk during reachability analysis. The current implementation of our reachability analyser does not implement such swapping of BDDs to and from disk, and consequently stores all BDDs in main memory. For large circuits, the total number of projections (and hence BDDs) becomes very large. While only a few of these are needed at any time for image computation, our current implementation suffers from memory bottlenecks since it stores all BDDs in main memory. We intend to implement swapping of BDDs to and from disk to allow scaling of our analysis to even larger circuits in the future.

The techniques presented in this paper effectively exploit locality of interactions in large networks of transition systems to scale reachability analysis to networks that are at least an order of magnitude larger than those amenable to existing tools. While other optimization techniques are being actively investigated for scaling reachability analysis, we believe locality of interactions is an important factor that, if properly exploited, can allow us to analyse very large networks of transition systems efficiently and fairly accurately. Our current work is a first step in this direction. We are also investigating ways to encode different search strategies in a meta-programming framework for reachability analysis that would allow one to mix and match different techniques for achieving a good balance of accuracy and efficiency.

References

1. R.E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.
2. G. Cabodi, P. Camurati, and S. Quer. Improving symbolic reachability analysis by means of activity profiles. *IEEE Transactions on Computers*, 19(9):1065–1075, 2000.
3. H. Cho, G.D. Hachtel, E. Macii, B. Plessier, and F. Somenzi. Algorithms for approximate FSM traversal based on state space decomposition. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 15(12):1465–1478, 1996.
4. A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV version 2: An opensource tool for symbolic model checking. In *Proceedings of CAV*, LNCS 2404, pages 359–364, 2002.
5. O. Coudert and J.C. Madre. A unified framework for the formal verification of sequential circuits. In *Proceedings of ICCAD*, pages 126–129, 1990.
6. Gaurishankar Govindaraju. *Approximate Symbolic Model Checking Using Overlapping Projections*. PhD thesis, Stanford University, August 2000.
7. I.-H. Moon, J.H. Kukula, K. Ravi, and F. Somenzi. To split or to conjoin: The question in image computation. In *Proceedings of DAC*, pages 23–28, 2000.
8. F. Somenzi. CUDD: Colorado University Decision Diagram Package Release 2.3.0., University of Colorado at Boulder, 1998.
9. D. Thomas, S. Chakraborty, and P.K. Pandya. Efficient guided symbolic reachability using reachability expressions. In *Proceedings of TACAS*, pages 120–134, 2006.

Schedule Verification and Synthesis for Embedded Real-Time Components*

Purandar Bhaduri

Abstract In this paper we address the problems of schedule synthesis and timing verification for component-based architectures in embedded systems. We consider a component to be a set of tasks with response times that lie within specified intervals. When a set of components is deployed to implement a desired functionality, we want to guarantee that the components can achieve the timing constraints of the application. We solve the associated synthesis and verification problems using the framework of timed interface automata and timed games.

Keywords: Component-based embedded real-time systems, real-time scheduling, timed interfaces, timed games, schedule synthesis.

1 Introduction

Component-based development has been proposed as a framework for dealing with the complexity of embedded control systems. It is based on the premise that generic components can be developed so as to be *reused* in different contexts. While the encapsulation of behaviour in component interfaces does lead to modularity and enhanced reuse, the verification of non-functional aspects (such as timing and resource constraints) of an assembly of components remains a major challenge.

In this paper we analyse whether a given set of components satisfies the timing constraints of an embedded control application. We consider a component to be a collection of *tasks*, which are *functionally* and *logically* related. In turn, each task has a response time (i.e., the time between task release and completion) that is guaranteed to lie within a specified interval by the component implementation. When a set of components is deployed to implement a desired functionality, we want to guarantee that the components can achieve the timing constraints of the application. The application-level timing properties we consider here are the *end-to-end* timing constraints of *transactions*. Each transaction is typically a loop consisting of reading sensors, computing control inputs and writing to actuators. The constituent tasks of a transaction may be part of different components. The specific problems we are

* This work was supported by a grant from GM R&D.

S. Ramesh and P. Sampath (eds.), *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*, 137–153.

interested in are (a) *timing verification*: to ascertain whether the given components can satisfy the end-to-end timing constraints of the application, and (b) *schedule synthesis*: if the answer to (a) is yes, to determine a sequence of task release actions that will lead to satisfaction of the constraints.

We refer to the above problem as *component scheduling*, to distinguish it from *task scheduling*, the staple of real-time scheduling theory. In task scheduling, we already know the deadlines, periods and execution times of tasks, and want to know whether the tasks can be scheduled to meet their deadlines. In component scheduling we know that the tasks can be scheduled to meet certain deadlines (which may *not* be related or derived from the application at hand), but want to know whether these tasks can be released in such a way that the end-to-end constraint of a transaction can be met. Task scheduling is a top-down analysis – from the real-time requirements we identify tasks and their characteristics, identify the platform and check whether the tasks can be scheduled. Component scheduling is bottom-up – given the components and the constituent tasks, along with their pattern of release and completion times, we want to verify whether they can satisfy the end-to-end constraints. The component scheduling problem becomes relevant when the tasks are not identified based on the real-time requirements of the particular application, but the application itself is built by composing pre-existing components.

Our approach to solving the timing verification and schedule synthesis problems for components is based on the formalism of timed interface automata (TIA) (de Alfaro et al., 2002). We view the problems as a *timed game* between two players – one representing the environment (the scheduler or *Input*) and an adversary representing the system (the component or *Output*). The environment can decide on when to release tasks for execution, but not their completion times, which can be decided only by the component. Both players make certain assumptions about the other player, and deliver certain guarantees. The overall goal is to check that there is a sequence of allowed moves by the environment (release of tasks) which leads to satisfaction of the high-level timing requirements; in other words, there is a winning strategy for the environment in the corresponding timed game. The existence of such a winning strategy guarantees that the components can be used together to satisfy the end-to-end timing constraints.

Timed games have been used to solve several scheduling problems (see Altisen et al., 1999, 2002 for example). Unlike these works, we solve a new scheduling problem that is unrelated to traditional task or job-shop scheduling. A key feature of our work is that all the timing requirements (both task characteristics and external timing constraints) are captured using the TIA formalism, a formalism for compositional reasoning about timed systems. Our techniques are therefore *modular*, and can be applied in a *compositional* and *incremental* manner.

The main novelty of this work is that we define a notion of component scheduling and propose methods for solving the associated verification and synthesis problems. Contrary to the classical notion of task scheduling, component scheduling deals with transactions involving a set of tasks rather than separate task instances. In our setting, checking for deadline violation corresponds to checking that the end-to-end

constraints of a transaction are satisfied. Component scheduling is motivated by the fact that modern embedded control systems are typically built out of existing components. Components consist of tasks representing component services; transactions are application specific jobs that span across a set of components. The main technical contribution of this work is twofold: *encoding* the specification of component scheduling problem as timed interface automata and *reduction* of the verification and synthesis problem for component scheduling to finding a winning strategy in the game structure for the associated timed interface automata. Our use of TIA for modelling both tasks and transactions is novel. So is our use of the formalism for solving scheduling problems, since we go beyond checking compatibility of timed components.

As an application, we apply our component scheduling framework to the problem of deriving a static time-triggered schedule for a set of periodic tasks. We are given a set of processors and a number of tasks with known frequencies, and execution times lying in fixed intervals. Each task is statically allocated to a processor, called a *TTA node*, and must communicate with other tasks through a shared bus. The problem is to find a static schedule on each processor along with a bus schedule, such that all task and communication deadlines are met without any task being preempted when executing. The solution using our approach is worked out on an automotive Adaptive Cruise Control (ACC) application.

2 The Component Scheduling Problem

The typical design flow in component-based development of embedded systems is as follows. To implement a given *feature* of the system to be built, such as the adaptive cruise control feature in an automobile, a number of transactions, each consisting of a related set of tasks, is identified. A transaction is actually a partial order on the tasks reflecting their interdependence. The tasks comprising a transaction usually span multiple components. The end-to-end timing constraints for each transaction are derived from the feature requirements, and must be met by the tasks from different components that constitute the transaction. This is the essence of the component scheduling problem.

2.1 Tasks and Task Graphs

According to our view, a component is a set of tasks, with each task satisfying certain timing constraints. A component is a black-box which hides the internal details of how tasks are actually scheduled. The interface only exposes the timing constraints in the form of *assumptions* about task release times and *guarantees* about task completion times. In our setting the release and execution times of a task may not be strictly periodic, but can lie within a specified interval. This facilitates modelling of

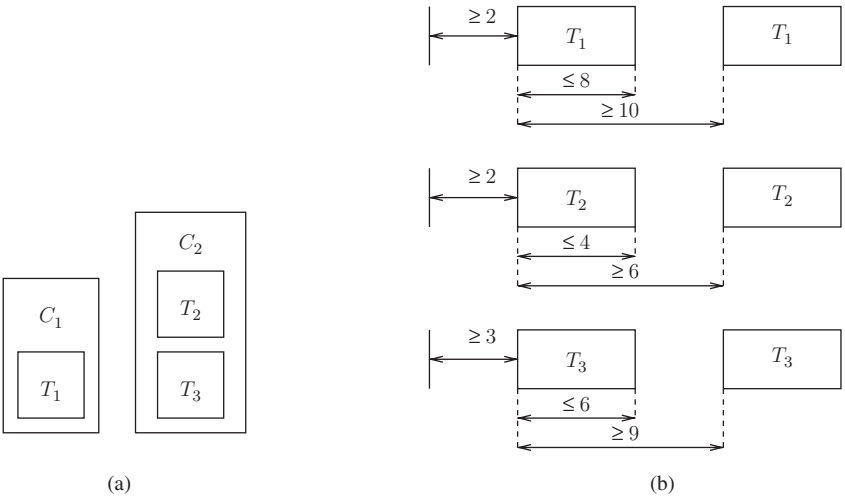


Fig. 1 Components, tasks and timing constraints

jitter and communication delays and leads to more flexibility in scheduling, as tasks with fixed periods are too simplistic and lead to pessimistic analysis.

Example 1 (Components and Tasks) *Figure 1(a) shows two components C_1 and C_2. Tasks T_1 belongs to component C_1, while tasks T_2 and T_3 belong to C_2.*

Figure 1(b) shows various timing constraints for the tasks T_1, T_2 and T_3. For instance, task T_1 cannot be released within the first 2 *time units, which is an assumption on the environment; we call such a constraint an* offset constraint*. Once the task T_1 is released, it must complete within 8 time units, a guarantee provided by the component; we call such a constraint an* execution time constraint*. Further, the delay between two successive task-releases has to be at least 10 time units, again an assumption on the environment; we call such a constraint a* period constraint*.*

Task Graphs We model transactions as *task graphs*, i.e., partial orders (or DAG's) on tasks. The partial ordering reflects the data dependencies between tasks in a particular transaction: an edge from task T_i to T_j indicates that task T_i must complete before task T_j begins. We associate an end-to-end deadline with each transaction, as well as constraints on inter-task separation to guarantee freshness of data. The constraints on deadline and the inter-task separation are collectively referred to as *end-to-end constraints*. Note that a transaction represented by a task graph is periodic, the period being determined by the sampling frequency of the associated control loop. We assume that the period of the transaction is given by the end-to-end deadline of the task graph.

Example 2 (Task Graph) *Figure 2 shows a task graph for a transaction involving components C_1 and C_2 in Example 1. It says task T_2 must be released after tasks T_1*

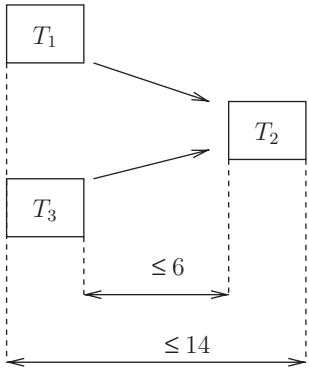


Fig. 2 Task graph and transaction constraints

and T_3 have completed. The transaction has an end-to-end deadline of 14 *time units, and a constraint that says T_2 must be released within* 6 *time units from the completion of T_3 (to ensure* freshness *of data, for instance).*

Another constraint that is implicit in the task-graph is that the transaction it represents is required to execute an infinite number of time, a liveness *constraint.*

Definition 3 *A* task graph with end-to-end constraints *is a triple $G = (\mathcal{T}, <_{\mathcal{T}}, d)$ where $\mathcal{T} = \{T_1, \ldots, T_m\}$ is a set of m tasks, $<_{\mathcal{T}}$ is a strict partial order (i.e., an irreflexive transitive relation) on $\mathcal{T}$ and d is a set of constraints of the form $d(\mathcal{T}) \leq C$ or $d(a, b) \leq C$ for $a, b \in \bigcup_{1 \leq i \leq m} \{r_i, c_i\}$, where C is an integer. The constraint $d(\mathcal{T}) \leq C$ represents an end-to-end deadline of C time units for the task graph, while the constraint $d(a, b) \leq C$ represents a maximum separation of C time units between the two actions a and b, which are either the release r_i of a task T_i or the completion c_j of a task T_j. We denote by $\Pi(T)$ the set of immediate predecessors of task T in the partial order $(\mathcal{T}, <_{\mathcal{T}})$.*

2.2 The Problem

The scheduling problem we are trying to solve is: given a set of tasks with timing constraints on their release and completion, and a task graph with end-to-end constraints, to find a schedule, i.e., a timed sequence of release actions (which may depend on the timed sequence of preceding completion actions), which satisfies the constraints imposed by the task graph. The latter constraints are: (1) a task can be released only if all its predecessors have completed; (2) the time duration between the earliest release and the latest completion action is bounded by the end-to-end deadline of the task graph; and (3) the time duration between each pair of actions in a specified list is bounded by the corresponding separation limit.

Definition 4 *A* timed trace *on an alphabet A of actions is a sequence* $\sigma = (a_0, t_0), (a_1, t_1), \ldots$, *where each* $a_j \in A$ *and each* $t_j \in \mathbb{R}^{\geq 0}$, *with* $t_0 <= t_1 <= t_2 \ldots$. *We call* t_j *the* time-stamp *of the* j^{th} *action occurrence in the sequence.*

Definition 5 *Given a set of tasks* $\mathcal{T}$ *with associated timing constraints on their release and completion actions, a* release-schedule σ *is a function, that given a time instant for the completion of the task instances released earlier, assigns a time instant* $\sigma(r_{ij}) \in \mathbb{R}^{\geq 0}$ *to the release of the* j^{th} *instance of task* T_i *for each* $i \in \{1, \ldots, m\}$ *and each* $j \geq 0$. *Such an assignment must satisfy the offset and period constraints of each task. Likewise, a* completion-schedule τ *is a function, that given the release times of the* j^{th} *instance of task* T_i *and other tasks started earlier, assigns a time instant* $\tau(c_{ij}) \in \mathbb{R}^{\geq 0}$ *to the completion of the* j^{th} *instance of task* T_i *for each* $i \in \{1, \ldots, m\}$ *and each* $j \geq 0$. *Such an assignment must satisfy the execution time constraint of each task.*

Given a release-schedule σ and a completion-schedule τ, we can define the outcome $Outcome(\sigma, \tau)$ of the two schedules in the usual inductive way. This is a set of timed traces over $\bigcup_{1 \leq i \leq m} \{r_i, c_i\}$.

Definition 6 *Given a set of tasks* $\mathcal{T}$ *with associated timing constraints on their release and completion actions, and a task graph G expressing end-to-end constraints of a transaction, a* schedule σ *is a release-schedule, such that for all completion-schedules* τ, *every timed trace* $\pi \in Outcome(\sigma, \tau)$ *satisfies the following conditions:*

1. Precedence*: For every pair* $T_i <_{\mathcal{T}} T_j$ *in G, the* n^{th} *occurrence of* r_j *is preceded by the* n^{th} *occurrence of* c_i *in* π, *for every n.*
2. End-to-end deadline*: For an end-to-end deadline constraint of the form* $d(\mathcal{T}) \leq C$, $max(\{ts(\alpha') - ts(\alpha)\}) \leq C$, *where* α, α' *range over all the* n^{th} *occurrences of actions* c_j,r_k *respectively in* π, *for all* $j, k \in \{1 \ldots m\}$ *and for all n. Here* $ts(\alpha)$ *denotes the time-stamp of action* α.
3. Separation constraints*: For every constraint of the form* $d(a, b) \leq C$, $ts(\alpha') - ts(\alpha) \leq C$, *where* α, α' *are the* n^{th} *occurrences of* a, b *respectively in* π, *for all n.*
4. Liveness*: There is an* n^{th} *occurrence of* r_i? *for every* $i \in \{1 \ldots m\}$, *for every n.*

Intuitively, the above definition captures the fact that a schedule must specify a correct timed sequence of releasing tasks, no matter how much time the tasks take for completion, as long as they are within specified bounds. We now formally define the verification and synthesis problem we are interested in.

Definition 7 *The* timing verification *and* schedule synthesis *problems for end-to-end constraints are defined as follows. Given a set of tasks* $\mathcal{T}$ *and a task graph G, verify that there exists a schedule (i.e., a way of generating release actions for tasks) that satisfies the end-to-end constraints in G,* no matter when the tasks complete, as long as they satisfy the given constraints*, and synthesise such a schedule if it exists.*

Example 8 *Consider the set of tasks specified in Figure 1 and the task graph in Figure 2. In this example, the components C_1 and C_2 do meet the end-to-end constraints of the transaction. A possible schedule for meeting the requirements would be to release each task according to the timed trace $(r_1?, 2), (r_3?, 4), (r_2?, t)$ where t is the maximum of the completion times of T_1 and T_3, which is guaranteed to be within 10 time units. Note that releasing the task T_3 earlier than 4 time units (say at 3 time units) can lead to a violation of the freshness constraint (depending on when T_1 completes its execution, which the environment cannot control), although the interface for T_3 does not itself rule out the possibility.*

From the above example it is clear that the two timing analyses mentioned above can be carried out at the level of tasks rather than components, since they involve the timing assumptions and guarantees of only individual tasks. However, the component view would be essential when we consider the following situations:

- Tasks in a component have resource conflicts due to shared resources such as buffers.
- Components may not be "reentrant", in which case, the execution of two tasks of the component cannot be overlapped.
- Two different transactions can share the computations of certain tasks; for example, a sensor component will typically not perform the sensing task for different transactions separately – the sensor data will be broadcast to all the components with tasks that depend on the data.

All these situations can be modelled using the TIA framework, though the resulting TIA models will be more complex in general. For instance, resource conflicts can be modelled by using an additional TIA for for modelling the resource access, and guaranteeing mutual exclusion by allowing synchronisation with the resource TIA. An example of this kind is treated in Section 5, where we apply our component scheduling framework to derive a static time-triggered schedule for a set of distributed tasks.

3 Modelling Component Scheduling with Timed Interfaces

In this section, we model the tasks, and task-graphs of the previous section using timed interface automata. Interface automata were presented in de Alfaro and Henzinger (2001) as a formalism for studying compatibility of components in an *open* system. Timed interface automata (TIA) (de Alfaro et al., 2002) were proposed as an extension to model real-time constraints on interacting components. Due to lack of space we cannot present all the relevant details of the TIA model here. The reader is referred to de Alfaro et al., (2002) for the formal definitions and the important properties of the TIA model. Our use of the TIA framework is novel, and is different from the one in de Alfaro et al., (2002): our goal is to synthesise schedules rather than to check compatibility of components.

3.1 Timed Interface Automata for Tasks

Timed interface automata are syntactically similar to traditional timed automata as in Alur and Dill (1994), with the exception that location invariants are classified as either *input* or *output* invariants. The crucial difference lies in the semantics – timed interface automata correspond to *games* between players *Input* and *Output*, rather than just labelled transition systems. It is the responsibility of player *Input* to ensure that all the input invariants are met; similarly for the output invariants with respect to player *Output*.

Example 9 (TIA) *Figure 3 shows timed interface automata corresponding to the tasks in Example 1.*

- *The release and completion events of tasks are described using actions $r_i?$ and $c_i!$ of the task T_i.*
- *The clock variable x_i in the timed interface automaton for task T_i keeps track of the time elapsed since the last release of the task.*
- *The guards on the transitions describe when the actions $r_i?$ and $c_i!$* may *take place.*
- *The location invariants describe when certain actions* must *take place; for example the location invariant $O : x_1 < 8$ is an* output-invariant *(indicated by the label O), indicating that the output $c_1!$ must be produced while $x_1 < 8$ holds, otherwise player Output loses the game.*
- *The guards on the transitions with input action $r_i?$ specify that a minimum inter-arrival time should be maintained, otherwise player Input loses the game.*

Definition 10 *Let $\mathcal{T} = \{T_1, \ldots, T_m\}$ be a set of m tasks. The* TIA for a task $T_i \in \mathcal{T}$ *(also denoted by T_i) is given by a TIA with a single clock x_i, input action $r_i?$ and output action $c_i!$. The clock constraints appearing as invariants and guards express the pattern of release and completion times of the task. We assume that each TIA T_i is well-formed, i.e., both players have a strategy to let time diverge, unless the other player is to be blamed for monopolising the game from some point on (see de Alfaro et al. 2002).*

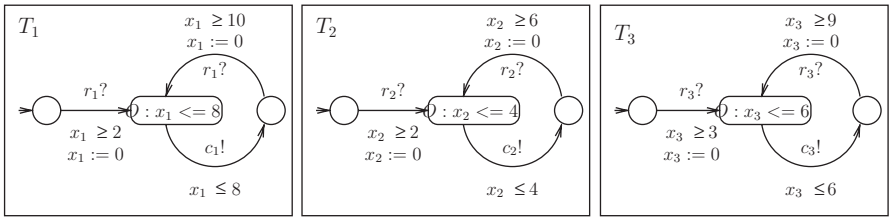


Fig. 3 TIA for tasks T_1, T_2 and T_3

3.2 From Task Graph to Specification Automaton

To solve the component scheduling problem, we use TIA in two distinct ways – first, to model the timing properties of tasks as presented above, and second to model a task graph for a transaction. We call the TIA for a task graph a *specification automaton*. Before describing the procedure for obtaining a specification automaton from a task graph, we give an example.

Example 11 (Specification Automaton) *The specification automaton corresponding to the task graph in Figure 2 is shown in Figure 4. It uses a clock x to record the time since the transaction was started, and a clock y to record the time since T_3 completed. The specification automaton has each r_i and c_i as* input *actions – it is an observer which detects violations of timing constraints by flagging an error state, and does not generate any output action (except the special action end*!*). It specifies all the legal runs of the environment (the scheduler) and the components that do not violate the end-to-end timing constraints.*

There is an input invariant $I_1 : x < 14$ associated with every location in the specification automaton, except the one on the extreme right (which is the final location*). This represents the fact that meeting the end-to-end timing deadline is the responsibility of player Input. For brevity, we use a statechart-like notation: an invariant associated with a super-location (the dotted oval in Figure 4) represents an invariant on all the locations contained in the super-location. Violation of the input invariant I_1 leads to a timed error state, where the progress of time is blocked. Similarly, the violation of the input invariant $I_2 : y \leq 6$ in the oval shaped location signifies violation of the freshness constraint and leads to a timed error state. The output action end*! *is a new action not shared by any other automaton which signifies the end of the transaction.*

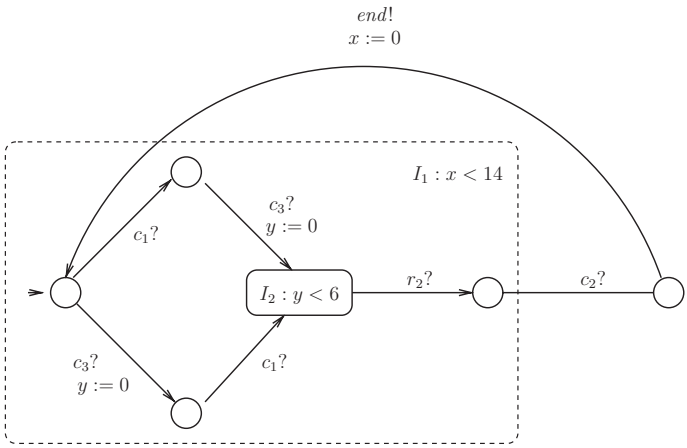


Fig. 4 Specification automaton for task graph in Figure 2

We follow the ideas in Abdeddaïm et al., (2003) to obtain the specification automaton from a task graph. First, we build a specification TIA P_i for each task T_i, consisting of three locations, corresponding to the task states *waiting*, *executing* and *completed*. The transition from the *waiting* to the *executing* state is taken when the specification TIA for the tasks in $\Pi(T_i)$ are all in their final locations.

Definition 12 *Let $G = (\mathcal{T}, <_{\mathcal{T}}, d)$ be a task graph. For every task $T_i \in \mathcal{T}$ its associated* specification TIA *is $P_i = (Q_i, q_i^{init}, q_i^{final}, \mathcal{C}_i, \mathcal{A}_i^I, \mathcal{A}_i^O, Inv_i^I, Inv_i^O, \tau_i)$ with the set of locations $Q_i = \{p_i^0, p_i^1, p_i^2\}$, the initial location $q_i^{init} = p_i^0$, the final location $q_i^{final} = p_i^2$, the set of input actions $\mathcal{A}_P^I = \{r_i?, c_i?\}$, the set of output actions $\mathcal{A}_i^O = \emptyset$, and the set of transitions τ_i include the tuples*

$$(p_i^0, \bigwedge_{T_j \in \Pi(T_i)} p_j^2, r_i?, \emptyset, p_i^1)$$

and

$$(p_i^1, true, c_i?, \emptyset, p_i^2).$$

The global specification automaton is obtained as a composition of the individual specification automata. The composition can be treated as composition of ordinary timed automata since the components have no shared actions. The composition ensures that the release actions of tasks do not violate the precedence constraints in the task graph. Next, we add some clocks and clock constraints, both as guards on transitions as well as location invariants, to take care of the end-to-end constraints in the task graph $G = (\mathcal{T}, <_{\mathcal{T}}, d)$. For the end-to-end deadline constraint $d_{\mathcal{T}} \leq C$, there is a clock t_e and an input invariant $I_e : t_e < C$ on all the locations of the composed automaton except its final location. For a separation constraint of the form $d(a, b) \leq C$, there is a clock t_{ab} which is reset on every transition with the action label a, and an input invariant $I_{ab} : t_{ab} < C$ on all locations that are sources of transitions labelled with action b. Finally, there is a transition labelled with the output action *end*! from the final location of the composed automaton to the initial location which resets the clock t_e.

The specification automaton in Figure 4 is actually obtained by applying some optimisations on the result of the above transformation on the task graph in Figure 2: the release actions $r_1?$ and $r_2?$ do not appear in Figure 4. A general optimisation scheme based on chain coverings of a partial order is presented in Abdeddaïm et al., (2003).

4 Timing Verification and Schedule Synthesis

In this section, we explain how the timing verification and schedule synthesis problems can be viewed as an instance of a timed game (see Maler et al., 1995; de Alfaro et al., 2002). between players *Input* (the environment) and *Output* (the system). Further, synthesising a schedule, i.e., a timed sequence of task release actions that obeys

the precedence constraints in the task graph and leads to all the end-to-end constraints being satisfied, corresponds to finding a winning strategy for *Input* in such a game.

As in all timed games, there are two kinds of moves available to each player: a player can either let time progress, as long as this does not violate an invariant for the player, or make a discrete transition to a new state when the associated guard becomes enabled. Thus a move of player *Input* (a *controllable* action) either triggers a task T_i via action $r_i?$ or allows time to elapse in a location. Similarly a move of *Output* (an *uncontrollable* action) either completes execution of a task T_i via action $c_i!$ or allows time to elapse in a location.

The game structure for the schedule synthesis and verification problem i.e., the graph on which the game is played (called a *timed interface* in de Alfaro et al., 2002), is obtained from the product of the timed interface automaton for each task and the specification automaton obtained from the task graph. The specification automaton has an *input invariant* on several locations capturing the end-to-end constraints. Violation of this invariant leads to a *timed error state*. The winning plays are those sequences of states in the game graph that avoid the error state, and in addition complete the transaction infinitely often, i.e., the goal involves both a *safety* and *liveness* condition. Finding a schedule for a given set of components that meets the end-to-end constraints of a task graph then amounts to finding a winning strategy for *Input* in the corresponding timed game.

Example 12 (Timed Game Structure) *For our running example, the product of the timed interface automata for the tasks T_1, T_2 and T_3 in Figure 3 with the specification automaton in Figure 4 represents the game structure on which the timed game is played. The fact that there exists a schedule satisfying the end-to-end constraints means that player Input has a winning strategy in the game.*

In the rest of this section we elaborate on the solution to the timing verification problem in terms of winning strategies for a timed game. In the following discussion, we assume we are given a set of tasks $\mathcal{T} = \{T_1, \ldots, T_m\}$ and a task graph $G = (\mathcal{T}, <_{\mathcal{T}}, d)$ on the set $\mathcal{T}$. The global specification automaton for the task graph G, defined in Section 3.2, is denoted T_G.

Consider the product TIA $T = T_1 \otimes T_2 \ldots \otimes T_m \otimes T_G$, i.e., the joint behaviour of all the TIA's corresponding to the tasks together with the specification automaton. Intuitively, the game structure $[\![T]\!]$ corresponding to the TIA T has the set of states $(s_1, s_2, \ldots, s_m, s)$ where each component s_i is a pair (q_i, v_i) of a location in T_i and a clock valuation over the single clock x_i, and likewise s is a pair (q, v) of a location in T_G and clock valuation over the clocks t_e and t_{ab}, where a, b range over the the actions r_i, c_j (see the paragraph in Section 3 following Definition 12). The input and output transition relations of $[\![T]\!]$ encode the possible moves of the corresponding player at a given state, and the new state that results, in the combined system of the m tasks and the specification automaton. Each transition is caused either by an immediate action (release or completion of a task T_i) or a timed action, where the player chooses to let time elapse. The available moves of a player in a state must conform to the location invariants for the player in the source and target location and

the enabled transition in the source location for each component TIA T_i. An input strategy is a partial function from sequences of states to the set of the enabled moves for *Input* in the final state of the sequence. So an input strategy is a way to specify the times at which the release actions for tasks occur, given the completion times for instances of tasks released earlier, while conforming to all the constraints imposed by the TIA for each task T_i. Likewise, one can define an output strategy.

Given an input and an output strategy, one can define the resulting set of outcomes starting from the initial state s_0 of $[\![T]\!]$ (see de Alfaro et al., 2002). These are finite and infinite sequences of the form $\sigma = s_0, \alpha_1, \gamma_1, s_1, \alpha_2, \gamma_2, \ldots$ where α_i is the move made by player $\gamma_i \in \{I, O\}$ in state s_i. A *winning* input strategy in $[\![T]\!]$ is one for which all possible output strategies lead to outcomes which avoid reaching all timed error states. Clearly, a winning input strategy corresponds to what we call a schedule (see Definition 8), except the liveness property may not be satisfied. In particular, an outcome can be empty – if no tasks are released there are no constraints to violate (assuming there are no input invariants in the TIA for the tasks, as is the case in Figure 3).

The following procedure takes care of the liveness problem. We take the composition (see de Alfaro et al., 2002) of the TIA corresponding to each task and the TIA for the task graph, and then find a winning input strategy for the goal $\Box\Diamond\ t^{final}$ (which says that the final location of the task graph component in the product is reachable) in the result. Intuitively, the composition $T_1 \parallel T_2 \ldots \parallel T_m \parallel T_G$ represents all schedules that satisfy the end-to-end constraints of the task graph G, without necessarily satisfying the liveness constraint; the latter is taken care of by the goal $\Box\Diamond\ t^{final}$. Details of how such games can be solved using symbolic fixed point computations can be found in Maler et al. (1995) and de Alfaro et al. (2002). The correctness of the procedure is captured by the following theorem.

Theorem 14 *A schedule satisfying the end-to-end constraints in G is a winning strategy for Input in the game structure* $[\![T]\!]$ *for the TIA given by the product* $T = T_1 \otimes T_2 \ldots \otimes T_m \otimes T_G$*, with the goal*

$$[\Box Good([\![T_1]\!], \ldots, [\![T_m]\!], [\![T_G]\!])] \cap (t\_div \cup blame^O) \cap \Box\Diamond\ t^{final}$$

where $Good([\![T_1]\!], \ldots, [\![T_m]\!], [\![T_G]\!])$ *is the set of all states in the game structure for the product T that are not immediate error states, t_div is the set of outcomes along which time diverges,* $blame^O$ *is the set of all outcomes where player Output monopolises the game, and* t^{final} *is the set of all states whose* T_G*-component has the final location of the specification automaton* T_G*.*

Implementation Currently, there is no implementation of timed interface automata. In order to experiment with our component scheduling framework, we hand-coded our TIA using the timed game automata (TGA) in the UPPAAL TIGA tool (Cassez et al., 2005; UPPAAL TIGA, 2006). Unfortunately, the synchronisation behaviour of TGA in UPPAAL TIGA is quite different from that of TIA. As a result, the task graphs cannot be represented as specification automata any more. Instead the precedence constraints have to be encoded using shared Boolean variables, and the end-to-end

deadline has to be specified as part of the winning condition (i.e., goal) for the controller. Note that this encoding in UPPAAL TIGA breaks the nice compositionality properties of the TIA framework. Also, the specification language used in UPPAAL TIGA for expressing goals is not very expressive, especially with respect to liveness constraints. The results of our experiments using the UPPAAL TIGA tool are described in the next section.

5 Application: Time-Triggered Schedule Synthesis

The time-triggered architecture (or TTA, see Kopetz and Bauer 2003) is a platform for distributed implementations of hard real-time systems used in automotive and avionics applications. It consists of a number of processors, called TTA nodes, that communicate by passing messages over a shared bus. The computation tasks running on the TTA nodes use the shared bus using a time-division multiple-access (TDMA) discipline based on a static schedule which recurs periodically. The problem of deriving a time-triggered schedule for a set of tasks is as follows (see Caspi 2003). We are given a set of m periodic tasks $\{T_1, \ldots, T_m\}$ and n processors. Every task is statically allocated to a processor. The task T_i has period P_i and is allocated to processor $host_i$. Its execution time lies in the interval $[l_i, u_i]$. Tasks can model computations as well as messages. There is a special processor modelling the bus – all tasks corresponding to messages are allocated to that processor. There is a precedence relation among tasks defined by data-flow constraints. This relation includes a computation task and a message task when the former is the sender of the message. Likewise, a message task precedes the computation task that is the receiver of the message. Tasks cannot be preempted once they start running. Tasks also have relative deadlines among them to model end-to-end constraints. These are of the form $\theta_i - \theta_j \leq C$, where $\theta_i \in \{s_i, e_i\}$, where s_i is the start time and e_i is the completion time of task P_i. The problem is to find a static schedule for the bus for transmission of messages, and a schedule for each TTA node for the tasks that are allocated to that node, so that all timing constraints are satisfied.

In order to apply our framework to this problem, we start with a TIA for each processor (TTA node or bus). Since each task is periodic, and has a best-case computation time l_i and a worst case execution time u_i, we can model it using TIA, as in Section 3. However, now we have the complication that several tasks can be allocated to a single processor, and tasks cannot be preempted. This constraint can be captured using a simple device: just take the composition of the tasks allocated to a single process with a synchronising automaton which enforces the execution of only one task at a time. For every two tasks T_i and T_j allocated to the same processor, such a synchronising automaton is shown in Figure 5. Intuitively, the automaton serialises the execution of T_i and T_j. This example illustrates the case where a component (see the description in Section 2) corresponds to a set of tasks with resource constraints

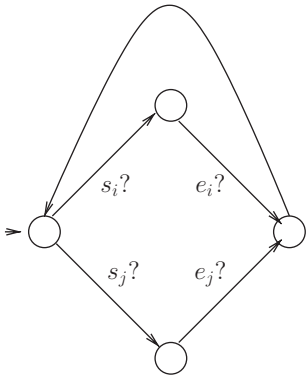


Fig. 5 Synchronisation automaton for enforcing non-preemptive serial execution of tasks T_i and T_j

among them. The resource constraint here is the non-preemptive nature of task execution, and a component describes the set of tasks allocated to a processor.

Note that the constraint that each task can run only in its allocated slot is taken care of by the strict periodicity constraint. If the tasks do not have the same period, we can take the lcm of the periods to be the working period, and create multiple instances of each task to fit the period. New precedence constraints must be added between these new instances to indicate their order.

The end-to-end constraints can be modelled as TIA as in Section 3. The composition of all the TIA involved, if defined, gives us a feasible schedule for the execution of the tasks. However, the schedule is not static, since it is an *Input* strategy in which input moves can depend on previous output moves. To extract a static schedule, we can take the specification of execution times as worst case execution times of tasks (worst case communication times for messages) instead of intervals. This restricts the choices for player *Output* – tasks can complete only after a fixed known duration after they start.

Discussion Various approaches to the problem of synthesising a static time-triggered schedule based on constraint solving, branch-and-bound techniques and mixed integer linear programming (MILP) have been proposed in the literature (see Schild and Würtz, 2000; Caspi et al., 2003; Zheg et al., 2005 for example). Because of disjunctions in mutual exclusion constraints, when posed as an optimisation problem, the feasible region is not convex (see Caspi et al., 2003). The typical workaround is either to use backtracking techniques based on branch-and-bound search (as in Caspi et al., 2003), or code the problem using binary decision variables and use a MILP solver (as in Zheng et al., 2005). The latter technique involves guessing a large constant M, which should be as small as possible for feasibility reasons.

It is not clear whether our approach is more scalable than the above approaches. For a definitive answer, we need an implementation of TIA that we can use to carry out experiments on real-life time-triggered systems. Our expectation is that using on-the-fly techniques of Cassez et al., (2005) we can effectively conquer the inherent

EXPTIME-complexity of the timed control synthesis problem for reachability and safety objectives.

Example 15 (Adaptive Cruise Control) *This example is adapted from Kandasamy et al., (2003) and Zheng et al., (2005) except we require task scheduling on an ECU to be non-preemptive. The* adaptive cruise-control *(ACC) feature in an automobile automatically adapts the speed of the vehicle to the speed and distance of the vehicle in front. The ACC application involves the timely interaction among a number of tasks that are distributed, and must interact by sending messages. These tasks can be grouped as follows:*

- Sensors:

  T_1: *Object distance and speed*
  T_2: *Vehicle speed*
  T_3: *Throttle position*

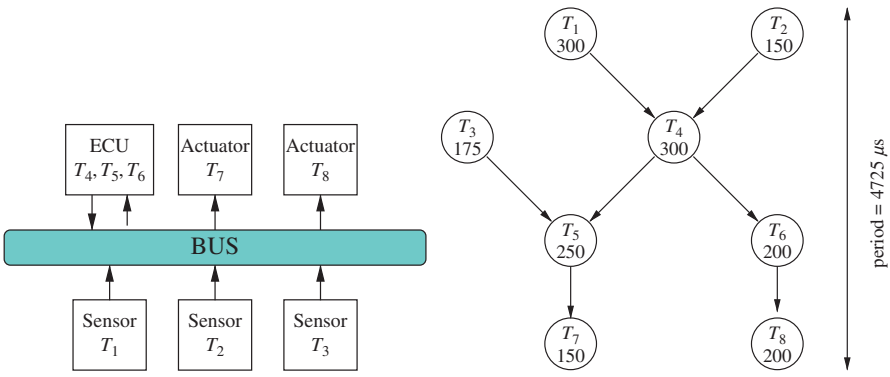
- Controllers:

  T_4: *Desired speed*
  T_5: *Desired throttle position*
  T_6: *Desired brake position*

- Actuators:

  T_7: *Throttle actuator*
  T_8: *Brake actuator*

Figure 6(a) shows the physical architecture of the system – all sensors and actuators are directly connected to the bus, and one ECU (electronic control unit) hosts all the controller tasks. Figure 6(b) shows the task graph, with the WCET (worst case



(a) Architecture (b) Task graph

Fig. 6 Adaptive cruise control

Source	Target	Delay (μs)
T_1	T_4	350
T_2	T_4	650
T_3	T_5	1425
T_4	T_5	500
T_4	T_6	500
T_5	T_7	500
T_6	T_8	500

Fig. 7 WCCT for messages in ACC

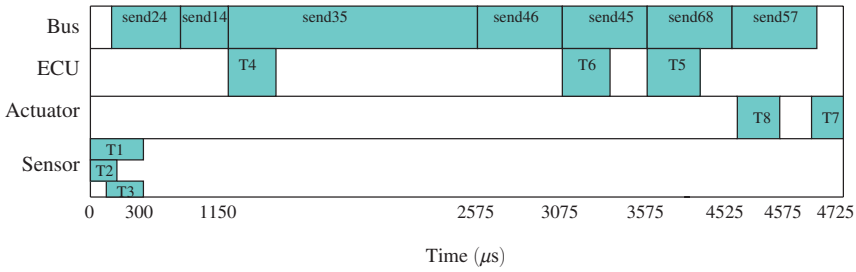


Fig. 8 Time-triggered schedule for ACC example

execution time) of each task appearing below the task name. The end-to-end deadline of the entire transaction is the same as the period, i.e., 4725 μs.

Figure 7 shows the WCCT (worst case communication time) of the messages. The time-triggered schedule synthesised by our method is shown in Figure 8.

6 Conclusion

Component-based development poses new problems for embedded control systems software. Traditional real-time scheduling theory has been successful in investigating whether a set of tasks can be scheduled on a given platform using the characteristics of the tasks and the platform. The underlying assumption is that the task characteristics have been derived from the application requirements. Since today's embedded systems are not monolithic, but are built using pre-designed components which are composed to realise a given functionality, what is needed is a new approach that combines task scheduling within a component with what we call component scheduling. This paper is an attempt to define and solve the component scheduling problem.

As future work, we would like to have an implementation of timed interface automata in order to carry out experiments to demonstrate the scalability of our approach. Experiments on small examples based on hand-coding of TIA using UPPAAL TIGA have been encouraging.

Acknowledgements The author wishes to thank Prahlad Sampath and S. Ramesh for their detailed discussions and feedback on this work. Thanks are due to Sri Satya Aravind Akella for carrying out some of the experiments using UPPAAL TIGA described in Section 5.

References

Abdeddaïm, Yasmina, Kerbaa, Abdelkarim, and Maler, Oded (2003). Task graph scheduling using timed automata. In *Parallel and Distributed Processing Symposium 2003*. IEEE Computer Society.

Altisen, Karine, Gößler, Gregor, Pnueli, Amir, Sifakis, Joseph, Tripakis, Stavros, and Yovine, Sergio (1999). A framework for scheduler synthesis. In *IEEE Real-Time Systems Symposium*, pages 154–163.

Altisen, Karine, Gößler, Gregor, and Sifakis, Joseph (2002). Scheduler modeling based on the controller synthesis paradigm. *Real-Time Systems*, 23(1-2):55–84.

Alur, Rajeev and Dill, David L. (1994). A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235.

Caspi, Paul, Curic, Adrian, Maignan, Aude, Sofronis, Christos, Tripakis, Stavros, and Niebert, Peter (2003). From simulink to SCADE/Lustre to TTA: a layered approach for distributed embedded applications. *ACM SIGPLAN Notices*, 38(7):153–162.

Cassez, Franck, David, Alexandre, Fleury, Emmanuel, Larsen, Kim Guldstrand, and Lime, Didier (2005). Efficient on-the-fly algorithms for the analysis of timed games. In *CONCUR 2005 – Concurrency Theory, 16th International Conference*, volume 3653 of *Lecture Notes in Computer Science*, pages 66–80. Springer.

de Alfaro, L. and Henzinger, T.A. (2001). Interface automata. In *Foundations of Software Engineering*, pages 109–120. ACM Press.

de Alfaro, Luca, Henzinger, Thomas A., and Stoelinga, Mariëlle (2002). Timed interfaces. In *Embedded Software, Second International Conference, EMSOFT 2002*, volume 2491 of *Lecture Notes in Computer Science*, pages 108–122. Springer.

Kandasamy, Nagarajan, Hayes, John P., and Murray, Brian T. (2003). Dependable communication synthesis for distributed embedded systems. In *SAFECOMP 2003 Proceedings*, volume 2788 of *Lecture Notes in Computer Science*, pages 275–288. Springer.

Kopetz, Hermann and Bauer, Günther (2003). The time-triggered architecture. *Proceedings of the IEEE*, 91(1):112–126.

Maler, Oded, Pnueli, Amir, and Sifakis, Joseph (1995). On the synthesis of discrete controllers for timed systems. In *Theoretical Aspects of Computer Science*, volume 900 of *LNCS*, pages 229–242. Springer-Verlag.

Schild, Klaus and Würtz, Jörg (2000). Scheduling of time-triggered real-time systems. *Constraints*, 5(4):335–357.

UPPAAL TIGA (2006). UPPAAL TIGA home page. `http://www.cs.auc.dk/~adavid/tiga/`.

Zheng, Wei, Chong, Jike, Pinello, Claudio, Kanajan, Sri, and Sangiovanni-Vincentelli, Alberto L. (2005). Extensible and scalable time triggered scheduling. In *Application of Concurrency to System Design (ACSD 2005)*, pages 132–141. IEEE Computer Society.

An Instrumentation-Based Approach to Controller Validation

Rance Cleaveland

Abstract This talk presents instrumentation-based validation (IBV) as a means to check whether models of controllers given in Simulink®/Stateflow® satisfy functional requirements. IBV relies on the formalization of requirements as small "observer models" whose purpose is to monitor, and detect violations of, single requirements. These models may then be used as instrumentation for larger controller models, and testing and other V&V activities performed in order to check for the presence of errors. This presentation discusses IBV in general and illustrates its implementation in the Reactis® model-based testing environment.

S. Ramesh and P. Sampath (eds.), *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*, 155.

A Design Methodology for Distributed Real-Time Automotive Applications*

Werner Damm and Alexander Metzner

Abstract This paper presents a survey on techniques for supporting a seamless development process of embedded automotive real-time systems. Starting from a set of requirements we show how to integrate early design space exploration, real-time requirements and the definition of component interfaces in a distributed organization of suppliers and OEMs. The main focus is to provide building blocks for a design methodology enabling an AUTOSAR driven process. We also present a method to formally specify requirements in terms of sequence diagrams and how these requirements can be formally checked against implementations by using a rich set of time analysis techniques. Finally, we present our approach of optimizing the implementation in order to reduce the number of ECUs or to increase robustness.

1 Introduction

Electronic system development for automotive applications is currently undergoing major changes to cope with the exponential growth of functionality offered by cars.

The old paradigm of equating one function to one electronic control unit (ECU) delivered by one supplier is broken: new functions "tap" information on the car status from multiple sources, and rely on the proper interplay of, e.g. power-train and brake systems in advanced stability protection applications. The implementation of such automotive functions involves distributed task sets running on multiple ECUs, with bus-based inter-task communication. The development of such intermeshed, complex systems which are built for product lines that deal with numerous different variants and demand lifetime maintainability is only feasible if a component based design methodology is applied. Moreover, the increasing architectural complexity of electronic systems consisting of eighty and more ECUs is no longer manageable neither for system integration nor for cost reasons. The AUTOSAR (Heinecke et al., 2004) initiative was founded exactly for this reason and aims at reducing the number of ECUs by making SW functions relocatable. This induces several challenges: Firstly, the verification of real-time requirements demands for a seamless design process,

*This work was partly supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS).

