First International Workshop on Formal Techniques for Safety-Critical Systems (FTSCS 2012)

Preliminary Proceedings

Editors: Cyrille Artho and Peter Csaba Ölveczky

Preface

This volume contains the preliminary proceedings of the *First International Workshop of Formal Techniques for Safety-Critical Systems* (FTSCS 2012), held in Kyoto, Japan, on November 12, 2012, as a satellite event of the ICFEM conference.

The aim of this workshop is to bring together researchers and engineers who are interested in the application of formal and semi-formal methods to improve the quality of safety-critical computer systems. FTSCS strives strives to promote research and development of formal methods and tools for industrial applications, and is particularly interested in industrial applications of formal methods. Specific topics include, but are not limited to:

- case studies and experience reports on the use of formal methods for analyzing safety-critical systems, including avionics, automotive, medical, and other kinds of safety-critical and QoS-critical systems;
- methods, techniques and tools to support automated analysis, certification, debugging, etc., of complex safety/QoS-critical systems;
- analysis methods that address the limitations of formal methods in industry (usability, scalability, etc.);
- formal analysis support for modeling languages used in industry, such as AADL, Ptolemy, SysML, SCADE, Modelica, etc.; and
- code generation from validated models.

The workshop received 25 submissions; 21 of these were regular papers and 4 were tool/work-in-progress/position papers. Each submission was reviewed by three referees; based on the reviews and extensive discussions, the program committee selected 10 regular papers, one work-in-progress paper, and two position papers for presentation at the workshop and inclusion in this volume. In addition, our program also includes an invited talk by Ralf Huuck.

Revised versions of accepted regular papers will appear in the post-proceedings of FTSCS 2012 that will be published as a volume in *Electronic Proceedings in Theoretical Computer Science*. Extended versions of selected papers from the workshop will also appear in a special issue of the *Science of Computer Programming* journal.

Many colleagues and friends have contributed to FTSCS 2012. First, we would like to thank Kokichi Futatsugi and Hitoshi Ohsaki for proposing to organize this workshop. We thank Ralf Huuck for accepting our invitation to give an invited talk and the authors who submitted their work to FTSCS 2012 and who, through their contributions, make this workshop an interesting event. We are particularly grateful that so many well known researchers agreed to serve on the program committee, and that they all provided timely, insightful, and detailed reviews.

We also thank the editors of *Electronic Proceedings in Theoretical Computer Science* for agreeing to publish the proceedings of FTSCS 2012 in their journal, Bas van Vlijmen for accepting our proposal to devote a special issue of the *Science of Computer Programming* journal to extended versions of selected papers from FTSCS 2012, Kenji Taguchi and Hiromi Hatanaka for their invaluable help with local and visa arrangements, and Andrei Voronkov for the excellent EasyChair conference systems.

We hope that you will all enjoy both the scientific program and the workshop venue!

Amagasaki and Oslo, November 2012 — Cyrille Artho and Peter Csaba Ölveczky

Program Chairs

Cyrille Artho	AIST
Peter Csaba Ölveczky	University of Oslo

Program Committee

Erika Ábrahám	RWTH Aachen University
Musab AlTurki	King Fahd University of Petroleum and Minerals
Farhad Arbab	CWI and Leiden University
Cyrille Artho	AIST
Saddek Bensalem	VERIMAG
Armin Biere	Johannes Kepler University
Peter Bokor	Technical University Darmstadt
Santiago Escobar	Technical University of Valencia
Bernd Fischer	University of Southampton
Klaus Havelund	Jet Propulsion Laboratory, California Institute of Technology
Marieke Huisman	University of Twente
Ralf Huuck	NICTA
Fuyuki Ishikawa	National Institute of Informatics
Takashi Kitamura	AIST
Alexander Knapp	Universität Augsburg
Yang Liu	National University of Singapore
Steven Miller	Rockwell Collins
Tang Nguyen	AIST
Thomas Noll	RWTH Aachen University
Peter Csaba Ölveczky	University of Oslo
Grigore Roşu	University of Illinois at Urbana-Champaign
Neha Rungta	NASA Ames Research Center
Carolyn Talcott	SRI International
Tatsuhiro Tsuchiya	Osaka University