S. Ramesh and P. Sampath (eds.), *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*, 157–174.

where formally defined requirements can be checked against the implementation of the system. Secondly, application-derived end-to-end latencies must now be established across shared bus systems, entailing the need to perform cross ECU network schedulability analysis incorporating models of the underlying communication architecture. Thirdly, such analysis must meet the requirements of a multi-organizational design process, in which multiple suppliers develop subsets of such a system, with system and integration responsibility resting with the OEM. This entails, that OEMs must be able to assess the overall design space for possible target architectures as well as the feasibility for cost efficient realizations of new functions prior to subcontracting suppliers for pieces of the design.

We address these challenges by a set of building blocks to support a distributed design methodology providing complete traceability from top-level requirements to real-time characteristics of implementations. These building blocks help to give partial solutions to the issues arising from applying an AUTOSAR driven development process by the use of incremental integration, formal requirement checking and optimization of embedded real-time applications. The solutions are partial in the sense of defining them under some assumptions: We assume a Flexray/TDMA based bus system used as backbone, an OSEK Time operating system running on all ECUs, a resource separation between suppliers based on slot allocation, and, finally, the usage of a fixed-priority scheduling of resources by suppliers. However, some of the building blocks are only partially dependent on these assumptions. These building blocks will be embedded in a development methodology driven by the application of so-called rich components (Damm et al., 2005), which aims at defining a seamless component based design process (cf. the SPEEDS project, http://speeds.eu.com).

In the following we start the presentation bottom up by describing use cases for optimization and analysis of deployment architectures with respect to real-time objectives. We shortly introduce our proposed technique on this topic and elaborate on constraints to higher layers of abstraction within the development process. Successively we describe techniques to solve the issues related to those constraints, elaborate new constraints, and go another step further climbing the hierarchy of the development process until we have defined a design methodology that is capable of enabling embedded real-time systems design in an AUTOSAR era of development. After starting with deployment optimization we present our approach to close the gap between an implementation and formal requirements by combining formal verification techniques with real-time analysis. Afterwards, the use of timed sequence diagrams for real-time specifications and their capabilities for formal verification is shown. Issues arising from that design stage deal mainly with integration related constraints, like design partitioning and putting together components coming from different suppliers. From a real-time perspective we present techniques and a methodology for system tailoring and incremental integration and implementation.

2 Design Optimization

As starting point we assume an executable networks of tasks that realize the functional behavior and a given architectural topology of the system. Hence we have to integrate and deploy software entities onto hardware entities while preserving validity of the complete design and while achieving optimality with respect to some objectives. How we can reach the assumption of having implementations while coming from requirements by applying a development methodology will be presented in the following sections. However, deployment synthesis is one key technique for future system design targeting at the reduction of architectural complexity by migrating to more software based systems. That is why we start our presentation with this topic which enables us to achieve the aims given by AUTOSAR.

Deployment synthesis assigns executable tasks to ECU nodes and messages to sequences of buses in a given system architecture while preserving their temporal requirements – given in terms of deadlines – by using middleware components, like a real-time operating system. These middleware components control the execution of tasks on a node, hence those behaviors are essential and we assume that they are given with the system architecture. A system architecture consists of a number of ECUs and a number of communication media the ECUs are connected to. In order to show the applicability of our approach we additionally introduce memory consumption as representative for other resources. Thus the architecture is described by a tuple $A = (P, \mu^A, K)$, where P is the set of ECUs, $\mu^A : P \rightarrow \mathbb{N}$ is the amount of memory attached to each ECU and $K \subseteq 2^P$ is the set of communication media. A task may send messages at the end of each computation to one or more other tasks. The arrival of a message on an ECU may activate the receiving task. The timing constraints exist for each task and each message. The task model is defined by a set T of tuples $(t_i, c_i, \mu_i^T, \gamma_i, \pi_i, \delta_i, d_i)$ describing the individual tasks. The elements are the activation period or minimal inter-arrival time $t_i \in \mathbb{N}$, the worst case execution times (WCET) $c_i : P \rightarrow \mathbb{N}$, memory consumption of the task on each ECU $\mu_i^T : P \rightarrow \mathbb{N}$, the messages (including their target, size and their deadline) the task is sending $\gamma_i \subseteq T \times \mathbb{N} \times \mathbb{N}$, and the ECUs the task is allowed to be allocated on $\pi_i \subseteq P$. Tasks from δ_i are not allowed to be allocated together with τ_i (set of redundant tasks), and $d_i \in \mathbb{N}$ is the deadline of τ_i. Task allocation is now defined by the following mappings: $\Pi : T \rightarrow P$ that assigns each task in T to an ECU in P, $\Phi : T \times T \rightarrow \{0, 1\}$ defines a priority ordering of tasks, and $\Gamma : \left(P \times \mathbb{N} \times \mathbb{N} \rightarrow 2^K\right)$ assigns each message to a set of communication media. The task of generating a deployment of software tasks is now to find these mappings Π, Γ, Φ for a given system while guaranteeing all requirements.

Beside the technical question of how to find these mapping functions there occur several use cases for optimization during the integration/optimization phase of a given development process, which have to be covered by an automatic deployment approach:

1. Aiming at reducing costs leads to a demand for using less ECUs, hence there is a need for architectural exploration. It is obvious, that during integration of already implemented components changing the basic architectural topology and implementation is impossible. This main topology has to be defined in advance (see Section 6). However, what can be achieved is the reduction of used ECUs in order to remove unused ECUs from the network.
2. Incremental integration – during development as well as after delivering systems in terms of applying upgrades – demands for preserving slackness and major parts of the already existent deployment in order to minimize changes on ECUs implementation while merging new functions and the initial system's implementation.
3. The consideration of uncertain parameter valuation, e.g. execution times, in system analysis is a key challenge for achieving robust systems. Therefore, deployment techniques should provide capabilities to cope with such variations.

In the following we present our approach for automatic deployment synthesis and afterwards we discuss how this approach can be used to cover these use cases.

Once we have given an architectural specification and more concrete component specifications we optimize the deployment of software tasks. This is done by generating a refinement of an initial calculated pre-allocation (see Section 6) by applying fine-grained analysis methods for extra-functional properties, like real-time properties, while minimizing or maximizing different objective functions according to the different use cases. We use a SAT checking based approach for optimization, that we will sketch in the following (for more details refer to Metzner et al., 2005, 2006a, b).

According to our system model we specify the allocation problem in terms of arithmetic inequalities over integers. Assume a set of ECUs P and a set of tasks $\tau_i \in T$. We introduce a variable a_i of type integer in the range of 1 to $|P|$, where a valuation of such a variable $a_i = x$ means that task τ_i is allocated on ECU $p_x \in P$, thus we implement Π in terms of finding valuations for these variables a_i. Similar encodings are used to define the functions Γ and ϕ (Metzher et al., 2006b). The optimization procedure now has to find values for all these variables such that the requirements are fulfilled. Furthermore, if we add an objective function, e.g. minimize the utilization U_B on a bus B, the optimization approach has to return a solution where the upper bound of the utilization is minimal. As driver for the optimization we apply a SAT checker modulo real-time theory, where the SAT checker is used to assign values to the placement variables. Whenever the SAT checking algorithm reaches a partial deployment configuration, it calls the real-time theory in order to check this partial solution for feasibility. If the partial solution turns out to be infeasible, the real-time theory returns an explanation of infeasibility in terms of the actual configuration which in the SAT part of the combined procedure is used to build contradictory clauses in order to avoid re-visiting this partial configuration. In order to find an optimal solution we apply this procedure in a binary search, driven by the bounds of the current optimization clause.

The advantage of this approach is that it returns the optimal solution while coping with discontinuous solution spaces, like, e.g. given in Davis and Burns (2005).

Furthermore, applying this technique to optimization problems with multiple, sometimes contradictory, objective functions, we are able to achieve optimal witnesses of a pareto frontier. The main drawback of this approach is that the complexity with respect to run-time is in principle exponential in the number of variables. However, our evaluations have shown that it nevertheless scales very well even for big problem instances (some ten ECUs and up to 100 tasks were solved in less than one hour). See Metzner et al. (2006a) for a detailed summary of our experiments.

Using this approach we can now apply different encodings of the optimization objective to cover the three use cases described above:

1. For minimizing the number of used ECUs we simply can introduce variables encoding the number of tasks allocated on each ECU. The applied optimization objective for this use case is defined by yielding as much ECUs with no tasks allocated on it as possible (Metzner et al., 2006a).
2. Preserving maximal slackness can be achieved by defining optimization objectives which minimize the loads of networks and ECUs (Metzner et al., 2005, 2006b). Changes with respect to a given initial deployment during incremental integration simply can be captured by encoding the initial deployment as a reference mapping Π' and use this for counting the differences. Minimization of changes in comparison to the initial deployment can then be achieved by an optimization objective that aims at minimizing the number of changes (Metzner et al., 2006a).
3. Aiming at robustness caused by uncertainties of parameter valuations can be achieved by exploiting the combination of a SAT encoding of the allocation function and the real-time theory that is in charge of validating the deployment: In addition to the encoding of Π, Γ and ϕ as part of the SAT inequalities we introduce variables for those parameters which possess uncertainties in order to let the SAT solver determine their valuation with respect to an optimization objective (e.g. for yielding high slackness). Since the real-time theory already has to be capable of validating partial deployments, we extend the underlying scheduling theory to be able to deal with partial assigned parameters (Metzner et al., 2006a).

Note that during final design optimization supplier's tasks can be moved from one to another ECU. As long as the execution time is not increased by using another type of ECU, the system remains sustainable. This holds even for jitter and offset values, which in general are known to be not sustainable (Baruah and Burns, 2006), because in a setting according to our system assumptions (using TDMA-based slot allocations for supplier separation), these values are not affected by reducing execution times.

A key enabler for performing automatic optimization is the possibility of assessing real-time behavior in an efficient but accurate way within the real-time theory which is coupled to the SAT solver. For our system setting of TDMA-based scheduling on ECUs encapsulating fixed-priority scheduling in a hierarchical way, we propose to use a schedulability analysis method which is tailored to such types of systems, and, therefore, provides accurate analysis results while yielding an efficiency that enables us to use it during system optimization, as described above. We present this method in the following section.

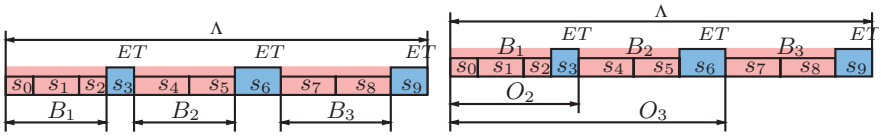
3 Implementables and Real-Time Analysis

Implementing functions means structuring the behavior into executable tasks, writing code for them and selecting a scheduling paradigm, hence defining the used real-time operating system. Since we are dealing with automotive applications, we restrict ourselves to the usage of OSEK and OSEKTime. We propose to use OSEKTime and a standard OSEK on top of the time triggered part. The OSEKTime part owns the control of the processor and triggers separated execution windows by timer interrupts. As described in Section 6, a TDMA schedule is constructed by a set of slots. Disjoint subsets of slots build execution windows, in each of them a set of tasks (assigned to one supplier) is locally scheduled. To simplify the analysis we assume that task sets assigned to an execution window are scheduled under a fixed priority preemptive scheduling. We call slots assigned to an execution window *ET* slots (slot s_i is in the set of ET slots S^{ET}). All other slots are called *TT* slots ($s_i \in S^{TT}$). In this way we are able to implement a hierarchical scheduling which will be a requirement coming from the way we implement the supplier-OEM relationship (see Section 6). Now that we have an implemented and integrated system, the issue is to analyze the temporal behavior of the integrated parts and look for optimization options. Therefore, we will derive a response time analysis based on traditional analysis approaches (cf. Audsley et al., 1995; Hamann et al., 2004) for tasks that are assigned to a set of ET slot.

The main idea of this analysis is to extend above cited scheduling analysis for fixed priority preemptive scheduling by adding blocks induced by interference of the tasks execution with TT slots of the TDMA round. Firstly we collect all successive slots $s_j \in S^{TT}$ that are not ET slots and construct new, bigger slots between two ET slots which we call *blocking slots*. Figure 1(a) shows the new blocking slots and their length which is constructed by the sum of the length $\lambda(s_j)$ of all merged slots.

For each of these slots B_i we can construct a fixed offset O_i. The value of O_i is constructed by considering the length of the merged slots and the offset of the first slot s_{i_1} within each B_i, as shown in Figure 1(b).

Because of the cyclic nature of the TDMA round, we can use the extended version of the response time analysis with offset presented in Palencia and Harbour (1998). If we use this extension, then we have to find out what is the critical instant. It is obvious that starting within an ET slot is not the worst case. Thus we observe – in analogy



(a) Slots that are not used for ET tasks can be treated as blocking slots

(b) Blocking slots between ET slots have fixed offsets

Fig. 1 Building blocking slots and offsets

to the critical instant from Liu and Layland (1973) – the set of worst case starting points for the analysis of ET tasks to be the beginning of each blocking slot B_i.

An important observation from this set of worst case scenarios is that it is not sufficient to consider a single critical instant but we have to regard all instants that are possible. Different starting points in time will shift the offsets of the blocking slots within the response time calculation. Furthermore, blocking slots without ET slots between can be merged. In order to calculate critical instant specific offsets and to consider all possible starting points, we define a rotation operation $rot(TDMA)$, that produces a new view of the TDMA round in which the TDMA round is rotated to the left until the nearest ET slot of the original TDMA round just has passed, i.e. the new view of the TDMA round starts with the first slot $s_i \in S^{TT}$ after the nearest (from the left) ET slot. After performing the $rot(TDMA_{k-1}) =: TDMA_k$ we construct the blocking slots B_j^k and use this view ($TDMA_k$) of the TDMA round as starting point of our response time analysis. Let O_j^k the offset of blocking slot B_j^k in the TDMA view $TDMA_k$, that can be calculated by summing up all blocking slots and all ET slots before B_j^k in $TDMA_k$. Let $\lambda(B_j^k)$ the length of a blocking slot B_j^k and n the number of blocking slots (and thus the number of ET slots). Then it holds:

$$I_i^{TT}(r_i, k) = \sum_{j=1}^{n} \left\lceil \frac{r_i - O_j^k}{\Lambda} \right\rceil_0 \cdot \lambda(B_j^k) \tag{1}$$

where r_i is an arbitrary time interval and $I_i^{TT}(r_i, k)$ represents the time amount that is consumed by blockings from TT slots. With the usually used equation for calculating the interference costs $I_i^{ET}(r_i^k)$ for a task τ_i in a fixed priority preemptive scheduling (cf. Tindell 1994) we can now calculate the response time r_i^k of ET tasks for a given TDMA view:

$$r_i^k = c_i + I_i^{ET}(r_i^k) + I_i^{TT}(r_i^k, k) \tag{2}$$

In order to determine the worst case response time for an ET task we have to apply the rotation and response time analysis for each possible instance iteratively. Hence, we perform the rotation operator:

$$TDMA_k := \underbrace{rot(rot(\ldots rot(TDMA)\ldots))}_{\text{k times}} \tag{3}$$

Finally, the worst case response time is determined by the maximal response time w.r.t. a certain view $TDMA_k$ of the TDMA round, i.e.

$$r_i = \max_{k \in \{1,\ldots,n\}} \left\{ r_i^k \right\} \tag{4}$$

Note, that n is the number of blocking slots after the first rotation and thus the number of ET slots. Therefore, only very few instants have to be considered. In Metzner (2005) we have proven that considering all rotations is sufficient for finding all possible critical instants.

Given this base of analysis methods for hierarchical OSEKTime applications, we implemented a tool set for validating real-time properties in many more facets than

explained in this paper. Finally, with these methods on-hand, the integrator of complex embedded systems has the capability of performing virtual integration and of searching for optimization possibilities as well as looking for robustness issues by varying the component's parameters. However, there is one main issue still open, which we will address in the next section: the capability of propagating the results of the real-time analysis into the abstraction level of real-time requirements, including their proofs.

4 Implementation Verification

Real-time scheduling analysis methods are applied to implementations and therefore are capable of predicting the temporal behavior of the system under design. While this is a very useful technique, there is still a gap between properties derived by those analytical approaches and the requirements specified in terms of formal models, like LSCs (see next section). In this section we briefly sketch how we close this gap by translating both sides in the same semantic domain, namely timed automata, and pave the way towards a seamless design process by allowing to backannotate results from scheduling analysis into pre-defined classes of timed automata which then can be used for the formal verification of temporal properties.

In order to combine real-time scheduling and formal methods, recent approaches model the underlying scheduling algorithms using the same or a similar formalism as the model itself. Beside others timed automata based approaches are widely used in this area (Fersman and Yi, 2004; Hendriks and Verhoef, 2006; Madl et al., 2006). Although those verification techniques provide exact results a problem here is the limited system size due to the computing power needed for verification. Moreover, compositional methods are usually of limited use because scheduling analysis is holistic, i.e. the temporal behavior of tasks cannot be considered in isolation.

We propose an approach, which is also based on timed automata, but utilizes dedicated verification techniques and tools used in the area of scheduling analysis, that can deal with heterogeneous, distributed system architectures, complex task networks, and mixes of different scheduling algorithms (Richtes et al., 2003). Our approach defines only the components needed to model task networks as timed automata, while the system architecture and scheduling algorithms are modeled using the underlying scheduling analysis framework. We provide a mapping from the respective task networks including their response times to our models, and we are able to show that by means of this mapping both models are semantically equivalent. With this our models subsume all possible temporal behavior due to task interferences, and in advance we assure that given schedulability of a task network model implies that the timing behavior of the respective timed automata model is valid.

Real-time analysis, as it was presented in the last chapter, can also be interpreted in another way, by so-called event streams (Hamann et al., 2004). The idea is to transform critical instant theorems used in scheduling analysis into the world of timed

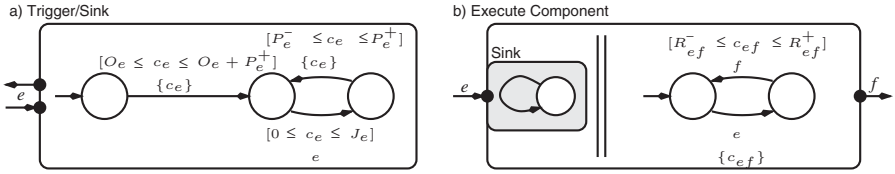


Fig. 2 CTA component definitions

traces in order to characterize task activation properties. Furthermore, the delay a task causes can be interpreted as a transformation function on such a triggering event stream leading to output event streams which then can be used to build a composition theory. Interestingly, these event streams and their transformers have been shown to be equivalent to scheduling analysis (Hamann et al., 2004), and, furthermore, have a timed language semantics.

The formalism we use for modeling temporal behavior of task networks is called Cyclic Timed Automata (CTA) and consists of a set of pre-defined parameterizable timed automata. Figure 2 gives an example, where each CTA contains a parallel composition of a trigger automaton (so-called sink) and an execution automaton. Sinks characterize the temporal activation of task according to the input event streams. Execution automata characterize the event stream transformer function. Each CTA is parameterizable with those parameters that define the shapes of event streams, i.e. period, jitter and response times. We have shown, that the timed language that sinks accept is equivalent to the language created by input event streams (Dierks et al., 2006). Simultaneously we have shown that the parallel composition of execution automaton and sinks is equivalent to the output event stream created by an event stream transformer on an input event stream. Obviously, this implies the equivalence between execution automaton and event stream transformer.

This result is very important because it closes the gap between scheduling analysis (now expressible in terms of timed automata) and high level specifications, which have also a timed automata based semantics. Hence we are now able to formally check requirements against implementations, which is key for providing a seamless development process. Our experiments show, that we can deal with large distributed systems while outperforming other timed automata based approaches by far (see Dierks et al., (2006) for detailed summary of these experiments).

To summarize, up to now we are able to deploy and integrate implementables on a given hardware architecture, we are able to analyze the temporal behavior, and we are able to automatically check the temporal behavior of an implementation against higher level specifications. Typically, the latter ones are derived from requirements given in a higher layer of abstraction. In the next section we propose to use a graphical specification language in which specifications can be modeled in a protocol-like style and which has a timed automaton based semantics, meaning that we may apply the techniques presented in this section in order to reach a seamless design process while crossing different layers of abstraction.

5 Requirement Checking

Increasing the quality of real-time systems is a main issue of the definition of development processes, often supported by automatic analysis tools, as described in the last section. Furthermore, the specification of formal requirements allows to reach the overall goal of a seamless process by allowing for formal verification. We propose to use a visual specification language that has a sound semantical base and is easy to use by engineers.

The Live Sequence Charts (LSC) (Damm and Harl, 2001) language is a formally rigorous variant of the well-known scenario language Message Sequence Charts (MSC). LSCs yield expressive power by means to distinguish mandatory and scenario behavior, means to characterize by another scenario the context in which a specification applies, and means to distinguish required from possible progress, i.e. to require liveness. In the context of this work we use LSCs as specification language for sequences of task and message activation and time relations over parts of these sequences. Since LSCs are an established visual formalism for requirements in formal, model-based development equipped with a formal semantics in terms of Timed Büchi Automata, they are aiming at formal verification as well. The subclass of the LSC language that is needed for real-time requirements we aim at falls into the class of time-bounded LSCs. For this class the complexity of formal verification by using model checking techniques is known to be practically efficient (Klose et al., 2006). Within our real-time framework we exploit this advantage and are able to provide an efficient formal verification for component specifications, taking into account their actual real-time behavior in a deployment architecture.

A special use case of these specifications is the characterization of temporal behavior of interfaces and compositions of components. Three different kind of specifications are used under the framework of a component-based development process (Damm et al., 2005): so-called black box specifications characterize the temporal behavior of executable components, e.g. the time delay between the triggering of an component by their inputs and the time instant a computational results at corresponding outputs becomes visible. Secondly, input and output ports can be characterized by assume/guarantee pairs of specifications, where the assume part defines the allowed environment applied to the component such that the guarantee at outputs are achievable. Such specifications are used for compatibility checks in compositions of components, e.g. an input assumes a fixed time period between successive activations on dedicated signals. Formal verification can then be used to check whether the sending components provides a guarantee that fits this assumption. Lastly, LSCs can be used for the specification of complex protocol behavior in the composition of components, here called gray box specification. Examples for this kinds of specifications are inter task communication protocols while guaranteeing end-to-end delays on paths within the composition. In this use case formal verification is used to establish the gray box specification from the characterization of the black box specifications of the connected subcomponents.

While such an approach is very useful in a design phase dealing with networks of functions, we now have to create a link to the earliest design phase in order to provide a seamless development process, where the OEM captures requirements and tailors the design to portions to be implemented by different suppliers. Such a tailoring is used to determine properties of sub-systems which can be checked by the approach described before. Crossing the borderline of responsibilities demands for solutions for two main issues: The partitioning of a systems into modules assigned to suppliers, including smooth integration capabilities at OEM site, and, dependent on that tailoring, refining the requirements to partial requirements for each supplier. Both issues will be addressed in the next section.

6 Design Tailoring and Pre-Allocation

Due to the different roles of OEM and supplier, the way forward to hardware independent feature implementations in an AUTOSAR driven development process demands for spending more effort in the specifications that will be delivered to the suppliers. Typically, these specifications have to be worked out for each supplier in an early phase of the development process, even if there is not yet any implementation of other parts available. Hence, there is a need for having virtual integration of the system, based on uncertain knowledge of extra-functional parameters. Therefore, we conclude to have the following approach for achieving a high quality specification:

1. A component based interface for real-time requirements and resource sharing which aims at an independent development of software functions by suppliers without the need of taking other system parts into account.
2. A design tailoring at system integrator's site that enables the integrator to derive resource constraints assigned to functions under uncertain knowledge of implementation parameters.

Item 2 is a prerequisite to item 1, because in a complete requirement specification for relocatable software the amount of available resources is needed. Consequently, the supplier of components will not need any system knowledge during development but may design his component according to their own rules (scheduling, etc.) on the hardware platform his components are assigned to. Therefore, the real-time interfaces must be generated in a way that the overall feasibility is not affected by integrating components at OEM site.

We propose a methodology containing 4 phases to achieve the goal of design tailoring and interface generation. Starting with a network of functions, we propose to perform the following steps:

1. Architectural design space exploration will assess the various solutions for ECU network topologies and ECU architectures, potentially based on a library of already existing parts of ECUs and network topologies (e.g. a given Flexray

system as backbone). The result is an architectural topology on which executable functions can be deployed.
2. Task synthesis will create executable fragments, called tasks, from the logical description in terms of networks of functions (e.g. a function that describes the behavior of a controller reading sensors and driving actuators, but sensors and actuators are located on different ECUs in a distributed architecture). The results are sets of deployable tasks, which are the elementary design entities that are scheduled on ECUs and buses.
3. Early deployment analysis will deploy the elements of task networks to elements of the architectural view while considering uncertainties and abstractions of different viewpoints (e.g. real-time). The result is a pre-allocation of functional fragments on a distributed architecture that is handed over to the supplier.
4. Time schedule definition will allocate time slots on each architectural element for the set of functional fragments to be executed on it. Based on the estimated loads of fragments within the preliminary deployment and the real-time requirements given in terms of end-to-end deadlines, the time amount each supplier is allowed to consume on each architectural element will be calculated.

In this paper, we assume that the first two items are given. We will address the last two topics in more detail in the next two subsections, assuming as result of step 1 a given architectural topology, and as a result of step 2 a set of possibly dependent tasks.

6.1 Design Tailoring

The aspect of generating resource shares in advance can be reduced to the task of generating deadlines and periodicities. This is in contrast to previous work (cf. Easwaran et al., 2006; Henzinger and Matic, 2006), where the resource assigned to components is formalized as a function of computation capacity over the time domain. Those approaches have a main drawback: During implementation of components it is not possible to use traditional validation techniques, like measuring. This is, because it is not feasible to slow down the hardware in a way that measurements are sufficiently performable regarding a capacity function. However, our approach aims at leaving the traditional implementation process untouched on the one hand and enabling the smooth composition of components developed outside on the other hand. Such an approach is achievable if we are able to translate the resource consumption of other components into real-time requirements for the component under design. Section 6.2 gives a detailed introduction to this topic and its integration in the whole system. Prerequisite for such a technique is the specification of time budgets each component is allowed to consume, i.e. deadlines, which strongly depend on the implementation. Hence, given the component structure there is a need for an estimated design space exploration for assessing the set of feasible solutions, and for predicting the software and hardware structure of the whole system in advance. Obviously, in an early phase

of the development process this will be driven by abstractions of hardware resources and software implementations. Once we have found such an estimated implementation, we can use the result and calculate the approximative load of each component which then is used as driver for budgeting the time intervals of end-to-end deadlines coming from system requirements.

We use the approach presented in Section 2, but instead of calculating accurate results for real-time parameters in pre-allocation only very rough estimations of the load are used (see Metzner, 2005) for a more comprehensive presentation). The optimization function is selected in a way that the slackness of the system is maximized. The outcome of this optimization step is a deployment for which we now are able to calculate the load of each task by using standard schedulability methods (see Section 3). This load is used to derive the needed parameters for resource sharing: deadlines and periods. For periods we assume that they are given by the high-level design (e.g. derived from the closed loop controller analysis for a plant model). Deadlines are given as requirements that are talking about end-to-end delays over chains of tasks, hence, we have to tailor these time intervals into smaller fragments for software parts of the chain that need not be allocated on the same ECU.

The approach of synthesizing deadlines is based on previous work (cf. Di Natale and Stankovic, 1994; Johnson and Shin, 1997), that is enhanced by a load driven metric. The task network we deal with is assumed to be a tree of tasks, which are deployed on a distributed architecture according to the pre-allocation described above. For each path $\pi \in \Pi$ from root tasks to leaf tasks, that is an ordered set of tasks T_π, we assume a given end-to-end deadline d_π. For simplicity, we assume that each task is triggered by an incoming message if it is within a path, or by an external event with some periodicity if it is a root task. In order to generate deadlines for these tasks we perform a load-based sharing of the path's deadline d_π, i.e. the generated deadline d_{τ_i} of a task $\tau_i \in \pi$ is given by $d_{\tau_i} = S^\pi_{\tau_i} \cdot d_\pi$, where $S^\pi_{\tau_i}$ is the load of task τ_i in path π. $S^\pi_{\tau_i}$ is derived by real-time analysis of the complete system using techniques presented in Section 3, i.e. a calculation of the response time of the paths r_π and the response time of each task r_{τ_i} (and each message) in the path. The share of each task on a path π can now be calculated by $S^\pi_{\tau_i} = r_{\tau_i}/r_\pi$. Since task networks are trees, it is possible for one task to occur in more than one path, each with a different end-to-end deadline d_π. In order to not violate shorter end-to-end deadlines by generated task deadlines coming from paths with longer deadlines, we first minimize the deadlines for all tasks that are part of more than one path. The collection of shared task T^{share} is defined by

$$T^{share} = \bigcup_{\pi_i, \pi_j \in \Pi, i \neq j} \pi_i \cap \pi_j \tag{5}$$

For these tasks we assign the minimal share-driven deadline on all paths:

$$\forall \tau_i \in T^{share} : d_{\tau_i} = \min \left\{ S^\pi_{\tau_i} \cdot d_\pi \mid \pi \in \Pi \wedge \tau_i \in \pi \right\} \tag{6}$$

Now we have generated the deadlines of tasks shared across multiple paths, we can generate the deadline of all other tasks:

$$\forall \pi \in \Pi : \forall \tau_i \notin T^{share}\ \tau_i \in \pi \rightarrow d_{\tau_i} = S^{\pi}_{\tau_i} \cdot \left(d_\pi - \sum_{\tau_j \in \pi \cap T^{share}} d_{\tau_j} \right) \quad (7)$$

In general, this approach may lead to non-optimal results due to the fixed pre-allocation. However, it turns out that for substantial benchmarks load-driven deadline generation in all cases is superior to the methods described in Di Natale and Stankovic (1994) and Johnson and Shin (1997) (cf. Metzner, 2005) for a more detailed description).

6.2 Generating Real-Time Interfaces

We now want to adjust the synthetic deadlines for tasks achieved by the pre-allocation towards localized deadlines which can be used by the supplier who is responsible for implementing the task. Localized deadlines means that the supplier may implement the task that guarantees these deadlines without the need of knowing more system related details. We assume – for simplicity reasons – that the OEM has tailored the system design to slots of computations in a TDMA round based scheduling, like provided by OSEKTime. Multiple slots can be assigned to a supplier during one TDMA round, where there is no need for suppliers to know about that fact. Hence, we have to adjust the deadlines in such a way that if all tasks of suppliers hold the deadlines of their tasks, then the whole system is feasible. In this first version of our synthesis approach we restrict the tasks to be implemented by suppliers in a way that they have to be fully preemptive.

Let $s_0 < \ldots < s_{n-1}$ be all slots allocated to supplier S in a TDMA round by the OEM. The TDMA round is a sorted set of n slots s_i, where each slot has a length $\lambda(s_i)$ and a fixed offset O_i. The offset determines the starting point in time of this slot. Without loss of generalization we only talk about slots destined to the task under design. The length of the whole TDMA round is given by Λ. Let the (synthesized) deadline of a task be given by d. We further assume, that the triggering event of the task arrives without any relation to the TDMA round. In order to calculate localized deadlines we now have to consider only those slots the deadline covers during potentially multiple TDMA rounds. We then will shorten d in a way, that the task remains safe with respect to d if it is implemented by fully exploiting the computation power of the foreseen architecture. However, we additionally have to consider the possible worst case scenarios of the arrival time during a TDMA round and take the minimal deadline. Worst case scenarios are those time instants that are coming directly after passing a slot s_i. For time instants between slots we have proved that these instants are not critical (Metzner, 2005). We now analyze for each slots s_i how d can be safely shortened assuming the triggering event of τ occurs directly at the end of s_i.

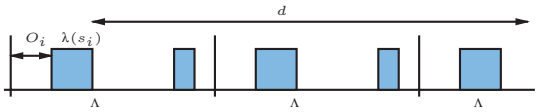


Fig. 3 Worst case example for TDMA based schedule of a set of task

As depicted in Figure 3, we can split the time interval covered by d into three parts. The first part collects all time intervals the TDMA round provides for the task set of the supplier after the beginning of d right after the end of slot s_i.

$$c_{\mathrm{f}}^{i} = \sum_{y=i+1}^{n-1} \lambda(s_y) \tag{8}$$

In the middle part we have to consider all slots destined to the set of tasks for each covered instance of the TDMA round after the first and without the last one, if the last one is partially covered by d:

$$c_{\mathrm{m}}^{i} = X_i \cdot \sum_{y=0}^{n-1} \lambda(s_y) \tag{9}$$

where X_i is the number of covered TDMA rounds without the first and the last one. It can easily be calculated by

$$X_i = \max\left(\left(\left\lfloor \frac{O_i + \lambda(s_i) + d}{\Lambda} \right\rfloor - 1\right), 0\right) \tag{10}$$

In the last covered TDMA round we have to consider all slots that are destined to the set of tasks until we reach the end of the deadline interval. Furthermore, if the deadline ends within a slot s_i we only are allowed to assign those part of the slots that are covered by the deadline interval. Therefore, the computation time for the set of tasks in the last covered TDMA slot is defined by

$$c_{\mathrm{l}}^{i} = \sum_{y=0}^{n-1} \left\lceil \frac{l_i - O_y}{\Lambda} \right\rceil_0 \cdot \left(\lambda(s_y) - \max\left(\left(\lambda(s_y) - \left(l_i - O_y\right)\right), 0\right)\right) \tag{11}$$

where l_i is the part of the deadline interval that covers the last partially covered TDMA round:

$$l_i = O_i + \lambda(s_i) + d - (X_i + 1) \cdot \Lambda \tag{12}$$

In order to get the portion of computation in the last TDMA round which is only partially covered, we have to look at 3 cases: If l_i ends before slot s_y, then $\left\lceil \frac{l_i - O_y}{\Lambda} \right\rceil_0$ becomes 0 and s_y is not added. If l_i ends somewhere in s_y, then $z = l_i - O_y$ calculates

the covered interval of s_y, hence the correction factor to be subtracted from $\lambda(s_y)$ is $\lambda(s_y) - z$. If l_i ends after s_y, then $z = l_i - Oy > \lambda(s_y)$, hence the max-operation forces to subtract 0 from $\lambda(s_y)$ and the computation time of s_y is added completely. Now we are able to calculate the localized deadline for the worst case scenario of an arriving trigger event just behind slot s_i, i.e. the maximal computation time for the set of tasks that is covered by d:

$$d^i = c_{\mathrm{f}}^i + c_{\mathrm{m}}^i + c_{\mathrm{l}}^i \tag{13}$$

Since we have to analyze this considering all possible worst case scenarios, which are all time instants right after each slot that is assigned to serve the supplier's set of task, we finally end up in

$$d_{\mathrm{loc}} = \min\left\{d^i \mid i \in \{0, \ldots, n-1\}\right\} \tag{14}$$

The implementation of the set of tasks can now be developed by the supplier on the foreseen hardware platform with the ability of using tools and methods (e.g. measuring execution times etc.) used in a stand-alone project. On the other hand the capability of seamless composition enables the OEM to integrate the delivered implementation within the intended time slots without any need for additional analysis.

7 Conclusion

In this paper we gave a survey on the techniques for the design of automotive real-time systems. Starting from requirements we have shown a set of methods and techniques that are dedicated to support a seamless development process while considering supplier-oriented industrial processes. Automatic and approximative design space exploration leads to an assessment of feasible solutions. These solution in advance are used by the OEM as key driver for budgeting the time domain for different supplier's components, which enables a component based integration at OEM's site, and an isolated development at supplier's site. Formal specifications in terms of LSCs are used to capture all requirements and provide the capability of proving correctness of the system in a strong mathematical sense. Together with accurate real-time analysis methods of implementations, the technique of cyclic timed automata is used to close the gap between formal requirements implementations. Finally, the application of design space exploration with exact analysis methods enables fine-grained optimizations of the system.

In the end, mastering a seamless, component based development processes, while simultaneously increasing the quality of designs, will be one key technology in upcoming embedded systems development processes, in the AUTOSAR era more than ever. Therefore, our approach can be seen as a first and important step into that direction.

References

N. Audsley, A. Burns, R. Davis, K. Tindell, and A. Wellings. Fixed priority pre-emptive scheduling: An historical perspective. *Real-Time Systems*, 8(2), 1995.

S. Baruah and A. Burns. Sustainable Scheduling Analysis. In *Proceedings of the IEEE Real-Time Systems Symposium*. IEEE Computer Society, 2006.

W. Damm and D. Harl. LSCs: Breathing Life into Message Sequence Charts. *Formal Methods in System Design*, 19, 2001.

W. Damm, E. Böde, A. Metzner, T. Peikenkamp, and A. Votintseva. Boosting Re-use of Embedded Automotive Applications Through Rich Components. *Proceedings Foundations of Interface Technologies*, 2005.

R. Davis and A. Burns. Hierarchical Fixed Priority Pre-emptive Scheduling. In *Proceedings of the IEEE Real-Time Systems Symposium*, 2005.

M. Di Natale and J. Stankovic. Dynamic End-To-End Guarantees in Distributed Real-Time Systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, 1994.

H. Dierks, , A. Metzner, and I. Stierand. Combining Timed Automata based Formal Specifications and Real-Time Scheduling. Technical report, Department of Computer Science, Carl-von-Ossietzky Universität Oldenburg, 2006.

A. Easwaran, I. Lee, O. Sokolsky, and I. Shin. Incremental Schedulability Analysis of Hierarchical Real-Time Components. In *Proceedings of the 6th ACM Conference on Embedded Software*, 2006.

E. Fersman and W. Yi. A generic approach to schedulability analysis of real time tasks. *Nordic Journal of Computing*, 11, 2004.

A. Hamann, M. Jersak, K. Richter, and R. Ernst. Design Space Exploration and System Optimization with SymTA/S. In *Proc. RTSS*, 2004.

H. Heinecke, K.-P. Schnelle, H. Fennel, J. Bortolazzi, L. Lundh, J. Leflour, J.-L. Maté, K. Nishikawa, and T. Scharnhorst. AUTomotive Open System ARchitecture – an industry-wide initiative to manage the complexity of emerging automotive E/E-architectures. In *Proceedings of Convergence 2004, International Congress on Transportation Electronics*, 2004.

M. Hendriks and M. Verhoef. Timed Automata Based Analysis of Embedded System Architectures. In *Proc. IPDPS*, 2006.

T.A. Henzinger and S. Matic. "An Interface Algebra for Real-Time Components". In *Proc. RTAS*, 2006.

J. Jonsson and K. Shin. Deadline Assignment in Distributed Hard Real-Time Systems with Relaxed Locality Constraints. In *Proceedings of the International Conference on Distributed Computing Systems*, 1997.

J. Klose, T. Toben, B. Westphal, and H. Wittke. Check It Out: On the Efficient Formal Verification of Life Sequence Charts. In *Proceedings of the 21st Conference on Computer Aided Verification*, 2006.

C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

G. Madl, S. Abdelwahed, and D.C. Schmidt. Verifying distributed real-time properties of embedded systems via graph transformations and model checking. *Real Time Systems*, 33, 2006.

A. Metzner. *Effizienter Entwurf verteilter eingebetteter Echtzeitsysteme*. PhD thesis, Carl-von-Ossietzky Universität Oldenburg, 2005.

A. Metzner and C. Herde. RTSAT – An Optimal and Efficient Approach to the Task Allocation Problem in Distributed Architectures. In *Proceedings of the IEEE Real-Time Systems Symposium*. IEEE Computer Society, 2006.

A. Metzner, M. Fränzle, C. Herde, and I. Stierand. Scheduling Distributed Real-Time Systems by Satisfiability Checking. In *Proceedings of the Conference on Embedded and Real-Time Computing Systems and Applications*, 2005.

A. Metzner, M. Fränzle, C. Herde, and I. Stierand. An Optimal Approach to the Task Allocation Problem on Hierarchical Architectures. In *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium*, 2006.

J. Palencia and M. Harbour. Schedulability Analysis for Tasks with Static and Dynamic Offsets. In *Proceeding of the 9th IEEE Real-Time Systems Symposium*, 1998.

K. Richter, R. Racu, and R. Ernst. Scheduling Analysis Integration for Heterogeneous Multiprocessor SoC. In *Proc. RTSS*, 2003.

K. Tindell. *Fixed Priority Scheduling of Hard Real-Time Systems*. PhD thesis, University of York, 1994.

Role of Formal Methods in the Automobile Industry

Thomas E. Fuhrman

Abstract The automobile industry presents some unique challenges for the application of formal methods. The automobile industry is rapidly changing from a mechanical industry to one driven by innovation in electronics and embedded software. Many new safety and convenience features are being designed that will have higher degrees of control authority over the motion of the vehicle, leading to increasingly autonomous operation of the vehicle. To achieve robust operation of these new features in the presence of variability in traffic and weather conditions, road conditions, driver skill level, and vehicle state of health, formal methods research is needed in the areas of requirements engineering, model-driven design and model translation, hybrid system modeling and verification, distributed electrical architecture, software architecture, communication protocols (such as LIN, CAN, and FlexRay), fault tolerance, and testing. Several preliminary case studies have been conducted in some of the above areas.

S. Ramesh and P. Sampath (eds.), *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*, 175.

Predicting Failures of and Repairing Inductive Proof Attempts*

Mahadevan Subramaniam, Deepak Kapur, and Stephan Falke

Abstract Inductive reasoning is critical for ensuring reliability of computational descriptions, especially of algorithms defined on recursive data structures. Despite advances made in automating inductive reasoning, proof attempts by theorem provers frequently fail while performing inductive reasoning. A user of such a system must scrutinize a failed proof attempt and do intensive debugging to understand the cause of failure, and then provide additional information to make a failed proof attempt succeed.

A method for predicting a priori failure of proof attempts by induction is proposed. It is based on analyzing the definitions of function symbols appearing in a conjecture. Further, failure analysis is shown to provide information that can be used to make those proof attempts succeed for valid conjectures. The failure of proof attempts could be because of a number of reasons even when a conjecture is believed to be valid. It might be that an induction scheme used in a proof attempt is not powerful enough to yield useful induction hypotheses which can be applied effectively. Or, even when induction hypotheses are applicable, the proof attempt might not succeed because of missing lemmas. A method for speculating intermediate lemmas which can make induction hypotheses applicable and/or lead to simplification obtaining validity is proposed. The analysis can be automated and is illustrated on several examples. A preliminary implementation demonstrates the effectiveness of the proposed approach.

1 Introduction

Induction plays an important role in reasoning about many applications that involve computation with recursive data structures and/or recursive algorithms. Among other things, this includes software, parameterized hardware and protocols, and, more recently, static program analysis. Theorem provers to mechanize induction have been investigated and successfully used in verifying properties of several applications. However, inductive proof attempts in theorem provers fail more often than succeed, thus requiring user intensive debugging of failed proof attempts. Automated

*Partially supported by the NSF award CCF-0541315.

S. Ramesh and P. Sampath (eds.), *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*, 177–191.

methods that aid users in handling failures of inductive proof attempts are of crucial importance.

In this paper, we describe a novel approach for dealing with failed proof attempts:

- it a priori determines the failure of a theorem prover to establish a given conjecture using induction, and
- under the assumption that the conjecture is valid, it can suggest intermediate lemmas which may be proved to complete a proof of the conjecture.

It is shown how a theorem prover can provably predict its own failure even before attempting an inductive proof of the conjecture. Syntactic conditions on the conjecture and on the definitions of the functions appearing in the conjecture are identified to achieve this goal. Based on such an analysis, a prover can provide useful feedback without the users having to resort to the tedious task of analyzing failed proof transcripts.

In this paper, we concentrate on predicting *inapplicability failures*, meaning that the induction hypotheses generated from an induction scheme are inapplicable in a subgoal generated from an induction scheme.

The proposed approach analyzes the definitions of function symbols appearing in a conjecture to check whether one function definition is *blocking* another function definition, causing the inapplicability of an induction hypothesis. In Section 3, this concept is used to define *flawed* induction schemes for the conjecture, using the terminology of Boyer and Moore [2]. It is shown in Section 4 that if a proof of a conjecture is attempted using a flawed induction scheme, then under certain conditions, such a proof attempt is guaranteed to fail.

The framework used for performing inductive reasoning is that of explicit induction using the cover set method as implemented in our theorem prover *RRL* (*Rewrite Rule Laboratory*) [9, 12].

Assuming that the conjecture is valid, the above analyses can also lead to the discovery of potentially useful bridge lemmas for making a proof attempt succeed. These lemmas are formulated to *unblock* function definitions and ensure further simplification. Using our earlier work on discovering lemmas for automating inductive proofs [7], we propose repair strategies for failed proof attempts; this is discussed in Section 5.

Section 6 includes a discussion of a preliminary implementation as well as a table of examples tried using the proposed approach. This is followed by a brief conclusion and ideas for future work.

This work is complementary to our work reported in [6, 8] where conditions on function definitions and conjectures are identified that ensure that the inductive validity of a subclass of conjectures can be decided automatically without any user interaction. *Compatibility* among function definitions (which is anti-thesis of blocking) plays a critical role there, albeit positively, in the sense that if functions appearing in a conjecture have compatible definitions, then under certain conditions, its inductive validity can be decided.

1.1 Two Illustrative Examples

Example 1 Consider proving the conjecture

$$C_1 : \mathsf{len}(\mathsf{rev}(x)) == \mathsf{len}(x)$$

from the following rules defining functions len, append, and rev:

1. $\mathsf{len}(\mathsf{nil}) \rightarrow 0$,
2. $\mathsf{len}(\mathsf{cons}(x, y)) \rightarrow \mathsf{s}(\mathsf{len}(y))$,
3. $\mathsf{append}(\mathsf{nil}, z) \rightarrow z$,
4. $\mathsf{append}(\mathsf{cons}(x, y), z) \rightarrow \mathsf{cons}(x, \mathsf{append}(y, z))$,
5. $\mathsf{rev}(\mathsf{nil}) \rightarrow \mathsf{nil}$,
6. $\mathsf{rev}(\mathsf{cons}(x, y)) \rightarrow \mathsf{append}(\mathsf{rev}(y), \mathsf{cons}(x, \mathsf{nil}))$.

with 0 and s (successor) as the constructors for natural numbers and nil and cons as the constructors for lists.

The conjecture C_1 cannot be decided by equational reasoning from the above rules. An inductive proof of C_1 can be attempted. Using the principle of structural induction for lists, x is instantiated to be nil for the base case; for the induction step case, x is instantiated to be $\mathsf{cons}(u, v)$ with a hypothesis generated by instantiating x to be v.[1]

The base case simplifies to $true$ using rules 5 and 1. Focussing on the step case, the conclusion in the subgoal is $\mathsf{len}(\mathsf{rev}(\mathsf{cons}(u, v))) == \mathsf{len}(\mathsf{cons}(u, v))$, with the induction hypothesis being $\mathsf{len}(\mathsf{rev}(v)) == \mathsf{len}(v)$. The conclusion simplifies using rules 6 and 2 to $\mathsf{len}(\mathsf{append}(\mathsf{rev}(v), \mathsf{cons}(u, \mathsf{nil}))) == \mathsf{s}(\mathsf{len}(v))$.