Additional Reviewers

Bentea, Lucian	Bozga, Marius
Corzilius, Florian	Khanh, Nguyen Truong
Liu, Shuang	Loup, Ulrich
Nellen, Johanna	Santiago, Sonia
Shi, Jiangfan	

Table of Contents

Formal Verification, Engineering and Business Value

Ralf Huuck
NICTA
Sydney, Australia
School of Computer Science and Engineering
University of New South Wales
Sydney, Australia
ralf.huuck@nicta.com.au

How to apply automated verification technology such as model checking and static program analysis to millions of lines of embedded C/C++ code? How to package this technology in a way that it can be used by software developers and engineers, who might have no background in formal verification? And how to convince business managers to actually pay for such a software? This work addresses a number of those questions. Based on our own experience on developing and distributing the Goanna source code analyzer for detecting software bugs and security vulnerabilities in C/C++ code, we explain the underlying technology of model checking, static analysis and SMT solving, steps involved in creating industrial-proof tools.

1 Motivation

Formal verification has come a long way from being a niche domain for mathematicians and logicians to an accepted practice, at least in academia, and frequently being taught in undergraduate courses. Moreover, starting out from a pen-and-paper approach, a range of supporting software tools have been developed over time including specification tools for (semi-)formal languages such as UML, Z or various process algebrae, interactive theorem-provers for formal specification, proof-generation and verification, as well as a large number of algorithmic software tools for model checking, run-time verification, static analysis and SMT solving to name a few [5].

Despite all the effort, however, there has been only limited penetration of verification tools into industrial environments, mostly remaining confined to the respective R&D laboratories of larger corporations, defense projects or selective avionics work. The use of verification tools by the average software engineer is rare and typically stops at formal techniques built into the compiler or debugger.

One of the key motivations for our work has been to contribute to some of the technology transfer from classical academic domains to industrial applications and use. In particular, we have been working on bringing verification techniques such as model checking [8, 2], abstract interpretation [3] and SMT solving [4] to professional software engineers.

To make verification technology applicable, we believe a number of core principles must be met: First of all, any verification tool has to be so simple to use that it does not require much or any learning from users outside the formal methods domain. Secondly, the performance of the tool has to match the typical workflow of the end-user. This means, if the end-user is accustomed to doing things in a particular order, those steps should remain largely the same. Moreover, run-time performance of any additional analysis or verification should be similar to existing processes. Finally, and most importantly, a new analysis tool should provide real value to an end-user. This means, it should deliver information or a degree of reliability that was previously not available, making the use of the tool worthwhile.

Submitted to:
FTSCS 2012

2 Software Tool Challenges

The result of our endeavor is *Goanna* [6], an automated software analysis tool for detecting software bugs, code anomalies and security vulnerabilities in C/C++ source code. Goanna is designed to be run at compile-time and does not require any annotations, code modifications or user interaction. Moreover, the tool can directly be integrated into common development environments such as Eclipse, Visual Studio or build systems based on, e.g., Makefiles. To achieve acceptance in industry, all formal techniques are hidden behind a typical programmer's interface, all of C/C++ is accepted as input (even, e.g., Microsoft specific compiler extensions) and scalability had to be ensured for many millions of lines of code in reasonable time.

To achieve this, a number of trade-offs had to be made: While using a range of formal verification techniques, Goanna is not a verification tool as such, but rather a bug detection tool. This means, it does no conclusively show the absence of errors, but does its best to find certain classes of bugs. Moreover, Goanna comes by default with a fixed set of pre-defined checks for errors such as buffer overruns, memory leaks, NULL-pointer dereferences, arithmetic errors, or C++ copy control mistakes as well as with support for certain safety-critical coding standards such as MISRA [1] or CERT [9] totaling over 200 individual checks. To achieve reasonable performance that is of the same order as the compiler, a number of assumptions and approximations (as well as refinements) are made. Naturally, this leads to missed errors (false negatives) as well as to spurious errors (false positives). Finding the right balance between precision, speed and number of supported checks is very much an engineering art supported by new verification and abstraction techniques. This work will focus on some key insights:

Core Analysis The core of our program analysis is based on CTL model checking. In particular, most static analysis tasks such as NULL-pointer detection, buffer overrun analysis and memory leak detection are outsourced to an explicit state CTL model checker and an abstract interpretation framework. The approach of model checking the distribution of syntactic elements in C/C++ code creates particular requirements such as handling many small verification tasks. We implemented our own explicit state model checker outperforming some existing state-of-the-art implementation by up to an order of magnitude.

Refinements & Heuristics Goanna, in its academic version, integrates with SMT solvers and applies a refinement loop for distinguishing spurious errors from real ones [7]. Moreover, a substantial engineering effort has been done to apply heuristics matching common programmer patterns and expectations to keep the overall false-alarm rate to a minimum.

Engineering Challenges While formal methods tend to be precise and define right and wrong, software practice is often much less rigorous and one person's bug is another person's feature. Moreover, C/C++ are complex languages with often tricky semantics and, more importantly, are neither stable with new features being added through the C++11 standard or compiler extensions, nor are these languages the same for everyone, but rather determined by how the individual compiler manufacturer supports and interprets C/C++.

Internally, we have to deal with huge number of verification tasks. This can range to hundreds of millions of model checking specifications for larger projects and sometimes billions of pre-processing pattern matching queries. This is typically contrary to an academic environment, where one is often interested in a few complex verification tasks only. Apart from new algorithms this requires dedicated engineering solutions for efficient caching, incremental analysis, inter-function summaries and databases for storage and look-up between runs.

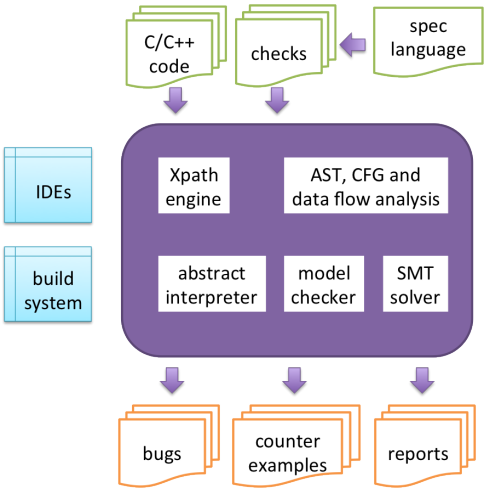


Figure 1: High-level Goanna Architecture

Business Value Interestingly, software analysis plays only a small role in the overall software development lifecycle and general product development process. Most importantly, organizations need to get products to market as fast as possible with as many features as possible. Bugs are of concern if they lead to critical failure and therefore a loss in business. A bug that never or very rarely manifests is often not of primary concern. However, of importance is being able to track the overall software quality and observe trending, see levels of compliance to standards and coding guidelines, and to obtain reports that can be understood by management. Most importantly, it is worth to note that the people making purchasing decisions are typically different from the software developers and motivated by quite different reasons.

Figure 1 depicts the high-level architecture of our software analysis tool. At the core are the different analysis engines for abstract interpretation, model checking, SMT solving and pattern matching. As input the tool takes C/C++ projects and a set of specifications that are written in our own domain-specific language and pre-defined for standard users. Goanna can be run on the command line, integrates with IDEs and can be embedded in the build system. It outputs warning messages classified by severity, displays error traces based on counter-examples, and can create summary reports.

3 Lessons

While it is a difficult endeavor to target industry with formal methods there are a range of valuable lessons that can be learned. This starts out with understanding the problem domain, the actual demands and challenges faced by the end-user as well as understanding their processes and potential *value add* by tools. Technology is only one piece of a larger puzzle.

Moreover, software developers face time constraints and pressure to deliver results. Learning a new tool from scratch is not only a process change management shies away from, but typically comes with some kind of formal training in many organization often being much more costly than the actual software tool. Hence, any tool that integrates into existing IDEs and processes that work “with the click of a button” gains much easier acceptance.