If the conjecture is viewed as oriented from left to right, then the left side of the induction hypothesis cannot be applied to the left side of the conclusion.

A key result of this paper is that such a failure of an inductive proof attempt can be predicted a priori (without attempting the proof) by analyzing the interaction among the rules defining len and rev. When rev as an argument to len is expanded using the recursive rule 6, the extra function append between len and rev cannot be eliminated using the existing rules. Consequently, the corresponding induction scheme is said to be *flawed*.

It will also be shown how the above analysis can be used to speculate bridge lemmas which can repair such a failed proof attempt. Such bridge lemmas can possibly repair failed proof attempts and lead to proofs for valid conjectures.

Proof attempts of conjectures that are not assumed to be oriented can also fail because neither side of the hypothesis is applicable to the corresponding conclusion side. The next example illustrates such a failure.

Example 2 Consider proving the conjecture

$$C_2 : x * (z * y) == z * (x * y)$$

[1] In the next section, we discuss the cover set induction method [12] for generating induction schemes from the terminating definitions of functions as implemented in our theorem prover *RRL*. Using the terminating definition of rev (or len), the cover set method generates the same induction scheme.

from the following rules defining $+$ and $*$ on natural numbers:

$$1.\ x + 0 \to x, \qquad 2.\ x + \mathsf{s}(y) \to \mathsf{s}(x + y),$$
$$3.\ x * 0 \to 0, \qquad 4.\ x * \mathsf{s}(y) \to x + (x * y).$$

The cover set method identifies y as the only possible induction variable, and the generated induction scheme is identical to the principle of mathematical induction. The base case is $x * (z * 0) == z * (x * 0)$, which simplifies to $true$ using rule 3. For the step case, the conclusion is $x * (z * \mathsf{s}(u)) == z * (x * \mathsf{s}(u))$, with the hypothesis being $x * (z * u) == z * (x * u)$. The conclusion simplifies using rule 4 to $x * (z + (z * u)) == z * (x + (x * u))$, to which the hypothesis (on either side) is not applicable. Hence, the proof attempt fails. This example cannot be handled by the repair strategies proposed in Section 5.

1.2 Related Work

The manual overhead of analyzing failed proof attempts of theorem provers is a well-known problem. Methods to aid users for analyzing failed inductive proof attempts have been investigated earlier, ranging from the development of interactive proof browsers [10], automatically patching faulty conjectures based on proof planning [4, 5], and using rippling techniques [3]. The approach discussed in this paper was first proposed in [11] and is radically different from the above approaches in its attempt to predict guaranteed failures, thereby avoiding wasteful analyses of failed proof attempts. Inspired by [2], flawed induction schemes for conjectures were defined using the concept of definitional blocking in [11]. Two key results of this paper stem from [11], with the difference that function definitions were assumed to be shown terminating using a recursive path ordering and its variants in [11]. In contrast, we only require that function definitions are given using an arbitrary terminating set of rewrite rules in this paper. As a result, we are able to extend the class of conjectures from those considered in [11]. Also, [11] does not consider the speculation of lemmas for repairing failed proof attempts.

2 Generating Induction Schemes

We briefly review the cover set method [12] for automating induction as implemented in our theorem prover *RRL*. We assume familiarity with the concepts of term rewriting [1].

Let $T(F, X)$ denote the set of terms built using a set of function symbols F, partitioned into *constructor* symbols and *defined* symbols, and a set of variables X. Let $t|_p$ stand for the subterm of t at position p, a sequence of positive integers. A term is an f-term if its outermost function symbol is f. A substitution σ is a finite map from variables to terms, written $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$.

Function definitions are given using a finite set $\mathcal{R}$ of ground-convergent, inter-reduced, sufficiently complete, terminating rewrite rules of the form $l \to r$, where l is assumed to be of the form $f(s_1, \ldots, s_k)$ such that none of $s_1, \ldots, s_k$ contains the function symbol f and the right side r contains zero or more recursive calls to f and to other functions. Non-inductive positions of a function f are those positions such that the argument in that position is the same variable in the left side and in all recursive calls to f on the right side. All other position are inductive positions.

In the cover set method [12], induction schemes are automatically generated for a conjecture C from its subterms of the form $f(x_1, \ldots, x_n, t_1, \ldots, t_m)$ where the $x_1, \ldots, x_n$ are distinct variables occurring on the inductive positions of f and which do not appear in any of $t_1, \ldots, t_m$. Let $\mathcal{IndVar}(f(x_1, \ldots, x_n, t_1, \ldots, t_m)) = \{x_1, \ldots, x_n\}$ denote the induction variables of the term $f(x_1, \ldots, x_n, t_1, \ldots, t_m)$. Each such subterm $f(x_1, \ldots, x_n, t_1, \ldots, t_m)$ in a conjecture suggests an induction scheme.

An induction scheme φ is a finite set of pairs $\langle \sigma, \{\vartheta_1, \ldots, \vartheta_n\}\rangle$ of substitutions where for each subgoal the ϑ_i's produce the induction hypotheses and σ produces the conclusion. For a subterm $s = f(x_1, \ldots, x_n, t_1, \ldots, t_m)$ in C, each pair $\langle \sigma, \{\}\rangle$ in φ corresponds to a non-recursive rule $l \to r$ in the definition of f, where σ is the mgu (most general unifier) of l and s. A pair $\langle \sigma, \{\vartheta_1, \ldots, \vartheta_n\}\rangle$ is produced corresponding to each recursive rule $l \to r$ where σ is the mgu of l and s, and each ϑ_i is the mgu of s and the i^{th} recursive call to f in r.

In an inductive proof attempt of C based on an induction scheme φ, a base case $C\sigma$ is generated from each $\langle \sigma, \{\}\rangle$. An induction step case with conclusion $C\sigma$ and n hypotheses $C\vartheta_i$ is generated for each pair $\langle \sigma, \{\vartheta_1, \ldots, \vartheta_n\}\rangle$ in φ. For example, the inductive proof attempt of the conjecture C_1 in Section 1.1 is based on the induction scheme $\varphi = \{\langle x \mapsto \mathsf{nil}, \{\}\rangle, \langle x \mapsto \mathsf{cons}(u, v), \{\{x \mapsto v\}\}\rangle\}$, which is suggested by the subterm $\mathsf{rev}(x)$ and x is the only induction variable. The induction scheme produces a base case by instantiating x with nil in C_1; it produces a step case where the conclusion is obtained by instantiating x with $\mathsf{cons}(u, v)$ in C_1 and the hypothesis is obtained by instantiating x with v.

To ensure that an induction proof attempt actually does get stuck whenever the generated hypotheses cannot be applied, we make some further assumptions about conjectures and induction schemes obtained from function definitions. We assume that the conclusion of an inductive step case cannot be proved without using an inductive hypothesis. We also assume that only one rewrite rule from $\mathcal{R}$ applies to the instantiation of the subterm suggesting the induction scheme in the conclusion, and that that subterm of the conclusion cannot be simplified any further afterwards. Furthermore, we assume that the inductive hypotheses are irreducible. Induction schemes satisfying these properties have been called *well-behaved* induction schemes in [11]; more details about well-behaved schemes and their properties can be found in [11]. Henceforth, all induction schemes are assumed to be well-behaved.

3 Flawed Induction Schemes

We identify conditions on induction schemes that make it likely that an inductive proof attempt based on a scheme satisfying these conditions fails. We define the concept of *flawed* induction scheme, borrowing the terminology from [2]. This is done based on identifying structural conditions on function definitions.

3.1 Blocking

The interaction among function symbols in a conjecture is captured using the concept of *blocking*. Intuitively, a function definition f blocks another function definition g if a term of the form $f(\ldots, g(\ldots), \ldots)$ cannot be simplified using $\mathcal{R}$, which includes rules defining functions as well as any additionally available lemmas about functions.

Definition 1 (Equational Blocking) *The definition of a function* f equationally blocks *the definition of a function* g *as its* k^{th} *argument with respect to the rules* $\mathcal{R}$ *iff the* k^{th} *argument of* l *in all* f*-rules or* f*-lemmas* $l \to r$ *is neither a variable nor a* g*-term.*[2]

As an example, append in Example 1 equationally blocks append as its 1st argument w.r.t. the rules 1–6. Also, len equationally blocks both append and rev as its argument.

By abuse of notation, we will simply say that f blocks g as its k^{th} argument if this holds true for the definitions of f and g in $\mathcal{R}$.

Based on equational blocking, it can be determined if a subterm at a particular position in a term is left unchanged when the term is simplified using $\mathcal{R}$.

Lemma 2 *Let* s *be an* f*-term with* $s|_k = t$*, where* t *is an irreducible* g*-term. If* f *equationally blocks* g *as its* k^{th} *argument w.r.t.* $\mathcal{R}$ *and* $s \to^*_{\mathcal{R}} s'$*, then* s' *is an* f*-term with* $s'|_k = t$*.*

Proof sketch Since t is irreducible, no rewrite rule or lemma applies to the term s at or inside the subterm t. Since f equationally blocks g, no rewrite rule or lemma applies to s at the root position. Hence, the only possible rewrite steps occur inside the remaining arguments to f, i.e., each simplified form of s has the form $f(\ldots, t, \ldots)$. □

The concept of equational blocking can be extended to an arbitrarily long sequence of function symbols.

Definition 3 (Equationally Blocked Sequences) *A sequence of function symbols* $\langle f_1, f_2, \ldots, f_d \rangle$ *is* equationally blocked *on arguments* $\langle k_1, k_2, \ldots, k_{d-1} \rangle$ *if for each* $1 \leq i < d$*,* f_i *equationally blocks* f_{i+1} *as its* k_i^{th} *argument.*

[2] Recall that in an f-rule or an f-lemma $l \to r$, the root of l is f.

Recall that an induction scheme for a conjecture in the cover set method of mechanizing induction is generated from the terminating complete recursive definition of a function symbol appearing in the conjecture. The definition of the function that is used for generating the induction scheme is expanded in an inductive proof attempt of the conjecture. To predict information about the simplified forms of conclusions in induction step cases, we extend equational blocking to rules defining functions that are used to generate induction schemes.

Let H_g denote the set of function symbols h different from g that can be obtained by expanding the definition of g, i.e., $H_g = \{\text{root}(r) \mid l \to r \in \mathcal{R}_g\} - \{g\}$, where $\mathcal{R}_g \subseteq \mathcal{R}$ contains all g-rules.

Definition 4 (Definitional Blocking) *A function f* definitionally blocks *a function g as its k^{th} argument if there is an $h \in H_g$ such that f equationally blocks h as its k^{th} argument.*

For example, append definitionally blocks rev as its 1st argument since it equationally blocks append $\in H_{\mathsf{rev}}$ as its 1st argument. The function len definitionally blocks rev as its argument since it equationally blocks append $\in H_{\mathsf{rev}}$.

3.2 Flawed Schemes

Definitional blocking is the key idea behind predicting failure of an inductive proof attempt based on an induction scheme generated from a terminating recursive definition, as illustrated by Example 1. In conjecture C_1, len has rev as its argument in len(rev(x)), but len definitionally blocks rev. The induction scheme generated from the definition of rev produces a step case where the conclusion contains the symbol append as the argument of len which cannot be simplified using rules 1–6. However, in the induction hypothesis, the argument of len will remain to be rev, the same as in the conjecture. Therefore, the hypothesis cannot be applied due to this difference. This failure can be predicted from the fact that len definitionally blocks rev.

In general, consider a conjecture that has a subterm $s = f(\ldots, g(\ldots), \ldots)$. Suppose that f definitionally blocks g and that we generate an induction scheme using the definition of g. An inductive proof attempt of the conjecture based on such a scheme will produce an induction step case in which the conclusion contains a term of the form $f(\ldots, h(\ldots, g(\ldots), \ldots), \ldots)$ where h belongs to H_g and is equationally blocked by f. However, in the hypotheses of this induction step case, the corresponding argument of f will be a subterm $g(\ldots)$.

In order to guarantee that the hypothesis cannot be applied, we additionally need to ensure that this difference between conclusion and hypothesis cannot be removed by simplifying subterms properly containing s. This can be ensured by requiring that the sequence of function symbols on the path from the root symbol of the term containing s to s in the conjecture is equationally blocked.

Definition 5 (Flawed Induction Scheme) *An induction scheme φ for a conjecture $s == t$ is* flawed *if there is a position $p = p_1 p_2 \ldots p_n$ in s such that $s|_p = g(\ldots)$ suggests the induction scheme φ and*

1. *the immediate superterm of $s|_p$ is an f-term and f definitionally blocks g as its p_n^{th} argument, and*
2. *$\langle f_1, f_2, \ldots, f_n \rangle$ is equationally blocked on arguments $\langle p_1, p_2, \ldots, p_{n-1} \rangle$, where $s|_{p_1 p_2 \ldots p_i} = f_i(\ldots)$ for $1 \leq i \leq n$.*

The induction scheme suggested by the term $\mathsf{rev}(x)$ in Example 1 is flawed in $\mathsf{len}(\mathsf{rev}(x))$ since len definitionally blocks rev. Since $p = 1$, the second condition in Definition 5 is trivially satisfied.

We will also make use of induction schemes that are flawed in both sides of a conjecture $s == t$. This is captured by the following definition.

Definition 6 (Doubly Flawed Scheme) *An induction scheme φ is* doubly-flawed *for the conjecture $s == t$ if it is flawed in s as well as flawed in t.*

From Example 2, the scheme φ suggested by both $z * y$ and $x * y$ is doubly flawed in the conjecture $x * (z * y) == z * (x * y)$ since $*$ definitionally blocks $*$ as its second argument.

4 Predicting Failure of Inductive Proof Attempts

We show below that an inductive proof attempt generated using a flawed induction scheme provably fails under certain conditions. We make distinction below between oriented and general conjectures. In an oriented conjecture, only the left sides of the induction hypotheses can be applied to the left side of the simplified conclusion. In case of a general conjecture, there is flexibility in applying an induction hypothesis, namely either the left side of an induction hypothesis is applied to the left side of the simplified conclusion, or the right side of an induction hypothesis is applied to the right side of the simplified conclusion.

4.1 Failure due to Inapplicability of Induction Hypotheses

Oriented Conjectures Inapplicability failures for oriented conjectures can be predicted as follows.

Theorem 7 (Inapplicability Failure for Oriented Conjectures) *An inductive proof attempt of an oriented conjecture $s == t$ based on an induction scheme φ fails due to an inapplicability failure if (i) φ is flawed in s and suggested by $s|_p$, and (ii) no $x \in \mathcal{IndVar}(s|_p)$ occurs more than once in s (i.e., none of the induction variables in φ occurs more than once in s).*

The second requirement ensures that the induction hypothesis is not applicable in the conclusion of a subgoal generated from the conjecture at any place other than the subterm suggesting the induction scheme. Note that we do not impose any restrictions on how the termination of $\mathcal{R}$ is shown. In contrast, [11] requires that termination can be shown using a recursive path ordering.

Proof sketch Let φ be flawed in s where $s|_p = g(\ldots)$ suggests φ and the immediate superterm $u = f(\ldots, g(\ldots), \ldots)$ of $s|_p$ is an f-term, where f definitionally blocks g as its p_n^{th} argument. Thus, there is an $h \in H_g$ such that f equationally blocks h. Consider an induction case of φ that is generated by a rule whose outermost function symbol on the right side is h. For this induction case, the conclusion contains the term $u' = f(\ldots, h(\ldots, g(\ldots), \ldots), \ldots)$ instead of u after that rule is applied. Since the induction scheme is well-behaved this is the only rule that can be applied. Since f equationally blocks h, every simplified form of u' has the same form by Lemma 2, i.e., h appears between f and g. Thus, none of the inductive hypotheses is applicable to a simplified form of u'. Since every proper superterm of u is equationally blocked, no rewrite rule applies to any proper superterm of u', either. Furthermore, none of the inductive hypotheses is applicable to those terms since they don't contain induction variables. For subterms of s that don't contain $s|_p$ as a subterm, the same argument applies. □

The conjecture $\mathsf{len}(\mathsf{rev}(x)) == \mathsf{rev}(x)$ in Example 1 is an oriented conjecture. As shown above, the scheme φ suggested by the term $\mathsf{rev}(x)$ is flawed in $\mathsf{len}(\mathsf{rev}(x))$. The induction variable x occurs only once in $\mathsf{len}(\mathsf{rev}(x))$. Hence, by Theorem 7, the proof attempt based on the induction scheme φ is predicted to fail, which is indeed the case as illustrated in Example 1.

General Conjectures For an inductive proof attempt of a general conjecture to provably fail, a stronger notion of flawed schemes is needed. Either side of a hypothesis can be applied to and replaced in the corresponding side of a simplified conclusion in a subgoal generated from a general conjecture. This can be taken care by requiring that an induction scheme φ is doubly flawed (i.e., flawed in both sides s and t of the conjecture $s == t$).

Similar to Theorem 7 for oriented conjectures, the following theorem for general conjectures holds true.

Theorem 8 (Inapplicability Failure for General Conjectures) *An inductive proof attempt of a general conjecture $s == t$ based on an induction scheme φ fails due to an inapplicability failure if (i) φ is doubly flawed for the conjecture $s == t$ and suggested by $s|_p$ and $t|_q$, (ii) no $x \in \mathcal{IndVar}(s|_p)$ occurs more than once in s, and (iii) no $x \in \mathcal{IndVar}(t|_q)$ occurs more than once in t.*

Consider the conjecture $x * (z * y) == z * (x * y)$ from Example 2. As shown above, the scheme φ suggested by both $z * y$ and $x * y$ is doubly flawed. The induction variable y occurs only once in both sides of the conjecture. Hence, by Theorem 8, the proof attempt based on the induction scheme φ is predicted to fail, which it does as illustrated in Example 2.

4.2 Simplification Failures

A proof attempt could still fail even if induction hypotheses are applicable to a conclusion in each subgoal. This type of failure of a proof attempt is when simplification of subgoals does not establish a conjecture. Below, due to lack of space, we only consider general conjectures; simplification failures of oriented conjectures can be predicted under similar conditions. The key idea is to identify a subterm in one side of a conclusion of a subgoal that persists even after simplification, whereas that subterm cannot appear in any simplification of the other side. For that, we introduce the notion of t-preserving rules for any given nonvariable term t.

Definition 9 (t-Preserving Rules) *A rule $l \to r$ is* t-preserving *iff any subterm of l (including variables) that matches t also occurs in r.*

Lemma 10 *If t is an irreducible subterm of s, all rules in $\mathcal{R}$ are t-preserving, and $s \to_{\mathcal{R}}^{*} s'$, then s' has t as a subterm.*

Proof sketch It suffices to show the claim for a single reduction $s \to_{\mathcal{R}} s'$ since the claim then follows by induction. Thus, let t be a subterm of s and assume that all rules in $\mathcal{R}$ are t-preserving. Since t is a subterm of s we get $s|_p = t$ for some position p. Let q be the position where the reduction $s \to_{\mathcal{R}} s'$ takes place. If $p \perp q$ or $p < q$, then $s'|_p = t$ as well. Otherwise, $p = qp'$ for some position p' and $s|_q = l\sigma$ for some rule $l \to r$ and some substitution σ. Since $t = s|_p$, we get $t = s|_{qp'} = l\sigma|_{p'}$. If p' is also a position in l, then $l\sigma|_{p'} = l|_{p'}\sigma$, i.e., $l|_{p'}$ matches t. By Definition 9, $l|_{p'}$ is also a subterm of r and $r\sigma$ contains t as a subterm. If p' is not a position in l, then $p' = p_1 p_2$ where $l|_{p_1}$ is a variable. Since $l \to r$ is t-preserving and this variable matches t, it occurs in r as well and $r\sigma$ contains t as a subterm. □

The following theorem then establishes that the simplification failure of a proof attempt can be provably predicted if the rules preserve the side of the hypothesis where the scheme appears flawed.

Theorem 11 *An inductive proof attempt of a general conjecture $s == t$ based on an induction scheme φ fails due to simplification if*

- *φ is suggested by subterms in both s and t, and*
- *φ is flawed only in s, and*
- *the rules applicable to the conjecture are s-preserving.*

5 Possibly Repairing Predicted Failures

This section describes a heuristic for automatically repairing predicted failed proof attempts by discovering bridge lemmas which possibly remove the cause of failure. This is based on our earlier work for automatic lemma generation for inductive proofs

[7]. We first briefly review the key ideas of the heuristic and then discuss how to discover bridge lemmas to possibly fix failures due to inapplicability of induction hypotheses.

The heuristic discussed in [7] assumes that the left side of a proposed lemma is known and the structure of the right side, i.e., subterms appearing in it, can be guessed. The right side is formulated as a template with parameters standing for terms. Then, constraints on the template are generated by attempting an inductive proof of the proposed lemma. For the heuristic to be applicable, it is thus necessary that an inductive proof of the lemma can be attempted, i.e., that there is at least one induction scheme suggested by the lemma that is not flawed. If that is not possible, the proposed lemma can be generalized by abstracting a subterm to be a variable, until an inductive proof can be attempted. Constraints on the template ensure that the inductive proof attempt will be successful if the constraints are satisfied. Based on constraints on the template and available function definitions, terms standing for parameters are speculated. Among possible techniques used for the speculation of these terms are higher-order unification, simple matching constraints on function templates with function definitions, etc.; the reader can consult [7] for details. This will be illustrated below using an example.

5.1 Speculating Bridge Lemmas

As discussed above, inapplicability failure arises because an induction hypothesis cannot be applied to the conclusion generated from a subgoal in a proof attempt. This failure can be repaired by speculating a bridge lemma that applies to the conclusion to produce a result containing an instance of the hypothesis so that the hypothesis indeed becomes applicable.

Consider a conjecture $f(\ldots, g(\ldots), \ldots) == t$, whose proof attempt based on a flawed scheme suggested by $g(\ldots)$ fails due to the inapplicability of the induction hypotheses. The function f definitionally blocks g so the left side of the hypothesis $lh = f(\ldots, g(\ldots), \ldots)$ does not apply to the left side of the conclusion $lc = f(\ldots, h(\ldots, g(\ldots), \ldots), \ldots)$ because f equationally blocks h. The speculated bridge lemma must reduce lc to a term which contains an instance of lh so that the hypothesis is applicable. The left side of the proposed lemma is speculated to be $f(\ldots, h(\ldots, g(\ldots), \ldots), \ldots)$ (or a general version of one of its subterms) since one way to make lh applicable is to remove the equationally blocked function h appearing between f and g. One way to ensure that the result of the proposed lemma has an instance of the induction hypothesis lh is to have the right side of the lemma to contain lh. The right side of the lemma is thus formulated as a template structure $F(lh, G(t_1, \ldots, t_n))$ where F and G are function templates standing for terms to be discovered, and the t_i are all subterms that appear as arguments to f and h in lc.

For the heuristic to be applicable, the proposed lemma must have an unflawed induction scheme. In case all induction schemes generated from the proposed lemma

are flawed, the lemma is generalized by abstracting subterms appearing in them to be variables. Using an unflawed induction scheme, a proof is attempted that generates constraints on F and G. These constraints are then used to match against known function definitions and known lemmas about them, to speculate what F and G should be.

Consider the conjecture C_1 from Section 1.1, which fails due to the inapplicability of the induction hypothesis. A possible bridge lemma is formulated as:

$$L : \mathsf{len}(\mathsf{append}(\mathsf{rev}(x), \mathsf{cons}(x_1, \mathsf{nil}))) == F(\mathsf{len}(\mathsf{rev}(x)), G(x, \mathsf{rev}(x))),$$

where $lc = \mathsf{len}(\mathsf{append}(\mathsf{rev}(x), \mathsf{cons}(x_1, \mathsf{nil})))$, $lh = \mathsf{len}(\mathsf{rev}(x))$, and F and G are meta-variables which must be instantiated by terms.[3]

The scheme suggested by $\mathsf{rev}(x)$ in the conjecture L above is flawed since append and len both definitionally block rev. The above lemma is generalized by uniformly replacing $\mathsf{rev}(x)$ by z in L:

$$L' : \mathsf{len}(\mathsf{append}(z, \mathsf{cons}(x_1, \mathsf{nil}))) == F(\mathsf{len}(z), G(z)),$$

with an unflawed scheme suggested by $\mathsf{append}(z, \mathsf{cons}(x_1, \mathsf{nil}))$ and $\mathsf{len}(z)$.[4]

To generate constraints on the meta-variables F and G, a proof of L' using the unflawed induction scheme suggested by $\mathsf{append}(z, \mathsf{cons}(x_1, \mathsf{nil}))$ is attempted. From the basis case, 1. $\mathsf{s}(\mathsf{0}) = F(\mathsf{0}, G(\mathsf{nil}))$. From the step case after the application of the hypothesis, the second constraint is 2. $\mathsf{s}(F(\mathsf{len}(z), G(z))) = F(\mathsf{s}(\mathsf{len}(z)), G(\mathsf{cons}(z_1, z)))$.

The above two constraints on F and G match the rules defining $+$: $\mathsf{0} + x \rightarrow x$, $\mathsf{s}(x) + y \rightarrow \mathsf{s}(x + y)$. If the template F is instantiated to be $+$, additional constraints on G are derived: $\{G(\mathsf{nil}) = \mathsf{s}(\mathsf{0}), G'(\mathsf{cons}(z_1, z)) = G(z)\}$. Based on these constraints, G can now be speculated by the constant function $\mathsf{s}(\mathsf{0})$. With F, G instantiated, we get

$$Lemma : \mathsf{len}(\mathsf{append}(z, \mathsf{cons}(x_1, \mathsf{nil}))) == \mathsf{len}(z) + \mathsf{s}(\mathsf{0}).$$

The proof of this bridge lemma follows from the reasoning employed, which can be verified. This in turn generates another lemma $\mathsf{len}(z) + \mathsf{s}(\mathsf{0}) == \mathsf{s}(\mathsf{len}(z))$, which can be proved after generalizing it to $y + \mathsf{s}(\mathsf{0}) = \mathsf{s}(y)$.

The failed proof attempt of C_1 can thus be repaired and C_1 is an inductive theorem.

If $+$ is defined by recursing on the second argument, then matching of constraints on F, G to rules defining $+$ will work by speculating the lemma by switching the arguments of F.

Several techniques can be employed to speculate instantiations for templates using constraints on them; in the example above we showed how constraints can be used to match against function definitions. Higher-order unification can be used instead. Narrowing techniques may be useful as well; see [7] for details.

---

[3] Subterms $\mathsf{cons}(x_1, \mathsf{nil})$, x_1, and nil do not have to be included as arguments to G since $\mathsf{cons}(x_1, \mathsf{nil})$ appears in C_1 only as the second argument to append, which does not change in a proof attempt. However, including them as arguments to G does not change the derivation.

[4] The variable x is removed from the argument of G in L' since it appears nowhere else in L'.

The above strategy can in general output multiple lemmas if there are multiple ways to instantiate F and G. If lemmas thus generated are found not to be valid, alternative proof attempts based on other unflawed schemes in the proposed bridge lemma (or its generalization) are performed.

In the above discussion, we started with a conjecture with $f(\cdots, g(\cdots), \cdots)$ as its left side where f definitionally blocks g. The repair strategy is easily adapted to repair inapplicability failures of conjectures with a more complex left side containing a sequence of equationally blocked symbols. In that case, many bridge lemmas may have to be speculated to remove blocking in order to deal with the failure due to the inapplicability of the induction hypotheses.

6 Implementation

The proposed approach has been implemented in the OCaml language. This preliminary implementation has been successfully tested to predict a priori failures of proof attempts for several conjectures over lists and numbers, due to inapplicability and simplification failures; some of these are listed below. To evaluate the effectiveness of the proposed approach, we also attempted to prove these conjectures from function definitions using the inductive theorem prover *RRL* [9]. The results are given in the table below. In the "Type" column, an "o" denotes an oriented conjecture, and a "g" denotes a general (unoriented) conjecture. The "Repair" column indicates whether the heuristic in Section 5 is successful in finding a lemma. The "n" column specifies the number of inductions performed by *RRL* in a proof attempt. If the number is less than 100, then the proof attempt succeeded using "n" inductions; if the reported number is 100, this implies that *RRL* did not succeed in finding a proof even after 100 induction attempts. In the "Remarks" column, "d" denotes divergence of *RRL*, while "d w/o g" denotes divergence of *RRL* if its generalization heuristic is not employed.

Conjecture	Type	Repair	*RRL*	n	Remarks
$\mathsf{len}(\mathsf{rev}(x)) == \mathsf{len}(x)$	o	Yes	+	6	d w/o g
$\mathsf{even}((x + x) * y) == \mathsf{true}$	o	No	−	100	d
$\mathsf{rev}(\mathsf{append}(x, y)) == \mathsf{append}(\mathsf{rev}(y), \mathsf{rev}(x))$	g	Yes	+	4	d w/o g
$x * (z * y) == z * (x * y)$	g	No	−	100	d
$x * (y * z) == (x * y) * z$	g	Yes	−	100	d

For all of the examples, the proposed approach can predict failure without attempting a proof. *RRL* will not succeed on any of these examples (even if up to 100 inductions are allowed) unless multiple inductions are attempted combined with the generalization heuristic for aggressively speculating lemmas, in

which case the conjectures $\mathsf{len}(\mathsf{rev}(x)) == \mathsf{len}(x)$ and $\mathsf{rev}(\mathsf{append}(x, y)) == \mathsf{append}(\mathsf{rev}(y), \mathsf{rev}(x))$ are eventually proved by *RRL*. The generalization heuristic generalizes common subterms by variables. For both of these conjectures, the lemmas generated by *RRL* using the generalization heuristic are similar to the lemmas generated by the repair heuristic discussed earlier.

7 Concluding Remarks and Future Work

An automated approach for a priori predicting provable failures of mechanized inductive proof attempts of a large class of conjectures is proposed. Syntactic conditions capturing the interaction among the definitions of function symbols appearing in a conjecture are identified in order to predict two kinds of failures: (i) the inapplicability of induction hypotheses and (ii) inability to simplify subgoals to valid formulas in case induction hypotheses are applicable. The notions of equational blocking and definitional blocking among function definitions are introduced to characterize flawed induction schemes suggested by function definitions. Failures of proof attempts for conjectures are provably predicted if a function symbol f whose definition leads to a flawed induction scheme, appear in contexts that definitionally block f.

For valid conjectures, the analysis for predicting failure can suggest bridge lemmas which, if proved, can lead to a successful proof. The approach has been implemented and effectively tried on several examples.

We believe that the proposed approach can also aid the user in case a conjecture does not follow from the definitions. The above analysis can help in reformulating the conjecture and/or the definitions to make a modified proof attempt which will not provably fail. In case a proposed conjecture is false, it can also help in fixing it. This will be investigated in future work.

Heuristics to support repair strategies for speculating lemmas from constraints on templates need to be investigated. The proposed approach predicts failures for proof attempts with a single application of induction. It will be interesting to extend this approach to multiple applications of induction.

This failure analysis is complementary to our work on identifying a subclass of conjectures about function definitions that can automatically be decided [6, 8]. If a conjecture satisfies syntactic conditions given in [6, 8] and the definitions of the function symbols occurring in the conjecture satisfy the stated structural conditions in [6, 8], then it falls in the decidable class and and we are guaranteed to get a yes or no answer by the theorem prover about its validity. If the conjecture is not in the decidable class, we can then check whether its proof can be predicted to fail. If yes, it can be further analyzed whether the failed proof attempt can be repaired, in which case lemmas are generated and proved before the conjecture is attempted. If the failed proof attempt cannot be repaired, then the conjecture can perhaps be reformulated or function definitions need to be reformulated or some lemmas need to be proved before attempting the conjecture. If it cannot be predicted that a proof

attempt will fail, then a proof can be attempted with no guarantee about whether it will fail or succeed. Or alternatively, it can be declared that nothing can be said about the conjecture.

References

1. F. Baader and T. Nipkow, *Term Rewriting and All That.* Cambridge University Press, 1998.
2. R. S. Boyer and J. S. Moore, *A Computational Logic.* ACM Monographs in Computer Science, 1979.
3. A. Bundy, The Automation of Proof by Mathematical Induction, *Handbook of Automated Reasoning*, 2001.
4. A. Bundy, Planning and Patching Proof, *Proc. AISC '04*, LNCS 3249, 26–37, 2004.
5. A. Ireland, Productive Use of Failure in Inductive Proof, *Journal of Automated Reasoning*, 16(1–2):79–111, 1996.
6. D. Kapur, J. Giesl, and M. Subramaniam, Induction and Decision Procedures, *Rev. R. Acad. Cien. Serie A: Mat.*, 2004.
7. D. Kapur and M. Subramaniam, Lemma Discovery in Automated Induction, *Proc. CADE '96*, LNCS 1104, 538–552, 1996.
8. D. Kapur and M. Subramaniam, Extending Decision Procedures with Induction Schemes, *Proc. CADE '00*, LNAI 1831, 324–345, 2000.
9. D. Kapur and H. Zhang, An Overview of Rewrite Rule Laboratory (RRL), *Proc. RTA '89*, LNCS 355, 559–563, 1989.
10. M. Kaufmann, An Extension of the Boyer–Moore Theorem prover to Support First-Order Quantification, *Journal of Automated Reasoning*, 9(3):355–372, 1992.
11. M. Subramaniam, *Failure Analyses in Inductive Theorem Provers*, Ph.D. Thesis, Department of Computer Science, University of Albany, New York, 1997.
12. H. Zhang, D. Kapur, and M. S. Krishnamoorthy, A Mechanizable Induction Principle for Equational Specifications, *Proc. CADE '88*, LNCS 310, 162–181, 1988.

Can Semi-Formal be Made More Formal?

Ansuman Banerjee, Pallab Dasgupta, and Partha P. Chakrabarti

Abstract Capacity limitations continue to impede widespread adoption of formal property verification in the design validation flow of software and hardware systems. The more popular choice (at least in the hardware domain) has been dynamic property verification (DPV), which is a semi-formal approach where the formal properties are checked over simulation runs. DPV is highly scalable and can support a significantly richer specification language as compared to languages supported by formal property verification tools. Though the main limitations of DPV revolve around its dependence on the coverage of the relevant scenarios by simulation, there appears to be ample scope of addressing these issues through new formal methods for coverage and consistency analysis. We survey some of the formal methods developed by us in this direction, that can aid DPV in becoming more effective in practice and more formal in nature. We also present a new platform for performing DPV over state-event based software systems.

Keywords: Dynamic property verification, assertion.

1 Introduction

In recent years, the number of Electronic Control Units (ECU) built into automobiles has increased dramatically, spanning all segments from power train, body electronics, active and passive safety, as well as navigation and infotainment [4]. Communicating over different bus systems (e.g. CAN, LIN, Bluetooth), these ECUs execute a multitude of functions to improve safety, reduce emission and fuel consumption or improve comfort and driver information. Such systems include safety-critical systems like a brake control or a dynamic drive control. A high-end car today contains 50 embedded electronic control units (or even more) [3].

The growing demand for feature-rich automotive systems and stringent time to market constraints has forced the manufacturing companies to adopt a distributed design methodology to facilitate component reuse at all levels of abstraction: increasingly one automotive function is shared across multiple subsystems consisting of multiple ECUs (a subsystem typically consists of one or more ECUs connected by a bus) and it requires the correct and timely interaction of these multiple distributed

S. Ramesh and P. Sampath (eds.), *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*, 193–211.

subsystems for achieving the system level performance. Such distributed hard real-time systems contrast dramatically in complexity with the typically single ECU based technologies. In addition, there has been a paradigm shift in the development model as well. In the past, electronic system companies used to maintain full control of the product development cycle from product definition to product manufacturing. This is not the case today. Most of the ECU development task is outsourced to different suppliers. The definition of the detailed system specifications, the development and assembly of the components, and the manufacturing of the final product are tasks performed more frequently by multiple suppliers (in contrast to the traditional scenario where one supplier used to provide a complete solution).

The challenge for the design of such distributed and networked control units is to define all requirements and constraints (at the subsystem level, for each individual ECU and at the system level), to understand and analyze the manifold interactions between the many control units, the car and the environment (road, weather, etc.) in normal as well as stress situations (crash), within a development process which is concurrent and distributed between the automobile manufacturers and suppliers. In such a heterogeneous development model, two of the most important tasks are as follows:

- The task of architecture analysis: This typically involves creating the subsystem functionalities from the system level functionality and mapping a set of subsystem functionalities into individual ECUs. This is of utmost importance since the automobile manufacturer needs to correctly and adequately specify the requirements of the ECU to be developed by a supplier. Companies need a specification model for the basis of contracts with suppliers.
- The task of integration verification: This involves the validation task to ascertain that the set of sub-functions being realized by individual subsystems, each consisting of multiple ECUs designed by different manufacturers, satisfies the system level requirement. This is of significant importance at the *hand-off* points of the design chain.

To address the first issue, the manufacturing industry is increasingly moving towards a *model-based* development process, in which the largely textual way of requirement capturing is replaced by executable specification models at different levels of abstraction. This allows creation of abstract models which are iteratively refined to arrive at correct and adequate specifications to be passed on to the supplier. The second and more important issue mandates the need for a platform of *system integration* that can validate the distributed realization of different automobile functions. This calls for a *sound and complete compositional validation* methodology that can validate/infer the functionalities of the integrated system, from the functionalities of the individual subsystems.

In the last few decades, formal property verification has established itself as an effective validation methodology, both in the hardware and software verification community for its ability of automatic and exhaustive reasoning. Researchers have

analyzed several historically significant failures and have shown that the use of formal verification could have detected the bug in the design. Verification practitioners have also been able to uncover flaws in the specifications of complex protocols and intricate bugs in live designs.

Unfortunately, the exponential increase in complexity and the increasingly distributed nature of functions renders the use of formal verification infeasible because of its inherent capacity bottleneck in handling large systems. A typical automotive system today consists of a number of subsystems, each responsible for executing a multitude of tasks. Each subsystem, in turn, consists of a number of ECUs. Each ECU, in turn, has an interface to one or two CAN-bus systems, as well as a number of sensors and actuators directly connected to the ECU (to sense brake pressure, acceleration, speed, RPM, etc.). Specification models of such systems would have typically some 10 to 30 sub-functions, each with about 20 to 100 states [4]. To be able to perform a validation of the integrated system, formal tools typically run into capacity problems. Purely formal methodologies, might however, be feasible at the subsystem level, particularly because of its philosophy of performing an exhaustive validation.

Reasoning only about the correctness of the subsystems in isolation is not sufficient, since most of the system level requirements needed to ensure safe and smooth execution of the automobile will typically span multiple functions with different responsible subsystems. As an example, consider the following requirement: *upon detecting a crash, the power steering mode will be deactivated and the side airbag will be unfolded within 5 ms*. Such a requirement cannot be validated considering the power steering unit or the airbag control unit in isolation, since it demands correct and timely interaction of both when a crash is detected. Validation of such system level requirements which involve multiple subsystems is therefore, non-trivial.

In the last decade, the more popular validation methodology (at least in the hardware domain) has been dynamic property verification (DPV). DPV is a semi-formal approach where the formal properties are checked over simulation runs. DPV is highly scalable and can support a significantly richer specification language as compared to languages supported by formal verification tools.

The success of DPV depends largely on the ability of the simulation test generator to create good testcases for validating the requirements. Classical approaches to testing are guided by the test engineer's intuition and experience in designing *truly effective* testcases which have a high likelihood in exposing errors. Typically, each ECU development effort involves a testing team who, on the basis of textual requirement documents, is responsible for designing testcases for *acceptance testing* (for subsystems delivered by the suppliers) and integration testing. Ideally, the set of testcases should cover all functionality, address boundary conditions, and cover all scenarios. The sheer astronomical complexity of today's feature-rich automotive systems renders the task of generating testcases notoriously difficult, even for the most qualified test engineer. Even random test generators are ineffective for this purpose. It is quite clear that the way out of this impasse cannot be achieved by purely formal or

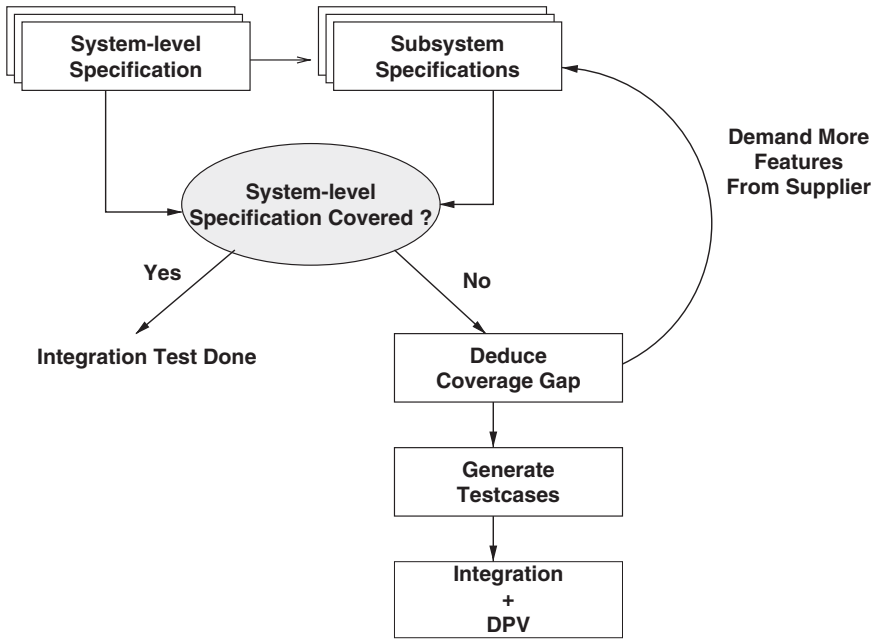


Fig. 1 The overall flow

simulation-based techniques. The challenge, therefore, is to architect a semi-formal reasoning framework, that is truly scalable and effective.

In this paper, we survey some of the formal methods developed by us in this direction that can aid DPV in becoming more effective in practice and more formal in nature. The overall architecture of the proposed flow is shown in Figure 1. To understand the philosophy behind the proposed flow, let us look at the design process adopted by an automobile manufacturer. In the first phase, the functional requirements of the system are identified. To facilitate component reuse, the overall system functionality is decomposed into a set of subsystem functionalities. The functionality of the subsystems and their interconnections defines the system architecture. The subsystem functionalities are in turn mapped to ECUs, many of which are available as *off-the-shelf* components (having an advertised specification). In the second phase, the manufacturer typically selects a set of ECUs (on the basis of the supplier promised specifications) and conducts an *acceptance test* followed by a *virtual integration* test. The acceptance test checks whether the ECU is really meeting the advertised requirements. The virtual integration test is performed to check whether the set of subsystems (each having multiple ECUs) and the way they are connected actually realizes the system requirement. The challenge lies in performing this validation task before creating the integrated design, so that the manufacturer does not waste effort on a flawed architecture. This is where the proposed methodology adds value. The proposed method of integration test is based on compositional reasoning, which attempts

to derive a system level property from the properties of its components. The basic steps are:

1. Determine whether the set of subsystem specifications guarantee the set of system-level specifications. If the answer is yes, integration test is successful, and the actual functional integration of the subsystems may be started.
2. If the answer to the above question is no, we propose to derive a set of *coverage gaps* which demonstrate system-level functionalities that cannot be implied from the subsystem guarantees. This has two major ramifications:
   - If we are able to characterize the coverage gap at the subsystem level, the manufacturer may demand more functionality from the respective supplier. This may continue in an iterative process till the coverage is achieved, provided that the supplier is willing to add-on/provide additional guarantees.
   - If we are able to characterize the coverage gap at the system level, the manufacturer can derive a set of scenarios that can exercise and verify these gaps during actual DPV of the integrated system. (Automatically derived test scenarios can be used for hardware-in-the-loop or software-in-the-loop testing. This addresses the key case of acceptance testing such as checking whether an ECU delivered by the supplier conforms to a specification model (an ECU will only be accepted if it reacts to all stimuli correctly).
   - Perform DPV of integrated system with existing test plan augmented with the set of testcases generated from coverage gaps.

In order to address the first two issues, we present here a methodology for comparing specifications to address coverage of one specification by another (the system specs and the subsystem specs). The methodology has been developed over Linear Temporal Logic (LTL) [8]. The specification coverage methodology determines whether the set of subsystem specifications cover the system level specification, and if not, produces a coverage gap, both in terms of the system level uncovered functionality or the desired guarantee required at the subsystem level that can close the coverage gap. In addition, we present a methodology for automatic testcase generation from LTL specifications. These testcases are used during DPV. We also present here a new platform for performing DPV over state-event based software systems. While the theory behind these have been presented in our earlier papers [1, 2], in this paper, we attempt to fit these methods into an integrated flow in the context of automotive control system design.

2 Comparing Specifications for Analyzing Coverage

In this section, we describe the notion of specification coverage, where we compare two formal specifications and determine the coverage of one by the other. We shall refer to one specification as the *target specification*, $\mathcal{T}$, and the other as the

achieved specification, $\mathcal{A}$. In our context, the ECU specification is the target specification and the subsystem specifications from the different suppliers constitute the achieved specification. Our aim is to determine whether the *achieved specification* covers the *target specification*. If the answer is negative, then we will further aim to determine the *coverage gap* or difference, $\mathcal{T} - \mathcal{A}$, between the specifications, and represent it in a way that is both legible and syntactically comparable with both $\mathcal{T}$ and $\mathcal{A}$. The problem is non-trivial, since both $\mathcal{T}$ and $\mathcal{A}$ may contain several temporal properties.

The problem of determining the coverage of the system level properties by the collective properties of the individual subsystems is computationally less expensive than formally verifying the system level properties on the integrated system. This is because the specification coverage problem does not involve the implementation (whose size is the main bottleneck in formal verification). In the automotive industry today, each subsystem typically comes with an SSTS (Sub System Technical Specification) stating the properties that they guarantee, and the proposed idea of specification matching can formally determine whether a given subsystem achieves the desired intent within a larger system.

Formally, the inputs to the specification coverage framework are:

1. The *target specification* $\mathcal{T}$ as a set of LTL properties.
2. The *achieved specification* $\mathcal{A}$ as another set of LTL properties.

Definition 1 (Coverage Definition) *The achieved specification covers the target specification iff there exists no run that refutes one or more properties of the target specification but does not refute any property of the achieved specification.* □

Our coverage problem is as follows:

- to determine whether the achieved specification covers the target specification; and
- if the answer to the previous question is *no*, determine a set of additional temporal properties that represent the coverage gap (these properties together with the achieved specification succeed in covering the target specification).

In the proposed methodology, to check for specification coverage, we check if every behavior that is inadmissible with respect to $\mathcal{T}$ is also inadmissible with respect to $\mathcal{A}$. The reverse need not be necessarily true, since additional design constraints imposed at the subsystem level may restrict some of the behaviors that are permitted by the integrated specification. For example, the target specification may require that *upon detecting a crash, the side airbag will be eventually unfolded* – in a specific airbag control subsystem implementation, we may guarantee that the airbag will be unfolded within 5 ms. Therefore, the target specs admit those behaviors where the airbag unfolds after 5 ms, but the subsystem specification restricts such behaviors. Behaviors where the airbag never unfolds are inadmissible with respect to the target specs, and therefore, inadmissible with respect to the achieved specs.

The following theorem shows us a way to answer the coverage question.

Theorem 2 (Reproduced from [2]) *The achieved specification, $\mathcal{A}$, covers the target specification $\mathcal{T}$, iff the temporal property $\mathcal{A} \Rightarrow \mathcal{T}$ is valid.* □

The theorem shows that the primary coverage question can be answered by testing the validity of $\mathcal{A} \Rightarrow \mathcal{T}$. Most model checking tools for LTL and its derivatives already have the capability of performing validity (or satisfiability) checks on temporal specifications, and can therefore be used to answer the primary coverage question. Note that the complexity of LTL model checking coincides with that of checking the satisfiability of LTL formulas (both being PSPACE-complete), but since the coverage question does not involve the system implementation, the proposed approach scales to much larger applications.

We shall use the following example to demonstrate the formal methods for specification coverage.

Example 3 This example has been adapted partially from the functional description [3] of the brake management system developed at BMW. Following are some of the system-level requirements:

- If the driver presses the disable hold function, while the hold function is active, the hold function must be deactivated in the next cycle. The *hold* function, when active, is responsible for deactivating any movement of the vehicle.
- If the driver pushes the brake and the clutch pedals and the vehicle stops, then the hold function remains active till the brake is released.
- When the brake is released and the accelerator is pressed, the gears must change and the car must no longer be in neutral in the next cycle.

The system-level requirements (target-spec) may be expressed in LTL as follows:

$$T_1: \quad G\,((\,hold \wedge disable\,) \Rightarrow X\,\neg hold)$$
$$T_2: \quad G\,((\,brake \wedge clutch \wedge stop\,) \Rightarrow X(\,hold\ U\ \neg brake\,)\,)$$
$$T_3: \quad G\,((\,\neg brake \wedge acc\,) \Rightarrow X\,\neg neutral\,)$$

The top-level activity chart of the STATEMATE model [3] of the brake management system consists of 14 charts, each having its own functionality. These correspond to the specifications of the different subsystems in terms of which the design was planned. There are some information flows to and from other subsystems, e.g. the sensor values come from the DME (Digital Motor Electronics), lamps in the cockpit, etc. As a simple example, we consider the specifications of the hold management subsystem and the automatic gear control subsystem. The inputs and outputs of these are shown in Figure 2. To sense the status of the car (brake pressed or not, car stopped or not, known from RPM, etc.), there are some sensory inputs. We describe the functionalities of the subsystems below.

- **Hold Management Unit:** If the disable input is high, while the hold function is active, the hold function is deactivated in the next cycle.
- **Hold Management Unit:** If the stop, clutch and brake inputs are high, the hold function is activated in the next cycle.

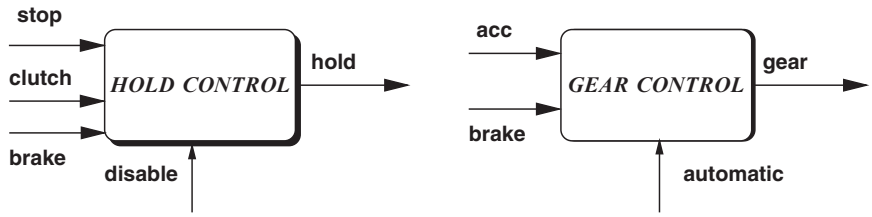


Fig. 2 The hold management and automatic gear subsystems

- **Automatic Gear Management Unit:** If the brake is released while the automatic gear is on, and the accelerator is pressed, the car must not be in neutral in the next cycle.

The above (achieved spec) may be expressed in LTL as follows:

$$A_1: \quad G\,((\,hold \wedge disable\,) \Rightarrow X\,\neg hold)$$
$$A_2: \quad G\,((\,brake \wedge clutch \wedge stop\,) \Rightarrow X\,hold\,)$$
$$A_3: \quad G\,((\,\neg brake \wedge automatic \wedge acc\,) \Rightarrow X\,\neg neutral\,)$$

The primary coverage problem is to determine whether $(A_1 \wedge A_2 \wedge A_3) \Rightarrow (T_1 \wedge T_2 \wedge T_3)$ is valid. In this case the answer is negative. It is clear that T_1 is implied, but we can see that neither T_2 nor T_3 is covered by the properties in the achieved specification.

For example, whenever we have a scenario where the brake is released, the automatic gear is off and accelerator is pressed, the target specification requires gears to change from neutral, but the achieved specification does not have this requirement. This shows that T_3 is not covered.

Consider those scenarios where brake, clutch and stop are active together, but clutch de-asserts before brake. In these scenarios, the target specification requires hold to remain high as long as brake remains pressed (T_2), but the achieved specification does not guarantee this. □

2.1 Where is the Coverage Gap?

Theorem 2 shows that the primary coverage question, that is, whether the achieved specification, $\mathcal{A}$, covers the target specification, $\mathcal{T}$, can be answered by checking the validity of the implication $\mathcal{A} \Rightarrow \mathcal{T}$. If the implication is not valid, then we know that there is a gap between the achieved specification and the target specification, but *how do we find out the gap?*

As we shall show in this section, it is not hard to compute the coverage gap between two temporal specifications and specify a property that theoretically represents the

coverage gap. The main challenge is in presenting the new property in a form that is syntactically similar and visually comparable with the original specification, so that the validation engineer is able to visually examine the new property and realize the set of architectural behaviors that have not been covered by the achieved specification. Let us first see how the coverage gap can be computed.

Example 4 Let us consider the coverage of the property, T_3 of Example 3 by the achieved specification. We have already established that T_3 is not covered. However this information does not accurately point out the coverage gap between T_3 and the achieved specification. Specifically the coverage gap lies only for those scenarios where the brake is released and the automatic gear is off at some point of time. In other words, the coverage gap can be accurately represented by the following property that considers exactly the above scenarios:

$$U_1: \quad G(\,(\,\neg brake \,\wedge\, \neg automatic \,\wedge\, acc\,)\;\Rightarrow\; X\,\neg neutral\,)$$

We have $A_3 \wedge U_1 \;\Rightarrow\; T_3$, and therefore U_1 closes the coverage gap between A and T_3. In general, the aim is to determine the *weakest* set of temporal properties that close the coverage gap between the achieved and target specifications. This is formally expressed below. □

Definition 5 (Strong and Weak Properties) *A property $\mathcal{F}_1$ is stronger than a property $\mathcal{F}_2$ iff $\mathcal{F}_1 \Rightarrow \mathcal{F}_2$ and $\mathcal{F}_2 \not\Rightarrow \mathcal{F}_1$. We also say that $\mathcal{F}_2$ is weaker than $\mathcal{F}_1$.* □

Definition 6 (Coverage Gap in Achieved Spec) *A coverage gap in the achieved specification is a property A_H such that $(\mathcal{A} \wedge A_H) \;\Rightarrow\; \mathcal{T}$, and there exists no property, A'_H such that A'_H is weaker. In other words, we find the weakest property that closes the coverage gap. Adding the weakest property strengthens the achieved specification in a minimal way.* □

Theorem 7 (Reproduced from [2]) *The coverage gap in the achieved specification is unique and is given by $\mathcal{T} \vee \neg\mathcal{A}$.* □

There is an intuitive explanation of the coverage gap as defined by Theorem 7. The goal of the coverage analysis is to find those behaviors that refute $\mathcal{T}$ but satisfy $\mathcal{A}$, that is, those behaviors that satisfy:

$$\varphi \;=\; \neg\mathcal{T} \;\wedge\; \mathcal{A}$$

The property representing the coverage hole must reject exactly these behaviors, hence the property is $\mathcal{T} \vee \neg\mathcal{A}$ which is $\neg\varphi$.
To demonstrate the part of the target specification that is not covered by the achieved specification, we need a further level of abstraction. The definition of the *uncovered target specification* is as follows.

Definition 8 (Uncovered Target Specification) *The uncovered target specification is a property $\mathcal{T}_H$ such that $(\mathcal{A} \wedge \mathcal{T}_H) \Rightarrow \mathcal{T}$ is valid, and there exists no property $\mathcal{T}'_H$ such that $\mathcal{T}'_H$ is weaker than $\mathcal{T}_H$ and $(\mathcal{A} \wedge \mathcal{T}'_H) \Rightarrow \mathcal{T}$ is valid.* □

2.2 How should we Present the Coverage Hole?

Theorem 7 gives us a formalism for computing the coverage gap, but does not convey the missing properties in a meaningful way. Our aim is to present the coverage gap and the uncovered target specification before the manufacturer in a form that is syntactically close to the target specification and is thereby amenable to visual comparison with the target specification. The expressibility of the logic used for specification does not always permit a succinct representation of the coverage gap. In such cases, we prefer to present the coverage gap as a succinct set of properties that closes the coverage gap, but may be marginally stronger than the coverage gap.

This methodology of representation of coverage gap is based on two key algorithms. The first algorithm enables the computation of the bounded terms in the coverage gap and then pushes these terms into the syntactic structure of the target properties to obtain the uncovered part. The second algorithm takes target properties having unbounded temporal operators (such as G, F and U) and systematically weakens them into structure preserving decompositions. It then checks the components that remain to be covered. Our intuitive idea in this step is to systematically weaken the intermediate uncovered target specification, $\mathcal{T}_{\mathcal{U}}$, while ensuring that it still closes the coverage hole. Our methods are based on the observation that a property can be weakened by appropriately weakening or strengthening a variable occurrence. The details of these algorithms is left out of this paper, but can be found in [2]. The following example illustrates the working of the proposed coverage deduction methodology.

Example 9 Let us return to the specifications shown in Example 3 and let us consider the coverage of T_3 by A_3:

T_3: $G\ ((\ \neg brake\ \wedge\ acc\ )\ \Rightarrow\ X\ \neg neutral\ )$
A_3: $G\ ((\ \neg brake\ \wedge\ automatic\ \wedge\ acc\ )\ \Rightarrow\ X\ \neg neutral\ )$

In the first step, we compute ψ as $((\neg brake \wedge automatic \wedge acc\ )\ \Rightarrow\ X\ \neg neutral\ )$. The set of terms, obtained by unfolding $T_3 \vee \neg G\psi$ are as follows:

$$\mathcal{U}_M = \{brake, \neg acc, X(\neg neutral),\ \neg brake \wedge automatic \wedge acc \wedge \neg X(\neg neutral)\ \}$$

We now distribute the terms in $\mathcal{U}_M$ into T_3 past the G operator (as shown in Figure 3). The figure shows how the terms in $\mathcal{U}_M$ distribute across the different operators in T_3. The value of ϑ shown besides the nodes indicate whether the incoming terms are in conjunction or disjunction with the property rooted at that node ($\vartheta = 0$ means disjunction). The uncovered part of T_3 after applying this is given by: $G\ (\ \neg brake\ \wedge\ \neg\ automatic\ \wedge acc\ \Rightarrow\ X\ \neg\ neutral\ )$. □

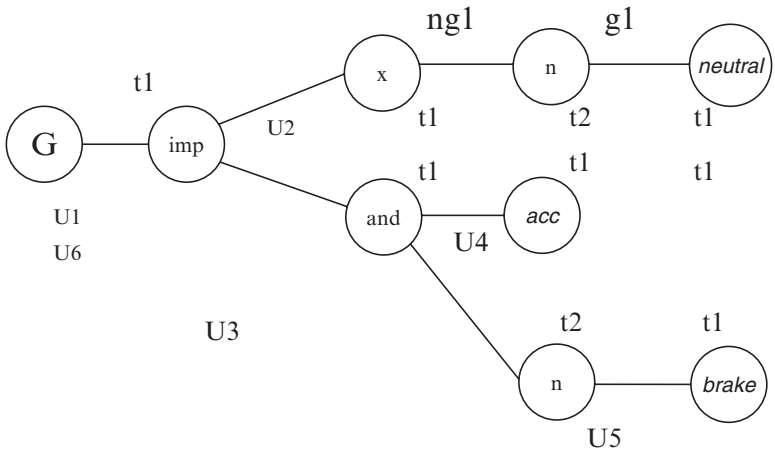


Fig. 3 Term distribution

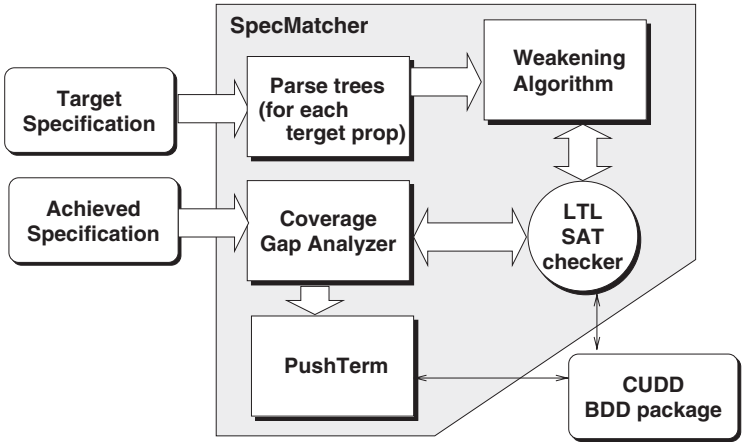


Fig. 4 The architecture of SpecMatcher

2.3 SpecMatcher – The Intent Coverage Tool

SpecMatcher is our in-house tool for specification coverage over LTL specifications. The architecture of the tool is shown in Figure 4. The tool accepts two specifications, namely the target specification and the achieved specification. Both should be in LTL. It produces the uncovered architectural specification (in LTL) as the output. SpecMatcher uses an in-house LTL satisfiability checker for checking the primary coverage question.

3 Testcase Generation for DPV

In the previous section, we presented a methodology to compare the system and the subsystem specifications and deduce the coverage gap. Typically this coverage gap is a function of the input scenarios. In other words, we often find that the subsystem specification succeeds in covering the system specification under some input constraints and fails for the rest. The input scenarios under which the coverage fails is to be targeted during DPV.

One option for solving this problem is by writing directed tests to cover the behaviors for which the gap exists. However, this is not practical. A property can be *triggered* in complex ways. It is not easy for a validation engineer to visualize all types of scenarios in which the property may become relevant.

The challenge is to find an automated solution to this problem. Given an assertion, the aim is to develop an automatic methodology for guiding DPV to *those behaviors for which the assertion is relevant*. The following discussion presents an approach towards this problem.

3.1 The Concept of Vacuity

Given an assertion, we can partition all possible behaviors of the system under test into two sets – the set of behaviors for which the assertion is relevant and the set of behaviors for which the assertion is not relevant. The assertion is *vacuously* satisfied in the second set of behaviors. Therefore, validation is meaningful with respect to a given assertion only if DPV traces a run from the first set. Consider the following example.

Example 10 Let us consider a slightly modified specification of the hold control unit (Figure 2). We wish to target the following properties during DPV of this unit:

P_1: $G\ ((brake\ \wedge\ stop\ )\ \Rightarrow\ X\ hold\ )$
P_2: $G\ ((hold\ \wedge\ disable\ )\Rightarrow\ X\ \neg hold)$
P_3: $G\ ((brake\ \wedge\ \neg\ stop\ )\Rightarrow\ (X\ (stop\ \vee\ hold)\vee\ XX\ hold\ ))$

It may be noted that except *hold*, all the other functions occurring in the above set of properties are outputs with respect to the hold control unit. We now explain the notion of vacuity through the above properties.

- The first property, P_1, is relevant in only those scenarios where $brake$ and $stop$ are both active. In all other scenarios (e.g. when $brake$ or $stop$ is low), P_1 is satisfied vacuously because *the system under test (the hold management unit) plays no part in satisfying the property in this way*. In order to cover P_1, DPV must drive both $brake$ and $stop$ to high.

- Let us consider the second property, P_2. Obviously P_2 is satisfied vacuously in all scenarios where disable is low.
  Is it vacuously satisfied when disable is high, but hold is low? In this case, the antecedent of the implication is false, but the property is satisfied non-vacuously. This is because the system under test played a role in falsifying the antecedent.
  The system can satisfy an implication property by satisfying the consequent or by refuting the antecedent. In both cases the property is satisfied non-vacuously, since the behavior of the system under test is responsible for the satisfaction.
  In order to cover P_2, the test generator must drive disable as high.
- The third property, P_3, is vacuously satisfied at all times where *brake* is not pressed or if the *brake* is pressed and the car stops. It is satisfied non-vacuously if *brake* is pressed and the car does not stop but the system asserts *hold* in the next cycle.
  What if the system does not assert hold in the next cycle? We have two cases, namely (a) the car stops in the next cycle, and (b) *stop* remains false in the next cycle. In the first case, the property is again satisfied vacuously. In the second case, the property is satisfied (non-vacuously) if *hold* is asserted in the subsequent cycle.
  What is the role of the test generator in this case? In order to cover P_3, it must drive *brake* to high and *stop* to low and study the response in the next cycle. If the system asserts *hold*, then the test generator has a non-vacuous hit. Otherwise, it must drive *stop* to low again, which is now guaranteed to result in a non-vacuous match or fail. □

The last case suggests that the test generator needs to be online and adaptive. The coverage of a temporal property may span over multiple cycles – in each cycle, the test generator needs to study the response of the system-under-test in order to determine the non-vacuous inputs for the next cycle.

3.2 Non-Vacuous Test Generation

Our procedure for non-vacuous test generation with respect to an assertion revolves around the unfolding of a temporal property during DPV. In the discussion below, we present the methodology over LTL. We first present a few definitions.

Definition 11 (X-Pushed Formula) *A formula is said to be X-pushed if all the X operators in the formula are pushed as far as possible to the left.*

Definition 12 (X-Guarded Formula) *A formula is said to be X-guarded if the corresponding X-pushed formula starts with an X operator whose scope covers the whole formula.*

Example 13 Let us consider the temporal property

$$\mathcal{P} = ((X\ p)\ U\ (X\ X\ q)) \wedge (X\ F\ r)$$

The X-pushed form of $\mathcal{P}$ is:

$$\mathcal{P}_X \;=\; X\,(\,(\,p\;U\;(\,X\;q\,)\,)\;\wedge\;(\,F\;r\,)\,)$$

Now $\mathcal{P}$ is a X-guarded formula as the corresponding X-pushed formula $\mathcal{P}_X$ starts with an X operator whose scope covers the whole formula. □

The task of monitoring the truth of a given property along a DPV run works as follows. If we are required to check an LTL property, φ, from a given time step, t, we rewrite the LTL property into a set of propositions over the signal values at time t and a set of X-guarded LTL properties over the run starting from time $t+1$. The rewriting rules are standard, and are as follows:

$F\varphi \;=\; \varphi \vee XF\varphi$
$G\varphi \;=\; \varphi \wedge XG\varphi$
$p\;U\;q \;=\; q \vee (p \wedge X(p\;U\;q))$

The property checker reads the signal values from the simulation at time t and substitutes these on the rewritten properties and derives a new property that must hold on the run starting from $t+1$, by dropping the leftmost X operator from each X-guarded term.

Example 14 To check the property $p\;U\;(q\;U\;r)$ at time t, we rewrite it as:

$$(r \;\vee\; (q \;\wedge\; X(q\;U\;r))) \;\vee\; (p \;\wedge\; X(p\;U\;(q\;U\;r)))$$

If the simulation at time t gives $p=0$, $q=1$, $r=0$, then by substituting these values, we obtain the property $X(q\;U\;r)$. Therefore at time $t+1$ we need to check the property $q\;U\;r$. We repeat the same methodology on $q\;U\;r$ at time $t+1$. □

For automatic test generation, the test generator should choose the values of the input signals at each time step t while monitoring the property. The following definition is useful to characterize the goal of the test generator.

Definition 15 (Vacuous Test) *A test stimulus, $\widehat{I}$, is vacuous at a given DPV step with respect to a property, φ, iff φ becomes true at that step on $\widehat{I}$ regardless of the values of the remaining variables.* □

For each property, the target is to drive a sequence of tests such that a non-vacuous success/failure of the assertion is reported. To develop the formal algorithm for test generation, let us study a DPV run between the system-under-test and a testcase with respect to a given property, $\mathcal{L}$. The *initial* system configuration is the first step. If the initial configuration is sufficient to satisfy or refute $\mathcal{L}$, then the test-generator has a non-vacuous success since the initial configuration contains assignment to non-inputs only. Otherwise, there must exist some non-vacuous test input with respect to $\mathcal{L}$. The test generator must choose one such assignment. Simulation is done for one cycle with the chosen input test and the system response (that is, the values of the non-inputs) is recorded. If $\mathcal{L}$ is now satisfied or refuted, then the test generator stops. We repeat the process till the end of simulation or till a success/failure is hit. The

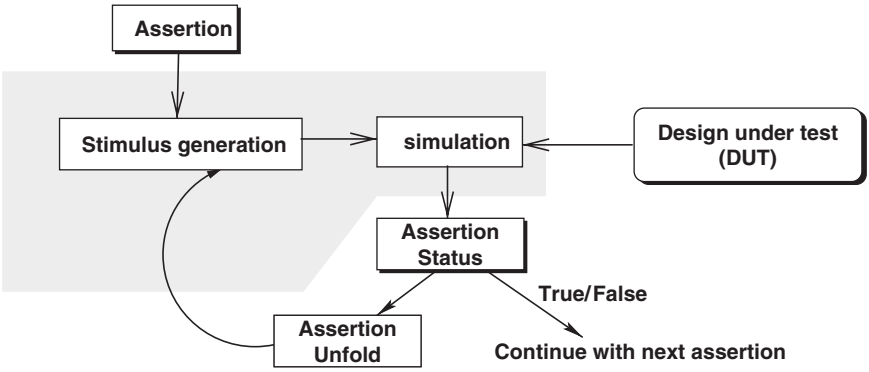


Fig. 5 Assertion-guided test-generator

details of the test generation algorithm can be found in [1]. The overall architecture of the test generation platform is shown in Figure 5. The following example illustrates a sample run.

Example 16 Consider the specification $P_3 : G\ ((brake\ \wedge\ \neg\ stop\ )\ \Rightarrow\ (X\ (stop\ \vee\ hold)\ \vee\ XX\ hold\ ))$ in Example 10. Let us assume a hypothetical implementation of the hold control unit, which is being tested through DPV. Initially, $hold$ is low. Substituting the values of $hold$, the specification does not evaluate to true or false. In the first step, the test generator is called with P_3 as argument, which discards the X-guarded terms and generates the non-vacuous input scenario $(brake\ \wedge\ \neg\ stop\ )$. One step simulation is performed and let us assume that the hold unit again fails to assert the $hold$ signal. The input condition pertaining to this time step is $stop$ which is generated and another step of simulation is run. Now, assuming that $hold$ remains de-asserted, a non-vacuous failure of P_3 is reported. On the other hand, if $hold$ is asserted, the process is repeated since P_3 is an invariant property. □

4 A Platform for DPV for Software Systems

In this section, we describe the details of a DPV framework for validating properties of automotive control subsystems designed using UML Statecharts. Our work was inspired by the fact that purely formal techniques are not suitable for verification in the context of automotive control due to the following factors:

- FPV is unable to handle large designs. Even with symbolic encodings of the finite state structure of the model, FPV cannot scale to the kind of systems arising out of composition of subsystems in automotive control.
- ECUs react on continuously changing sensor readings, monitoring the controlled system or its environment. This may be modeled using real valued variables with

unbounded domains (this is one of the factors that contributes to the undecidability result in software validation). For FPV, we need to abstract from unbounded domains to get a finite representation of the system model. FPV tools, therefore, cannot be both sound and complete when applied in the context of software validation unless the specification language is severely constrained.
- FPV specification languages have limited features (properties over real-value comparisons, software events, etc. are not supported).

DPV is a semi-formal technique in which the assertions are validated over simulation runs. DPV is scalable, and can support a much more enriched specification language as compared to FPV. DPV gels well with the model development process as well, considering the fact that model development languages like UML [7] have support for a wide variety of modeling constructs.

The main idea of DPV is based on validating the truth/falsification of the assertions on the basis of the responses of the system under test during simulation. These responses are based on valuations assigned by the models to its attributes. Therefore, one of the major requirements in building DPV is to define an *interface* for accessing values of model variables during simulation. The system under verification must allow some external hooks to access the model attributes and events needed by the verification engine. Adding hooks inside the source code of a model requires perturbation of source code and may introduce coding and execution overheads. In addition, the granularity of model attribute sampling is also important, due to the absence of a universal clock (as in the hardware domain) and values of data attributes and events sampled at different points may result in different validation results. Fortunately, current software simulators like Rhapsody for UML allow external hooks for sampling model parameters at the appropriate level of granularity, without having to disturb the model. We exploit this feature in the proposed DPV approach.

In the following subsection, we present a DPV platform for verifying behavioral properties of communicating concurrent software systems described using UML Statecharts. Our work is based on the semantics of Statecharts, as employed by Rhapsody from I-logix [5].

4.1 DPV for UML over Rhapsody

The development of DPV over Rhapsody posed some non-trivial challenges. We explain these with respect to the basic requirements for DPV in the context of ECU or subsystem validation. These are as follows:

- A modeling language for the system under test and a simulator.
- An assertion specification language: The language should be rich enough to support correctness requirements arising in software systems.

- An interface to monitor model responses: The verification engine should be able to sample model attributes and events after every step.
- A verification engine running on top of the simulator: The verification engine should be easily integrable into any model.

To address the first issue, we considered UML as the model development language, and Rhapsody as the UML simulator. For this, we needed to understand the simulation semantics of Rhapsody, the way it handles concurrency, communication and other features.

To describe correctness properties for ECU validation, one needs to describe properties over data attributes and events as well. Property specification languages that have been widely used in the formal verification community are predominantly either state-based or event-based. However, in the context of automotive control, we need to specify both *state* information and events (communication among components). For example, the Local Interconnect Network (LIN) [6] protocol specification has the following requirement: *In slave node, detection of break/synch frame shall abort the transfer in progress and processing of the new frame shall commence*. As this shows, both states (for describing a transfer in progress) and events (break/synch event) are required to capture the desired behavior. In our work, we extended Linear Temporal Logic (LTL) with some interesting features, specifically, the ability to express requirements over events, ability to express arithmetic and relational queries over data attributes (both Boolean and integer valued), the concept of local variables and the concept of parameterized events. Our logic is called Action-LTL and is used within the DPV framework for specifying assertions.

To address the third issue, we overloaded the event handler of Rhapsody (RicGEN) to send the verification engine a copy of every event generated after every simulation step. The verification engine developed by us which was integrated inside Rhapsody exploits this. The model parameters are sampled by the assertion engine separately by calling appropriate methods.

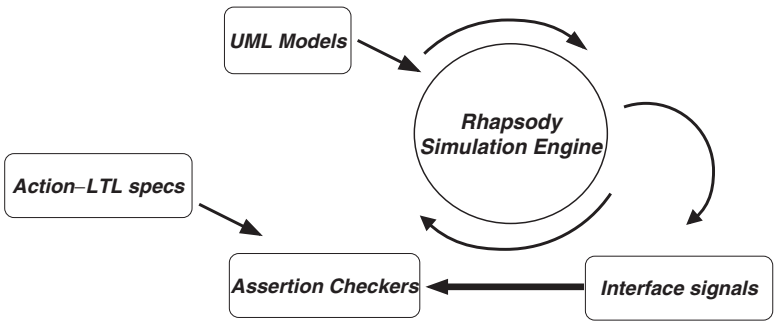


Fig. 6 The DPV platform over Rhapsody

Specifications described in Action-LTL are compiled into an assertion monitor and integrated with the Rhapsody models of the system under test. The assertion monitor is built as a Rhapsody object with embedded C routines The verification procedure enjoys a symbolic implementation (based on the CUDD package) and is therefore very space efficient. The assertion monitor is then cosimulated with the Rhapsody model of the design-under-test and success/failures are shown as events on the generated sequence diagram. This facilitates debug. Figure 6 shows the overall architecture.

Our main contributions may be summarized as follows:

- We have extended LTL to be able to express a wide range of requirements using local variables and parameterized events.
- We have developed a complete DPV solution for UML models over Rhapsody.
- We have developed the complete property suite of the Local Interconnect Protocol (LIN) [6] in Action-LTL and tested the concept of DPV on Rhapsody-based LIN models.

5 Conclusion

Existing static verification techniques do not scale beyond components of moderate size, thereby posing a major limitation in the applicability of FPV within the design validation flow. The more promising technique in this context has been Dynamic Property Verification, but it relies on appropriate and exhaustive modeling of test-cases. The verification by specification coverage approach presented in this paper is a serious attempt towards extending the frontiers of DPV, by enabling the coverage of system level properties by the collection of subsystem guarantees, and generating testcases to aid DPV.

In the context of automotive control, systems are far too complex, and they typically consist of both software and hardware. Especially software execution properties must be timely taken into account, since software modules more and more dominate the functionality. The execution of software in such real-time systems is, in general, controlled by a real-time operating system (RTOS). The presence of the RTOS creates a major verification challenge, since properties to be verified now critically depend on the strategies of the RTOS, and we need to consider this in the specification coverage framework as well. In addition, there are issues like bus latency, interrupts, etc. which also need to be considered. We believe the simplicity and effectiveness of the approach will encourage further research on the verification by specification coverage paradigm in the context of automotive control.

References

1. Banerjee, A., Pal, B., Das, S., Kumar, A., and Dasgupta, P., Test Generation Games from Formal Specifications, In *Proc. of Design Automation Conference* (DAC'2006), San Francisco, pages 827–832.
2. Basu, P., Das, S., Banerjee, A., Dasgupta, P., Chakrabarti, P.P., Mohan, C.R., Fix, L., and Armoni, R., Design Intent Coverage – A new paradigm for Formal Property Verification. *IEEE Trans. on CAD*, pages 1922–1934, 2006.
3. Bienmüller et al., Verification of Automotive Control Units, In Ernst-Rdiger Olderog and Bernd Steffen, editors, Correct System Design, vol. 1710 of LNCS, pages 319–341.
4. Eckard et al., Adding Value to Automotive Models, In Automotive Software – Connected Services in Mobile Networks, vol. 4147/2006 of LNCS, pages 86–102.
5. http://www.ilogix.com
6. http://www.lin-subbus.org/
7. Object Management Group, *Unified Modeling Language Specification*, Version 1.4, Draft, OMG(2001), http://cgi.omg.org/cgibin/docad/018214.
8. Pnueli, A., The Temporal Logic of Programs. In *Proc. of Foundations of Computer Science*, pages 46–57, 1977.
9. Reference Verification Methodology for Vera, http://www.synopsys.com/products/simulation/pdf/va_vol4_iss1_vera.pdf
10. System Verilog. http://www.systemverilog.org/

Beyond Satisfiability: Extensions and Applications*

Natarajan Shankar

Abstract Satisfiability procedures are used to check if a formula representing a constraint has a solution. They are gaining popularity as core engines for a number of applications. These procedures can be adapted for uses beyond testing satisfiability. We describe the underlying ideas and enumerate some of the applications and extensions of satisfiability procedures for verification, test generation, planning, and scheduling.

Keywords: boolean satisfiability, satisfiability modulo theories, decision procedures, automated theorem proving

Propositional satisfiability (SAT) procedures date back to the work of Post (1921) and Bernays (Zach, 1999) around 1920, but recent advances have rendered them useful for an impressive array of applications. The increased efficiency of these procedures stems mainly from representations and algorithms that are cache-aware and support low-overhead backtracking. SAT procedures have been used for testing both combinational and state machine equivalence of hardware circuits, for generating test cases and plans, and for the bounded model checking of both hardware and software systems. They can also be instrumented to generate proofs corresponding to unsatisfiability and interpolant formulas that can be used to partition such proofs. Given a collection of constraints that are not simultaneously satisfiable, satisfiability procedures can be extended to construct the solution that violates the fewest constraints. SAT procedures can be extended beyond the propositional realm to decide the satisfiability of quantifier-free formulas in a theory such as equality or arithmetic, or even a combination of such theories. The resulting procedure decides satisfiability modulo theories (SMT), and can be applied to problems in real-time and hybrid systems as well as assertion checking, predicate abstraction, and model checking of programs.

1 Propositional Satisfiability

The principles of modern SAT solving have their origin in the 1960 procedure of Davis and Putnam (1960), as simplified in 1962 by Davis et al. (1983). A propositional formula ϕ can be a propositional variable p or a negation $\neg\phi$, a conjunction

*This research was supported NSF Grants CCR-ITR-0326540 and CCR-ITR-0325808.

S. Ramesh and P. Sampath (eds.), *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*, 213–225.

$\phi_0 \wedge \phi_1$, a disjunction $\phi_0 \vee \phi_1$, or an implication $\phi_0 \Rightarrow \phi_1$ of smaller formulas ϕ_0, ϕ_1. A truth assignment ρ for a formula ϕ maps the propositional variables in ϕ to $\{\top, \bot\}$. A given formula ϕ is *satisfiable* if there is a truth assignment ρ such that $\rho \models \phi$ under the usual truth table interpretation of the connectives.

A literal is either a propositional variable p or its negation $\neg p$. The negation of a literal p is $\neg p$, and the negation of $\neg p$ is just p. A formula is in negation normal form (NNF) if the negation operation appears only in literals of the form $\neg p$. to propositional variables. Any formula can be converted to NNF by transforming $\neg\neg\phi$ to ϕ, $\neg(\phi_0 \wedge \phi_1)$ to $\neg\phi_0 \vee \neg\phi_1$, $\neg(\phi_0 \vee \phi_1)$ to $\neg\phi_0 \wedge \neg\phi_1$, and $\neg(\phi_0 \Rightarrow \phi_1)$ to $\phi_0 \wedge \neg\phi_1$, wherever such subformulas occur in the formula. A formula is a clause if it is the iterated disjunction of literals of the form $l_1 \vee \cdots \vee l_n$ for literals l_i, where $1 \leq i \leq n$. A formula is in conjunctive normal form (CNF) if it is the iterated conjunction of clauses $\Gamma_1 \wedge \cdots \wedge \Gamma_m$ for clauses Γ_i, where $1 \leq i \leq m$.

The first step in the Davis–Putnam–Logemann–Loveland (DPLL) procedure is to convert the formula to conjunctive normal form (CNF) by introducing new variables to label the subformulas. A formula can be converted to clausal form by introducing fresh variables for each compound subformula and adding suitable clauses, e.g., in converting $\neg p \vee (\neg q \wedge r)$, we label $\neg q \wedge r$ as b and $\neg p \vee b$ as a to obtain the clauses $a, a \vee p, a \vee \neg b, \neg a \vee \neg p \vee b, b \vee q \vee \neg r, \neg b \vee \neg q, \neg b \vee r$.

The input to the satisfiability procedure is given as a set of clauses K representing the CNF formula $\bigwedge K$. The DPLL procedure works by building up a partial truth assignment to the variables by successively guessing an assignment for a literal, propagating the consequences of the partial assignment with respect to the clauses, and backtracking on the partial assignment when a conflict is detected in the form of a falsified clause. The procedure terminates either with a truth assignment satisfying each of the clauses, or with a demonstration that no such assignment can be constructed. The state of the search procedure is a 4-tuple $\langle h, M, K, C\rangle$ consisting of the decision level h, the partial assignment M, the input clause set K, and a set C of conflict clauses derived from K that are constructed during the search.

At a decision level h, the partial assignment consists of a sequence $\langle M_0, \ldots, M_h\rangle$. Each M_i at decision level i is of the form $d; \langle l_1[\Gamma_1], \ldots, l_k[\Gamma_k]\rangle$ for some k, where d is the decision literal at level i, and each l_i is an implied literal. The assignment M_0 contains no decision literal. A decision literal or implied literal in M is said to be an assigned literal in M. No assigned literal in M occurs twice in M, nor does it occur negated in M. The assignment corresponding to M maps a variable p to $\top$ (respectively, $\bot$) if p (respectively, $\neg p$) is an assigned literal in M. If neither p nor $\neg p$ occcurs in M, then the assignment is undefined. Given an assigned literal l occurring in M at level i, the assignment preceding l, written as $M_{<l}$, consists of $\langle M_0, \ldots, M_{i-1}, M_i^{<l}\rangle$, where $M_i^{<l}$ consists of the part of the assignment of M_i preceding the occurrence of l. For each entry $l[\Gamma]$ in M, the clause Γ occurs in $K \cup C$ and is of the form $l \vee \Gamma'$, where $M_{<l} \models \neg\Gamma'$.

The DPLL search algorithm works by constructing the partial assignment M through the use of *propagation*, *analysis/backjumping*, and decision literal *selection*, until it has constructed an assignment satisfying the input clauses K or it can be shown that there is no such assignment. For decision level h, propagation works by

adding $l[\Gamma]$ to M_h, where $\Gamma \in K \cup C$ is of the form $l \vee \Gamma'$, where $M_0 \models \neg\Gamma'$. When $h0$, each unit clause l in $K \cup C$ is placed in M_0 as $l[l]$. Propagation can also detect a conflict where there is a clause of the form Γ such that $M \models \neg\Gamma$. If a conflict is detected at decision level 0, then the algorithm reports unsatisfiability. Otherwise, each conflict can be analyzed to construct a conflict clause that is added to C. If clause Γ in $K \cup C$ is the source of the conflict, then it is of the form $l_1 \vee \cdots \vee l_n$ where M contains $\neg l_i[\neg l_i \vee \Gamma_i]$, for $1 \leq i \leq n$. The analysis phase successively replaces Γ with the result of resolving Γ with each clause $\neg l_i \vee \Gamma_i$ for l_i occurring at level h until Γ contains a unique literal l at level h.

Note that $M \models \neg\Gamma$ for each such clause Γ generated through analysis. Furthermore, the clause Γ contains at least one literal at level h since the conflict is detected at level h. The analysis process will arrive at a point where there is a unique literal at level h in Γ since the negation of the decision literal at level h will eventually be generated from any other literal at level h.

The clause Γ constructed by the analysis phase is added as a conflict clause to C to obtain the new conflict clause set C'. Let h' be the level of the next highest literal l such that $\neg l$ occurs in Γ. The search is resumed with the state $\langle h', M_{h'}, K, C'\rangle$. Note that the unique literal l at level h in Γ is now an implied literal at level h'. Though the partial assignment has shrunk, it now contains more implied literals at level h'. On the other hand, if no conflict is detected at level h, then an unassigned literal d is *selected* as the decision literal at level $h+1$, and the search is resumed with the state $\langle h+1, \langle M; d\rangle, K, C\rangle$. If no unassigned literals remain, then the algorithm terminates with a satisfying assignment M for K. Termination (Nieuwenhuis et al., 2005; Shankar, 2005) follows since each step of propagation, backjumping (with propagation), or selection increases the quantity $\Sigma_i^h |M_i| * N^{(N-h)}$ towards the bound $N^{(N+1)}$ for $N|vars(K)|$.

The algorithm can either terminate with an assignment M satisfying the input clause set K, or with an unsatisfiability when a conflict is reported at the decision level 0. The SAT procedure can also generate a proof of unsatisfiability since a conflict at level 0 implies that some clause Γ in K when resolved with other clauses from $K \cup C$ yields a contradiction. The clauses in C are themselves derived by resolution. An example computation of the DPLL algorithm for demonstrating the unsatisfiability of the input K given by $\{p \vee q, \neg p \vee q, p \vee \neg q, s \vee \neg p \vee q, \neg s \vee p \vee \neg q, \neg p \vee r, \neg q \vee \neg r\}$. is shown in Figure 1. In this example, there are no unit input clauses. The partial assignment M_0 is therefore empty. The literal s is selected as the decision literal at level 1. Propagation does not yield any new implied literals at level 1. Then, literal r is selected as the decision literal at level 2. Now propagation adds the literals $\neg q$ and p, but then detects the conflict with clause $\neg p \vee q$. Analyzing this conflict, we obtain the conflict clause q which is added to C while backjumping to level 0. Now there is a unit clause q, and propagation adds the literals p and r to M_0 before detecting the conflict on the clause $\neg q \vee \neg r$. Since this conflict is at level 0, the input clause is judged to be unsatisfiable.

The proof of unsatisfiability for the example in Figure 1 can be constructed by resolution. The conflict clause q is proved by resolving $\neg p \vee q$ with $p \vee q$. The proof of unsatisfiability is constructed by resolving $\neg q \vee \neg r$ with $\neg p \vee r$ to obtain $\neg p \vee \neg q$

Step	h	M	K	C	Γ
selection s	1	$; s$	K	$\emptyset$	–
selection r	2	$; s; r$	K	$\emptyset$	–
propagation	2	$; s; r, \neg q[\neg q \vee \neg r]$	K	$\emptyset$	–
propagation	2	$; s; r, \neg q, p[p \vee q]$	K	$\emptyset$	–
conflict	2	$; s; r, \neg q, p$	K	$\emptyset$	$\neg p \vee q$
analysis	0	$\emptyset$	K	q	–
propagation	0	$q[q]$	K	q	–
propagation	0	$q, p[p \vee \neg q]$	K	q	–
propagation	0	$q, p, r[\neg p \vee r]$	K	q	–
conflict	0	q, p, r	K	q	$\neg q \vee \neg r$

Fig. 1 Example of the DPLL satisfiability procedure

which is in turn resolved with $p \vee \neg q$ to obtain $\neg q$ which is resolved with the conflict clause q to derive $\perp$.

Key ideas in the development of efficient SAT solvers originate from SATO (Zhang, 1997), GRASP (Marques-Silva and Sakallah, 1999), and Chaff (Moskewicz et al., 2007). Efficient implementations of SAT algorithms include ZChaff (Zhang and Malik, 2002; Zhang, 2003), Berkmin (Goldberg and Novikov, 2002), Siege (Ryan, 2004), and MiniSAT (Eén and Sörensson, 2003).

1.1 Extensions and Applications

The satisfying assignment $\rho(M)$ generated by a satisfiability procedure has many uses since it can be used as a solution to the input constraints encoded by the clause set K. The formulas given to a SAT solver can represent

1. Combinational logic $oF(x_1, \ldots, x_n)$
2. Combinational equivalence $F(x_1, \ldots, x_n)G(x_1, \ldots, x_n)$
3. The k-fold unwinding of a state machine with initial state I and transition relation R: $I(s_0) \wedge N(s_0, s_1) \wedge \ldots \wedge N(s_{k-1}, s_k)$
4. Bounded model checking (Biere et al., 1999): $I(s_0) \wedge N(s_0, s_1) \wedge \cdots \wedge N(s_{k-1}, s_k) \wedge (\neg P(s_0) \vee \cdots \vee \neg P(s_k))$
5. An invariance proof obligation for a state machine: $P(s) \wedge N(s, s') \wedge \neg P(s')$
6. A k-induction proof (Sheeran et al., 2000) $I(s_0) \wedge N(s_0, s_1) \wedge \cdots \wedge N(s_{k-1}, s_k) \wedge P(s_0) \wedge \cdots \wedge P(s_{k-1}) \wedge \neg P(s_k)$

The satisfying assignment in these instances can be a counterexample or a test case. For the case of invariant proof obligations, the satisfying instance can be used to suggest a strengthening of the invariant.

AllSAT The AllSAT version of the problem requires a representation of all possible satisfying instances, for example, as a binary decision diagram or in disjunctive normal form (DNF). Let $l_1, \ldots, l_n$ be a satisfying assignment M found by the satisfiability procedure, then $\neg d_1 \vee \cdots \vee \neg d_n$ for the decision literals $d_1, \ldots, d_n$ can be added as a *blocking clause* to C and the search can be continued until the resulting set of clauses is unsatisfiable. The complete set of blocking clauses B can be extracted from the conflict clause set C. The DNF equivalent of the input formula ϕ is then just $\bigvee\{\neg\Gamma \mid \Gamma \in B\}$. For example, with a failed invariant proof, the AllSAT procedure can be used to enumerate a symbolic representation of the set of states s such that $P(s) \wedge N(s, s') \wedge \neg P(s')$. The AllSAT procedure can be used to get the CNF equivalent of a given formula without the extra variables introduced by the syntactic CNF conversion shown above (McMillan, 2002). The CNF equivalent of ϕ is obtained by just negating the DNF equivalent of $\neg\phi$.

For example, the DPLL search with the input clause set $\neg a \vee b, c$ yields an assignment $Mc; a, b$. The negation $\neg c \vee \neg a \vee \neg b$ is added as a blocking clause. The new clause yields a conflict so that backjumping followed by search yields a new assignment $c; a, \neg b$, which creates a conflict. The search is continued with the addition of the conflict clause $\neg c \vee \neg a$ to C following analysis and backjumping. A new satisfying assignment $c, \neg a; b$ is generated, and the resulting blocking clause $\neg c \vee a \vee \neg b$ is added to B. Further conflict analysis, backjumping, and search yields the satisfying assignment $c, \neg a, \neg b$ and the blocking clause $\neg c \vee a \vee b$. With this, there is now a conflict at level 0. The resulting DNF is

$$(c \wedge a \wedge b) \vee (c \wedge \neg a \vee b) \vee (c \wedge \neg a \wedge \neg b).$$

The procedure can be optimized by pruning each satisfying assignment M to a minimal set of literals $\overline{M}$ such that each clause in K contains at least one literal in $\overline{M}$. In the above example, the first blocking clause can be reduced to $\neg c \vee \neg b$. Now, propagation at level 0 yields the assignment $c, \neg b[\neg c \vee \neg b], \neg a[\neg a \vee b]$. The resulting blocking clause is $\neg c \vee a$, yielding a conflict at level 0. The DNF returned is the more compact $(c \wedge b) \vee (c \wedge \wedge a)$.

Boolean quantification can also be computed using the normal form conversion so that if $CNF(\phi) \equiv \Gamma_1 \wedge \ldots \wedge \Gamma_n$, then $\forall \boldsymbol{x} : \phi \equiv \forall \boldsymbol{x} : CNF(\phi) \equiv (\forall \boldsymbol{x} : \Gamma_1) \wedge \ldots \wedge (\forall \boldsymbol{x} : \Gamma_n) \equiv \Gamma_1 \backslash \boldsymbol{x} \wedge \ldots \wedge \Gamma_n \backslash \boldsymbol{x}$. Existential quantification $\exists \boldsymbol{x} : \phi$ can be computed as $\neg\forall \boldsymbol{x} : \neg\phi$. SAT-based Boolean quantification can be used as an alternative to BDDs in symbolic model checking (McMillan, 2002).

Interpolation Given a proof of unsatisfiability, it is possible to generate a Craig interpolant (Craig, 1957). The Craig interpolation theorem for first-order logic asserts that when two formulas $A \Rightarrow B$ is valid, then there must be an interpolant I so that the language of I, namely the set of non-logical function and predicate symbols occurring in I, is contained in the languages of A and B. Equivalently, if $A \wedge B$ is unsatisfiable, then there is an interpolant I that is entailed by A such that $I \wedge B$ is unsatisfiable.