From a formal methods researcher’s point of view demands are often sobering. Real life applications

require to deal with huge amounts of code in relatively short time. This will require not only abstractions, but all various heuristics, both on an algorithmic level as well as on an engineering level predicting *typical* developer behavior. Generally, the engineering requirements can easily become a roadblock for the acceptance of more advanced formal analysis techniques.

Finally, with much less effort we believe that many existing academic tools could see a larger industry uptake by following sometimes simple rules. This includes: providing a clear and end-user friendly license agreement, providing a well documented introduction to a tool, as well as a delivering degree of reliability that the software works in *most* cases. This is, however, easier said than done as academic environments typically do no provide for dedicated software engineers to develop and maintain tools over longer period of time.

References

[1] Motor Industry Software Reliability Association et al. (1998): *Guidelines for the use of the C language in vehicle based software*.

[2] Edmund M. Clarke & E. Allen Emerson (1982): *Design and Synthesis of synchronization skeletons for branching time temporal logic*. In: *Logics of Programs Workshop, New York, May 1981*, *LNCS* 131, Springer Verlag, pp. 52–71.

[3] P. Cousot & R. Cousot (1979): *Systematic design of program analysis frameworks*. In: *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM Press, New York, NY, San Antonio, Texas, pp. 269–282.

[4] Leonardo De Moura & Nikolaj Bjørner (2011): *Satisfiability modulo theories: introduction and applications*. *Communications of the ACM* 54(9).

[5] Vijay D'Silva, Daniel Kroening & Georg Weissenbacher (2008): *A Survey of Automated Techniques for Formal Software Verification*. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 27(7), pp. 1165–1178.

[6] A. Fehnker, R. Huuck, P. Jayet, M. Lussenburg & F. Rauch (2007): *Model Checking Software at Compile Time*. In: *Proc. TASE 2007*, IEEE Computer Society.

[7] Maximilian Junker, Ralf Huuck, Ansgar Fehnker & Alexander Knapp (2012): *SMT-based False Positive Elimination in Static Program Analysis*. In: *14th International Conference on Formal Engineering Methods, Kyoto Japan*, *Lecture Notes in Computer Science* 7635, Springer Berlin Heidelberg.

[8] Jean-Pierre Queille & Joseph Sifakis (1982): *Specification and verification of concurrent systems in CESAR*. In: *Proc. Intl. Symposium on Programming, April 6–8*, Springer Verlag, pp. 337–350.

[9] R.C. Seacord (2008): *The CERT C secure coding standard*. Addison-Wesley Professional.

A Timed Calculus for Mobile Ad Hoc Networks

Mengying Wang
Software Engineering Institute
East China Normal University
Shanghai, China
mywang@sei.ecnu.edu.cn

Yang Lu
Department of Computer Science
Shanghai Jiaotong University
Shanghai, China
luyang0415@sjtu.edu.cn

We develop a timed calculus for Mobile Ad Hoc Networks embodying the peculiarities of local broadcast, node mobility and communication interference. We present a Reduction Semantics and a Labelled Transition Semantics and prove the equivalence between them. We then apply our calculus to model and study some MAC-layer protocols with special emphasis on node mobility and communication interference.
A main purpose of the semantics is to describe the various forms of interference while nodes change their locations in the network. Such interference only occurs when a node is simultaneously reached by more than one ongoing transmission over the same channel.

1 Introduction

Mobile ad hoc networks (MANETs) are complex distributed systems that consist of a collection of wireless mobile nodes that can dynamically self-organize into arbitrary network topologies, so as to allow people and devices to seamlessly interwork in areas without pre-existing communication infrastructures [1]. Owing to the flexibility and convenience, their applications have been extended from traditional military domain to a variety of commercial areas, e.g., ambient intelligence [2], personal area networks [4] and location-based services [3].