In the propositional case, the interpolant I will be a propositional formula whose propositional variables are included in those of both A and B. If we have two sets of clauses K_1 and K_2 with a proof of unsatisfiability for $K_1 \cup K_2$, the interpolant can be constructed from the proof of unsatisfiability. Each clause Γ in the proof can be decomposed as $\Gamma_1 \vee \Gamma_2$, where $vars(\Gamma_2) \subseteq vars(K_2)$ and $vars(\Gamma_1) \bigcap vars(\Gamma_2)$. If we let $\neg\Gamma_1$ represent the formula that is the negation of Γ_1, then a clause $\Gamma_1 \vee \Gamma_2$ can be read as the implication $\neg\Gamma_1 \Rightarrow \Gamma_2$. Each such clause Γ in the proof has an associated interpolant $\mathcal{I}(\Gamma)$ with $vars(\mathcal{I}(\Gamma)) \subseteq vars(K_1) \bigcap vars(K_2)$ such that $K_1 \vdash \neg\Gamma_1 \Rightarrow \mathcal{I}(\Gamma)$ and $K_2 \vdash \mathcal{I}(\Gamma) \Rightarrow \Gamma_2$. For each clause (Γ) in K_1, the interpolant $\mathcal{I}(\Gamma)\Gamma_2$, and for each clause Γ in K_2, $\mathcal{I}(\Gamma)\top$, where $\top$ is the empty clause. Whenever a clause Γ is derived by resolution from clauses Γ' and Γ'', we know that we have interpolants $\mathcal{I}(\Gamma')$ such that $K_1 \vdash \neg\Gamma'_1 \Rightarrow \mathcal{I}(\Gamma')$ and $K_2 \vdash \mathcal{I}(\Gamma') \Rightarrow \Gamma'_2$, and similarly for $\mathcal{I}(\Gamma'')$. If the resolution is on a literal p such that p occurs in Γ'_1 and $\neg p$ in Γ''_1, then since $K_1 \vdash \neg(\Gamma'_1) \Rightarrow \mathcal{I}(\Gamma')$ and $K_2 \vdash \mathcal{I}(\Gamma') \Rightarrow \Gamma'_2$, and similarly for Γ'', we have that $\mathcal{I}(\Gamma)\mathcal{I}(\Gamma') \vee \mathcal{I}(\Gamma'')$ since $K_1 \vdash \neg\Gamma \Rightarrow p \vee \mathcal{I}(\Gamma')$ and $K_1 \vdash \neg\Gamma \Rightarrow \neg p \vee \mathcal{I}(\Gamma')$. Correspondingly, if p occurs in Γ'_2 and Γ''_2, then the interpolant $\mathcal{I}(\Gamma)$ is $\mathcal{I}(\Gamma') \wedge \mathcal{I}(\Gamma'')$ since $K_1 \vdash \neg\Gamma \Rightarrow \mathcal{I}(\Gamma')$ and $K_1 \vdash \neg\Gamma \Rightarrow \mathcal{I}(\Gamma'')$ and $K_2 \vdash \mathcal{I}(\Gamma) \Rightarrow p \vee \Gamma'_2$ and $K_2 \vdash \mathcal{I}(\Gamma) \Rightarrow \neg p \vee \Gamma''_2$.

For example, consider the input clause set K partitioned as $K_1 \cup K_2$, where $K_1\{a \vee e[e], \neg a \vee b[b], \neg a \vee c[c]\}$, and $K_2\{\neg b \vee \neg c \vee d[\top], \neg d[\top], \neg e[\top]\}$. Here K_1 and K_2 share the variables, b, c, and e. The interpolation can be constructed from the annotated proof as shown below.

Γ	Γ'	Γ''
$a[e]$	$a \vee e[e]$	$\neg e[\top]$
$b[e \vee b]$	$\neg a \vee b[b]$	$a[e]$
$c[e \vee c]$	$a[e]$	$\neg a \vee c[c]$
$\neg c \vee d[e \vee b]$	$b[e \vee b]$	$\neg b \vee \neg c \vee d[\top]$
$d[(e \vee b) \wedge (e \vee c)]$	$\neg c \vee d[e \vee b]$	$c[e \vee c]$
$\bot[(e \vee b) \wedge (e \vee c)]$	$\neg d[\top]$	$d[(e \vee b) \wedge (e \vee c)]$

Interpolation has a number of uses (McMillan, 2005; Jhala and McMillan, 2006). It can be used in bounded model checking to identify assertions that are implied by the prior computation that are sufficient to achieve the desired unsatisfiability.

Planning A planning problem involves a set of actions, various domain constraints, an initial state, and a goal state (Kautz et al., 1996). The task is to find a schedule of actions leading from the initial state to the goal state that is consistent with the domain constraints. Examples of planning problems include blocks worlds, transportation, resource usage, and even human activity. A typical problem would be as follows: Given cities A, B, C, and D, where only A and C can be used for refueling. Given trucks u and v, where u is initially at A, and v is at B. The packages AB, AD, BC, BA, CA, CD, and DB, are labeled by source and destination. The trucks consume

half a tank of fuel in moving from one city to another, and they can fill their tanks when they are at either A or C. The actions are those of

1. Loading a package on to a truck if both the package and the truck are in the same city
2. Unloading a package from a truck to a city if the package is on the truck and the truck is at the city
3. Moving a truck from one city to another provided there is at least half a tank of fuel in the truck
4. Refueling a truck, i.e., filling the tank, provided the truck is at either A or C

A plan is a trajectory consisting of a series of steps, where each step consists of one or more independent actions. A plan is optimal if it takes the fewest steps.

The above planning problem can be coded as a SAT problem using Boolean variables

1. *location*(*truck*, *city*, *step*) for each truck, city, and step
2. *at*(*package*, *city*, *step*) for each package, city, and step number
3. in(*package*, *truck*, *step*) for each package, truck, and step number
4. *fulltank*(*truck*, *step*) indicating whether the fuel tank is full
5. *halftank*(*truck*, *step*) indicating the tank is empty

There are constraints asserting that a truck can have at most one location, a package can have at most one location either in a truck or a city, and a fuel tank cannot both be full and half full.

The actions are indicated by Boolean variables

1. *fill*(*truck*, *step*): Initiates the action of filling the truck at the given step.
2. *go*(*truck*, *city*, *step*): The truck is driven to the destination city.
3. *load*(*package*, *truck*, *step*): Loads the package on to the truck.
4. *unload*(*package*, *truck*, *step*): Unloads the package from the truck to the city where the truck is located.

Constraints are asserted binding the Boolean variables corresponding to each action to the actual action. Note that the *fill*, *load*, and *unload* actions cannot occur concurrently with the *go* action for the truck in question. Finally, the initial configuration is asserted of the initial state at step 0, and the final state which can be chosen as some number n.

The existence of an n step plan can be established by solving the constraints. By starting from a conservative target n for the number of steps and progressively decreasing the value of n by 1, when there is a value of n for which the constraints are unsatisfiable, we know that the plan at length $n + 1$ is optimal.

MaxSAT and Weighted MaxSAT Many satisfiability problems contain a mixture of hard constraints that have to be satisfied, and soft constraints that should be satisfied if possible. The goal with MaxSAT is to find the assignment that satisfies all the hard constraints and the maximum number of soft constraints. In the weighted case, there

are numerical penalties associated with the soft constraints, and the goal is to find an assignment with minimum overall penalty, where the latter quantity is the sum of the penalties over each of the falsified constraints. There are several ways in which weighted and unweighted MaxSAT problems can be solved with similar techniques. Each soft constraint A with weight w is represented using a fresh Boolean variable a as the formula $a \vee A$, and the weight constraint is specified as $\Sigma_i\, p_i * a_i \leq M$, for some limit M. The latter pseudo-Boolean constraint can be turned into an ordinary Boolean constraint by means of conditional addition and comparison. As with planning, we can start with a conservative bound M that is progressively narrowed until the set of constraints is unsatisfiable.

MaxSAT and its variants are useful for problems where some constraints are either lower in reliability or importance than others. It is also useful where there is a preferred solution and one would like to minimize the distance of the obtained solution from the preferred one. A simple but common example of MaxSAT is that of minimizing abnormality. For example, given a circuit with a battery B in series with two lamps L_1 and L_2 in parallel. If the battery is normal, then each normal lamp will light up. If lamp L_1 and L_2 both fail to light up, the abnormality can be attributed to the battery, and otherwise, if at least one of the lamps is lit, then the abnormality is only attributable to the other lamp.

2 Theory Satisfiability

While many practical problems can be encoded in terms of propositional satisfiability, there are lots of problems where the constraints are not purely propositional. They involve other logical symbols such as equality and inequality and function symbols that are interpreted in theories such as equality over uninterpreted function symbols, arithmetic, arrays, and datatypes. The resulting problem of satisfiability modulo theories (SMT) was first worked on by Shostak (1979) and Shostak et al. (1982) and Nelson and Oppen (1979) and Nelson (1981). The goal with such constraints is to find a satisfying assignment for the variables, including propositional variables, that satisfies the constraints when the symbols are interpreted according to specific theories. For example, many proof obligations involve propositional logic with the theory of uninterpreted equality. For hardware proofs showing the correspondence between the unpipelined instruction set architecture (ISA) and the pipelined register-transfer level (RTL), the datatype operations can be modeled as uninterpreted function symbols since these remain the same in both descriptions and only the control operations have changed. Verification conditions for imperative programs and induction steps in proofs can also be established by treating certain operations as uninterpreted. For example, in proving the associativity of the *append* operation, we may have to prove the induction subgoal

$$append(append(cons(u, x), y), z) append(cons(u, x), append(y, z))$$

in the context of the assertions

$$append(cons(u, x), y) \quad cons(u, append(x, y)) \tag{1}$$
$$append(cons(u, append(x, y)), z) \quad cons(u, append(append(x, y), z)) \tag{2}$$
$$append(cons(u, x), append(y, z)) \quad cons(u, append(x, append(y, z))) \tag{3}$$

Here, the function symbols *cons* and *append* can be treated as uninterpreted.

The significance of For the theory of uninterpreted function symbols, the Ackermann reduction can be used to reduced formulas ϕ to Boolean form by introducing individual variables $x_{f(y_1,\ldots,y_n)}$ and asserting for each pair of terms $f(y_1, \ldots, y_n)$ and $f(z_1, \ldots, z_n)$, the clause $y_1 \neq z_1 \vee \cdots \vee y_n \neq z_n \vee x_{f(y_1,\ldots,y_n)} x_{f(z_1,\ldots,z_n)}$. The resulting formula ϕ' is equivalent to ϕ but contains only propositional variables and equalities between individual variables. Such formulas can be further reduced to Boolean form by finite instantiation by identifying a finite domain D for the individual variables that is sufficient for exhibiting satisfiability.

In general, when deciding theory satisfiability of a formula ϕ with respect to a theory T, one can identify a finite set L_ϕ of formulas (lemmas) valid in the theory T concerning the theory atoms in ϕ such that the propositional satisfiability of $L_\phi \wedge \phi$ implies the satisfiability of ϕ. This *eager* approach (Bryant et al., 2002) to theory satisfiability suffers from the problem that the lemmas are generated indiscriminately and can overwhelm the SAT solver.

In the lazy approach (Stump et al., 2002; de Moura et al., 2002; Flanagan et al., 2003), the DPLL procedure is modified so that the partial assignment M is now a decision procedure context. The decision procedures can then decide if a literal l is entailed in the context M and generate a lemma clause $\neg l \vee \neg l_1 \vee \ldots \vee l_n$ encapsulating the explanation, where $l_1, \ldots, l_n$ are literals asserted into M. The contextual entailment can be used for detecting conflicts and enriching Boolean constraint propagation to theory propagation. Since contextual entailment is expensive, it is usually better to use an incomplete but efficient entailment procedure for theory propagation, and a complete procedure for detecting conflicts.

Typically, the theory T is a combination of theories $T_1, \ldots, T_n$, where these theories can include uninterpreted equality, arithmetic, arrays, datatypes, bit-vectors, among other theories. Variable abstraction can be used to convert each theory atom A in ϕ to a pure atom $\overline{A}$ by replacing each flat theory term $f(x_1, \ldots, x_n)$ by the fresh (interface) variable y and adding the conjunct $yf(x_1, \ldots, x_n)$ to ϕ. The decision procedure for the theory T can be obtained from those for the individual T_i provided we introduce case splits $y_1 y_2 \vee y_1 \neq y_2$ for pairs of interface variables y_1, y_2. These tautologous case split clauses can also be added as clauses in the DPLL search procedure. The theory decision procedures are also allowed to generate clauses. The Ackermann reduction can be seen as a generated clause from the equality theory. The array theory can generate the clause $ij \vee (A[i : v][j]A[j]$ whenever a term equivalent to $A[i : v][j]$ appears in the context. This kind of case-splitting was introduced in Shostak's STP solver (Shostak et al., 1982) for satisfiability modulo theories.

The Simplify theorem prover is a widely used SMT solver (Detlefs et al., 2003). Yices (Dutertre and de Moura, 2006) is a high-performance SMT solver with a

powerful range of capabilities developed at SRI. It supports an expressive language with higher-order types, dependent types, and predicate subtypes, for capturing constraints. Yices supports constraint solving in a combination of theories including Booleans, uninterpreted functions, linear arithmetic, records, tuples, datatypes, arrays, and bit-vectors. Yices also supports quantifier instantiation through e-graph matching (Nelson, 1981), incremental and interactive use, MaxSMT, and model construction. Yices also includes a competitive SAT solver. It can be used interactively through a command language with incremental definitions, assertions, context creation and examination, pushing/popping contexts, and MaxSAT-checking. Yices is integrated with SAL (http://sal.csl.sri.com) and PVS (http://pvs.csl.sri.com). It is used in hardware/software verification, bounded model checking, planning, probabilistic consistency using MaxSAT, and symbolic execution.

Procedures for satisfiability modulo theories have a number of interesting applications. We have already mentioned hardware verification as a motivating application. Since programs with variables ranging over integers, arrays, datatypes, and bit-vectors can also be captured directly with theory formulas, the procedure can be used for bounded model checking for infinite state systems including timed and hybrid systems, assertion checking, extended type checking, k-induction, test case generation, predicate abstraction, scheduling, and plan generation and validation (Rushby, 2006). We illustrate the use of SMT solvers for predicate abstraction (Lahiri et al., 2006).

Predicate Abstraction The goal in predicate abstraction is to construct a Boolean approximation $\alpha(\phi)$ of a formula ϕ with respect to some atomic formulas $A_1, \ldots, A_n$. The formula $\alpha(\phi)$ uses the Boolean propositions $p_1, \ldots, p_n$ to stand for $A_1, \ldots, A_n$, respectively, so that $\vdash \phi \Rightarrow \alpha(\phi)[A_1/p_1, \ldots, A_n/p_n]$. The formula $\alpha(\phi)$ can be constructed using the CNF construction strategy from the previous section with the formula $\neg\phi \wedge \bigwedge_i p_i A_i$ can be used to generate the set of blocking clauses Γ such that $\neg\phi \wedge \bigwedge_i p_i A_i \wedge \neg\Gamma$ is satisfiable. The conjunction of the blocking clauses in the Boolean variables $p_1, \ldots, p_n$ when reduced to its prime implicates yields the appropriate $\alpha(\phi)$.

Abstract Reachability Predicate abstraction can be used to over-approximate the reachable set of states for a transition system (Graf and Saïdi, 1997; Das and Dill, 2001; Henzinger et al., 2002) with initial state predicate I and next-state relations N, where $S_0 \alpha(I)$, and $S_{i+1} S_i \bigcup \alpha(N(\gamma(S_i)))$. Since the abstract state space is infinite, the iteration must converge to a set S. A concrete condition E is abstractly reachable if $S \bigcap \alpha(E)$ is nonempty. This yields an abstract trace $t_0, \ldots, t_m$ such that $t_0 \in S_0$ and each t_{i+1} is in $S_{i+1} - S_i$. The concrete version of this trace can be converted into a formula $\gamma(S_0) \wedge N(s_0, s_1) \wedge \gamma(S_1) \wedge \cdots \wedge N(s_{m-1}, s_m) \wedge \gamma(S_m)$. If the latter formula is satisfiable, then there is a concrete trace exhibiting the reachability of the condition E. Otherwise, if the formula is unsatisfiable, then the abstract trace is a spurious one, and the proof of unsatisfiability can be used to suggest new predicate to refine the abstraction. The interpolants at the individual states $s_1, \ldots, s_{m-1}$ can be used to suggest new predicates (Jhala and McMillan, 2006). The interpolant ψ at state s_i is

entailed by the formula $\gamma(S_0) \wedge N(s_0, s_1) \wedge \gamma(S_1) \wedge \cdots \gamma(S_i)$ and is unsatisfiable in conjunction with $N(s_i, s_{i+1}) \wedge \gamma(s_{i+1}) \wedge \cdots \wedge \gamma(s_m)$. Including the predicates contained in the interpolants thus ensures that the spurious trace is ruled out under the refined abstraction. Quantifier-free interpolants can be constructed for theories such as linear arithmetic and equality. For some theories, such as arrays, there may not be any quantifier-free interpolants even for quantifier-free input formulas.

Quantification When the given formula contains quantifiers, the satisfiability problem may not be decidable. Quantifiers can be eliminated by

1. Skolemization to replace the existential quantifiers by Skolem constants.
2. Quantifier elimination for formulas within theories that support the elimination of quantifiers.
3. E-graph matching (Nelson, 1981) to instantiate universally quantified variables by matching modulo the term equalities in the context M. The use of e-graph matching is heuristic and therefore incomplete.

3 Conclusions

We have outlined the basic ideas in the construction of satisfiability procedures for the case of propositional formulas and first-order formulas, including quantifier-free and quantified formulas. We have examined extensions to the basic satisfiability paradigm including the construction of proofs of unsatisfiability, the generation of all possible satisfying assignments, satisfiability under soft and weighted constraints, and the generation of interpolants. We have also examined applications of satisfiability in planning, test case generation, and verification. Satisfiability is the core technology for a number of applications and is likely to remain a fertile and exciting research discipline into the foreseeable future.

References

A. Biere, A. Cimatti, E. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Proceedings of the ACM Design Automation Conference (DAC'99)*. ACM Press, 1999.

Randal E. Bryant, Shuvendu K. Lahiri, and Sanjit A. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In *Proceedings of CAV'02*, volume 2404 of *Lecture Notes in Computer Science*, 2002.

W. Craig. Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. *Journal of Symbolic Logic*, 22(3):269–285, 1957.

Satyaki Das and David L. Dill. Successive approximation of abstract transition relations. In *Annual IEEE Symposium on Logic in Computer Science01*, pages 51–60. The Institute of Electrical and Electronics Engineers, 2001.

Bruno Dutertre and Leonardo de Moura. The Yices SMT solver. 2006.

M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, July 1962. Reprinted in Siekmann and Wrightson [SW83], pages 267–270, 1983.

Leonardo de Moura, Harald Rueß, and Maria Sorea. Lazy theorem proving for bounded model checking over infinite domains. In A. Voronkov, editor, *18th International Conference on Automated Deduction (CADE)*, volume 2392 of *Lecture Notes in Computer Science*, pages 438–455, Copenhagen, Denmark, July 2002. Springer-Verlag.

D. Detlefs, G. Nelson, and J.B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, Hewlett-Packard Systems Research Center, 2003.

M. Davis and H. Putnam. A computing procedure for quantification theory. *JACM*, 7(3):201–215, 1960.

Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *Proceedings of SAT 2003*, 2003.

Cormac Flanagan, Rajeev Joshi, Xinming Ou, and James B. Saxe. Theorem proving using lazy proof explication. In *Proceedings of the 15th International Conference on Computer-Aided Verification (CAV 2003)*, volume 2725 of *Lecture Notes in Computer Science*, pages 355–367. Springer-Verlag, 2003.

E. Goldberg and Y. Novikov. Berkmin: A fast and robust sat solver, 2002.

S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *Conference on Computer Aided Verification CAV'97*, Volume 1254 of *Lecture Notes in Computer Science*, Springer Verlag, 1997.

Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy abstraction. In *ACM Symposium on Principles of Programming Languages02*, pages 58–70. Association for Computing Machinery, January 2002.

Ranjit Jhala and Kenneth L. McMillan. Lazy abstraction with interpolants. In *Proceedings of CAV'06*, pages 123–136. Springer-Verlag, 2006.

Henry Kautz, David McAllester, and Bart Selman. Encoding plans in propositional logic. In *Proceedings of the International Conference on Knowledge Representation (KR'96)*, 1996.

Shuvendu Lahiri, Robert Nieuwenhuis, and Albert Oliveras. SMT techniqes for fast predicate abstraction. In *Proceedings of CAV'06*, 2006.

Kenneth L. McMillan. Applying SAT methods in unbounded symbolic model checking. In *Proceedings of CAV 2002*, pages 250–264. Springer-Verlag, 2002.

Kenneth L. McMillan. An interpolating theorem prover. *Theoretical Computer Science*, 345(1):101–121, 2005.

Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, June 2001.

J. Marques-Silva and K. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, May 1999.

G. Nelson. Techniques for program verification. Technical Report CSL-81-10, Xerox Palo Alto Research Center, Palo Alto, Ca., 1981.

G. Nelson and D.C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, October 1979.

Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Abstract DPLL and abstract DPLL modulo theories. In F. Baader and A. Voronkov, editors, *Proceedings of the 11th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR'04), Montevideo, Uruguay*, volume 3452 of *Lecture Notes in Computer Science*, pages 36–50. Springer, 2005.

E.L. Post. Introduction to a general theory of elementary propositions. *American Journal of Mathematics*, 43:163–185, 1921. Reprinted in [J. van Heijenoort, editor., pages 264–283].

John Rushby. Harnessing disruptive innovation in formal verification. In Dang Van Hung and Paritosh Pandya, editors, *Fourth International Conference on Software Engineering and Formal Methods (SEFM)*, pages 21–28, Pune, India, September 2006. IEEE Computer Society.

Lawrence Ryan. Efficient algorithms for clause-learning SAT solvers. Master's thesis, Simon Fraser University, 2004. M.Sc. Thesis.

Aaron Stump, Clark W. Barrett, and David L. Dill. CVC: A cooperating validity checker. In *Proceedings of CAV'02*, volume 2404 of *Lecture Notes in Computer Science*, 2002.

Natarajan Shankar. Inference systems for logical algorithms. In R. Ramanujam and Sandeep Sen, editors, *FSTTCS 2005: Foundations of Software Technology and Theoretical Computer Science*, volume 3821 of *Lecture Notes in Computer Science*, pages 60–78. Springer-Verlag, 2005.

Robert E. Shostak. A practical decision procedure for arithmetic with function symbols. *Journal of the ACM*, 26(2):351–360, April 1979.

R.E. Shostak, R. Schwartz, and P.M. Melliar-Smith. STP: A mechanized logic for specification and verification. In D. Loveland, editor, *6th International Conference on Automated Deduction (CADE)*, volume 138 of *Lecture Notes in Computer Science*, New York, NY, 1982. Springer-Verlag.

Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a SAT-solver. In Warren A. Hunt, Jr. and Steven D. Johnson, editors, *Formal Methods in Computer-Aided Design (FMCAD 2000)*, volume 1954 of *Lecture Notes in Computer Science*, pages 108–125, Austin, TX, November 2000. Springer-Verlag.

J. Siekmann and G. Wrightson, editors. *Automation of Reasoning: Classical Papers on Computational Logic, Volumes 1 & 2*. Springer-Verlag, 1983.

J. van Heijenoort, editor. *From Frege to Gödel: A Sourcebook of Mathematical Logic, 1879–1931*. Harvard University Press, Cambridge, MA, 1967.

Richard Zach Completeness before Post: Bernays, Hilbert, and the development of propositional logic. *Bulletin of Symbolic Logic*, 5:331–366, 1999.

Hantao Zhang. SATO: An efficient propositional prover. In *Conference on Automated Deduction*, pages 272–275, 1997.

Lintao Zhang. *Searching for Truth: Techniques for Satisfiability of Boolean Formulas*. PhD thesis, Princeton University, 2003.

L. Zhang and S. Malik. The quest for efficient boolean satisfiability solvers. In A. Voronkov, editor, *Proceedings of CADE-19*, Berlin, Germany, 2002. Springer-Verlag.

Compositional Reactive Semantics of SystemC and Verification with RuleBase

Rudrapatna K. Shyamasundar, Frederic Doucet, Rajesh K. Gupta, and Ingolf H. Krüger

Abstract We present a behavioral semantics of SystemC that succinctly captures its reactive features, clock and time references, macro- and micro-time model, and allows the specification of a network of synchronous and asynchronous components communicating through either high-level transactions or low-level signal and event communications. The proposed semantic framework demonstrates the anomalies introduced by the simulation kernel, in spite of the macro- and micro-time scales. The framework further relates the simulation and logical correctness and provides a technique for scaling up the verification while keeping the correctness intact. Furthermore, we translate SystemC components to RuleBase using our semantic characterization that permits testing and verification of heterogenous designs. We illustrate the verification of a Central Locking System (CLS) designed in SystemC.

Keywords: SystemC, semantics, verification, model checking

1 Introduction

System-level modeling using SystemC facilitates the use of various features and concepts such as perfect synchrony, asynchrony, reactive, and time specifications through a C++ class library. SystemC provides a bridge between hardware and software design and thus, provides a unifying framework for hardware/software design. SystemC consists of C++ libraries and a simulation kernel for creating behavioral and register-transfer level (RTL) designs. It provides a common development environment needed to support software engineers working in C/C++, and hardware engineers working in HDLs such as VHDL, Verilog, etc., particularly system-on-a-chip designs. While the simulation behavior of a SystemC description is well understood by engineers, existing frameworks have not catered to a comparative evaluation of simulation and verification particularly to the various perfect-synchrony features as well as macro- and micro-time scales. We also show that the two time scales, while intended to avoid some of the anomalous behaviors possible in perfectly synchronous languages like Esterel, cannot indeed be avoided. In fact, it also throws open the question whether the δ-cycle can indeed be avoided to speed up simulation without foregoing correctness.

S. Ramesh and P. Sampath (eds.), *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*, 227–243.

In this paper, we describe a compositional semantics using the rewrite framework of Esterel. A sound semantic model provides the ability to reason regarding issues with composition of SystemC models without adding global restrictions other than those imposed by SystemC language itself. With the ever increasing of complexity of systems, it is important to exploit the notion of compositionality that is deeply embedded in the rationale of SystemC. A clean separation of process reaction, and computing the next environment provides a basis for simulation and verification without flattening the composed model into a single uniform model. The main contributions of the paper are:

1. a semantic foundation that captures (i) the synchronous and asynchronous process composition, (ii) all levels of abstractions for communications, and (iii) relation between simulation correctness and logical correctness;
2. a way to scale up the verification while ensuring the simulation and logical correctness and equivalence are intact; and
3. an automatic translation to the model checkers for verification of SystemC components, providing a powerful workbench for testing and verification.

2 Overview of SystemC

SystemC is essentially a C++ library that provides macros to model hardware and software systems. The difference between a system-level modeling language rooted in C++ such as SystemC, and C++ itself, is that the system-level modeling macros are used to model a system, but not to implement it. Figure 1 shows an abstract syntax for SystemC, where we consider only the statements specific to modeling with SystemC and not the general statements in the C++ language. A SystemC program is a set of interconnected modules communicating through channels, signals, events, and shared variables. A module is composed of a set of ports, variables, a process and a set of methods. A process is sensitive to a set of events, and optionally can have

```
program       := { modules, channels, signals, events, variables }
module        := { ports, variables, process-decl, process-body, methods }
process-decl  := <process name> <sensitivity> <reset-condition>
process-body  := <event-comm | signal-comm | chan-comm | control-flow |
                          arithmetic>*
event-comm    := wait(event) | wait(event,time) |
                 wait(time) | wait(event_list)
                 notify(event) | notify-delayed(event) |
signal-comm   := signal.read | signal.write |
chan-comm     := tlm_port->put(value) | tlm_port->get(var) |
                 tlm_port->method(parameters)
control-flow  := <C++ control flow>
arithmetic    := <C++ arithmetic>
```

Fig. 1 Simplified abstract syntax for SystemC.

a reset condition. Some of the distinctive characteristics are informally summarized below:

- A process is in the ready state when either the SystemC program starts or there is an event that the process is waiting for. A process is in the waiting state when it is waiting for an event. A process is unblocked when it is notified by an event. Events can be notified immediately, or the notification can be delayed until all processes are waiting.
- Time is modeled through macro-time in some pre-defined quantifiable unit; a process waits for a given amount of time, expiration of which is notified through an event.
- Between processes, the basic communication is by reading and writing signals. During the execution of a SystemC program, all signal values are stable until all processes reach the waiting state. When all processes are waiting, signals are updated with the new values.
- Transaction-level communications are through channels, which are accessed using an interface defined by a set of methods. The transaction-level model (TLM) interface can be put and get methods, to connect to channels like FIFO buffers, etc., or a custom set of methods to connect to specific channels. In the body of the methods, communication is done by using the shared variables, events, signals or other channels defined in the channel.

When executing a SystemC program, the illusion of concurrency is provided to the user by a simulation kernel implementing a discrete event simulation loop. The simulation loop divides time into macro-time and micro-time, where micro-time is used for creating a partial ordering of the events that can occur in a macro-time unit. Micro-time events are called delta events, processed into the simulation queue, advancing micro-time as needed without advancing macro-time (to simulate synchronous reactions). Micro-time events are not observable in a macro-time scale because they are used only to simulate a synchronous concurrent reaction on a sequential computer.

The discrete-event simulation loop is used to compute the next environment which contains the events to which the processes synchronize. The simulation kernel first synchronizes all the processes with immediate events, and then picks one process to react. This reaction loop is repeated until there are no more immediate events and all processes are waiting. Then, the processes are synchronized with micro-time events, followed by a new reaction loop. When there are no more immediate nor micro-time events, the processes are synchronized with the macro-time events, leading to another reaction loop.

Note that simulation cannot guarantee correctness due to:

1. the inability of simulation to produce all possible behaviors, and
2. the simulation loop can introduce anomalous behaviors that cannot happen logically. For example, due to the underlying scheduler (as we discuss in Section 4), the simulation kernel can introduce nondeterminism and causality cycles in a design description.

3 Semantic Framework

We now define the behavioral semantics of SystemC compositionally to capture all possible behaviors that can be computed by a SystemC program by composing the semantics of its components. First, we divide the observables as controllable variables and environment variables. For a given SystemC module, the controllable variables are output signals, internal variables, output channels, output events, and the program counter for the process. The environment variables are input signals, input events, input channels, and global variables.

At any point during the program, there is at most one process that is reacting to the environment. One can locally visualize instants during which reactions occur by observing the state (C++ variables and program counters for each processes) of the program, denoted σ, or the modeling environment (events, channels, signals, processes, etc.), denoted E. An environment only lasts an instant; i.e., it is not persistent like the state and an event occurs only right after the instant it is emitted. For describing, how a statement changes the configurations of the observables, we use rewrite rules of the form $(\langle stmt \rangle, \sigma) \xrightarrow[E]{E_O, b} (\langle stmt' \rangle, \sigma')$ where:

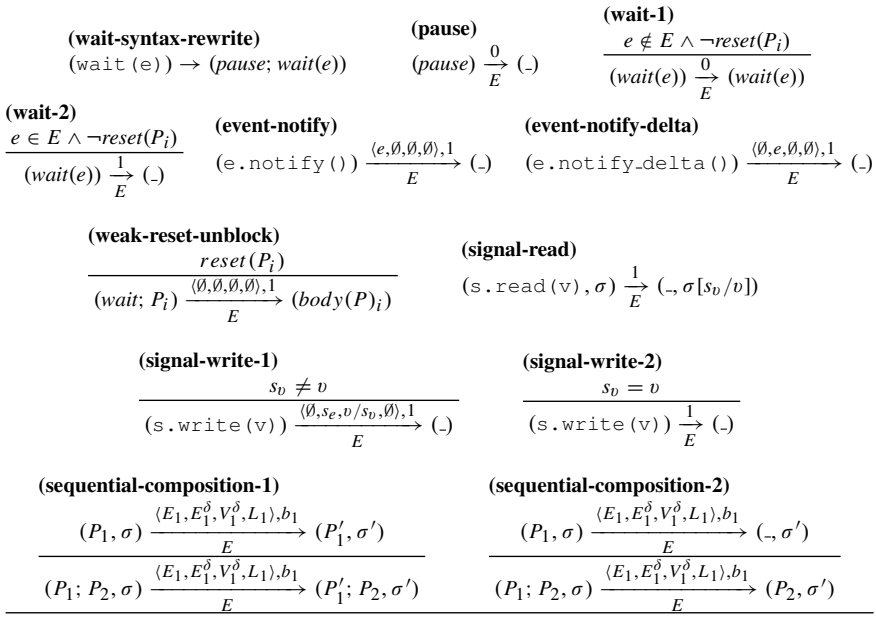
- $stmt$ is a SystemC program text with the location of the program counter, before the reaction, and $stmt'$ is the program text with the location of the program counter after the transition,
- σ and σ' are the states before and after the reaction respectively,
- E is the environment while taking the transition, E_O is the output emitted during the transition; in general, an environment is a 4-tuple $E = \langle E^I, E^\delta, V^\delta, L \rangle$ where:
  - E^I is the set of immediate events,
  - E^δ is the set of next delta events,
  - V^δ is the set of next delta updates for variable,
  - L is a set of pending transactions or pending asynchronous tasks,
- b is a Boolean flag indicating if the process completed in the instant or not.

Keeping in view the simulation explained above, a SystemC model behaves in an alternating sequence of synchronizations and reactions $(\rightarrow_{sync} \rightarrow_{react})^*$), observable as a sequence of environments and states $(E_0\sigma_0)(E_1\sigma_1)(E_2\sigma_2)\ldots$.

3.1 Reactive Statements

The reactive semantics forms the crux of the simulation. Some of the rules are given Table 1. The wait-syntactic rule defines that a `wait` statement is syntactically reduced to the sequence of a *pause* statement followed by a semantic *wait* statement. The wait argument `e` is passed verbatim through the reduction. A *pause* statement pauses a reaction until the next environment – it does not terminate in the current instant and reduces to nothing. The behavior of the semantic *wait* statement is to wait

Table 1 Semantics of reactive statements.



for an event e to be in the environment. Rule wait-1 defines that when event e is not in the environment and the reset condition specific to the current process (defined in by variable P_i) is false, the process continues to wait without doing anything. In rule wait-2, when event e is present in the environment and the reset is not asserted, the wait statement terminates and reduces to nothing.

The event notification statement immediately emits an event e in the next environment, and terminates. The delayed notification statement emits event e to be in the next delta environment. The processes waiting on these events will unblock in either the synchronization with the next environment and the synchronization with the next delta environment respectively. The weak-reset-unblock rule shows that when a process is waiting for some event and the reset variable is asserted, the process resets to the initial value for its program counter.

Now let us look at statements concerned with signal communications. A SystemC signal s is persistent and is associated with a variable s_v which holds the data value, and to an event s_e to notify signal value transitions. For a signal write operation, if the value v being written to a signal is different than the current value, the statement terminates, reduces to nothing, and emits event s_e in E^δ and v/s_v in V^δ. Otherwise the statement terminates without doing anything. Finally, there are two cases for sequential composition. If statement P_1 does not terminate in the current instant, then P_2 cannot start. If P_1 terminates then P_2 starts in the environment in which P_1 terminates.

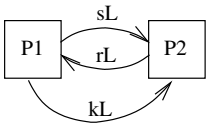


Fig. 2 Asynchrony interface.

Table 2 Semantics of timed statements.

(wait-timeout)

$$(\texttt{wait}(t)) \xrightarrow[E]{\langle sL_{P_i}(t),\emptyset,\emptyset,L_{P_i}\rangle,0} (wait(rL_{P_i}))$$

(wait-event-timeout)

$$(\texttt{wait}(t,e)) \xrightarrow[E]{\langle sL_{P_i}(t),\emptyset,\emptyset,L_{P_i}\rangle,0} (wait(rL_{P_i}|e))$$

(wait-timeout-event-1)

$$\frac{(rL_{P_i} \in E) \wedge \neg reset(P_i)}{(wait(rL_{P_i}|e)) \xrightarrow[E]{1} (\_)}$$

(wait-timeout-event-2)

$$\frac{(rL_{P_i} \notin E) \wedge (e \in E) \wedge \neg reset(P_i)}{(wait(rL_{P_i}|e)) \xrightarrow[E]{\langle kL_{P_i},\emptyset,\emptyset,\emptyset\rangle,1} (\_)}$$

3.2 Statements for Time

Wait-timeout statements are used to request a notification at a later macro-time. For this purpose, we shall assume the presence of an asynchronous timer process in the environment as in CRP [2], that can be called from any process in need of setting an alarm.

Figure 2 shows the asynchronous gateway interface. The timer is an asynchronous task, which is started and controlled, indirectly, by the process. The timer has the general interface of the asynchronous task which is described as follows. The asynchronous task is started with an event sL, and the completion of the asynchronous task is notified with event rL. Event kL is used to kill an asynchronous task. The set L contains the labels of all the currently active asynchronous tasks.

The semantic functions for the timed statements are given in Table 2. The wait-timeout statement requests an alarm after t units of time by sending t on signal sL_{P_i} to a timer L_{P_i} in the environment, with, P_i being the label for the current process. The process then proceeds to wait for an event rL_{P_i} which is to be sent by the timer after t time units. A process can also wait for an event e, with a timeout t, as showed in rule wait-event-timeout. If event e occurs before the time out (before receiving event rL_{P_i}), the process will resume and kill the pending timer by notifying event kL_{P_i}. It is necessary to kill the pending timer so that, after time t the process will not receive any unnecessary timeout event.

3.3 Rules for Parallel Composition

In SystemC, the parallel composition of the processes is defined as each module is instantiated. All modules are to be executed concurrently once the simulation is

Table 3 Semantics of parallel composition.

(sync-imm)

$$\frac{\forall i \in \{1..l\} : \exists e \in E^I : waiting(P_i, e) \qquad \forall j \in \{l+1..m\} : \forall e \in E^I : \neg waiting(P_j, e)}{(P_1 \| ... \| P_l \| P_{l+1} \| ... \| P_m) \xrightarrow[\langle E^I, E^\delta, V^\delta, L\rangle]{\langle \emptyset, E^\delta, V^\delta, L\rangle, 1}_I (P_1' \| ... \| P_l' \| P_{l+1} \| ... \| P_m)}$$

(async-react)

$$\frac{\begin{array}{c}\forall i \in \{1...l\} : waiting(P_i) \qquad \forall j \in \{l+1..m\} : ready(P_j) \\ select\ x \in \{l+1..m\} : (P_x, \sigma) \xrightarrow[E]{\langle E_x, E_x^\delta, V_x^\delta, L_x\rangle, 0} (P_x', \sigma') \\ merge(\langle E_x^\delta, E^\delta\rangle, \langle V_x^\delta, V^\delta\rangle, 1)\end{array}}{(P_1 \| ... \| P_l \| P_{l+1} \| ... \| P_m) \xrightarrow[\langle E^I, E^\delta, V^\delta, L\rangle]{\langle E_x, E_x^\delta \cup E^\delta, V_x^\delta \cup V^\delta, L_s \cup L\rangle, 1} (P_1' \| ... \| P_l' \| P_{l+1} \| ... \| P_x' \| ... \| P_m)}$$

(sync-micro)

$$\frac{\forall i \in \{1..n\} : waiting(P_i)}{(P_1 \| ... \| P_n, \sigma) \xrightarrow[\langle \emptyset, E^\delta, V^\delta, L\rangle]{\langle E^\delta, \emptyset, \emptyset, L\rangle, 1}_\delta (P_1 \| ... \| P_n, \sigma[V^\delta / V])}$$

(sync-macro)

$$\frac{\forall i \in \{1..n\} : waiting(P_i) \qquad e_t = next\_time()}{(P_1 \| ... \| P_n, \sigma) \xrightarrow[\langle \emptyset, \emptyset, \emptyset, L\rangle]{\langle e_t, \emptyset, \emptyset, L\rangle, 1}_T (P_1 \| ... \| P_n, \sigma[V^\delta / V])}$$

started. The various booking operations for building the environment can be partitioned as:

1. synchronizing processes with the events in the environment (denoted $\rightarrow_I$),
2. reaction of the selected process (denoted $\rightarrow$),
3. building next micro-environment (denoted $\rightarrow_\delta$), and
4. building next macro-environment (denoted $\rightarrow_T$).

The complete simulation loop can then be captured as iterative composition of relations given by: $((\rightarrow_I \rightarrow)^* \rightarrow_\delta)^* \rightarrow_T)^*$.

The various semantic rules of composition are given in Table 3. Rule sync-imm is defined to unblock all processes that are waiting for events that are in the environment. We use the notation $waiting(P, e)$ to mean that P is waiting on event e, meaning P is of the form $wait; P'$. In other words, it is a synchronous composition, but only for the wait statements.

Rule asynch-react defines the reactivity. A process which is ready, is selected to run until it reaches the next pause. The merge predicate provides a check on whether or not to allow nondeterministic environments in the composition. Nondeterministic environment are possible when two different values can be written to a signal in the same reaction. The $merge$ predicate checks the feasibility of partially ordering the events in the delta cycle. Setting the third parameter to "1" indicates that the partial

order has to be consistent. One can allow nondeterministic environment by setting the third parameter to -1.

Rule sync-micro defines the synchronization on delta events to build the next micro-environment. The rule proceeds only when there is no immediate events and there exists some delta events. The transition makes the delta events in E^{δ} become the immediate events in the next instant, and updates the state variables.

The rule for the synchronization on timed events builds the next environment from time events and advance macro-time. It is effective when all processes are blocked, where there are no immediate event nor delta event. Timed events are posted by `wait(time)` statements, timers and clocks. For simplicity in this rules, we use $next\_time()$ to broadly indicate moving to the next time.

3.4 Statements for Transaction-Level Modeling

For transaction-level method calls, we simply inline the body of the method inside the caller module. The put/get transaction-level channels from the SystemC TLM library, when used as a single place buffer in a point-to-point connection, do not cause nondeterministic behaviors. This is because the state changes in the transaction-level buffers are visible immediately for the calling process, but only at the next delta cycle for the other processes. This behavior is useful to avoid the kind of nondeterministic behavior described in Section 4. Furthermore, using these channels in combination with the step scheduler (described in Section 5) enables efficient transaction-level verification.

Table 4 lists the semantic rules for the TLM statements. We consider only the rules for communication with single place TLM FIFO buffers. Rule tlm-put writes data on the buffer if the buffer is empty. Otherwise, it waits that the data already on the buffer is read. Rule tlm-get works similarly in the complementary fashion. Note that these rules are to be used only with the step scheduler as their generalization for the full scheduler would significantly complicate the semantics and require extra copies of the variables in the verification environment (see Section 5).

Table 4 Semantics of transaction-level statements.

(tlm-put-1)

$$\frac{\neg full(f)}{(\texttt{f->put(v)}, \sigma) \xrightarrow[E]{\langle \emptyset, f_w, \emptyset, L\rangle, 1} (\_, \sigma[v/put(f)])}$$

(tlm-put-2)

$$\frac{full(f)}{(\texttt{f->put(v)}) \xrightarrow[E]{0} (wait(f_r); \texttt{f->put(v)})}$$

(tlm-get-1)

$$\frac{\neg empty(f)}{(\texttt{f->get(v)}, \sigma) \xrightarrow[E]{\langle \emptyset, f_r, \emptyset, L\rangle, 1} (\_, \sigma[get(f)/v])}$$

(tlm-get-2)

$$\frac{empty(f)}{(\texttt{f->get(v)}) \xrightarrow[E]{0} (wait(f_w); \texttt{f->get(v)})}$$

3.5 Computing the Semantics of SystemC Components

We now show how we generate the transition system for a SystemC component by applying the semantic rules. For a process P, the transformation yields the reactive sequences from initial state σ_0 to state σ_n such as:

$$(stmt, \sigma_0) \xrightarrow[E]{E_0,1} \ldots \xrightarrow[E]{E_{n-1},0} (stmt_n, \sigma_n)$$

where from a given state, the process will react until the next wait statement (or up until termination). During the reaction, the states in the sequence between σ_0 and σ_n are observable only from within P, and no other process in the environment can observe the intermediate states. Hence, from another SystemC process, only the first and last states of the reaction are observable:

$$(stmt, \sigma_0) \xrightarrow[E]{E_O,0} (stmt_n, \sigma_n)$$

Therefore, when building the transition system for a process, we can reduce a detailed graph to an observable graph with all the intermediate transitions, form an initial state σ_0 to a state σ_n, collapsed to one single transition.

From the process description, we use the semantic rules to construct the control-flow graph, and add the predicate for the conditions and assignments. Since we do not keep the state implicitly in the semantic structure, we use the weakest precondition on observable paths to convert the control-flow graph to a graph with only the observable reactive steps. We use the standard definitions for constructively computing weakest precondition. For all the pair of observable points (paths from a pause statement to the next pause statement in the graph) we compute the weakest precondition between the points, and add it to the observable graph if the weakest precondition is satisfiable. Note that one cannot constructively compute the weakest precondition for loops that cannot be unrolled. This is a well known limitation of the approach, but we can alleviate it by requiring wait statements inside loop bodies.

Figure 3 shows an example of the semantic translation of a SystemC process. The process is first converted to the control-flow graph, and then to the observable graph. Every transition is labeled with a guard (the conjunction of the labels starting with a G), and the variable assignments should the transition be taken. The process initializes a counter variable to 0, waits on the clock, re-initialize it to 10, and then counts up until it is reset.

4 Anomalous Behaviors

We illustrate various anomalous behaviors such as causality (as in Esterel), non-determinism, etc.

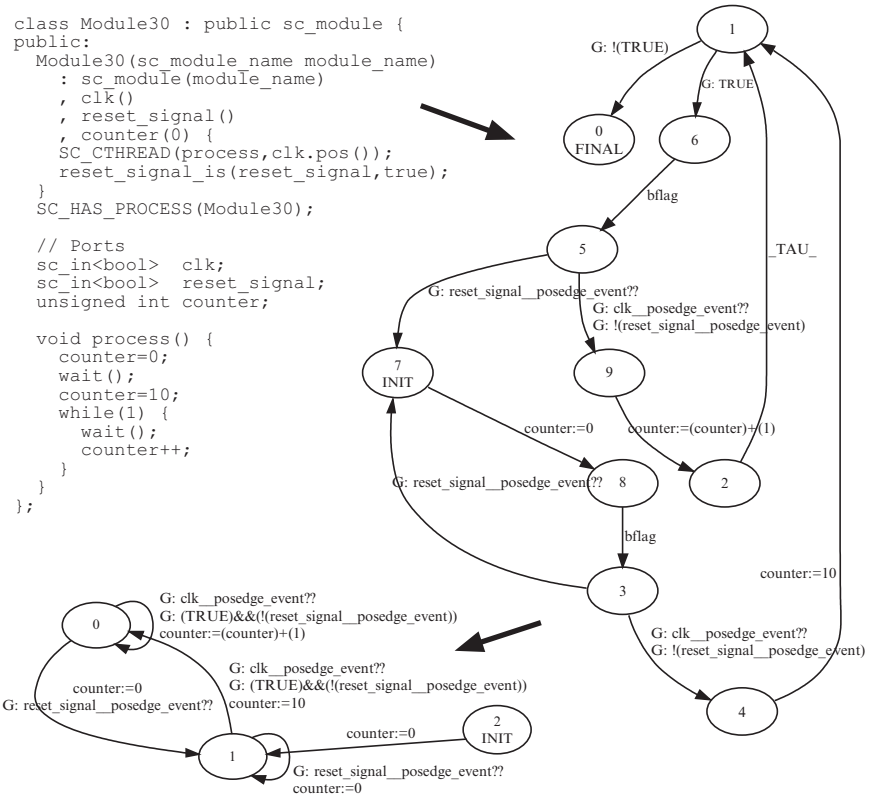


Fig. 3 Example of semantic translation.

4.1 Causality Cycle

A causality cycle is a behavior that triggers an infinite amount of action in a finite amount of time. It is important to look for causality cycles in a SystemC design as causality cycles are not always triggered in simulation. Indeed, a causality cycle can be triggered by a corner case condition in the behavior of the composition of a system of asynchronous components. In a simulation, one can observe a causality cycle when a computation does not stabilize to specific output values in an instant and keeps re-triggering itself. In our semantics rules, a causality cycle occurs when it never gets to the next synchronization with the macro-time events. Indeed, the causal cycle occurs in the reaction $(((\rightarrow_I \rightarrow)^* \rightarrow_\delta)^* \rightarrow_T)^*$ when step $\rightarrow_T$ is never taken.

We now give an example of a SystemC program that has a causality cycle. The component definition for a "watching process" is given in Figure 4(a). The behavior of the watching process is to watch another module, and if the other module has not been triggered, the watching process will trigger it. The component by itself does not

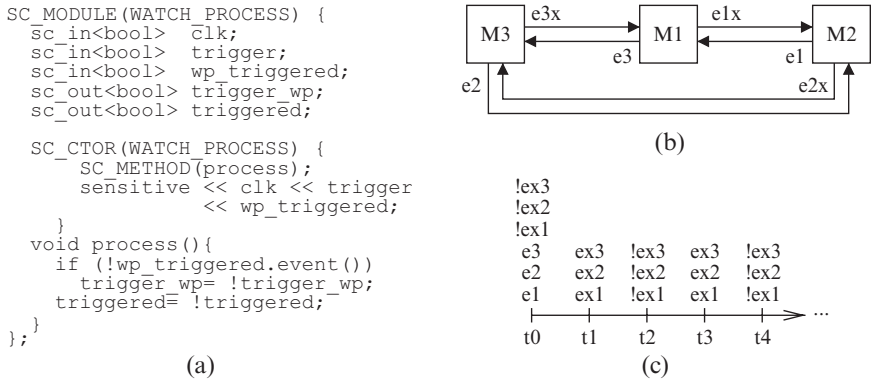


Fig. 4 Example of causality cycle: (a) watching component (b) three such components watching each others, and (c) depiction of the causality cycle.

have a causality cycle. However, if one builds a system where instances of watching processes in are connected in a cycle such as depicted in Figure 4(b), there will be a causality cycle. When executing the program, the system will enter a causality cycle as depicted in Figure 4(c). The reason is that the modules will keep triggering each other.

Note that causal loop can sometimes be desirable when used to produce an oscillating behavior (i.e., generate a clock). For instance, a SystemC untimed model never takes the $\rightarrow_T$ step by construction. We find a causality loop by searching for a loop in the state graph where the next time is never reached. We use the model checker to verify that, for a given design, it is always possible to reach the next clock – thus no divergence.

4.2 Nondeterminism

None of the SystemC syntactic constructs are meant to explicitly produce nondeterministic behavior, in the sense of a nondeterministic choice operator. A program is nondeterministic if, for a given input, it is possible to observe multiple different output behaviors. Causes of nondeterministic behavior in SystemC can be communications with shared variables, immediate event notification, or using uninitialized signals and variables. Nondeterministic behavior may not be observable in the SystemC simulation of a program because the output behavior is decided by the implicit process selection priorities in the scheduler.

We now give an example of a nondeterministic SystemC program, depicted in Figure 5. The program is composed of two modules which communicate through an event e. The first module notifies the second module, and the second module terminates the simulation upon reception of the event notification. To effectively receive

```
sc_event e;

SC_MODULE(M1) {
  SC_CTOR(M1) {
    SC_THREAD(a);
  }
  void p1() {
    e.notify();
  }
};
```

```
SC_MODULE(M2) {
  SC_CTOR(M2) {
    SC_THREAD(b);
  }
  void p2() {
    wait(e);
    sc_stop();
  }
};
```

```
int sc_main(int argc,
  char* argv[]) {
  M1 m1("m1");
  M2 m2("m2");
  //  M2 m2("m2");
  //M1 m1("m1");

  sc_start(10);
  return 1;
};
```

Fig. 5 SystemC code whose behavior is dependant on the scheduler.

```
SC_MODULE(Buffer) {
  bool data_avail;
  int  data_value;

  SC_CTOR(Buffer) {
    data_avail= false;
  }

  void read(int& val) {
    while(!data_avail)
       wait();
    val= data_value;
    data_avail=false;
  }

  void write(int val) {
    while(data_avail)
      wait();
    data_value= val;
    data_avail=true;
  }

  bool peek_data(int& val) {
    if (data_avail) {
      val= data_value;
      return true;
    } else
      return false;
  }
};
```

Fig. 6 Definition of a buffer which can cause nondeterministic behavior.

the event notification, process p2 must wait on the event before the event is notified. If the notification is done before p2 runs, then the event notification will be lost. The behavior of this program is dependent on the scheduler because the scheduler will decide which process will run first. If process p1 is run before process p2, then event *e* will be missed by p2. With the reference implementation of the SystemC simulation kernel, the initial process triggering schedule is determined by the order into which the modules are instantiated. If we permute the order of instantiation (as commented), then the problem is not observable as process p2 will be waiting for the event, and will terminate the simulation. An example of a nondeterministic TLM buffer using shared variables is also illustrated in Figure 6.

5 Verification Framework

For a given SystemC program, we automatically build an SMV description that allows verification of desired properties. We have built the prototype for both IBM RuleBase and NuSMV. We do not build a global transition system explicitly, but rather we only translate the modules and use the model checker to compose the environment model with the transition systems of each module (process). We automate

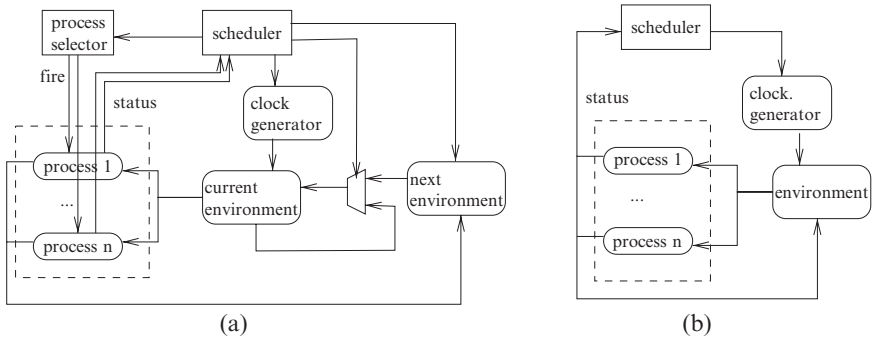


Fig. 7 Example of verification setup: (a) full scheduler and (b) step scheduler.

the composition rules by building an environment that manages the events and fires the processes. For this purpose, it is possible to use two different schedulers:

1. Full scheduler: fires the process one by one and compute all the possible interleavings and synchronizations.
2. Step scheduler: fire all processes synchronously and synchronize them at the same time on the delta events (no immediate events).

Figure 7 depicts the difference between the schedulers. The full scheduler captures the effect of nondeterminism in shared variable communications, while the step scheduler does not. But the good thing is that the verification algorithms are significantly more efficient with the step scheduler because there is no process selection and it does not require duplicating the variables in the environment. However, using the step scheduler introduces the following restrictions on the design. First, nondeterminism, immediate notifications are not allowed. Second, there can be no interference on shared variable communication: this means that, at every cycle, only one process can write to a shared variable.

It happens that a large family of SystemC program satisfy these conditions. This is because these restrictions match the traditional synchronous semantics, to which all RTL models and many TLM models are built. Since the size of the state space is exponential to the number of variables this has the potential for significant performance improvements.

6 An Example: Central Locking System

We illustrate our approach by verifying the transaction-level model of a the design of a design of a Central Locking System (CLS), a system used to lock and unlock the doors, and validate who can perform these operation and when. We consider it to be an interesting example since it is distributed over many components in the car, and

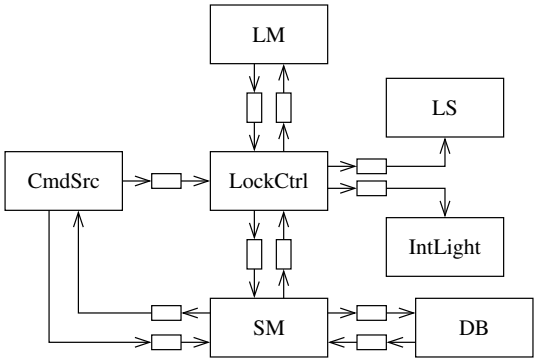


Fig. 8 Structural specification for the Central Locking System.

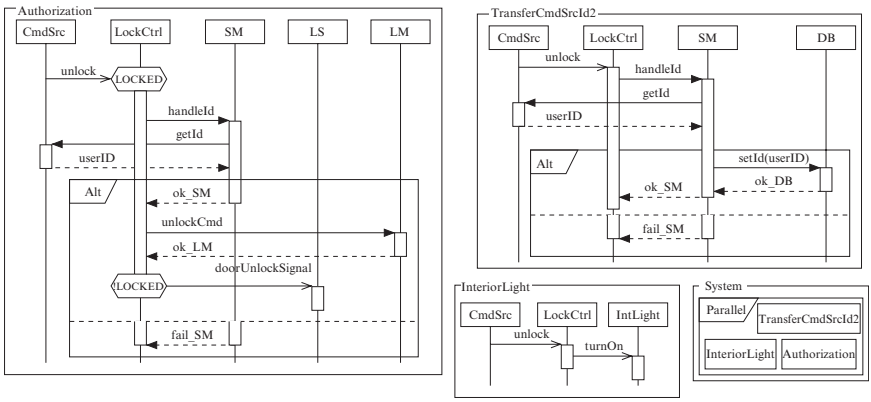


Fig. 9 Behavioral specification for the Central Locking System.

one has to be careful to avoid synchronization or deadlock problems. The structural specification of the CLS is illustrated in Figure 8 and the behavioral specification of the CLS is depicted in Figure 9. The high-level system specification describes the system as being the parallel composition of the three basic services illustrated in the interaction diagrams.

We implemented the CLS with the SystemC language and we replicated the local structure of the system into the component structure. Table 5 shows the verification statistics for the verification of the SystemC implementation of the CLS against the interaction specifications. The data domain for the keys has been reduced from undefined integer range from -1 to 9, which corresponds to the values transmitted between the modules.

Table 5 Verification of SystemC CLS with the services.

	NuSMV 2.4.1		RuleBase	
Property	*Time (s)*	*Memory Used*	*Time (s)*	*Memory Used*
No causal loops	2200.234	47172 K	632	n/a
All services	2031.379	46880 K	218	743 MB
TransferKeyId2 service	418.482	19832K	136	480 MB
IntLight service	185.940	18064 K	178	288 MB
Authorization service	500.835	19456 K	142	447 MB

The verification runs are much faster using RuleBase but require more memory. The first entry is for the verification of the conjunction of all services together, while the next three entries are for the verification of the individual services. All the runs are with the cone of influence reduction and with dynamic BDD variable re-ordering. Using NuSMV, we always verify that the FSM has fair paths and has no deadlocks, increasing the verification times.

7 Related Work

Our approach incorporates the reactive synchronous features of SystemC succinctly and distinguishes from other works and further, it is based on frameworks proposed in [1], [2], and [8].

Recently, there has been two interesting approaches based on synchronous languages. The first one, advocated in [6], is based on deriving the transactional-level models of SystemC into a automata encoded in Lustre, and refining the channel interfaces through a protocol automata. Subsequently, it uses the Lustre model checking tools for analysis. While the approach is nice, it carries over the full delta-cycle details which compromises the scalability of the verification. We improve on their results by defining the TLM abstraction by restricting the design and mapping the semantics to a synchronous transition system ala Esterel, which enables us to improve on the scalability. The approach in [9] describes how to translate the body of a SystemC process into a set of Signal equations, and synchronously compose all the equations into the transition system. The problem with this work is that it ignores the simulation anomalies, timing statements and the TLM statements. The work in [5] uses a process algebra but ignores the need of distinctions between synchronous and asynchronous compositions, ignores δ cycles, etc.

Approaches such as [3, 7] translate SystemC program simulation into ASML units of compiled code. While it appeals as a nice global executional simulation system, the reactive features of the SystemC such as reset (that has priority as well), broadcast to waiting processes, cannot be succinctly captured in synchronous fashion like an Esterel transition system. Indeed, by going to the ASML, one does not have access to the nice results of the synchronous community and it is also unclear how one can

exploit the power of modern model checkers. Our translation is similar to the work of [4] that similarly generates SMV transition relations for SystemC code. Our work fully uses the reactive rules and with the addition of TLM abstractions and uses the step semantics.

8 Summary and Conclusions

In this paper, we have presented a compositional reactive semantics for SystemC that captures both at signal and TLM levels of abstractions. We are able to reduce the design without anomalies to pure synchronous transition systems to exploit the full power of the SMV-based verifiers. Further, the framework provides techniques to detect anomalies and enable relating simulation to logical correctness. It also has brought to light how verification can be speeded up without foregoing correctness. Another interesting question that has been brought to light is the need to arrive at quantitative/qualitative comparisons of δ-cycles of SystemC with respect to the constructive semantics of Esterel. Our system translates a given SystemC program into RuleBase – industrial strength verifier. This allows a variety of design integrations and exploits the power of industrial strength model checkers. Further, our results improve on previous published results and we are now able to verify TLM within some reasonable (scalable) times, while enjoying all the capabilities of the modern model checkers. The work needs to enriched to take into account dynamic memory allocations, exceptions, custom channels, etc. Furthermore, the computation of the weakest precondition can be inefficient and yield unnecessary overhead for medium and large processes.

The focus for our future work is to generalize the methodologies to support custom channels, develop compositional design methodologies, and provide appealing abstractions for application-level transactions and compositions of transactions.

Acknowledgements Thanks go to Mr. Saurabh Joshi, IBM India Research Lab, for various experiments with the system developed for RuleBase.

References

1. G. Berry. *The Foundations of Esterel*. MIT Press, 2000.
2. G. Berry, S. Ramesh, and R.K. Shyamasundar. Communicating Reactive Processes. In *Proc. Symp. on Principles of Programming Languages*, 1993.
3. A. Habibi and S. Tahar. Design and Verification of SystemC Transaction-level Models. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 14(1):57–68, January 2006.
4. D. Kroening and N. Sharygina. Formal Verification of SystemC by Automatic Hardware/ Software Partitioning. In *Proc. of Formal Methods and Models for Codesign*, 2005.
5. K.L. Man. SystemC^{FL}: Formalization of SystemC. In *Proc. of 12th IEEE Mediterranean Electrotechnical Conference*, 2004.

6. M. Moy, F. Maraninchi, and L. Maillet-Contoz. LusSy: A Toolbox for the Analysis of Systems-on-a-Chip at the Transactional Level. In *Proc. of Application of Concurrency to System Design*, 2005.
7. W. Mueller, J. Ruf, D. Hofmann, J. Gerlach, T. Kropf, and W. Rosenstiel. The Simulation Semantics of SystemC. In *Proc. Design Automation and Test in Europe Conf.*, 2001.
8. B. Rajan and R.K. Shyamasundar. Multiclock Esterel: A Reactive Framework for Asynchronous Design. In *Proc. of 13th Intl. Conf. on VLSI Design*, 2000.
9. J.-P. Talpin, D. Berner, P. Le Guernic, A. Gamatie, S. Shukla, and R. Gupta. A Behavioural Type Inference System for Compositional System-on-Chip Design. In *Proc. of Application of Concurrency to System Design*, 2004.

PSL: Beyond Hardware Verification

Ziv Glazberg, Mark Moulin, Avigail Orni, Sitvanit Ruah, and Emmanuel Zarpas

Abstract In recent years, the language PSL (Property Specification Language, a.k.a. IEEE P1850) has been embraced and put to successful use by chip design/verification engineers across the electronics industry. While PSL is mainly used for hardware verification, it can, in fact, be used to verify a wide variety of systems, including missile interception systems, railway interlocking protocols, system automation policies, and even business processes. We discuss and exemplify how PSL can be used as a general purpose language for the specification of models and properties, beyond hardware systems.

Keywords: PSL, nonlinear controllers, concurrent reactive systems, policy-based system automation.

1 Introduction

Since its approval by the IEEE, the Property Specification Language (PSL IEEE P1850, [21]) has met with huge success in the hardware verification community. It is widely used for industrial hardware verification and is supported by a wide range of vendors. PSL is mainly considered a hardware specification language; however, its use is not restricted to hardware verification. While some features such as clocking are close to hardware, on the whole PSL is a general property specification language.

The goal of this paper is to illustrate how PSL can be used outside the hardware verification scope. Section 2 describes PSL. Section 3 focuses on the use of PSL for the modeling and verification of antimissile interception hybrid control system. Section 4 describes our work in statically checking the output of IBM Rational Rose Real-Time (RoseRT), a widely used model-driven development tool for concurrent reactive systems. Section 5 deals with the modeling and verification of Tivoli System Automation policies.

2 Property Specification Language (PSL)

PSL, the Property Specification Language, is a language for specifying properties. It is typically used for specifying temporal properties of systems, i.e., properties that deal with the behavior of a system over time. The main part of PSL is based on the

S. Ramesh and P. Sampath (eds.), *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*, 245–260.

temporal logic Linear Time Logic (LTL) [13], augmented with regular expressions. PSL originated as the Sugar language, which was used by the IBM RuleBase PE model checking tool [22], and later evolved into an IEEE standard.

This section presents a brief overview of PSL. Only a small selection of the language is shown here. A clear and comprehensive introduction to PSL can be found in [7]. The official definition of PSL is in IEEE Std. 1850-2005 [21].

2.1 Simple PSL Examples

We consider a system that accepts requests of some sort and processes them. The assumption is that our system has some definition of time points, which may be points at which a system clock ticks (if the system is synchronous), or points at which certain chosen events occur. PSL only requires that we have a sequence (finite or infinite) of discrete time points. (The notion of time in PSL is discussed more fully in the "The Granularity of Time" section.)

Our system has variables such as `req`, `ack`, `start`, `busy`, and `done`. Each variable is true at certain time points. We demonstrate how each of the following English statements, which describe system behavior, can be formulated in PSL.

- "Whenever `start` is true at a time point, `busy` will be true at the following time point."

```
always (start -> next busy)                    (1)
```

- "For every occurrence of `req` that is immediately followed by `ack`, processing of the acknowledged request begins at the next time point after the `ack`. The processing sequence begins with `start`, which is followed by `busy` for some number of time points, and ends with `done`."

```
{[*]; req; ack} |=> {start; busy[*]; done}     (2)
```

PSL is mathematically rigorous, therefore the properties in PSL are precise and unambiguous. However, they are also easy to read. Thus, a specification written in PSL can be used as input for automatic tools and may also serve as part of a human-readable specification document. In the following section, we present some PSL constructs and operators. Many of the PSL operators are based on LTL operators. Other PSL constructs are based on SEREs (discussed below), which are a type of regular expression. Another set of PSL operators is based on the CTL language [9] and is not discussed here.

2.2 SEREs – Regular Expressions in PSL

PSL includes a type of regular expression called a *SERE*, a *Sequential Extended Regular Expression*. SEREs are used to describe scenarios. The simplest type of SERE is a sequence of Boolean expressions separated by semicolons, such as

{req; !ack; ack}. This SERE describes a scenario spanning three time points, in which req holds at the first time point, ack does not hold at the second, and ack does hold at the third.

Generally, a SERE may describe a *set* of scenarios. For example, the operator [*] indicates an interval of zero or more time points, in which anything may occur. Therefore, the SERE {start;[*];done} describes any scenario that begins with start and ends with done. The [*] operator may also be attached to a Boolean expression. The expression busy[*] describes an interval of zero or more time points in which busy is true.

Additional operators serve as shorthand for longer constructions. For example, {busy[*4]} is equivalent to {busy; busy; busy; busy}. For any constant number n, the expression busy[*n] describes a sequence of exactly n time points, where busy holds for all the points. The expression req[=n] describes n occurrences of req, which may be non-consecutive.

SERE conjunction and disjunction operators create compound SEREs. The conjunction of two SEREs, using the && operator, describes two scenarios that happen simultaneously.

For example, in {start; busy[*]; done} && {cancel[=1]}
the left-hand side sub-SERE states that processing takes place (starting with start and ending with done). On the right-hand side, cancel occurs exactly once. In the conjunction, both sides happen simultaneously, so cancel occurs exactly once, at some time point, while processing is in progress.

The disjunction of two SEREs, using the | operator, describes a scenario in which either the left-hand side or the right-hand side of the disjunction (or both) must occur.

SERE operators may also be nested and combined, as shown:

```
{{busy[*]} && {cancel[=0]}} | {{req; ack}[*2]}
```

2.3 PSL Properties with SEREs

SEREs may be used as building blocks of PSL properties. Typically, a property may be composed of SEREs using the *suffix implication* operator |=>. For example

```
{[*]; req; ack} |=> {start; busy[*]; done}        (3)
```

This property states that any occurrence of the left-hand side scenario must be followed by an occurrence of the right-hand side scenario. In this particular case, {[*]; req; ack} describes a sequence of req followed immediately by ack, which may occur at any time point (due to the [*] at the beginning of the SERE). The property states that such a sequence must immediately be followed (starting at the next time point) by a scenario matching {start; busy[*]; done}. This property makes a requirement for any occurrence of a {req; ack} sequence,

at any time point, including overlapping occurrences. The following property is very similar to Formula 3:

```
{[*]; req; ack} |=> {start ;busy[*]; done}!        (4)
```

This property uses the *strong* version of the right-hand side SERE. Generally, a SERE is *strong* if it is followed by an exclamation point (!), and *weak* if it is not. The property in Formula 3 is satisfied by a right-hand side scenario in which done never occurs, and busy stays true until the end of the scenario. The property in Formula 4 is only matched by a scenario in which done eventually occurs.

2.4 Other Property Styles

A PSL property may be written without using SEREs at all. For example:

```
always (start -> next busy)                        (5)
```

The sub-property `(start -> next busy)` uses the logical implication operator `->`, which has the standard "if-then" meaning. The `next` operator refers to the time point that immediately follows the current one. So `(start -> next busy)` means, "if `start` is true at the current time point, then `busy` must be true at the next time point". Applying `always` to this sub-property means that the sub-property must hold at every time point. So the entire property means, "Whenever `start` is true, `busy` must be true at the time point that immediately follows".

The `eventually!` operator can be used to state that `start` must occur at some time point after `req` occurs (or simultaneously with `req`), as follows:

```
always (req -> eventually! start)                  (6)
```

In addition, we can combine non-SERE operators with SEREs, creating properties such as `(always {req; ack}|=> (start && eventually! done))`

2.5 PSL Layers and Flavors

PSL is structured in four layers: the Boolean layer, which contains the Boolean expressions used in properties; the temporal layer, which contains the temporal properties and SEREs; the verification layer, for directing the use of PSL by a tool; and the modeling layer, for modeling behavior of inputs and auxiliary variables.

Based on this layered structure, several flavors of PSL have been defined. The most generic is the GDL flavor, which is based on the General Description Language (GDL). GDL was designed especially for use in the PSL modeling layer, and can be used for modeling systems in diverse problem domains, at various levels of abstraction.

The other PSL flavors are based on hardware description languages (HDLs). In each flavor, the Boolean and modeling layers follow the syntax of the underlying HDL (or of GDL). The temporal and verification layers are not affected by flavors.

2.6 The Granularity of Time

Time in PSL is discrete; that is, time advances in pre-defined units. For example, the property `((size == 3) -> next (size > 5))` requires that if size equals 3 now, then at the next time point `(size > 5)` must hold. The time points are set by the system, that is, the system being verified has some function that advances by one time point.

Some PSL operators are not affected by the granularity of time. For example:

- `never (at_critical [1] && at_critical [2])` requires that `at_critical [1]` and `at_critical [2]` are never true at the same time.
- `eventually!(p)` requires that `p` holds now or at some point in the future.
- `next_event(p)(f)` requires that at the first time point at which `p` holds, `f` holds, regardless of the granularity of time.

The PSL operators that are affected by the granularity of time are next, next_e, next_a and some of the SERE operators.

The user can change the granularity of time given by the system using the @ operator. For example, assume you want to advance one time point whenever (status==OK). You can use the PSL @ operator on your property as follows:

```
(always ((size==3) -> next (size > 5)))@(status==OK)
```
(7)

3 Missile Interception Control System

It is a considerable challenge to verify aerospace hybrid control systems, especially when they include a significant amount of nonlinear dynamics. Recently, in [14], different nonlinear controllers were applied to a complex aircraft design problem. While each of the controllers demonstrated some ability to achieve the design criteria, it was noted that the controllers are labor intensive to design and would require new approaches for verification other than simulations. In this section, we present PSL-enhanced formal verification of antimissile control system.

An initial step in formal verification of a hybrid system is to make a reasonable approximation (discretization) of the nonlinear dynamics to reduce the possibly infinite state space system into a finite state space system. A weakness of such a conservative approximation is that it may require a large number of samples, and both the memory requirements and the computation time of a formal verification tool may

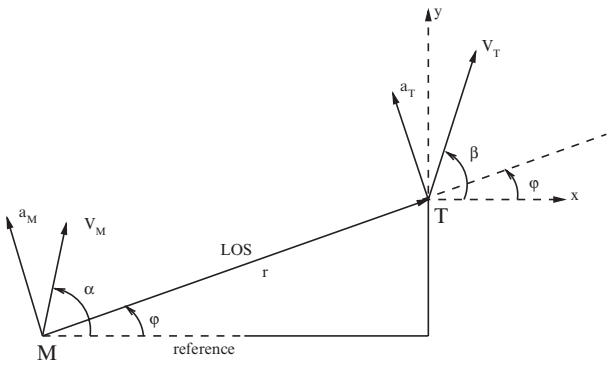


Fig. 1 Geometry of the planar tracking problem

soon become impractical [15]. Usually, the number of states to examine is huge in applications of practical interest, but in the case of missile tracking, the physical process is time limited, and can be translated into a bounded model checking application. Formal methods verify the antimissile interception using the same difference equations representation of the system used by the Matlab simulation tool, and the system properties have straightforward descriptions in PSL.

Consider the problem of a planar moving target interception, depicted in Figure 1 [10, 12]. Both the target T and pursuing missile M are assumed to be point masses moving in a plane. These are the polar system equations of motion that appear in [10]:

$$\begin{aligned} \dot{r} &= V_T \cos(\beta - \varphi) - V_M \cos(\alpha - \varphi) \\ \dot{\varphi} &= V_T \sin(\beta - \varphi) - V_M \sin(\alpha - \varphi) \end{aligned} \tag{8}$$

where LOS is the instantaneous missile-target line-of-sight, a time-variant vector from the pursuer to target; r is the range i.e., the length of the LOS; φ is the bearing angle i.e., the angle between the LOS and the reference line; α is the missile heading angle; and β is the target heading angle.

The missile is governed by its guidance system, i.e., a compensation network placed in series with engagement process Eq. (8) to accomplish an interception. Most of the applied guidance laws belong to the family of Proportional Navigation guidance laws [11], which have shown good performance against moderately-maneuvering targets. In the True Proportional Navigation guidance laws, missile acceleration is usually applied normal to LOS:

$$\begin{cases} a_M = -\lambda \dot{r} \dot{\varphi} \\ \dot{V}_M = a_M \sin(\alpha - \varphi) \\ \dot{\alpha} = a_M / V_M \cos(\alpha - \varphi) \end{cases} \tag{9}$$

where $\lambda > 0$ is a navigation constant. In practice, the control dynamic is implemented as a sampled control system (a computer-based control).

The closed-loop hybrid tracking system has a free system input; this is the target acceleration. An intelligent target is expected to perform evasive maneuvers

to increase the probability of its escape. The target maneuver considered below in Eq. (10) is restricted to the application of the lateral target acceleration normal to the target velocity; this governs the following input dynamics for the target:

$$\begin{cases} a_T = b/(r\dot{\varphi}) \\ \dot{V}_T = a_T \sin(\beta - \varphi) \\ \dot{\beta} = a_T / V_T \cos(\beta - \varphi) \end{cases} \tag{10}$$

where $b > 0$ is a positive constant.

The tuning of this hybrid control system is a very tedious task. The performance of the derived guidance law is characterized by the capturability (i.e., the ability of the guidance law to ensure the capture or interception of a target), which is translated into the capture region bounds. The capture regions may not exist when the initial conditions on range, bearing angle, and their rates are high. Usually, the qualitative analysis technique is used to obtain the capture region from the chosen target maneuver, final time to intercept, and sufficient initial conditions for interception [10]. The resulting controller is relevant only to this specifically chosen target maneuver according to the specific b parameter value.

In contrast to this, formal methods provide a full coverage of the impacts of b parameter perturbations by verifying a logical model of the system to satisfy/dissatisfy the particular properties. Formal verification formulates the design problem as follows: define the particular final time and initial conditions, and find the guidance system parameters that could prevent the escape of the target under these conditions. This property is realistic and helps design robust controllers, which take into account realistic target maneuvers.

Systems that have been traditionally analyzed by formal methods are discrete; therefore, the continuous-time Eq. (8) is transformed into the difference equation presentation of the continuous-time systems as a periodically-updated system. The overall system is described in the Verilog language, which has a powerful compiler that can synthesize the Verilog model into the logical circuit of basic logical gates. All numbers are represented by 32-bit vectors.

The tracking algorithm is applied to a realistic interception scenario with the initial range varying around 30 kilometers for a maneuvering target. For this case, the property is based on a necessary capturability condition that after 10 seconds from the start of the interception process (or after the k sequences of state transitions), the distance r between the missile and target must always decrease from the initial range of 30000 meters to less than 20000 meters. The specially-chosen gain of the controller (navigation constant) $\lambda = 3$ must ensure this. The simulation procedure usually used to check controller consistency is the launch of Monte Carlo trials. A formal verification engine found a counterexample in a single run after verifying the following property formulated in PSL:

```
Property 1:
define λ = 3;
always (range(0)=30000 -> next[k] (range(k)<20000))
```
(11)

relative to all possible perturbations of target acceleration parameter $b = [950 \ldots 1000]$ with granularity 5.

The SAT solver was launched for $k = 5$ sequences of state transitions (cycles). The counterexample provides the target acceleration that caused the range to be at least 21000 meters after the first 10 seconds of the interception process. Because the designed controller with $\lambda = 3$ has not met the requirements of the intermediate range value, we must tune the navigation constant of the guidance law Eq. (9). The following property claims that it is impossible to find the navigation constant $\lambda = [3 \ldots 5]$ with granularity 0.1 that ensures the condition $r(5) < 20000$ meters.

```
Property 2:
forall λ = [3...5]
always (range=30000 -> next[k] (range>20000))
```
(12)

After a 94 second run, the RuleBase PE provides a counterexample to Property 2, saying that navigation constant $\lambda = 3.8$ is a suitable control gain for this case.

This successful example illustrates a general methodology for formal analysis of a control system:

1. Create a discrete model of the system and control law.
2. In PSL, specify the system properties.
3. Verify the initial solution (model vs. properties).
4. If system properties hold, the system design is acceptable.
5. Otherwise, select a (possibly new) candidate control law.
6. Choose a new set of candidate values for control parameters.
7. Create a model of the system with the new control law.
8. Specify the fail claim property: the selected control law, for all perturbations of control parameters, always fails.
9. Check the model against the fail claim property.
10. If the fail claim property holds, go back to Step 5 or 6.
11. If the fail claim property does not hold, conclude that the values of the control parameters provided in the counterexample are robust.

This methodology provides a possibility of efficient application of formal analysis to design and verification of control systems.

4 SMARRT: Static Model Checking and Analysis for Rose Real-Time

IBM Rational Rose Real-Time (RoseRT) is a widely used model-driven development tool for concurrent reactive systems. The system's behavior is specified using a collection of UML [5] diagrams. RoseRT generates code based on the given model. Since RoseRT is intended to support the entire development process and not just the design stage, it allows the user to execute and debug the system at the model level.

The model defines the system in an exact and complete representation so that there is no disparity between the code and the model. In the SMARRT project, we set a goal to allow users of RoseRT to formally verify the model. As opposed to traditional testing, formal verification can prove the absence of a bug, and not only the existence of a feature. Often in the model checking process, verification expertise is required to build the model, define the specifications, and observe and understand the results. We, instead, intend to equip RoseRT users with the advantages of model checking, without requiring the user to be an expert in the field.

4.1 Defining the Model

The RoseRT model describes the entire system, therefore it can be used as the model that is checked. Using a simple transformation [8], the same behavior is expressed in a PSL model. In RoseRT, the user can work with certain building-blocks that are available for describing the behavior of the system. For example, one building block is a simple message queue. Generated code based on the RoseRT model uses an efficient code template to perform the desired behavior. The efficiency of this code should not be misunderstood—it is optimized for execution but not for verification. Thus, we have hand-modeled these building-blocks in a verification-efficient template. When the RoseRT model is translated into PSL, the PSL template is used, just as the code template is used when the model is translated to code.

Figure 2 shows a small client-server protocol example [6]. This example has two state machines: one for the client and one for the server. They are connected using

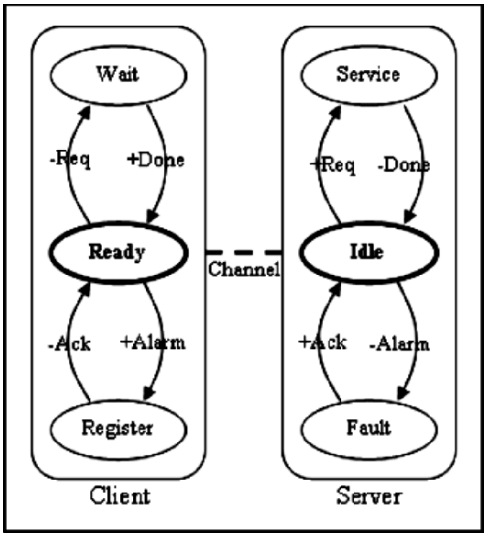


Fig. 2 Client–server protocol

a two-way message queue. The client state machine has three states: Ready, Wait, and Register. The Server state machine also has three states: Idle, Service, and Fault. A state machine changes the state, either due to a message it receives or an internal event. In this protocol, each transition is either associated with the reception or dispatch of a message. A state transition is represented by an arc. The arc is tagged by a sign: plus (+m) represents message reception and minus (−m) represents message dispatch. Initially, the client is in the Ready state and the server is in the Idle state.

SMARRT generates a PSL model that includes a state variable for each state machine defined in a state diagram, and a variable for each state machine that defines non-deterministically which port the state machine will check (the environment port, a communication port, or none). The variable that defines the state of the state machine changes according to the content of the checked port and the current state. If a transition occurs, the appropriate port action is executed (whether sending a message or getting a message). Different interleavings are modeled by allowing the state machine to not examine any port, and thus forcing it to stay in the same state.

4.2 Defining the Specification

SMARRT takes advantage of the fact that RoseRT users are comfortable working with models. The specification is written using an extension of the UML sequence diagram. Sequence diagrams depict the interactions between objects and their states. They define a clear timeline, allowing the user to express temporal specifications with a user-friendly interface. The specification is later translated into a PSL formula that needs to be verified. Even if users are not familiar with the PSL, they can express complex constraints that the verified system needs to hold. For a verification process to be successful, the user must understand which specification to verify. This is key to the usefulness of the entire process. Though the user should know which "questions" to ask, SMARRT allows the users to express these questions without prior PSL expertise.

Figure 3 shows a specification requirement over the client–server protocol specifying that the client does not enter into a deadlock state. A PSL translation of this requirement would be `eventually! Client_State=Ready`.

4.3 Model Checking the PSL Model

The translated PSL model and specification is fed to IBM RuleBase PE [4]. RuleBase PE can utilize different model checking engines (e.g., SAT-based, BDD-based, abstraction-refinement, and others) to verify the model. Though a specialized software-oriented engine exists for RuleBase PE [2, 3] we do not utilize it in verifying these models. We observe that concurrent reactive systems are more similar to hardware than common software. Typically, in software systems, a relatively small

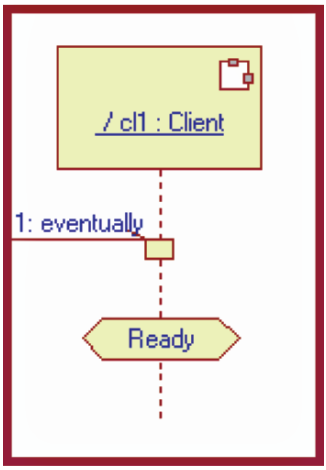


Fig. 3 Specification equivalent to "eventually! Client_State=Ready"

set of variables changes in every cycle, but in hardware, all variables may change in every cycle. Similarly, in communicating state machines for concurrent reactive systems, all machines may change their states in every cycle. Due to this similarity, we believe that hardware model checking techniques are more likely to succeed on these systems. RoseRT models sometimes introduce hierarchy into the model. Once the code is generated, the entire hierarchical structure is flattened. Though the model that needs to be verified is the flattened model, the hierarchy may be used for abstraction purposes if the model is too big for the model checker to cope with [16].

4.4 Counterexample Generation

If the model is verified and a counterexample is found, it is presented to the user in a simple and straightforward manner. The counterexample is described using a standard UML sequence diagram. As every sequence diagram depicts a possible execution of the system, the counterexample is exactly that—an execution of the system that violates the specification.

5 System Automation

System automation through policy-based management allows IT administrators to define high-level policies for various management tasks, such as networked systems and applications for business environments, network planning, problem detection, and quality of service provisions. This approach (e.g., [17]) to system management

allows the separation of the rules that govern behavioral choices of the system from the functionality provided by that system. In a very general way, policies are plans of an organization to achieve its objectives. A policy can be understood as a high-level specification of the system to be automated. It is, therefore, natural to translate it into a formal language and then verify it. Here, we consider policies for the IBM Tivoli System Automation (TSA) for Multi-Platform [19]. This section focuses on how to model and check TSA policy with PSL. We translate real-life industrial policies into PSL and then verify the system with the RuleBase PE model checker.

A TSA policy is a collection of relationships that describes the automated behavior to be enforced by TSA. TSA describes temporal relationships (e.g., A should start after B) or topological relationships (e.g., A is co-located with B) between resources that should be enforced by the system. The building blocks of TSA policies are resources, which can be any piece of hardware or software in the TSA management scope, located on several nodes of the system. There are three types of resources in the TSA policy language: fixed resources (`Resource`), floating resources (`MoveGroup`), and references to a resource outside the management scope of TSA (`ResourceReference`). Resources can be grouped using the `ResourceGroup` or `Equivalency` constructs, so that they are easier to handle. TSA policies are described with XML syntax. See [20] for a detailed description of the TSA policy language.

5.1 Modeling TSA Policies with PSL

To model TSA policies, fixed and floating resources are modeled as state machines in the GDL flavor of the PSL modeling layer and the relationships are modeled as PSL assumptions. In the systems we are modeling, time is continuous, while PSL time is discrete. We deal with this by allowing events to happen at a non-deterministic time and by considering the atomic unit of time to be the minimum possible time between two events in the system.

The TSA description provides the name and node of a resource. A resource can have five states: Unknown, Online, Offline, FailedOffline, and StuckOnline. A resource state is Unknown when its state is not known by TSA for some reason; a resource is Online when it is running and Offline when it is not running. A resource is FailedOffline when it is down with a fatal failure and StuckOnline when it is running with a fatal failure. Possible transitions (where a transition takes one atomic unit of time) for the resource state:

Unknown -> *Unknown* | *Online* | *Offline* | *FailedOffline* | *StuckOnline*
Online -> *Unknown* | *Online* | *Offline* | *FailedOffline* | *StuckOnline*
Offline -> *Unknown* | *Online* | *Offline* | *FailedOffline*
FailedOffline -> *FailedOffline*
StuckOnline -> *StuckOnline*

The amount of time a resource stays in a specific state is non-deterministic and independent of the behavior of other resources. Resources, resource groups, move groups, and equivalencies are coded in the PSL modeling language as an array. The first part of the array codes the node and the second part codes the state. For simplicity's sake, we denote the node and the state of a resource "r" by r.node and r.state. Resource transitions are constrained by the relationship.

Relationships are modeled as constraints using the PSL verification layer directive assume (the assume statement allows specifying an invariant). This allows us to provide a formal description of TSA policy relationships that are only informally described in [19] and [20]. We give a few examples of the way relationships are modeled in PSL.

A StartAfter B means that A must start after B starts. More precisely, when A starts, B should already be online. This translates to the following PSL verification directive:

```
assume always(rose(A.state=Online) -> (B.state=Online &
!rose(B.state=Online))) ;
```

This means the following property should be an invariant of the model: when A goes online, B should be online but did not go online at the same moment as A (*always p* is PSL syntax for the LTL *Gp*). This is more complex than expected. Translation to PSL allows a clearer and non-ambiguous description of the relationships.

A StopAfter B means that A must stop after B does, i.e., when A stops, B is already offline:

```
assume always (fell(A.state=Online) -> (B.state in
{Offline, FailedOffline} & !rose(B.state in {Offline,
FailedOffline}))) ;
```

A Collocated B means that if A is online, A and B are on the same node:

```
assume always ((A.state=Online) -> A.node=B.node) ;
```

A Anti Collocated B means that if A is online, A and B are not on the same node:

```
assume always ((A.state=Online) -> A.node!=B.node) ;
```

5.2 Verification

Once the model is built, we can check PSL properties against it to perform conflict detection, validation of the specified policy to ensure it is consistent with the capabilities of the system, deadlock detection, and loop detection.

We don't check how the system managed by TSA behaves; rather, we check properties on the policy controlling its behavior. For example, we check that the policy is not over-constrained in ways that prevent the system from running satisfactorily, we check that the system can reach the desired state, and we identify whether there exists a single point of failure with regard to these properties. The following PSL properties should hold for every policy:

```
1. assert EF nominal_state ;
2. assert AG EX true ;
3. assert AG (desired_state1 -> EF desired_state2) ;
4. assert AG (desired_state1 -> EX desired_state1) ;
```

where `nominal_state` is true when all resource groups are in the desired states specified in the policy, and `desired_state1` and `desired_state2` are chosen non-deterministically from all the desired states of the system. A desired state of the system is a state in which each resource group is in a known state, and not failed or stuck. Thus there are two "good" values per resource group: Online and Offline, and 2^n possible values of `desired_state1 and desired_state2`.

Property 1 means the system can reach the nominal state specified by the policy. Property 2 means the system can always follow the policy; i.e., there is no truncated path. Property 3 means that while running and in a desired state, the system can reach any other desired state (for instance, if resource group X is offline, all other resource groups are online, and it is possible to bring the desired resource group X online and take groups Y and Z offline). Property 4 means that once the system reaches a desirable state, it can stay there forever (this can be seen as some sort of termination property; it ensures, for instance, that no loop prevents the system from staying as long as needed in the desired state). RuleBase automatically checks that the model is not empty. These properties are rather different to the properties commonly used in hardware verification; as, for example, in [18]. The most commonly used properties for hardware verification are safety properties; non-LTL properties are uncommon. The properties we have shown so far should hold for every policy, and thus checking them can be completely automated. In addition, it is possible to perform policy-specific checks using RuleBase PE.

We built an *ad hoc* translator that semi-automatically translates the XML TSA policy into a model (described in the previous section) and extracts the definitions needed for the automated properties. We then checked these properties with the RuleBase PE model checker. Our work was used to verify several real-life TSA policies.

6 Conclusion

In this paper, we encourage readers to view PSL from a novel perspective, such that its use should not be limited to hardware verification. We do this by reviewing the application of PSL to various fields, such as missile interception algorithms, generated code for concurrent systems, or policies from policy-based middleware. There is

a large amount of literature available about CTL and LTL use; this is also relevant for PSL since CTL and LTL are sub-languages of PSL. As PSL is an IEEE standard, we believe it can be used successfully for a wide variety of problems beyond hardware verification.

Acknowledgements The authors wish to thank Cindy Eisner for her helpful suggestions.

References

1. Dakshi Agrawal et al. Policy Management of Networked Systems and Applications. In *Proc. of 9th Intl. Symp. on Integrated Network Management*, IFIP/IEEE 2005.
2. S. Barner, Z. Glazberg, and I. Rabinovitz. Wolf—Bug Hunter for Concurrent Software Using Formal Methods. In *Proc. of 17th International Conference on Computer Aided Verification, LNCS 3576*, Springer, 2005.
3. S. Barner and I. Rabinovitz. Efficient symbolic model checking of software using partial disjunctive partitioning. *CHARME, LNCS 2860*, 2003.
4. I. Beer et al. RuleBase: An Industry-Oriented Formal Verification Tool. In *Proc. of the 33rd Design Automation Conference*, 1996.
5. G. Booch, J. E. Rumbaugh, and I. Jacobson. *Unified Modeling Language User Guide*. Addison-Wesley, 1999.
6. D. Brand and P. Zafiropulo. On communicating finite-state machines. *Journal of the Association for Computing Machinery*, 30(2), 1983.
7. Cindy Eisner, Dana Fisman. *A Practical Introduction to PSL*. Springer, August 2005.
8. Janees Elamkulam et al. Detecting Design Flaws in UML State Charts for Embedded Software, to *In Proc. of Haifa Verification Conference HVC 2006.* LNCS 4383, Springer, 2006.
9. E.M. Clarke and E.A. Emerson, Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic. In *Proc. of Workshop on Logics of Programs, LNCS 131*, Springer, 1981.
10. D. Ghose. True Proportional Navigation with Maneuvering Target, *IEEE Trans. on Aerospace and Electronic Systems*, 1994.
11. C.-F. Lin. Modern Navigation, Guidance and Control Processing. Prentice Hall, 1991.
12. M. Moulin, L. Gluhovsky, and E. Bendersky. Formal Verification of Maneuvering Target Tracking. *Proc. of the AIAA Conf. of Guidance, Navigation and Control*, Austin, TX, 2003.
13. A. Pnueli. A Temporal Logic of Concurrent Programs. In *Theoretical Computer Science*, Vol 13, 1981.
14. M. L. Steinberg. Comparison of Intelligent, Adaptive, and Nonlinear Flight Control Laws, *Journal of Guidance, Control and Dynamics*, 2001.
15. A. van der Schaft, H. Schumacher. An Introduction to Hybrid Dynamical Systems. Springer, 2000. V.251 of Lecture Notes in Control and Information Sciences.
16. A. Wasowski. Flattening Statecharts without Explosions. In *Proc. of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, 2004.
17. S. Wright, R. Chadha, and G. Lapiotis (eds): *Special Issue on Policy Based Networking*, *IEEE Networking 16*, 2002.
18. Emmanuel Zarpas. A Case Study: Formal Verification of Processor Critical Properties, *Correct Hardware Design and Verification Methods: CHARME 2005, LNCS 3725*, Springer 2005.
19. *IBM Tivoli System Automation for Multi-platforms, Guide and Reference, version 1.2,* IBM, 2004.