Wireless nodes use radio frequency channels to broadcast messages. Compared to the conventional wired-based broadcasts like Ethernet networks, this form of broadcast has some special features. First, broadcasting is *local*, i.e., a transmission covers only a limited area, called a *cell*, and hence reaches a (possibly empty) subset of the nodes in the network. Second, channels are *half-duplex*: on a given channel, a node can either transmit or receive, but cannot do both simultaneously. As a result, communication interference can only be detected at the destination. Further, nodes in MANETs can move arbitrarily, which makes the network easily suffer from interference. Since interference plays an important role in evaluating the performance of a network, it becomes a delicate aspect of MANETs that is handled by a great quantity of protocols (e.g., MACA/R-T[5]).

Over the last two decades, a number of process calculi have been proposed to model MANETs [6, 7, 8, 9, 10, 11, 12, 13, 14, 15]. These calculi can be divided into two categories according to their attentions to the network. The first group contains CBS# [6], CMAN [7, 10], RBPT [8], CNT [9], CMN [11], ω-calculus [13] and CSDT [15]. They attempt to depict *local broadcast* and *node mobility*. Take CMN as an example, each node is equipped with a location and a radius that define the cell over which the node can transmit. When a sender broadcasts messages, only nodes that are within its transmission cell could receive. Furthermore, nodes are marked mobile or stationary, and mobile nodes can change their locations randomly. Then CWS [12] and TCWS [14] constitute the second group. They focus on *local broadcast* and *communication interference*. The former abstracts the transmission into two state change events: begin transmission and end transmission, while the latter regards the transmission as a

Submitted to:
FTSCS 2012

time consuming procedure. To our knowledge, no calculus has integrated all of the three peculiarities, especially including node mobility and communication interference.

In this paper, we present a *timed calculus for mobile ad hoc networks* (TCMN), which extends CWS [12] and deals with all of the three issues. A central concern of our calculus is to describe the forms of interference while nodes move their locations in the network. Towards local broadcast, we write $\mathsf{n}[\mathsf{Q}]^{\mathsf{c}}_{\mathsf{l},\mathsf{r}}$ to stand for a node identified by n, located at l, executing process Q, and which can transmit data over channel c in a cell centered at l with radius r. As for node mobility, measures vary according to the specific situation. For instance, nodes that presently participate in no transmission could move arbitrarily without any impact on the environment. However, the movement of an active transmitter may affect the receptions of active receivers: some may get an error or interference, since they passively leave or enter the transmitter's transmission cell. Finally, with regard to communication interference, we assume all wireless nodes have been synchronized by some clock synchronization protocol [16, 17]. Then we consider a transmission proceeds in discrete steps which are represented by occurrences of a simple action σ to denote passing of one time unit. And if a receiver is exposed to more than one ongoing transmission over the same channel, it detects an interference.

In concurrency theory, Labelled Transition Semantics (LTS) is the most popular way of giving operational semantics since the transitions of a LTS expose the full behavior of the system (its internal activities as well as the interactions with the environment) which is required for defining behavioral equivalences and providing powerful proof techniques. However, sometimes the rules of a LTS may be difficult to understand particularly when the calculi relates to node mobility like [7, 10, 11]. Hence, a different form of operational semantics, named Reduction Semantics (RS), is introduced. RS only concerns the internal activities of a system, so it is easier to grasp. Besides, RS can be used to check the correctness of a LTS, by proving consistency with the LTS. For these reasons, we define both RS and LTS semantics for our TCMN and prove that they coincide.

We end this section with an outline of the paper. In Section 2, we define the syntax of our core language. Then in Section 3, we provide a RS for our calculus which specifies how an unbounded number of system components can be involved in an atomic interaction. Next a LTS that captures all the possible interactions of a term with its environment is proposed in Section 4. The equivalence between the RS and the LTS semantics is proved in Section 5. In Section 6 and 7, we extend our core language by adding some new operators to model some MAC-layer collision avoidance protocols: CSMA and MACA/R-T. We prove that the CSMA protocol doesn't solve the issue of node mobility while the MACA/R-T protocol is robust against node mobility. Finally, in section 8, we summarize our contributions and present the future work.