20. *IBM Tivoli System Automation for Multi-platforms, Base Component Reference, version 2.1.1*, 2006.
21. *IEEE Standard for Property Specification Language IEEE Std. 1850-2005*, 2005.
22. RuleBase PE homepage.
http://www.haifa.il.ibm.com/projects/verification/RB_Homepage/index.html, 2006

On the Polychronous Approach to Embedded Software Design

A POMSET Interpretation of Endochrony

Sandeep K. Shukla, Syed M. Suhaib, Deepak A. Mathaikutty, and Jean-Pierre Talpin

Abstract Formal approaches for designing mission critical embedded software are gaining importance due to the complexity of concurrent nature and the asynchronous interaction with the environment by such software. "After-the-fact" formal verification is one way to provide correctness guarantees, but is plagued with state-space explosion and other problems. "Correct-by-construction" design approach is therefore often the methodological choice for such software design. *Polychronous* or "multi-clock" model of computation (MoC) in the context of synchronous programming has been successfully used in many safety critical embedded software design in avionics, and other industries in France. SIGNAL is an example of such an embedded software system description language that captures a polychronous MoC. SIGNAL compiler generates deterministic embedded software with provable properties from polychronous specifications. However, an embedded software often interacts with operating systems, hardware interrupt controllers, and other software applications. Therefore, determinism itself may not provide enough guarantee for the correct operation of a software component designed this way. Tighter characterizations beyond determinacy have been invented to guarantee safe usage of such software in an embedded context with lesser restrictive requirements on the environment. "Endochrony" is one such characterization, which is often hard to understand by regular embedded software designers due to the complications of the semantic domain in which such characterization has been expressed in the past. In this paper we provide a true concurrency based semantics which we believe makes the notion of endochrony easier for real system engineers to comprehend and use as a technique to guarantee usage safety.

Keywords: Concurrency, synchronous programming, polychrony, pomset, partial order, semantics, embedded systems and software, true concurrency.

S. Ramesh and P. Sampath (eds.), *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*, 261–273.

1 Introduction

Embedded software is often safety-critical, and hence their correctness is of paramount importance. For example, safety of human lives may be compromised if the software used in avionics applications, and automotive applications are erroneous. Furthermore, the software in such applications are not stand-alone components. They interact with an environment of software and hardware components. Therefore, proving correctness of such software components in isolation is not enough. Since the behavior of such environments may not be known a priori, for the strongest correctness guarantees, one has to prove that the components under design work under fully asynchronous environmental interactions. Of course, if the environment is also designed by the same group of designers, one could assess if such strong assumption (e.g., full asynchrony) is required, and might be able to assume more predictable environmental conditions. However, for the maximal robustness against all possible environments, one would want such components to satisfy some extra conditions.

"Determinacy" of embedded software components is one such important condition. Designers do not want a component to behave differently under identical input stimuli from the environment, at identical internal states. Moreover, if the environment is fully asynchronous, the subsequent stimuli on each input may arrive with unspecified interarrival delays. in order for the component to be safely used in such fully asynchronous environment without showing any change in behavior one has to further tighten the characterization of the components. In the context of Polychrony and SIGNAL (Guernic et al., 2003; Benveniste et al., 2005) this has been thoroughly understood, and the notion of "endochrony" has been introduced. The software components that are endochronous are safe to use in a fully asynchronous environment without worrying about any change in behavior due to delays in interarrival times on the inputs from the environment. Surely, endochrony is a very restrictive criteria, and is needed when the designers do not make any assumptions on the timing behavior of the environment. However, as timing behaviors of the environments are made explicit, one can loosen the requirements for the components to be endochronous. However, such characterizations will be even more complex, and dependent on the timing characteristics of the environment.

Since this article is about polychrony and SIGNAL, one relevant question is why SIGNAL and polychrony are used exclusively inside industries (e.g., AirBus) where the users are closely working with the inventors of polychrony and related semantic models. In fact, a number of concepts germane in polychrony and its characterizations such as endochrony, isochrony, etc., are often rediscovered in many contexts elsewhere. These include the notion of interface automata based theory for safe usage of components, latency insensitive design in the context of hardware design, etc. We believe that the main reason for such useful theory not being more widely utilized is the abstruseness of the semantic theory in which polychrony has been embedded. In particular, a tagged-signal model is used to provide semantics for polychrony. However, the presence of tags in the synchronous interpretation of that model, and

their absence in the asynchronous case leads to non-uniform semantic domains which make the model difficult to comprehend thwarting wide usage by engineers.

Here, we describe a uniform and intuitive semantic framework in which synchronous and asynchronous interpretations of polychronous models can be made uniformly, facilitating understanding, and hopefully wider usage by various embedded systems applications.

1.1 Polychrony and Synchronous Programming

Synchronous programming languages implement a model of computation (MoC) in which time is abstracted by symbolic synchronization and scheduling relations to facilitate behavioral reasoning and functional verification. In the particular case of Polychrony (Guernic et al., 2003), time is represented by partially ordered synchronization and scheduling relations, to provide an additional ability to model high-level abstractions of system paced by multiple clocks: globally asynchronous systems. Polychrony favors the progressive design of "correct by construction" systems by means of well-defined model transformations that preserve the intended semantics of early requirement specifications and eventually provide a functionally correct deployment on target architectures. The notions of determinacy and endochrony (Benveniste et al., 2005) that exist in polychronous model of computation are used to decide the feasibility of single clock implementation of a specification, or of a distributed implementation over asynchronous channels.

For embedded software determinism is an important criterion, because we do not want a piece of embedded software to produce distinct outputs when the same inputs are provided. However, such value-determinism or functional-determinism is not the only property for such software that guarantees its safe usage in an embedded environment. If we have not designed the environment ourselves, or if we have been provided a precompiled software environment whose timing behavior is not under our control, then the timing of input signal arrivals may also be arbitrary, and hence asynchronous with respect to the component we design. So we need a notion of timing-determinism. Timing-determinism is guaranteed when a piece of software produces the same sequences of output values given the same sequences of input values, irrespective of the delay between subsequent values on each inputs, and irrespective of the relative timings of occurrences of the input values on the different input lines. This of course, is a very restrictive requirement on the component under design. One may consider this as a worst case scenario based design, where the designer is not making any timing assumption on the environments. However, if embedded components under design is planned to be reused in many different environments, such strong criteria may be appropriate. As more about a specific environment is known, the requirement can be gradually relaxed. In this paper, however, we only consider the characterization of components with such timing-deterministic behavior, and in the literature on Polychrony, it is called endochrony (Guernic et al., 2003).

This property of endochrony has been defined with distinct semantic domains (synchronous transition systems or STS) by Benveniste et al. (2000) , where they state that for endochronous systems, the presence and absence of all variables can be inferred incrementally from the known values of the present variables and the state variables. Le Guernic et al. (2003) provide a tag-signal model based definition of endochrony. We attempt to provide a different viewpoint to make this notion easily usable in design.

In our view, the most important aspect of this paper is to bring forth the pomset based semantic model which removes the need for a tagged-signal model (Lee and Sangiovanni-Vincentelli, 1996). That model has an artifact of tags with each new value of the variables/signals during the program execution. Our exposition here shows that these tags were unnecessary for the purpose of characterizing polychronous programs. In fact their usage makes asynchronous interpretation of programs unduly complicated, resulting in difficult to understand theories for characterizing polychronous programs. Such characterizations include deterministic programs, endochronous programs (Guernic et al., 2003), self-synchronous programs (Benveniste et al., 2005), etc. As we have discussed earlier, one of the most important question is when can one place a program in an asynchronous environment and expect to obtain the correct output sequence, similar to what one would expect from the program in a synchronous environment. Such question is related to the issue of asynchronous interpretation of a program. Our semantic model is naturally an asynchronous interpretation, and its synchronous interpretation is constructible for deterministic polychronous programs by way of "levelling functions" defined in Section 4. Interestingly, the same deterministic program may have multiple possible synchronous interpretations, and a choice one makes leads to a refinement of the original polychronous specification. However, having multiple possible synchronous interpretation would mean that the program does not have a unique single clocked implementation. So if one is interested only in the class of programs that has unique single clocked possible implementation, one would be interested in the class of endochronous processes. So in this sense, this paper demystifies the notion of endochrony in very simple terms.

The main exports of this paper are as follows: (i) We show a true concurrency semantics for polychronous specifications using POMSETS (Pratt, 1986). (ii) We formalize properties such as determinism, and endochrony in terms of pomset semantics, and show that they are intuitive and visually relatable in this formalism.

2 Related Work

While multi-clocked synchrony offers a way to model both synchronous systems and concurrent ones within the same semantic framework, casting it into a model of computation that homogeneously captures both synchrony and concurrency has, to our knowledge, never been achieved. Meanwhile, the aim of capturing both synchrony and asynchrony in a unifying model of computation is shared by several approaches: interaction categories of Abramsky (1996), communicating sequential

processes of Hoare (1978), Kahn networks (Kahn, 1974), models of computation (Lee and Sangiovanni-Vincentelli, 1996), latency insensitive systems (Carloni et al., 2001), the polychronous model of computation (Guernic et al., 2003), heterogeneous systems (Benveniste et al., 2005). In these models, synchrony and asynchrony are partitioned into related yet disjoint mathematical domains: one for synchrony with a relational structure for time, one for asynchrony with a relational structure for temporal causality. For instance, heterogeneous systems consists of a tag-less model for asynchrony and a time-tagged model for synchrony (Lee and Sangiovanni-Vincentelli, 1996; Jantsch, 2003). One can relate one to the other through specific morphisms but one is not included in the other.

The polychronous model of computation slightly differs by considering a domain of tagged traces and of a semi-lattice structure that renders the synchronous hypothesis using a timing equivalence relation: clock equivalence. Asynchrony can be superimposed on this model by considering a flow equivalence relation as well as heterogeneous systems (Benveniste et al., 2005) by parameterizing composition using arbitrary timing relations. Still, synchrony is modeled by tag equality that is irrelevant to asynchrony. The notion of self-synchrony, endochrony (Benveniste et al., 2005) and isochrony (Guernic et al., 2003), etc. have been formulated for synchronous/polychronous MoCs to characterize classes of programs whose asynchronous interaction can be predicted by analyzing synchronous interactions. These are interesting subclasses of deterministic programs that are implementable in an asynchronous networked environment without having to insert special interfaces and protocols.

Pomsets have been used to model true concurrency in many contexts. A good introduction to pomsets can be found in (Pratt, 1986). A large body of literature exists on using pomsets to give true concurrency semantics to Petrinets, to parallel functional programs (Hudak and Anderson, 1987), for synchronization and recursion (Meyer and de Vink, 1989), etc. Complexity analysis of pomset equivalences have been studied (Jategaonkar and Meyer, 1993). A number of category theoretic models based on true concurrency representations have been proposed. However, in this paper, we take the concept of pomsets to obtain a simple semantic domain which allows us to understand the semantics of polychrony model of computation, and characterize some subclasses of polychronous processes. We believe that our characterizations clarifies much of the complex world of polychronous model of computation, which often gates the wide usage of this powerful model of computation.

3 Background

In this section, we introduce the background and preliminary definition necessary to understand our formalism. We first briefly introduced the tagged signal trace model for polychrony (Guernic et al., 2003) and the pomset based semantic model (Pratt, 1986).

Let $\mathcal{V}$ denote the universal set of all variables and D be the set of all data values.

3.1 A Polychronous Model of Computation

Polychronous MoC (Guernic et al., 2003) facilitates the description of systems in which components obey multiple clock rates. It provides a mathematical foundation to a notion of refinement and the ability to model a system from the early stages of its requirement specifications to the late stages of its synthesis and deployment.

In the tagged-signal model, a partially-ordered set $(\mathcal{T}, \leq, 0)$ of tags is considered. A tag $t \in \mathcal{T}$ denotes a symbolic instant or a period in time. $C \in \mathcal{C}$ is a chain of values in $\mathcal{T}$. Signals, behaviors and processes are defined starting as follows:

An event $e \in \mathcal{T} \times D$ is the pair of a tag and a value. The absence of an event, is denoted as ϵ. A *signal* $s \in C \rightarrow D$ is a function from a chain of tags to values. A behavior $b \in \mathcal{B}$ is a function from names $x \in \mathcal{V}$ to signals $s \in \mathcal{S}$. A *process* $p \in \mathcal{P}$ is a set of behaviors that have the same domain.

We write tags(s) and tags(b) for the tags of a signal s and of a behavior b; $b|_X$ for the projection of a behavior b on $X \subset \mathcal{V}$ and $b/X = b|_{\text{vars}(b)\setminus X}$ for its complementary; vars(b) and vars(p) for the domains of b and p.

The synchronous composition $r \,|\, s$ of two processes p and q is defined by the union of all behaviors b_r (from r) and b_s (from s) which carry the same values at the same time tags.

$$r \,|\, s = \{b_r \cup b_s \mid (b_r, b_s) \in r \times s, I = \text{vars}(r) \cap \text{vars}(s), b_r|_I = b_s|_I\}$$

A trace consists of a sequence of events ordered based on their time instants. A clock for the trace can be derived based on these time instants. Therefore, each variable is associated with a clock that denotes the chronology of the occurrence of events. For a variable v, we denote its clock by $\hat{v}$. Consider the following example: Consider the observation of the trace of variable a shown below. The data values that are assigned to variable 'a' are 1, 2, and 3, and their respective tags are t_1, t_3, and t_5. The clock $\hat{a}$ is $\langle t_1, t_3, t_5 \rangle$, which is based on the occurrence of events for a.

$time:$	t_1	t_3	t_5
$a:$	1	2	3

3.2 Pomsets

Definition 1 *A pomset (Pratt, 1986) is an isomorphism class of a labelled partial order (lpo) defined as a 4-tuple as* $\langle V, \leq, \Sigma, \mu \rangle$ *where:*

- V is the set of vertices modeling events. Each event $(\bar{e})$ in V is an instance of an action $\in \Sigma$.
- $\leq$ is the partial order defined on V which expresses precedence between events. If $\bar{a}, \bar{b} \in V$, then $\bar{a} \leq \bar{b}$ is interpreted as event $\bar{a}$ preceding event $\bar{b}$ in time. (So $\leq$ is really a pre-order).

- Σ is the alphabet modeling actions. For example an action can be an assignment of a data value to a variable (e.g., reset:=true).
- $\mu : V \rightarrow \Sigma$ is a labelling function that assigns the actions to vertices.

4 Pomset Representation of Polychrony

An event of a pomset is represented as $\bar{e}$, where $\bar{e} = (x, d)$, for $x \in \mathcal{V}$ and $d \in D$. An event set is a set of such events that are related by a partial ordering relation ($\leq$). A pomset is visualized as a graph, where each node corresponds an event from the vertex set V, and the arcs order the events based on ($\leq$). In the graphical representation, we only show arcs between two events $\bar{e_i}, \bar{e}_j \in V$, iff $\bar{e_i} \leq \bar{e}_j, \nexists \bar{e_k} : \bar{e_i} \neq \bar{e_k} \neq \bar{e}_j$ s.t. $\bar{e_i} \leq \bar{e_k} \wedge \bar{e_k} \leq \bar{e}_j$. We denote the partial order $\leq$ by $\mapsto$ if there is an edge between the event vertices in the pomset graph.

The pomset P for the trace of variable 'a' in the previous example is $\langle V_P, \leq_P, \Sigma_P, \mu_P \rangle$, where $V_P = \{\bar{a}_1, \bar{a}_2, \bar{a}_3\}$ and $\bar{a}_i = (a, i)$, $i = 1, 2, 3 \in \mathbb{N}$. The partial order, $\leq_P$ is defined on V_P, where $\bar{a}_1 \mapsto \bar{a}_2 \mapsto \bar{a}_3$, and Σ_P the action set is $\{a := 1, a := 2, a := 3\}$. μ_P is the labelling function that maps the events to their actions. For example, $\bar{a}_1$ is mapped to action $a := 1$. The pomset visualization of polychronous variable a is shown in Figure 1. We now need to extend a pomset with the notion of functional pomset, that captures functional dependency between events in the set. The partial order only captures temporal precedence of events, and does not capture the dependency caused by the fact that some events occur due to a function computation on some other events. Later in the paper, when we show how to construct a new pomset from pomset representations of two processes which interact in the sense that events in them participate together in a function computation, we need to introduce a new pre-order ($\leq^f$) on the resulting event set, that expresses the ordering based on functional dependency. If $\bar{b} = f(\bar{a})$ for $\bar{a}, \bar{b} \in V$, then $\bar{a} \leq^f \bar{b}$ is interpreted as event $\bar{a}$ precedes in functional sense event $\bar{b}$ and $\bar{b}$ is functionally dependent on $\bar{a}$.

Definition 2 (Functional Dependence) *An event $\bar{a}_i$ is functionally dependent on an events $\bar{b}_{j_1}, \bar{b}_{j_2}, \ldots, \bar{b}_{j_n}$ if $\bar{a}_i = f(\bar{b}_{j_1}, \bar{b}_{j_2}, \ldots, \bar{b}_{j_n})$. We denote functional dependence with $\leq^f$ and represent the corresponding edge in the pomset visualization with $\hookrightarrow$ arc.*

The addition of functional dependence order among events in a pomset means that we have a new semantic object which is a pomset together with another non-reflexive

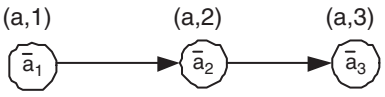


Fig. 1 Visualization of pomset P

partial order $\leq^f$. For the rest of the paper for ease of writing, we call this functional pomset also pomset, with the understanding that a $\leq^f$ may or may not be attached to it. If one considers, a single input variable of a program, and all the events happening for that variable, we usually get a totally ordered pomset, or a tomset. Such a pomset does not have any $\leq^f$ associated with it. It is only when we construct the pomsets corresponding to a program that computes functions based on such input variables, that we obtain pomsets with such $\leq^f$ relations in addition to the causal partial order.

One important construction fact to know is how to capture synchronization between two events in the $\leq$. Let x and y be two variables in the program, and events happening in those variables are expressed as tomsets $\{x_1, x_2, \ldots, x_i, \ldots\}$ and $\{y_1, y_2, \ldots, y_j, \ldots\}$. Suppose the program requires that x_i and y_j to be synchronized. This can be captured by adding (x_i, y_{j+1}) and (y_j, x_{i+1}) in $\leq$.

Furthermore, we define the notion of levelization of the events in the event set by using a level function l. Levelization involves arranging the events into levels based on their partial ordering.

Definition 3 (Levelling Function) *Given a pomset* $p : \langle V_p, \leq_p, \Sigma_p, \mu_p \rangle$ *and its associated functional ordering* $\leq^f_p$, *a level function* $l : V \rightarrow \mathbb{N}$ *has the following properties:*

1. $\forall \bar{e}_i, \bar{e}_j$ if $\bar{e}_i \leq_p \bar{e}_j$, then $l(\bar{e}_j) > l(\bar{e}_i)$
2. Let $V(\bar{e}_j) = \{\bar{e}_{i_1}, \bar{e}_{i_2}, ..., \bar{e}_{i_k}\}$ be the set of events s.t. $\forall \bar{e}_i \in V(\bar{e}_j)$ if $\bar{e}_i \leq^f_p \bar{e}_j$, $\nexists \bar{e}_k \notin V(\bar{e}_j)$ $(\bar{e}_k \leq^f_p \bar{e}_j)$, then $l(\bar{e}_j) = max_{\bar{e}_i \in V(\bar{e}_j)}\, l(\bar{e}_i)$

Let $\mathcal{L}^p$ be the set of all levelling functions $l : V \rightarrow \mathbb{N}$ for the pomset, as defined above. Let $\mathcal{L}^p_1 \subset \mathcal{L}^p$ denote the set of all levelization functions l with two extra conditions:

1. Let $V_1 = \{\bar{e}_i | \nexists \bar{e}_j, \bar{e}_j \leq_p \bar{e}_i\}$ then for at least one $\bar{e} \in V_1$, $l(\bar{e}) = 1$
2. Let $V(\bar{e}_j) = \{\bar{e}_{i_1}, \bar{e}_{i_2}, ..., \bar{e}_{i_k}\}$ be the set of events s.t. $\forall \bar{e}_i \in V(\bar{e}_j)$ if $\bar{e}_i \leq_p \bar{e}_j$, $\nexists \bar{e}_k \notin V(\bar{e}_j)$ $(\bar{e}_k \leq_p \bar{e}_j)$, then $l(\bar{e}_j) = max_{\bar{e}_i \in V(\bar{e}_j)}\, l(\bar{e}_i) + 1$

Figure 2 shows a graphical representation of a pomset, where events $\bar{a}$ and $\bar{b}$ may happen concurrently followed by event $\bar{c}$, and then event $\bar{d}$. It also shows one possible levelization for the pomset.

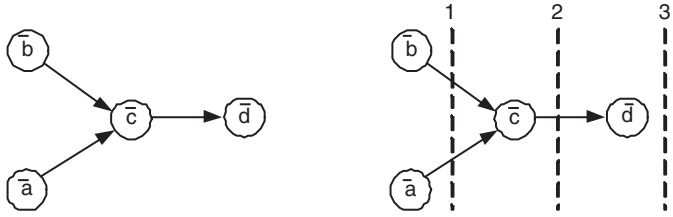


Fig. 2 Graphical representation of pomset

Given a pomset $p : \langle V, \leq, \Sigma, \mu \rangle$, and a levelling function $l \in \mathcal{L}^p$, (p, l) is an l-synchronous view of pomset p. A pomset may have multiple l-synchronous views. If $l \in \mathcal{L}_1^p$, then we call it a asap-synchronous view of the pomset. Note that "asap" stands for "as soon as possible". One can imagine a levelling function as a schedule for the events of the pomset. In this sense, an asap-synchronous view is provided by a levelling function which makes at least some of the events occur as soon as they are permissible by the constraints of $\leq_p^f$ and $\leq_p$.

Intuitively, given a pomset without an associated level function, we have an asynchronous view of the event occurrences, without caring about whether the non-causal events happen synchronously or not. In an l-synchronous view, one can imagine that events that have the same level happen at the same synchronous step. So an l-synchronous view imposes a scheduling on the events of the pomset.

Now, if one replaces the clause (1) in extra conditions on $\mathcal{L}_1^p$ by the condition $\forall \bar{e} \in V_1, l(\bar{e}) = 1$, then it provides an "exactly-asap" view. We will call the set of such levelling functions as $\mathcal{L}_1^p\text{-}exact$. So the corresponding "exactly-asap" schedule, therefore, makes events that results from computing a function on other events, synchronously occurring with the events it depends on. It also makes event sequences in each input to occur in consecutive steps (levels). All events that are inputs (events that do not functionally depend on other events in the pomsets), their sequence happens synchronously in a lock step. So an "exactly-asap" view of a pomset is a synchronous scheduling of events, assuming input events all come in the lock step. In Figure 2, one can easily see that right-hand side we have an exactly-asap-synchronous view of the pomset on the left-hand side. The level function l in this view maps $\bar{a}$ and $\bar{b}$ to level 1, and $\bar{c}$ to level 2 and $\bar{d}$ to level 3. Since this pomset does not have an associated $\leq^f$, this level function can be easily seen to be in $\mathcal{L}_1^p$.

A process can have multiple behaviors, and can be represented by either a single pomset or by multiple pomsets.

Definition 4 (Process) *A process $P = \{p_1, p_2, \ldots\}$ is a collection of pomsets, where $p_1, p_2, \ldots$ are pomsets with disjoint event sets of the process.*

Note that the different pomsets of a process capture different possible behaviors.

Definition 5 (Pomset Projection) *A projection of a pomset on a variable set I denoted by denoted by $p|_I$ is defined as follows. Given $p = \langle V, \leq, \Sigma, \mu \rangle$ and a variable set $I \subseteq \mathcal{V}$, $p|_I = \langle V|_I, \leq|_I, \Sigma|_I, \mu|_I \rangle$, where,*

1. $\Sigma|_I \subseteq \Sigma$ denotes the set of all actions that pertain to variables only in I,
2. $V|_I \subseteq V$ denotes all events that pertain to the variable set I only, ($\forall \bar{e} \in V$, $\bar{e} \in V|_I$ iff $\mu(e) \in \Sigma|_I$),
3. $\mu|_I : V|_I \rightarrow \Sigma|_I$ is restriction on μ to $V|_I$, and
4. $\leq|_I \subseteq \leq$ is a restriction of $\leq$ to the events only in V^I.

Intuitively, a restriction of a pomset p on variable set I can be obtained by deleting all events not pertaining to variables in I, and all dependency arrows that shows dependency between such events.

A process P for all variables in I considered as input to the process and given their corresponding tomsets is deterministic, iff the pomset corresponding to the process behaviors are unique up to isomorphism.

Definition 6 (Deterministic Process) *A process P is deterministic iff $\exists I \subset \mathcal{V}_P$ s.t. $\forall p, q \in P$, if $p|_I = q|_I$ then $p = q$.*

For every given input sequence, in the pomset representation of a deterministic process there exists a single pomset p such that $p|_I$ matches that given input sequence. Note that for independent input variable set I, the input sequence correspond to a set of tomsets, each tomset representing each input variable's event sequence. For non-deterministic processes, multiple pomsets may represent possible behaviors of the process for the same given input sequences. For the rest of the paper, we only talk about deterministic processes.

5 Flow and Clock Equivalence

The synchronous structure of polychrony is defined by a clock equivalence relation. Two behaviors are clock equivalent if they have the same partial order up to an isomorphic choice of time tags.

Definition 7 (Clock Equivalence) *A behavior c is a* stretching *of b, written $b \leq c$, iff* $\text{vars}(b) = \text{vars}(c)$ *and there exists a bijection f on $\mathcal{T}$ which satisfies*
$\forall t, t' \in \text{tags}(b), t \leq f(t) \wedge (t < t' \Leftrightarrow f(t) < f(t'))$
$\forall x \in \text{vars}(b), \text{tags}(c|_{\{x\}}) = f(\text{tags}(b|_{\{x\}})) \wedge \forall t \in \text{tags}(b|_{\{x\}}), b|_{\{x\}}(t) = c|_{\{x\}}(f(t))$
b and c are clock equivalent*, written $b \sim c$, iff there exists d s.t. $c \geq d \leq b$.*

The asynchronous structure of polychrony is modeled by a flow equivalence relation. Two behaviors are flow equivalent if they hold the same values in the same signal-wise total order.

Definition 8 (Flow Equivalence) *A behavior c is a* relaxation *of b, written $b \sqsubseteq c$, iff* $\text{vars}(b) = \text{vars}(c)$ *and, for all $x \in \text{vars}(b)$, $b|_{\{x\}} \leq c|_{\{x\}}$. b and c are* flow-equivalent*, written $b \approx c$, iff there exists d s.t. $b \sqsupseteq d$ and $d \sqsubseteq c$.*

Asynchronous composition $r \parallel s$ is defined by considering the partial-order structure induced by the relaxation relation. The parallel composition of r and s consists of behaviors d that relax the behaviors b_r and b_s from r and s along shared signals $I = \text{vars}(r) \cap \text{vars}(s)$ and that stretch b_r and b_s along independent signals of r and s.

$$p \parallel q = \left\{ d \in \mathcal{B}|_{\text{vars}(r) \cup \text{vars}(s)} \mid \exists (b_r, b_s) \in r \times s, d/I \geq (b_r/I \mid b_s/I) \wedge b_r|_I \sqsubseteq d|_I \sqsupseteq b_s|_I \right\}$$

We now extend these notions of flow and clock equivalence to pomsets. Let $p : \langle V_p, \leq_p, \Sigma_p, \mu_p \rangle$ and $q : \langle V_q, \leq_q, \Sigma_q, \mu_q \rangle$ be two pomset and P a process, such that $p, q \in P$. We denote the variable set of p and q by $\mathcal{V}_p$ and $\mathcal{V}_q$. Let $\mathcal{V}_P$ be the variable set of the process, such that $\mathcal{V}_P(p)$ is the set of variable associated with pomset p. Two pomsets are flow equivalent iff the tomsets representing the event sequences for the variables are isomorphic for each variable.

Definition 9 (Flow Map between Pomsets) *Given two pomsets p and q with their associated functional dependency pre-orders $\leq_p^f$ and $\leq_q^f$, a flow map between p and q is an injective map $f_{pq} : V_p \rightarrow V_q$ such that*

1. $\mathcal{V}_p = \mathcal{V}_q$
2. $\forall v \in \mathcal{V}_p, \forall a, b \in V_p|_{\{v\}}$ if $a \leq_p b$ then $f(a) \leq_q f(b)$, $f(a), f(b) \in V_q|_{\{v\}}$
3. $\forall a, b \in V_p$ if $a \leq_p^f b$ then $f(a) \leq_q^f f(b)$

Note that in the above definition $V_p|_{\{v\}}$ denotes only the events in V_p that pertain to a variable v only. So a flow map basically requires that if one observes events happening on each individual variable in one behavior (pomset), then there is a corresponding event happening in the same order in the other behavior (pomset). Moreover, if one observes the events across variables that have functional dependence among themselves, such dependence is preserved in the corresponding mapped events.

Definition 10 (Flow Equivalent Pomsets) *Two pomsets p and q with their associated functional dependency pre-order $\leq_p^f$ and $\leq_q^f$, p and q will be called flow equivalent iff there exists a flow map f_{pq} from p to q such that f_{pq}^{-1} is also a flow map from q to p.*

Two pomsets (with their functional dependency) are flow equivalent iff by just observing event sequences for each variables separately (without considering their causality with events from other variables, except for functional dependency), one cannot distinguish between the behaviors represented by the two pomsets.

Theorem 11 *For a deterministic process, the flow equivalence classes are singleton sets.*

Proof sketch Since the pomsets representing the behaviors of the processes are unique for a given input behavior, and pomset events does not have any associated time tag, the theorem follows. □

Let $\mathcal{L}^P$ denote the set of levelling functions for a pomsets in process P, then $\mathcal{L}^p$ and $\mathcal{L}^q$ denote the sets of levelling functions for pomsets p and q. Two pomset are clock equivalent if they have isomorphic levels.

Definition 12 (Clock Equivalent Pomsets) *Given two pomsets $p = \langle V_p, \leq_p, \Sigma_p, \mu_p \rangle$ and $q = \langle V_q, \leq_q, \Sigma_q, \mu_q \rangle$ and their associated functional dependency $\leq_p^f$ and $\leq_q^f$, we say p and q are clock equivalent, denoted as $p \sim q$, iff there exists a bijection $f : V_p \rightarrow V_q$, and levelling functions $l_p \in \mathcal{L}_1^p$ and $l_q \in \mathcal{L}_1^p$ s.t. $\forall x \in V_p$, $l_p(x) = l_q(f(x)) \wedge \forall y \in V_q$, $l_q(y) = l_p(f^{-1}(y))$*

What this means is that two pomsets are clock equivalent if asap-synchronous views of both are created, then a bijection between the events in both would exist with the following property. Events in one pomset that are in the same level under its levelling function, will be mapped to events in the other in a way that they all have the same level under that pomset's levelling function. In other words, if one thinks of events in the same level happening in the same cycle, then in both pomsets events that correspond to each other would happen in the same cycle.

5.1 Understanding Endochrony

A pomset represents a single behavior of a polychronous program for a given input event sequence. So from an asynchronous observer's point of view, a pomset provides a specification such that his/her observations of event sequences (without looking into the program itself) must never violate the pomset's $\leq$ and $\leq^f$. On the other hand, a levelling function l provides a view of the program execution with the scheduling of the events, which means the observer can also see inside the program's execution schedule with respect to some clock. Each possible levelling function may correspond to a different clock, and schedule. In polychrony, an interesting class of processes/programs are called endochronous processes/programs. This is the class of programs that can be scheduled with a single clock uniquely.

Definition 13 (Guernic et al., 2003) *A process is endochronous iff,* $\forall p, q \in P$, $p \approx q$ *implies* $p \sim q$.

What this means is that if we have behaviors that are flow equivalent, we could schedule them uniquely with respect to one clock. The way to schedule them will be given by the $\mathcal{L}_1$ levelling function set for the behaviors.

Now we state the following theorem which demystifies the idea of endochrony. A more detailed proof will be given in a future version of the paper.

Theorem 14 *A process is endochronous iff* $\forall p \in P$ *the set* $\mathcal{L}_1^p$ *is singleton and coincides with* $\mathcal{L}_1^p - exact$.

Proof sketch By the definition of endochrony, all flow equivalent pomsets of an endochronous process must be clock-equivalent. Since the flow equivalence class of pomsets are singletons, so we only have to consider if the same pomset can have multiple clock inequivalent scheduling. If that happens, then that means flow equivalence does not imply clock equivalence. On the other hand, if flow equivalence does imply clock equivalence, then these singleton sets cannot have more than one way of scheduling. □

6 Concluding Remarks

In contrast to these approaches based on tagged traces, we propose a formulation of polychronous system specifications using the truly concurrent structure of pomsets. This formulation achieves an unambiguous characterization of both synchrony and asynchrony by exclusively considering the scheduling structure of a pomset. Instead of considering a tag structure to give a model-specific time-stamp to events, we simply consider events to be timely unique instances of generic actions (as a pomset requires). The synchrony and asynchrony are captured by considering the structure of the partial order relation that models the causal relations between events.

References

Abramsky, S. (1996). Semantics of interaction. In *Trees in Algebra and Programming, Lecture Notes in Computer Science, Springer Verlag*, volume 1059.

Benveniste, A., Caillaud, B., and Le Guernic, P. (2000) Compositionality in dataflow synchronous languages: specification & distributed code generation. Information and Computation, 163, 125–171.

Benveniste, A., Caillaud, D., Carloni, L.P., and Sangiovanni-Vincentelli, A.L. (2005). Tag machines. In *Proceedings of Embedded Software Conference, Lecture Notes in Computer Science, Springer-Verlag*, October.

Carloni, L., McMillan, K., and Sangiovanni-Vincentelli, A. (2001). The theory of latency insensitive design. *IEEE Transactions on Computer Aided Design of Integrated Circuits and System*, 20(9):1059–1076.

Guernic, P. Le, Talpin, J.-P., and Lann, J.-C. Le (2003). Polychrony for system design. *Journal of Circuits, Systems, and Computers – Special Issue: Application Specific Hardware Design*, 12(3):261–303.

Hoare, C. (1978). Communicating sequential processes. *Communications of the ACM*, 21(8).

Holzmann, Gerard (2003). *The SPIN Model Checker*. Addison-Wesley Professional.

Hudak, Paul and Anderson, Steven (1987). Pomset interpretations of parallel functional programs. In *Proceedings of the Functional Programming Languages and Computer Architecture*, pages 234–256, London, UK. Springer-Verlag.

Jantsch, A. (2003). *Modeling Embedded Systems and SOC's Concurrency and Time in Models of Computation*. Morgan Kaufmann Publishers.

Jategaonkar, L. and Meyer, A. R. (1993). Deciding true concurrency equivalences on finite safe nets. In *Proceedings of ICALP*, pages 519–531. Springer-Verlag LNCS.

Kahn, G. (1974). The semantics of a simple language for parallel programming. In *proceedings of Information Processing*, pages 471–475.

Lee, E.A. and Sangiovanni-Vincentelli, A.L. (1996). Comparing models of computation. In *International Conference on Computer-Aided Design (ICCAD)*, pages 234–241.

McMillan, K.L. (1993). Symbolic Model Checking. PhD thesis, Boston.

Meyer, John-Jules Ch. and de Vink, Erik P. (1989). Pomset semantics for true concurrency with synchronization and recursion (extended abstract). In *MFCS '89: Proceedings on Mathematical Foundations of Computer Science 1989*, pages 360–369, London, UK. Springer-Verlag.

Pratt, Vaughan R. (1986). Modelling concurrency with partial orders. *International Journal of Parallel Programming*, 15(1):33–71.

Scaling up Model-checking

A Case Study

Aniket Kulkarni, Ravindra Metta, Ulka Shrotri, and R. Venkatesh

Abstract A typical formal development method includes specification of the functionality, formal analysis of the specification and finally code generation on to a platform. Often formal analysis is done using model-checking and scalability of model-checking is an area of concern. In this paper we describe our work on integrating two specific tools – Statemate and SAL, to scale up model-checking. More specifically we highlight the benefits, in terms of scalability, that can be obtained by exploiting peculiar usage patterns in the specifications under consideration. The paper briefly introduces the tools and their respective notations, describes a translation strategy as a means to integrate the notations, and presents how we achieved improved scalability of verification using SAL by exploiting peculiar usage of language constructs in the Statecharts of interest. We also present the results of using our tool on some randomly selected Statecharts demonstrating the scalability of our approach.

Keywords: statemate, statecharts, SAL, translation, model-checking.

1 Introduction

There has been a significant increase in the number of embedded systems being developed in the automotive [3], and other industries such as avionics and home appliances. Clear and executable models of such systems are constructed and analyzed before investing heavily in the implementation stages. Typically, these models are specified using formalisms such as Statecharts [4] and Esterel [1] enabling automated analysis. These formal models are verified, using model-checkers, for properties such as state reachability and absence of non-determinism before generating code for the chosen platform. Scalability of model-checkers is an important issue. In this paper we describe the integration of two tools – Statemate and SAL, to scale up model-checking of Statecharts and list challenges that need to be addressed to make the effort more rigorous.

S. Ramesh and P. Sampath (eds.), *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*, 275–283.

We integrated the two tools by building a translator that implements transformations between their notations. To ensure scalability of verification various optimizations have been implemented in the Statechart to SAL translator. Many of these exploit the patterns present in the source specifications under consideration. This exploitation of the structure and language usage within a given set of specifications and the support needed for it is the main focus of this paper.

The rest of the paper is organized as follows – The first few sections briefly present Statecharts and SAL, and a translation strategy from Statecharts to SAL. The next few sections describe the source specific optimizations and the result obtained due to these. We end by listing interesting problems to be solved in building a tool-chain and a possible approach to these.

2 Statecharts

In Statemate a reactive system maybe specified as a collection of Statecharts, which are extensions of conventional state transition diagrams. The main extension is hierarchy of states supported by means of *and* and *or* states. An *and* state has multiple direct sub-states all of which are active if the *and* state is active; i.e. they execute in parallel. An *or* state has non-zero direct sub-states of which exactly one will be active if the *or* state is active. A state that does not contain any other state is a *basic* state. The state at the highest level is the *root state*. In Figure 1, S2 is an *and* state, and both its sub-states and S1 are *or* states, and the rest of the states are *basic* states.

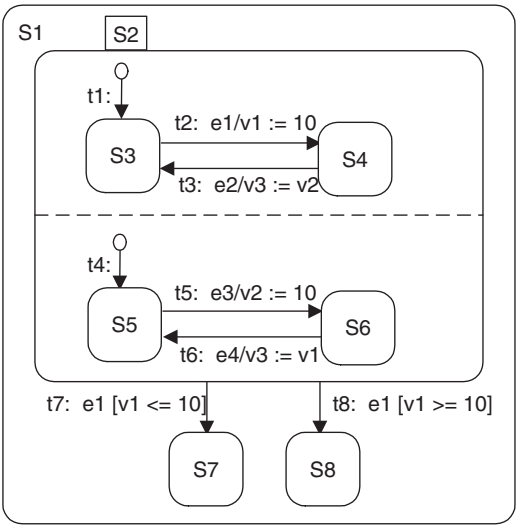


Fig. 1 Statechart C1

A transition is specified using the syntax *e[c]/a*, where *e* is an event, *c* is a condition, *a* is an action and all are optional. A transition may cut through a hierarchy of states and has a priority equal to that of the highest source state it exits. When a transition's target state is a non-basic state, there are multiple ways to specify which sub-state(s) to enter. The most commonly used one is *default* transition, which is a transition with a target state but no source state. Whenever its parent state is entered by a transition, by default it enters the target sub-state of the *default* transition. In Figure 1, t1 and t4 are default transitions and the rest are normal transitions. Note that transitions t7 and t8 have the same priority as they exit the same outermost state viz. S2. But both t7 and t8 have higher priority over transitions t2, t3, t5 and t6 as they exit states lower than S2.

A transition *e[c]/a* is enabled if its source state is active, event *e* occurs, condition *c* holds true and no higher priority transition is enabled. Whenever a transition is taken, the corresponding action *a* is executed, the source state exited and target state entered. It is also possible to associate actions to special events like state entry or exit and a state being active.

Statechart also has support for data-items that store values with specific types.

Execution of a Statechart model proceeds in a sequence of steps. Events live for exactly one step, the next step to the one in which they are generated. All transitions that are enabled in a step are taken simultaneously and their corresponding actions executed, which may result in new internal events being raised. In the next step if any transitions are enabled, due to either some internal events or internal data changes, those are taken. This continues till no more transitions are enabled. At this point external events are sensed and the process continues. The steps at which external events are sensed are called super-steps.

3 SAL

Symbolic Analysis Laboratory (SAL) is a framework developed at SRI for combining different tools for program analysis, abstraction, theorem proving, and model-checking of transition systems. SAL has a language for describing transition systems. Following is a short list of important features in the SAL language. For more details, the reader may refer to the SAL language manual [2].

The basic building block of any SAL program is a *module*. A SAL module consists of: a state, an initialization condition on this state and a binary transition relation of a specific form on the state. The state consists of INPUT, OUTPUT, GLOBAL and LOCAL variables. The input variables are read-only variables. The rest are read–write variables. INITIALIZATION is carried out exactly once and that is when the system execution starts. All transitions are defined in the TRANSITION section. In the transition section, variables from the next state can be referred to by using a '′' suffix.

Modules can be composed synchronously or asynchronously using || and [] respectively. Properties expressed as LTL formulas can be specified in SAL as theorems. SAL has a suite of model-checkers of which we used SAL-smc, which is a symbolic model-checker and SAL-bmc, which is a bounded model-checker that can be used to verify the properties. A sample SAL specification is given below for illustration.

```
C1_C1 : CONTEXT =
BEGIN
   C1_ENUM: TYPE = {S1, S2, <upto>, S8};
   ...
   C1: MODULE =
   BEGIN
      INPUT e1, e2, e3, e4 : BOOLEAN
      OUTPUT t1Enabled, t2Enabled, <upto>, t8Enabled: BOOLEAN
      OUTPUT v1, v2, v3: INTEGER
LOCAL state : { s3, s4, s5, s6, s7, s8 }
   INITIALIZATION
v1 = 0; ...
DEFINITION
t1Enabled = e1 = TRUE && state = s3;
...
   TRANSITION
   [
      ...
      []
      t2Enabled = TRUE --> v1' = 10;
      []
      t3Enabled = TRUE --> v3' = v2;
      []
      ...
   ]
   END;  %% END OF MODULE
   ...
END   %% END OF CONTEXT
```

(Sample SAL Example for Statechart C1 of Figure 1)

3.1 Analysis

Statechart specifications may have non-determinism, races (both read–write and write–write) and unreachable states. These are detected using a model-checker and scalability of model-checkers is an issue. The default model-checker bundled as a plug-in with Statemate cannot analyze large models. We integrated Statemate

with SAL to improve scalability of model-checking. Some such properties in the Statechart C1 include:

- Non-determinism: t7 and t8 can be simultaneously enable hence leads to non-determinism.
- Read–Write Race: When transitions t3 and t6 are taken in the same step, variables v2 gets read and written respective in the actions of t4 and t5.
- Write–Write Race: When transitions t3 and t6 are taken in the same step, variables v3 is assigned to in the actions parts of both t3 and t6.
- Drive to State: There is no input C1 which will drive i to reach state S4! This is because, t2 never gets enabled as whenever the system is in state S3 and the even e1 occurs either t7 or t8 or both will be enabled and both are of higher priority than t2.

4 Translating Statecharts to SAL

A Statechart with the root state as an *or* state or a *basic* state is translated into one single SAL MODULE while Statecharts with *and* root state with *n* direct sub-states is translated into *n* different SAL MODULEs composed synchronously. A SAL variable of *enum* type is generated for capturing current state of the Statechart with *enum literals* corresponding to states.

- All the data-items modified by the Statechart are translated as *OUTPUT* variables of appropriate SAL type.
- All conditions modified and the events raised by the Statechart are translated as *OUTPUT* Boolean variables.
- All data-items. conditions and events *used* in the Statechart are translated as *INPUT* variables of appropriate type.

Additionally each such MODULE will contain Boolean variables corresponding to each transition. this variable is defined to be true whenever the Statechart is in the source state of the transition and it's enabling condition is true. A special Boolean *ElseTransition* variable is defined to be true when none of the transitions are enabled. The state transitions and actions present in the Statechart are translated as guarded actions in the body of the corresponding MODULE. Following is a template SAL specification that illustrates some of these concepts.

Structure of SAL-code for Statechart 1:

```
MODULE S
BEGIN
INPUT
< all variables used in S >
< variables corresponding to events and guards used
  in S >
```

```
OUTPUT
< all variables assigned in S variables generated for
  keeping track of states and transitions >
< variables corresponding to events and guards
  generated by S and used in other charts >
LOCAL
< variables for storing the true/false/changed values
  of conditions used in S >
< variables corresponding to events and guards >
< generated by S and NOT used in other charts >
DEFINITION
< Invariant definitions such as transition enabling
  condition>
TRANSITION
< transition-action pairs for C1's transitions >
< code to set the state variables, new events, etc. >
< code to disable active events in the current step >
END
```

A Statemate model M consisting of many Statechart specifications is translated to a SAL *CONTEXT* that contains variables corresponding to the globals of M and *MODULES* corresponding to M's charts. Execution of these MODULES in parallel is done using the synchronous composition provided by SAL.

Corresponding to each property to be analyzed we generate a Boolean variable that is defined to be true whenever the property is violated, and a theorem that asserts that the property is globally false. For instance, in the case of read–write race the corresponding variable is true whenever an action that reads the variable and an action that writes to it are simultaneously enabled.

4.1 Key Issues

Both Statecharts and SAL have a notion of states and transitions from one state to another. However Statecharts have the following features that do not have a direct mapping in SAL.

- A notion of step and super step where external events are sensed only at the start of the next super step.
- A rich hierarchical structuring mechanism in the form of *and* states and *or* states.

Our solution to these are described below.

4.2 Step and Super Step

Statechart models take the environment events/changes every super step. SAL treats each INPUT variable of any module that is not OUTPUT of any module as an environment input and generates random values for them every single step. To implement super step semantics, we capture all environmental inputs as OUTPUT of one special module *ASYNC_SYS_MOD* and update them with SAL environment inputs only when the system is stable. For example, if there are n environment variables $v1$ to vn, then the special module ASYNC_SYS_MOD will have n INPUT variables env_$v1$ to env_vn which SAL updates every single step. The actual variables representing the Statechart environment variables ($v1$ to vn) will be updated with these only when the system reaches a stable state.

4.3 Implementation of Hierarchy

State transitions are implemented in SAL by having a state variable in a module that takes the value of the current basic state of the Statechart.