2 The Core Language

In Table 1, we present the core of TCMN. The syntax is defined in a two-level structure: a lower one for *processes* which describes the possible status of a node, and an upper one for *networks*. For easy understanding, in this section we only focus on those operators that are necessary for communication while the extended language will be presented in Section 6.

Generally, we use letters $\mathsf{a}...\mathsf{c}$ for channels, $\mathsf{m}...\mathsf{o}$ for identifiers, $\mathsf{x}...\mathsf{z}$ for variables, u for values that can be transmitted over channels: these include variables and closed values, and v for closed values, i.e. values that contain no variables. $\wr\mathsf{u}\wr$ is a unary function designed to estimate the number of time units required for the transmission of the value u. Since only closed values will be used in transmissions, we assume the existence of an evaluation function $[\![.]\!]$ to return the closed form of a value. Finally, we do

Table 1. The Syntax

Networks:			
$N \stackrel{\text{def}}{=} 0$	empty network	$\mid n[Q]^c_{l,r}$	node
$\mid N \mid N$	parallel composition		
Processes:			
$Q \stackrel{\text{def}}{=} P$	non-active process	$\mid A$	active process
$P \stackrel{\text{def}}{=} 0$	termination	$\mid \text{out}\langle u\rangle.P$	output
$\mid \text{in}(x).P$	input		
$A \stackrel{\text{def}}{=} \langle v\rangle^{\delta}.P$	active output	$\mid (x)^{\delta}_{v}.P$	active input
Values:			
$u \stackrel{\text{def}}{=} x$	variable	$\mid v$	closed value
Functions:			
$f \stackrel{\text{def}}{=} \wr u \int$	time function	$\mid [\![u]\!]$	evaluation function
$\mid d(l_1, l_2)$	distance function		
		where δ is a positive integer greater than 0	

not set how locations should be specified, the only assumption is that they should be comparable, so to determine whether a node is in or out of the transmission cell of another node. We do so by introducing a function d which takes two locations as parameters and returns the distance between them.

Networks are collections of nodes (which actually represent devices) that run in parallel and use same channels to communicate with each other. We use the symbol 0 to stand for the empty network, and $n[Q]^c_{l,r}$ to denote a node identified by n, located at l, executing process Q, and which can transmit data over channel c in a cell centered at l with radius r. We write $N|N$ to indicate a parallel composition of two sub-networks N.

Processes, living within the nodes, are sequential. For convenience, we divide processes into two categories: *non-active* and *active*. An active process is a process that is currently transmitting or receiving data, e.g., an active output process $\langle v\rangle^{\delta}.P$ denotes a transmitting process, and its transmission of value v will complete after δ time units. Similarly, an active input process $(x)^{\delta}_{v}.P$ represents a receiving process, and its reception of value v will last for the next δ instants of time. In the non-active process constructs, the symbol 0 stands for a terminated process. $\text{out}\langle u\rangle.P$ is an output process willing to broadcast the value $v = [\![u]\!]$, and once the transmission starts, the process evolves into the active output process $\langle v\rangle^{\delta}.P$, where $\delta = \wr u \int$ is the time necessary to transmit the value v. $\text{in}(x).P$ indicates an input process willing to receive data, and when the beginning of a transmission v in the following δ time units is captured clearly (i.e. without interference), the process becomes the active input process $(x)^{\delta}_{v}.P$. A node with an active output process inside is named *active transmitter*. Similarly, active input processes and non-active processes are included separately in *active receivers* and *non-active nodes*.

We assume that each node has a unique identifier, and different nodes cannot be located at the same position at the same time. We consider such networks *well-formed*. Since nodes cannot be created or destroyed, the well-formedness of a network is always preserved as the network evolves. In the remainder of the paper, all networks are well formed, and we use a number of notational conventions. Process Q stands for either a non-active or an active process while P and A represent non-active and active processes separately. We identify $\langle v\rangle^{\delta}.P = P$ and $(x)^{\delta}_{v}.P = P\{v/x\}$ if $\delta = 0$. We write $\text{out}\langle u\rangle$ for $\text{out}\langle u\rangle.0$, and $\langle v\rangle^{\delta}$ for $\langle v\rangle^{\delta}.0$.

3 Reduction Semantics

In this section, we study the reduction semantics (RS) for TCMN. In the literature [12], the only internal activity is a broadcast which is modelled by two events: begin transmission event and end transmission event. Yet in our system, a new type of internal activity: a migration is appended to depict node movement. In our model, the broadcast will be described by a begin transmission event and several time passing events (as shown in Table 2), while the migration will be represented by a node movement event from a specific node (as shown in Table 3). Among these three types of events, the *begin transmission event* (i.e. a node initiates a transmission) has the same meaning as that in [12], while the *time passing event* (i.e., a unit of time delays) is imported to replace the end transmission event in [12], and the *node movement event* (i.e., a node moves from one location to another) is a newly added event.

In our RS for core TCMN, a reduction denotes either a begin transmission event, or a time passing event, or a node movement event. In order to handle the interaction among an unbounded number of processes, we use rule schemas instead of simple rules to demonstrate the reductions. Also, since a reduction, e.g. a begin transmission event, cannot be performed inside arbitrary contexts: one should guarantee that the current context meets the specific conditions, the minimal information about the communication is attached to the reduction. Further, in order to model communication interference, we store all the needed active transmitters' information in a global set T which displays in any reduction to determine whether a node is simultaneously reached by more than one transmission over the same channel. For this reason, the reduction semantics is named RST: RS with parameter T. The component T is a set of triples $(\mathsf{l},\mathsf{r},\mathsf{c})$ with each l,r,c in a triple represents location, radius and channel of an active transmitter separately. For simplicity, the semantics does not automatically update the set T. Therefore, when a reduction is performed, the new T which will be used in the next one has to be manually computed. However, it is not difficult to modify the rules so that they also produce the new T.

As usual in process calculi, the reduction semantics relies on an auxiliary relation, called *structural congruence*, denoted by $\equiv$, to allow the manipulation of the term structure so as to bring the participants of a potential interaction into contiguous positions. Here we define a smallest congruence including associativity, commutativity and identity over the empty network:

$$\mathsf{N}|(\mathsf{N}'|\mathsf{N}'') \equiv (\mathsf{N}|\mathsf{N}')|\mathsf{N}'' \quad \mathsf{N}|\mathsf{N}' \equiv \mathsf{N}'|\mathsf{N} \quad \mathsf{N}|0 \equiv \mathsf{N}$$

Next are some useful notations that will be used in RST:

- $\mathsf{T}|_{\mathsf{l},\mathsf{c}}$ is the subset of the active transmitters T whose transmissions are synchronized on channel c and can reach a node located at l. Formally,

$$\mathsf{T}|_{\mathsf{l},\mathsf{c}} = \{(\mathsf{l}',\mathsf{r}',\mathsf{c}')|(\mathsf{l}',\mathsf{r}',\mathsf{c}') \in \mathsf{T} \wedge \mathsf{d}(\mathsf{l}',\mathsf{l}) \leq \mathsf{r}' \wedge \mathsf{c}' = \mathsf{c}\}$$

- $(\mathsf{l},\mathsf{r},\mathsf{c}) \not\Vdash_{\mathsf{i}} \mathsf{N}$ holds if Network N contains no input nodes $\mathsf{n}[\mathsf{in}(\mathsf{x}).\mathsf{P}]^{\mathsf{c}}_{\mathsf{l}',\mathsf{r}'}$ or $\mathsf{n}[(\mathsf{x})^{\delta}_{\mathsf{v}}.\mathsf{P}]^{\mathsf{c}}_{\mathsf{l}',\mathsf{r}'}$ for which $\mathsf{d}(\mathsf{l},\mathsf{l}') \leq \mathsf{r}$ is true (i.e., a transmission from a node located at l, with radius r, synchronized on c reaches no input nodes in N).