Or state maybe a source or target of a transition. For every transition t, of which this is a source state we generate a guarded transition that is enabled whenever the Statechart is in a basic state that is a child of the *or* state and the other enabling conditions of the transition are satisfied. Additionally for every transition that has a lower priority than t we output an additional condition that t should not be enabled thus implementing the priority of transitions. In the case of transitions that end at this state we set the target state as the target state of the default transition of the *or* state.

And state is flattened by taking a cross product of the sub-states of the direct children of the *and* state, and treating the elements of the cross-product as basic states of the Statechart.

5 Optimizations

The naive translation from Statecharts to SAL did not scale up to desired levels and we had to implement some optimizations. Two standard optimizations that gave us maximum benefits are:

- Generating a different SAL specification for each property.
- Partially retaining the hierarchical structure in the case of non-basic *or* states as described above.

However these were not enough and we analyzed all the available specifications for some common syntactic patterns that we could translate differently to achieve better scaling. Following are the ones that gave the maximum benefits.

5.1 Slicing

The Statecharts that were being analyzed were all well structured with different Statecharts implementing different functionality. Therefore for a given functionality only a small subset of the Statecharts were relevant. As a corollary given a Statechart it forms part of a small subset that implements a functionality. More formally, given a Statechart the variable/event set that it used or modified intersected with the variable/event set of only a few other Statecharts. So for a given property our slicing algorithm forms a slice-set of all Statecharts whose variable/event set intersected with the variable/event set of the Statechart on which the property is to be checked or belongs to the slice-set. More formally the sliced set of Statecharts S_s, is the smallest set that contains the Statechart on which the property is to be checked and

$$\forall s \cdot \exists s' \in S_s \cdot input(s') \cap output(s) \neq \{\}$$

5.2 And State as Child of Root or State

In most of the Statecharts we found that the *and* state was the child of the root state. Since this translates naturally to SAL as synchronous composition of two modules, we treated this case separately from a deeply nested *and* state.

5.3 Variable Type Abstraction

Many variables with type *real* or *integer* were used only to compare against a specific constant or to check equality or inequality against other variables of the same type. In the first case we abstracted the type of the variable to *Boolean* and in the second case to the SAL type *SCALARSET*.

6 Experimental Results

The results of running our tool on eight randomly chosen real-life models are presented in Table 1. As can be seen from the entries marked **abort** our tool analyzed more models than the standard model-checker could.

Table 1 Timing results

Model	Read–Write Race		Write–Write Race		Non-Determinism	
	Commercial	SAL-smc	Commercial	SAL-smc	Commercial	SAL-smc
M1	0	0	**abort**	29 m	**abort**	14 m
M2	45 m	11 m:7 s	2 m	9 m	1 m:8 s	5 m:30 s
M3	1 m:20 s	11 m:20 s	14 m:49 s	2 m:26 s	1 m:8 s	3 m:26 s
M4	1 m:20 s	32 s	2 m:9 s	15 s	16 s	5 m:6 s
M5	1 m	10 s	1 m	2 s	0	3 s
M6	43 m	18 s	0	0	**abort**	50 s
M7	1 m	2 s	0	0	25 s	4 s
M8	**abort**	6 m:17 s	**abort**	**abort**	**abort**	19 m

7 Problems and Future Work

We believe there is going to be an increasing need to integrate different tools into a tool-chain, this gives rise to the problem of transformation between their notations being correct. A high level notation to specify analyzable transformations between specification languages will help. We plan to experiment with Object Management Group's Query, View and Transformation [5] language as a notation for this. Since application specific transformations can give a lot of benefit. The ability to independently specify such transformations and analyze these incrementally will be very useful.

References

1. Gerard Berry and Georges Gonthier. The esterel synchronous programming language: design, semantics, implementation. *Sci. Comput. Program.*, 19(2):87–152, 1992.
2. Leonardo de Moura, Sam Owre, and N. Shankar. The sal language manual. Technical Report SRI-CSL-01-02 (Rev. 2), SRI International, Computer Science Laboratory, August 2003.
3. Hans-Georg Frischkorn. Automotive software – the silent revolution. In *Automotive Software Workshop San Diego*, San Diego, United States of America, Jan. 10–12 2004.
4. D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3), June 1987.
5. OMG. Mof qvt final adopted specification. Technical report, 2005.

Performance Debugging of Heterogeneous Real-Time Systems

Unmesh D. Bordoloi, Samarjit Chakraborty, and Andrei Hagiescu

Abstract Today most real-time embedded systems are made up of a heterogeneous collection of processors, communication subsystems and partially programmable or fixed-function components. These are typically supplied by different vendors and as a result have different interfaces, require different programming models and implement different resource scheduling and arbitration policies. Hence, performance analysis and debugging of such systems is increasingly becoming complex. Although a lot of work exists in the real-time systems literature on timing and schedulability analysis of specific task and event models—which can be applied to analyze individual subsystems—the issue of compositionality has not received sufficient attention so far. In this paper we discuss a framework which can help in the analysis and performance debugging of such heterogeneous real-time systems. It can account for a variety of combinations of task and event models and scheduling policies and does not require any global state-space construction. As a result, it is highly scalable and can be used to analyze real-life hardware/software architectures. The main focus of this paper is on illustrating the utility of this framework in analyzing a heterogeneous collection of electronic control units that communicate via a FlexRay bus.

Keywords: Performance modeling and debugging, heterogeneous embedded systems, FlexRay.

1 Introduction

Modern real-time embedded systems are highly complex and distributed in nature. They often consist of a collection of processing and communication elements and run multiple concurrent, communicating tasks. An underlying characteristic of such systems is their heterogeneity—which stems from the different activation rates and execution demands of the different tasks, and the different interfaces and resource scheduling/arbitration policies of the processing and communication elements. Although timing/performance analysis of such individual elements is now fairly well-established in the real-time systems literature, compositional performance analysis is still a challenging problem.

S. Ramesh and P. Sampath (eds.), *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*, 285–300.

Recently, there has been a number of efforts to address this problem (see [8] for an overview). For example, [9] and [7] proposed techniques for composing timing analysis results for individual components of a system which were analyzed using potentially different analysis techniques. However, such a composition technique is only applicable to architectures whose components interact using fairly standard event models such as periodic, periodic with bounded jitter and sporadic. More complex interaction patterns are supported by the framework presented in [1], albeit at the cost of increased analysis complexity. This framework was then extended in a number of subsequent papers to allow the modeling of variable activation rates and execution demands of tasks and complex scheduling policies (see, for example, [2, 4, 5, 10]).

The focus of this paper is on illustrating the utility of this framework through a real-life case study. Towards this, we model a heterogeneous collection of electronic control units (ECUs) that communicate via a FlexRay bus [3]. Each ECU runs a set of tasks and communicates with other ECUs via signals and data streams that are transmitted over the bus. The main challenge is to analyze the different scheduling policies implemented on the ECUs, as well as the FlexRay protocol, and compose these analysis results for end-to-end timing guarantees.

The rest of this paper is organized as follows. In the next section we briefly introduce this framework. This is followed by an overview of the FlexRay protocol in Section 3. In Section 4 we discuss how this protocol is formally modeled for timing analysis. Using an Adaptive Cruise Control application which is mapped onto multiple communicating ECUs, we then show how to perform compositional timing analysis and performance debugging using the abovementioned framework. Finally, Section 6 lists some directions for future work.

2 The Basic Framework

Our system architecture consists of multiple processing elements (PEs) which are connected to a bus. One or more applications are partitioned into tasks and are mapped onto these PEs. Some of these tasks are triggered by external events at a prespecified rate, while the remaining are triggered by data or signals generated by other tasks. Such data/signals might travel over the bus when the two communicating tasks are on different PEs. Once activated, a task needs to be processed and hence consumes a fixed number of processor cycles from the PE it is running on. Finally, each PE might use a different scheduling policy and multiple data streams might attempt to access the communication bus at the same time, and this contention is resolved using some bus arbitration policy. At the heart of the framework being discussed lies the modeling of (i) the triggering pattern of tasks (or the *event model*) which generates an execution demand on a PE and communication demand on the bus, and (ii) the *service* offered by a PE (or the bus) to each task running on it (i.e. the resource model).

Event Model The arrival rate of any event stream triggering a task is upper- and lower-bounded by two functions $\alpha^u(\Delta)$ and $\alpha^l(\Delta)$. Let $R(t)$ be the total number of events that arrive during the time interval $[0, t]$. Then $\alpha^l(\Delta) = \min_{t \geq 0}\{R(t + \Delta) - R(t)\}$ for any Δ. Similarly, $\alpha^u(\Delta) = \max_{t \geq 0}\{R(t + \Delta) - R(t)\}$. Hence, $\alpha^u(\Delta)$ and $\alpha^l(\Delta)$ denote the maximum and minimum number of events that might arrive within *any* interval of length Δ. The timing properties of standard event models—like *periodic, periodic with jitter* and *sporadic*—as well as more arbitrary arrival patterns can be represented by an appropriate choice of α^u and α^l. For example, a periodic event stream with period 9 can be represented by an upper and lower bound shown in Figure 1(a). It is also possible to determine the values of α^u and α^l corresponding to any given arbitrary event trace which was, for example, obtained from a simulation.

Resource Model Similarly, let $\beta^u(\Delta)$ and $\beta^l(\Delta)$ denote upper and lower bounds on the *service* available to a task. Let $S(t)$ be the number of activations of this task that were serviced during the time interval $[0, t]$. Then, $\beta^l(\Delta) = \min_{t \geq 0}\{S(t+\Delta) - S(t)\}$ for any Δ, and $\beta^u(\Delta) = \max_{t \geq 0}\{S(t+\Delta) - S(t)\}$. If there are multiple tasks running on a PE, the service bounds β^u and β^l available to any task will clearly depend on the scheduling policy being used. Further, if $\beta^u(\Delta)$ and $\beta^l(\Delta)$ are expressed in terms of the maximum and minimum number of available *processor cycles*, then they can easily be converted to represent service expressed as the number of task activations that can be serviced within any Δ. This is done by scaling $\beta^u(\Delta)$ and $\beta^l(\Delta)$ with the execution requirement incurred by the task due to each activation.

As an example, the upper and lower bounds on the service in the case of an unloaded PE can be represented as two straight lines that coincide with each other (see Figure 1(b)). The slope of these lines denotes the clock frequency of the PE. Communication resources (e.g. buses) can be similarly modeled, the *service curves* in this case typically bound the number of transmittable bits within any given time interval. Such service curves can be derived from a formal model of the resource, or from data sheets, or in some cases by simple measurements.

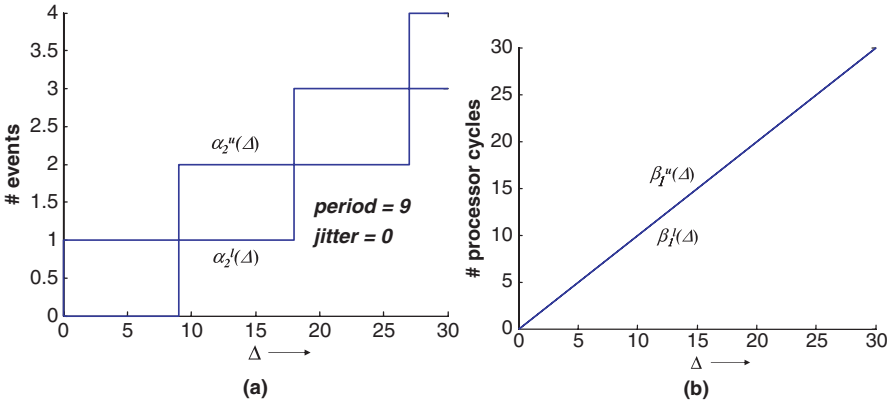


Fig. 1 (a) α^u and α^l corresponding to a periodic activation. (b) β^u and β^l of an unloaded processor

System Composition and Analysis An event stream entering a resource gets processed, thereby generating an outgoing stream of events/data which can activate other tasks on the same PE, or might be transferred over the bus to trigger tasks running on other PEs. Let $\alpha^{u\prime}(\Delta)$ and $\alpha^{l\prime}(\Delta)$ denote upper and lower bounds on the number of such events generated within *any* time interval of length Δ. It can be shown that (see [10]):

$$\alpha^{l\prime}(\Delta) = \min\{\inf_{0\le\mu\le\Delta}\{\sup_{\lambda>0}\{\alpha^{l}(\mu+\lambda)-\beta^{u}(\lambda)\}+\beta^{l}(\Delta-\mu)\},\beta^{l}(\Delta)\}$$
$$\alpha^{u\prime}(\Delta) = \min\{\sup_{\lambda>0}\{\inf_{0\le\mu<\lambda+\Delta}\{\alpha^{u}(\mu)+\beta^{u}(\lambda+\Delta-\mu)\}-\beta^{l}(\lambda)\},\beta^{u}(\Delta)\}$$

Similarly, the bounds on the *remaining service* after processing the activations of a task are given by:

$$\beta^{l\prime}(\Delta) = \sup_{0\le\lambda\le\Delta}\{\beta^{l}(\lambda)-\alpha^{u}(\lambda)\}$$
$$\beta^{u\prime}(\Delta) = \max\{\inf_{\lambda>\Delta}\{\beta^{u}(\lambda)-\alpha^{l}(\lambda)\},0\}$$

Given α^u, α^l and β^u, β^l, it is also possible to compute the maximum *delay* experienced by a task before its activation is serviced and the maximum number of *backlogged* activations. These are: delay $\le \sup_{t\ge0}\{\inf_{\tau\ge0}\{\alpha^u(t) \le \beta^l(t+\tau)\}\}$ and backlog $\le \sup_{t\ge0}\{\alpha^u(t)-\beta^l(t)\}$.

With the help of an example, we now show how a system architecture may be modeled using the above results. Consider the setup shown in Figure 2(a). It consists of two tasks T_1 and T_2 which are being scheduled using a rate monotonic scheduler. Both T_1 and T_2 are activated periodically, with T_1's period being 4 time units and T_2's period being 9 time units. Each activation of T_1 and T_2 requires 1 and 2 processor cycles respectively to process. The upper and lower bounds on the activation of T_2 (i.e. α_2^u and α_2^l) were shown in Figure 1(a). They are similar for T_1, except for the

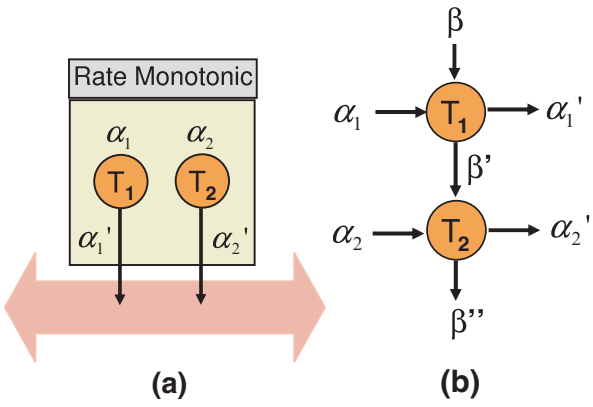


Fig. 2 (a) Rate monotonic scheduling of two tasks. (b) Corresponding scheduling network

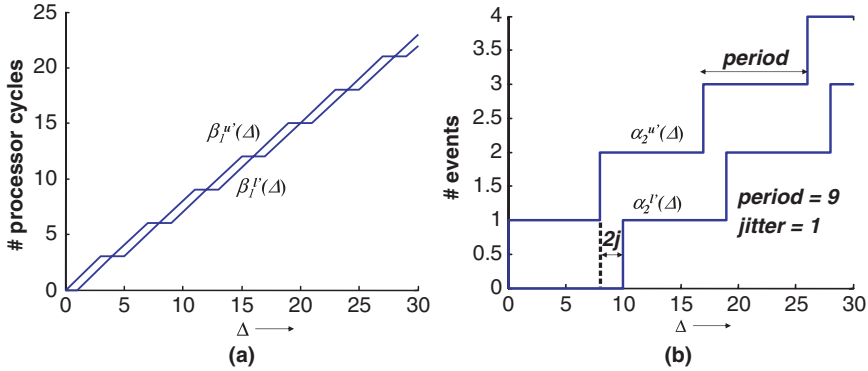


Fig. 3 (a) Bounds on the *remaining* service after processing task T_1. (b) Bounds on the messages generated by T_2

difference in the length of the period. The upper and lower bounds on the service offered by the unloaded PE (in terms of the number of processor cycles available over any time interval) were shown in Figure 1(b). Since T_1 has a smaller activation period, it has a higher priority (because of rate monotonic scheduling) and hence the full service offered by the unloaded PE is available to it.

As discussed above, using α_1^u, α_1^l and β_1^u, β_1^l, we can compute $\beta_1^{u\prime}$ and $\beta_1^{l\prime}$, which are bounds on the *remaining* service (that is left over after processing T_1). This remaining service is now available to the lower-priority task (i.e. T_2). This concept is illustrated in the form of a *scheduling network* for a rate monotonic (or any fixed priority) scheduler in Figure 2(b).

$\beta_1{}'$ is used for servicing task T_2 (see Figure 3(a)), which along with α_2 can be used to compute upper and lower bounds on the events generated by each serviced activation of T_2 (β and α often refer to the tuples β^u, β^l and α^u, α^l). These bounds are shown in Figure 3(b). From this figure, note that this event stream is periodic with a period of 9 time units and a jitter of 1 time unit. It is straightforward to see that the distance between $\alpha_2^{u\prime}$, $\alpha_2^{l\prime}$ is equal to twice the jitter of the event stream.

So far we described how to use this framework to analyze a PE, but the same technique is also applicable to communication resources (e.g. buses). To illustrate this, we now model the complete architecture (along with the communication bus) shown in Figure 2(a). Assume that the bus transmits the processed streams $\alpha_1{}'$ and $\alpha_2{}'$ as messages to another PE (which is not shown in this architecture). The performance model of the complete architecure including the bus is now shown in Figure 4(a). Suppose that each serviced activation of T_1 and T_2 generates a message of size 1 byte that is to be transmitted over the bus. The $TDMA$ scheduler running on the bus has a cycle length of 10 time units and provides slot sizes that are suitable for transmitting 4 and 3 bytes of data from T_1 and T_2 respectively during every cycle. The service curves corresponding to this bus availability to T_1 is shown in Figure 4(b).

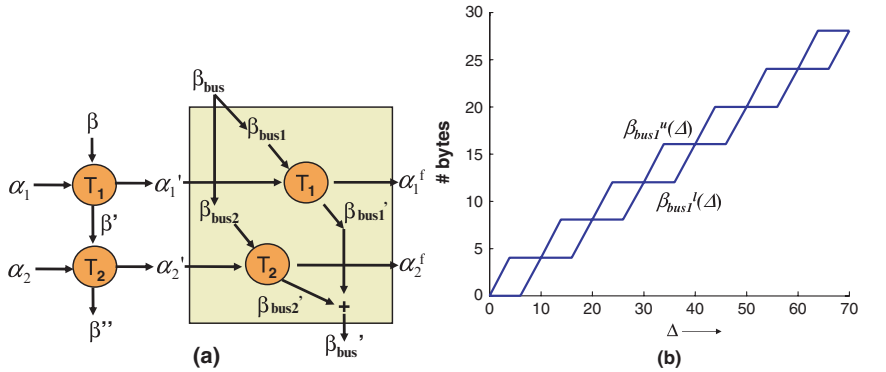


Fig. 4 (a) Performance model of the complete architecture. (b) The bounds on the service available on the *TDMA* bus to messages from T_1

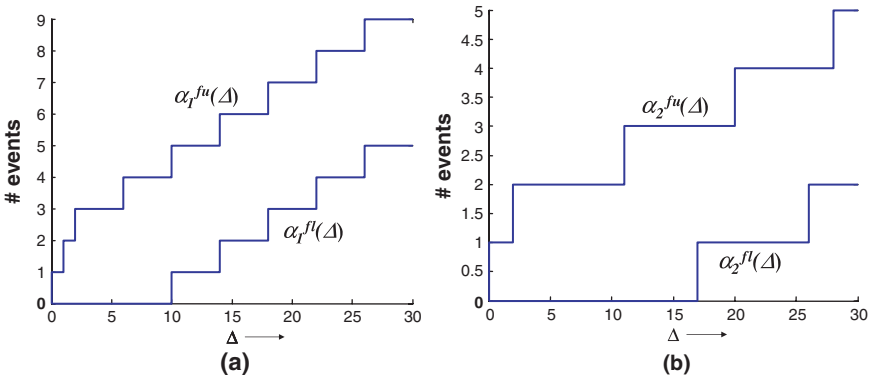


Fig. 5 (a) Upper and lower bounds on the transmitted messages over the bus arising from T_1. (b) Bounds on the transmitted messages from T_2

Finally, Figure 5 shows the timing properties (or bounds on the arrival rate) of the transmitted messages from T_1 and T_2. From the timing properties of the message stream injected by T_2 on the bus (Figure 3(b)) and the timing properties of these transmitted messages (Figure 5(b)), it may be noted that the jitter increases from 1 to 7.5 time units. These transmitted messages can now trigger tasks running on other PEs and the same procedure may be applied to analyze them as well.

3 The FlexRay Protocol

As mentioned in Section 1, the rest of this paper is concerned with modeling an Adaptive Cruise Control (ACC) application running on multiple ECUs that communicate via a FlexRay bus. Towards this, in this section we give a brief overview of the FlexRay communication protocol.

Communication in FlexRay takes place in periodic cycles and each communication cycle is partitioned into an ST and a DYN segment. The lengths of these segments need not be equal, but are fixed over the different cycles (hence these lengths are among the parameters that need to be determined when the FlexRay schedule is synthesized). The ST segment is further partitioned into a fixed number of equal-length slots. Each slot is allocated to a specific task and a task is allowed to send a message only during its allocated slot. If a task has no messages to send, then its slot goes empty (i.e. other tasks are not allowed to use it).

The DYN segment is also partitioned into equal-length slots, but each slot size is much smaller and is referred to as a *minislot*. Tasks which send messages on the DYN segment are assigned fixed priorities. At the beginning of each DYN segment, the highest priority task is allowed to send a message. The length of such a message can be arbitrarily long (i.e. can occupy an arbitrary number of minislots), but has to fit within one DYN segment. However, if the task has no messages to send, then only one minislot goes empty. In either case, the bus is then given to the next highest-priority task and the same process is repeated till the end of the DYN segment. Further, when its turn comes, a task is only allowed to send a message if it fits into the remaining portion of the DYN segment. For further details of this protocol, we refer the reader to [6] or to the full specification [3].

As an example, consider eight tasks $T_1, \ldots, T_8$ mapped onto different ECUs, which send messages on the FlexRay bus. Any message sent by a task T_i is labeled as m_i. Tasks T_1, T_2 and T_3 send messages over the ST segment and T_4 to T_8 over the DYN segment. For the DYN segment, the priorities of the tasks decrease from T_4 to T_8. Figure 6 shows two consecutive FlexRay communication cycles resulting from this mapping. In the first cycle, task T_2 has no message to send (hence the corresponding slot in the ST segment is empty) and in the second cycle T_1 and T_3 have nothing to send.

Similarly, in the first cycle, tasks T_5, T_6 and T_7 have messages to send, but not T_4 and T_8. Hence, there is one empty minislot corresponding to T_4 in the DYN segment, followed by the message m_5. The size of m_6 is bigger than the remaining length of the DYN segment, hence it is not sent; instead there is one empty minislot in its place. This is followed by m_7 and another empty minislot resulting out of no message from T_8. In the second cycle, T_4 and T_5 have no messages to send, which results in two

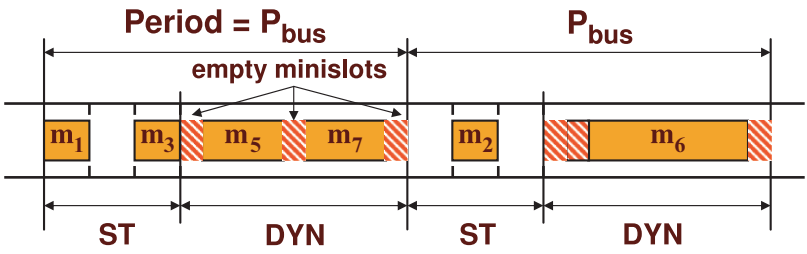


Fig. 6 Two typical FlexRay communication cycles

empty minislots. These are followed by m_6 which could not be sent in the first cycle. The DYN segment ends with one empty minislot which might either be because T_7 had nothing to send or its message was longer than 1 minislot.

Below, we only highlight the modeling of the DYN segment of FlexRay. The ST segment uses a TDMA scheme which can easily modeled, as illustrated in the last section.

Difficulties in Modeling FlexRay We just saw that in FlexRay, the task with the highest priority is offered access to the bus at the start of the DYN segment. Further, once given access to the bus, a task can occupy it till the end of the current DYN segment. Hence, the most straightforward approach would be to model this protocol as a fixed priority scheduler, as shown in Figure 2(b). Here, β would be used to model the total service offered by the DYN segment and successive βs would be computed from the message sizes and message generation rates of the different tasks. However, this approach does not work because of the following properties of FlexRay: (i) A task is only allowed to send a message if it fits into the remaining portion of the DYN segment, i.e. a message cannot straddle two communication cycles. (ii) Once a task misses its turn in the DYN segment (because there were no ready messages), it has to wait till the next communication cycle before it can access the bus (which is the TDMA-like property of the DYN segment). (iii) A task can send at most one message in each DYN segment (where the maximum length of the message can be equal to the length of the DYN segment).

The modeling framework presented in Section 2 does not incorporate these restrictions when representing the service availability of a resource using the upper and lower bounds $\beta^u(\Delta)$ and $\beta^l(\Delta)$. To see this, consider Figure 7(a), which shows α^u corresponding to the arrival of a single message (of length equal to 10 minislots) that is to be transmitted over the DYN segment (of length 8 minislots). Here, the length of each communication cycle (or period) is assumed to be p time units and the length of the DYN segment is equal to d time units. The lower bound on the service β^l corresponding to the DYN segment is also shown in this figure. Note that over time intervals Δ of length less than or equal to $p - d$, no service might be available from

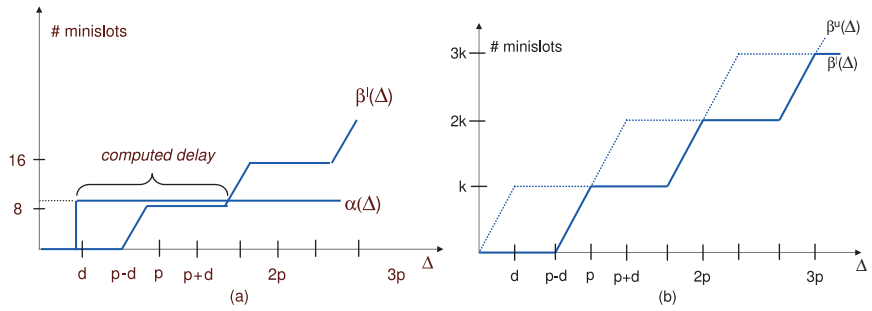


Fig. 7 (a) Computing maximum delay from α^u and β^l. (b) Total service offered by the DYN segment

the DYN segment due to the blocking by the ST segment. Since the length of the message in this case is longer than the length of the DYN segment, this message will never get transmitted. However, the framework we described in Section 2 models the message to be transmitted over two communication cycles, thereby incurring a delay equal to the maximum horizontal distance between α^u and β^l (see Figure 7(a)).

4 Formal Timing Analysis of FlexRay

To correctly model the DYN segment of FlexRay, we need to modify our representation of the service bounds $\beta^u(\Delta)$ and $\beta^l(\Delta)$ to reflect the FlexRay-specific properties listed in Section 3. Towards this, assume that tasks $T_1, \ldots, T_n$ send messages over the DYN segment with any message from task T_i being denoted by m_i and has a length of k_i minislots. The length of the DYN segment is assumed to be equal to k minislots (or d time units) and the length of a communication cycle, as before, is equal to p time units. Each minislot is assumed to be MS time units long.

Let $\beta^l(\Delta)$ be the lower bound on the service (expressed in terms of number of minislots) offered by the unloaded DYN segment to all the tasks. Further, let β_i^l be the service offered by the DYN segment to task T_i. To obtain β_1^l, the function β^l needs to be transformed using the following steps.

1. Extract k_1 minislots of service during each communication cycle from β^l. This is because during any communication cycle at most k_1 minislots are available to T_1 (since a task can send at most one message).
2. Discretize the service bound obtained from Step (i), i.e. convert it into a step-function. This is to model that a message cannot straddle two communication cycles. Steps (1) and (2) are shown in Figure 8(a).
3. The resulting service bound is shifted by d time units. This is again to model that a message has to be completely sent within a single DYN segment. Note from Figure 8(c) that any interval Δ of length less than $p + \mathtt{MS} \times k_1$ can be positioned to straddle two communication cycles. Hence, the minimum service available from the DYN segment over intervals of such length is equal to 0. The shifted service bound in Figure 8(b) reflects this. It also reflects the property that once a task misses its turn in the DYN segment, it has to wait for the next communication cycle.

The resulting service bound, which we denote as β_1^l correctly represents the minimum or guaranteed service from the DYN segment that is available to messages from T_1. This β_1^l can now be plugged into the framework outlined in Section 2 to compute the maximum delay suffered by any m_1, the maximum number of backlogged m_1s and the timing properties of the transmitted messages (which might trigger other tasks). Towards this $\alpha_1^u(\Delta)$ is used as an upper bound on the number of messages generated by T_1 within any interval of length Δ.

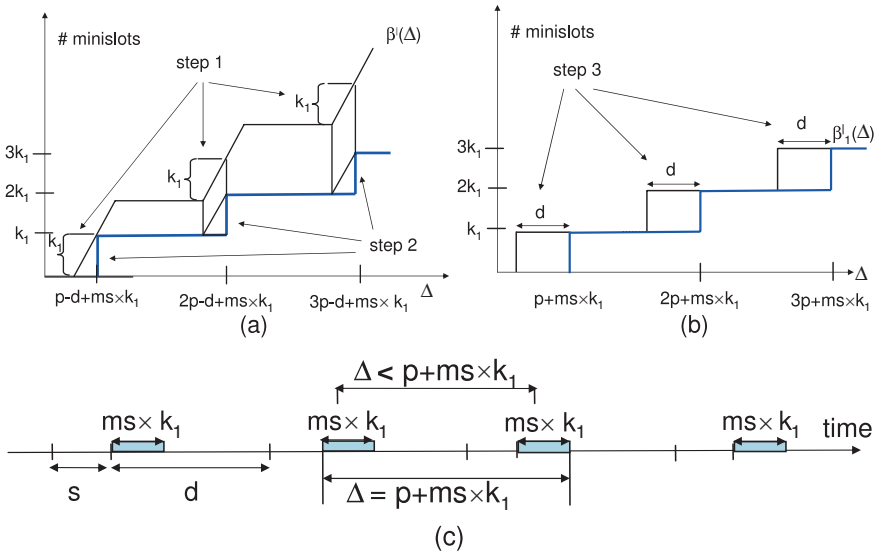


Fig. 8 (a) Steps (1) and (2) for transforming β^l. (b) Shifting the resulting service bound. (c) Blocking time

The service available to the lower priority tasks (i.e. $T_2, \ldots, T_n$) is made up of two components: (i) The remaining service left after performing transformation 1 (i.e. the service that was *unavailable* to T_1). This is given by $\bar{\beta}^l(\Delta) = \sup_{0 \le \lambda \le \Delta}\{\beta^l(\lambda) - \beta_1^l(\lambda)\}$. (ii) The service that was *unutilized* by T_1. This can be computed from β_1^l and α_1^u and is denoted by $\beta_1^{l\prime}$. However, $\beta_1^{l\prime}$ cannot be directly added to $\bar{\beta}^l$ because it is specific to messages from task T_1 (i.e. incorporates message size information). So it first needs to be transformed by applying the "inverse" of Steps (2) and (3) that were applied to β^l, and the resulting function is added to $\bar{\beta}^l$. This sum then represents the service available to the lower priority tasks, which is transformed in the same way as β^l, but using information specific to messages from task T_2. This procedure is then repeated for all the tasks $T_3, \ldots, T_n$.

To illustrate this scheme, consider the architecture shown in Figure 9(a), consisting of two tasks T_1 and T_2 running on two different ECUs. T_1 generates a periodic stream of messages (denoted by m_1) which are transmitted over the DYN segment of a FlexRay bus. The transmitted messages trigger T_2 on ECU_2, which in turn generates a stream of messages m_2 which are also transmitted over the DYN segment of the same bus to an actuator. m_1 is assigned a higher priority than m_2 and both m_1 and m_2 cannot fit into one DYN segment. In this context, Figure 9(b) show an application of our model. Here, α_1 bounds the arrival rate of m_1 at the bus and β is the service offered by the unloaded bus. β_1 is the service available to m_1. β' is the service remaining from β (i.e. *unavailable* to m_1). $\overline{\beta}'$ is the service that is *unutilized* by m_1 (from what was available to it). The sum of $\overline{\beta}$ and $\beta_1{}'$ is the service available to m_2. Finally, the triggering rate of T_2 (which is equal to the arrival rate of m_2 at the bus) is bounded by $\alpha_1{}'$, that is computed from α_1 and β_1.

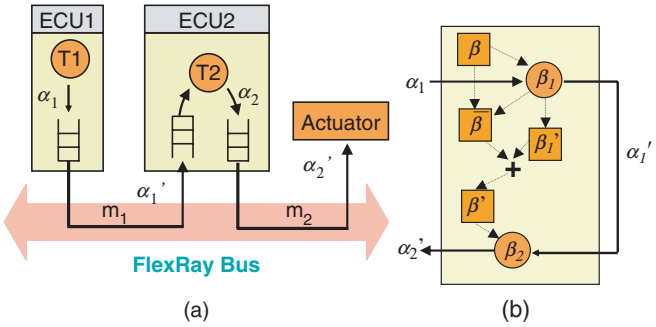


Fig. 9 (a) Example architecture. (b) Overview of the FlexRay model

5 Adaptive Cruise Control Application: A Case Study

We will now show the utility of the framework discussed in Section 2 in modeling an Adaptive Cruise Control (ACC) application. This is followed by an illustration of how this model can be used for formal performance analysis and debugging of an architecture consisting of multiple heterogeneous ECUs communicating via a FlexRay bus. When compared to simulation-oriented approaches—which can be time consuming and do not provide any formal guarantees—our framework can be used to quickly evaluate multiple design choices to determine whether they meet the performance constraints at hand. The main challenge here is to determine end-to-end timing properties of event/data streams which pass through multiple ECUs (implementing different scheduling policies) and the FlexRay bus. Each of these processing/communication elements modify the timing properties of the stream as it passes through it.

System Description As shown in Figure 10, the ACC subsystem consists of four ECUs communicating via a FlexRay bus. The bus has a communication cycle of 14 ms. The length of the DYN segment is of 6 ms and consists of 84 minislots, and the length of ST segment is 8 ms. Each minislot in the DYN segment can accommodate 2 bytes of data. $ECU1$ receives data from two radar sensors periodically every 50 ms, and $ECU3$ periodically receives data from a wheel sensor every 25 ms.

The data received by $ECU1$ from each radar is processed by an *Object Detection* task. The processed data streams m_1 and m_2 are sent over the DYN segment of the FlexRay bus to $ECU2$ to be processed by the *Data Fusion*, *Object Selection* and *Adaptive Cruise Control* tasks. The resulting data stream m_3 is transmitted over the ST segment of the bus to $ECU3$ which runs the *Throttle and Brake Arbitration* task. This task also receives input from the *Anti-lock Braking System* which runs on $ECU4$. This input is sent as the message stream m_7 over the ST segment of the bus. The output from the *Throttle and Brake Arbitration* task is fed into the *Brake Control* and *Throttle Control* tasks, which in turn sent their outputs to two different actuators. These final output control signals are bounded by the functions α_B^f and

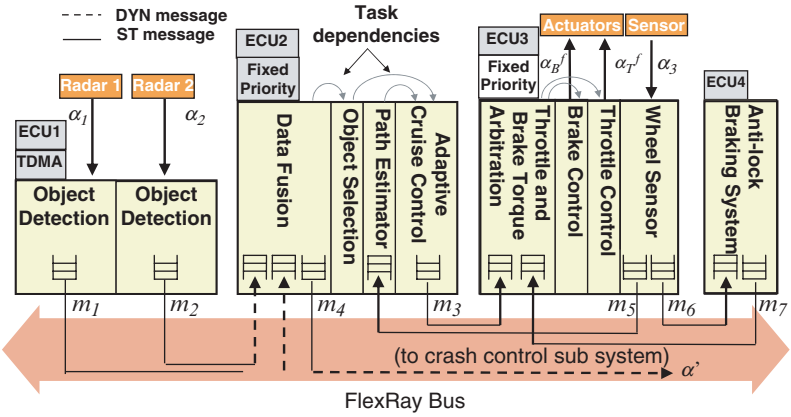


Fig. 10 The system architecure of an Adaptive Cruise Control subsystem

Table 1 The workload on the FlexRay bus and the ECUs for the ACC subsystem

Bus		**ECUs**	
Message	*# Bytes*	*Task*	*WCET*
m_1	128	Data Fusion	14.78 ms
m_2	128	Object Selection	1.45 ms
m_3	4	Adaptive Cruise Control	0.66 ms
m_4	64	Arbitration	0.88 ms
m_5	4	Path Estimation	1.67 ms
m_6	4	Brake Control	0.45 ms
m_7	4	Throttle Control	0.30 ms
		Anti-Lock Braking System	7 ms
		Wheel Sensor	0.13 ms
		Object Detection	6 ms

α_T^f respectively. $ECU2$ also transmits a message stream m_4 to a *Crash Control* subsystem via the DYN segment of the bus. Finally, the *Wheel Sensor* task running on $ECU3$ receives its input from a sensor, processes it, and sends messages m_5 and m_6 to $ECU2$ and $ECU4$ respectively via the ST segment.

In Figure 10, the dashed lines represent messages transmitted via the DYN segment of the FlexRay bus (m_1 has the highest priority, followed by m_2 and then m_4), while the solid lines represent messages transmitted via its ST segment. The arrows between tasks in $ECU2$ and $ECU3$ represent data dependencies (i.e. data from the incoming arrow flows into the task pointed to by the arrow). It may be noted that $ECU1$ uses a TDMA policy to schedule the tasks running on it, and ECUs 2 and 3 use a fixed-priority scheduler. Finally, Table 1 shows the lengths of the different messages and the execution times of the various tasks running on the different ECUs.

Performance Debugging For above the ACC subsystem, we computed performance metrics like end-to-end delays (radar to actuators), delays experienced by individual message streams, and buffer requirements at the ECUs. If some of these metrics violate prespecified constraints, we show how to use our framework to debug a design in order to satisfy these constraints.

Towards this, we implemented the FlexRay model described in Section 4 using a combination of Java and Matlab. Our implementation also models basic scheduling policies like fixed priority and TDMA, as outlined in Section 2. Based on the setup outlined above, Figure 11(a) shows the lower bounds on the resource availability for the DYN segment of the FlexRay bus. In this figure, β denotes the lower bound on the availability of the *unloaded* DYN segment of the bus. Similarly, β^f denotes the lower bound on the *remaining capacity* of this segment after accommodating all the message streams that have been mapped onto it. β'_{m1} and β'_{m2} denote lower bounds on the availability of the DYN segment *after* accommodating the message streams m_1 and m_2.

It may be noted from Table 1 that messages from both m_1 and m_2 cannot fit into the same DYN segment because their total length exceeds the capacity of a single DYN segment (168 bytes). From β'_{m1} (i.e. the bus capacity available to m_2) and β'_{m2} it can be observed that there can be time intervals of up to 35 ms during which m_2 cannot access the bus, although the bus is already available. This is because a message can be sent over the DYN segment only if it fits into a single segment. Finally, Figure 11(b) shows the lower bounds on the arrival rates of the data from the two radars and the wheel sensor. Since these data streams are periodic, the upper bounds would be similar. This figure also shows the upper (α_T^{fu}) and lower (α_T^{fl}) bounds on the final output stream that feed into the throttle actuator. As explained in Section 2, from these bounds it is possible to compute the maximum jitter of this stream.

The computed end-to-end delay along the path from *Object Detection* to *Data Fusion* (via the FlexRay bus) to the crash control subsystem is equal to 243.11 ms. This delay includes the waiting time of a message at the two ECUs ($ECU1$ and

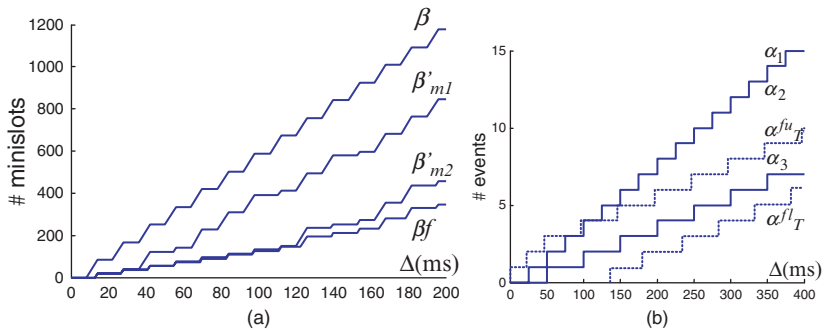


Fig. 11 (a) The bounds on the resource curves for the DYN segment. (b) The bounds on the input and the output signals for the system

Table 2 Delay and buffer requirement of each message stream on the FlexRay bus

ST Segment			DYN Segment		
Message	*Delay*	*Buffer*	*Message*	*Delay*	*Buffer*
m_3	37.08 ms	8 Bytes	m_1	18.57 ms	128 Bytes
m_5	21.8 ms	4 Bytes	m_2	54.64 ms	256 Bytes
m_6	21.8 ms	4 Bytes	m_4	151.39 ms	256 Bytes
m_7	23.2 ms	8 Bytes			

$ECU2$), as well as the delay experienced in the bus. On the other hand, the maximum end-to-end delay from any of the radars or the wheel sensor, to an actuator is equal to 141.62 ms. The messages sent to the crash control subsystem experience a higher delay because of the DYN segment of the bus being more heavily loaded than the ST segment. The delay and buffer requirements of all the different message streams are listed in Table 2. These buffer sizes refer to the input buffers in which messages are stored while they wait to access the FlexRay bus.

Given a set of performance constraints, this framework can now be used to quickly evaluate whether a given design meets specified constraints. Further, it can also be used to evaluate delay and buffer requirements of individual message streams and ECUs, which can provide insights into performance bottlenecks and potential hotspots in an architecture. Such insights can also help in appropriate resource dimensioning. Finally, this framework can also help in determining appropriate combinations of scheduling parameters and activation rates of the different tasks for optimal performance under specified resource constraints. The design of all modern embedded systems involve determining the values of many system parameters, which influence each other in complex ways. As a result, their impact on various performance metrics is not immediately clear.

In what follows, we illustrate how this framework can be used to evaluate the impact of various parameters on the end-to-end delay from the radar to an actuator in the ACC subsystem. Two such parameters that directly affect the delay directly are: (i) the lengths of the DYN and ST segment of the FlexRay bus, and (ii) the data arrival rates from the radars/sensors. Figure 12(a) shows how the end-to-end delay varies for various combinations of ST and DYN segment lengths for a bus cycle length (or period) of 14 ms (with all the other parameters being as described above). It turns out that this delay attains its minimum value when the length of the DYN segment is equal to 9 ms and the length of the ST segment is 5 ms. Finally, Figure 12(b) shows the variations in the end-to-end delay for different sampling rates (or periods) of the radar and the wheel sensor. The lengths of the ST and DYN segments are assumed to be as described earlier (i.e. 8 and 6 ms respectively). It can be seen that a smaller period of the sensor leads to smaller end-to-end delays, albeit at the cost of some information loss due to the lower sampling rate. However, the decrease in the delay is not always linearly proportional to the reduction in the sampling rate.

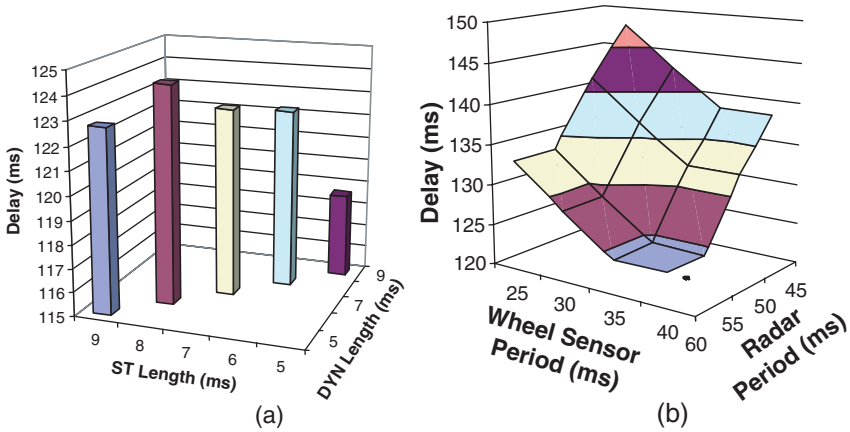


Fig. 12 Performance Debugging: (a) Influence of ST and DYN segment lengths on the end-to-end delay. (b) Influence of sampling rates on the end-to-end delay

6 Concluding Remarks

As a part of future work, we plan to implement this framework as a plug-in for standard FlexRay-based tools such as those from DECOMSYS. On the modeling side, we plan to extend it to be able to model correlations between different streams. In the current form, each data or message stream is specified independent of the other streams. In cases where two or more streams are correlated, such correlation information might lead to tighter bounds on timing and resource usage.

Acknowledgements Thanks are due to P. Sampath, V. Ganesan and S. Ramesh for numerous discussions and help with the case study presented here.

References

1. S. Chakraborty, S. Kunzli, and L. Thiele. A general framework for analysing system properties in platform-based embedded system designs. In *DATE*, 2003.
2. S. Chakraborty, L. T. X. Phan, and P. S. Thiagarajan. event count automata: a state-based model for stream processing systems. In *RTSS*, 2005.
3. The FlexRay Communications System Specifications, Ver 2.1. `www.flexray.com`, 2005.
4. A. Maxiaguine, S. Kunzli, and L. Thiele. Workload characterization model for tasks with variable execution demand. In *DATE*, 2004.
5. A. Maxiaguine, Y. Zhu, S. Chakraborty, and W.-F. Wong. Tuning soc platforms for multimedia processing: identifying limits and tradeoffs. In *CODES+ISSS*, 2004.
6. T. Pop, P. Pop, P. Eles, Z. Peng, and A. Andrei. Timing analysis of the FlexRay communication protocol. In *ECRTS*, 2006.
7. K. Richter and R. Ernst. Model interfaces for heterogeneous system analysis. In *DATE*, 2002.

8. K. Richter, M. Jersak, and R. Ernst. A formal approach to MpSoC performance verification. *IEEE Computer*, 36(4):60–67, 2003.
9. K. Richter, D. Ziegenbein, M. Jersak, and R. Ernst. Model composition for scheduling analysis in platform design. In *DAC*, 2002.
10. E. Wandeler, A. Maxiaguine, and L. Thiele. Quantitative characterization of event streams in analysis of hard real-time applications. *Real-Time Systems*, 29(2-3):205–225, 2005.