- $(\mathsf{l},\mathsf{r},\mathsf{c}) \not\Vdash_{\mathsf{ai}} \mathsf{N}$ holds if Network N contains no active input nodes $\mathsf{n}[(\mathsf{x})^{\delta}_{\mathsf{v}}.\mathsf{P}]^{\mathsf{c}}_{\mathsf{l}',\mathsf{r}'}$ for which $\mathsf{d}(\mathsf{l},\mathsf{l}') \leq \mathsf{r}$ is true (i.e., a transmission from a node located at l, with radius r, synchronized on c reaches no active input nodes in N).

Let's explain the rules in Table 2 and 3. Rule RST-BEGIN is used to derive begin transmission reduction. As in [12], it rewrites atomically an output node $\mathsf{n}[\mathsf{out}\langle\mathsf{u}\rangle.\mathsf{P}]^{\mathsf{c}}_{\mathsf{l},\mathsf{r}}$ which is intending to initiate a transmission and all the receiver nodes that are not only in its transmission cell but also synchronized on

Table 2. Reduction Semantics - Begin transmission and time passing event

$$\text{[RST-BEGIN]}\quad \frac{\forall h\in I\cup J\cup K.d(l,l_h)\le r \quad \forall i\in I.T|_{l_i,c}=\emptyset \quad \forall j\in J.T|_{l_j,c}\neq\emptyset}{T\rhd n[\mathsf{out}\langle u\rangle.P]^c_{l,r}|\prod_{h\in I\cup J} n_h[\mathsf{in}(x_h).P_h]^c_{l_h,r_h}|\prod_{k\in K} n_k[(x_k)^{\delta_k}_{v_k}.P_k]^c_{l_k,r_k}\hookrightarrow^c_{l,r} n[\langle[\![u]\!]\rangle^{\wr u\wr}.P]^c_{l,r}|\prod_{i\in I} n_i[(x_i)^{\wr u\wr}_{[\![u]\!]}.P_i]^c_{l_i,r_i}|\prod_{j\in J} n_j[\mathsf{in}(x_j).P_j]^c_{l_j,r_j}|\prod_{k\in K} n_k[P_k\{\perp/x_k\}]^c_{l_k,r_k}}$$

$$\text{[RST-PASS-NULL]}\quad T\rhd 0\hookrightarrow^\sigma 0$$

$$\text{[RST-SENDING]}\quad \frac{\delta>0 \quad \forall i\in I.d(l,l_i)\le r}{T\rhd n[\langle v\rangle^\delta.P]^c_{l,r}|\prod_{i\in I} n_i[(x_i)^\delta_v.P_i]^c_{l_i,r_i}\hookrightarrow^\sigma n[\langle v\rangle^{\delta-1}.P]^c_{l,r}|\prod_{i\in I} n_i[(x_i)^{\delta-1}_v.P_i]^c_{l_i,r_i}}$$

$$\text{[RST-PASS-NA]}\quad T\rhd n[P]^c_{l,r}\hookrightarrow^\sigma n[P]^c_{l,r}$$

$$\text{[RST-CONT]}\quad \frac{T\rhd N\hookrightarrow^c_{l,r} N' \quad (l,r,c)\not\Vdash_i N''}{T\rhd N|N''\hookrightarrow^c_{l,r} N'|N''}$$

$$\text{[RST-CONT-PASS]}\quad \frac{T\rhd N\hookrightarrow^\sigma N' \quad T\rhd N''\hookrightarrow^\sigma N'''}{T\rhd N|N''\hookrightarrow^\sigma N'|N'''}$$

$$\text{[RST-CONGR]}\quad \frac{N\equiv N' \quad T\rhd N'\hookrightarrow^\& N'' \quad N''\equiv N'''}{T\rhd N\hookrightarrow^\& N'''}$$

the same channel c. After this reduction, the output process evolves into $\langle[\![u]\!]\rangle^{\wr u\wr}.P$ indicating an active output process that will transmit the evaluation result $[\![u]\!]$ of value u in the following $\wr u\wr$ time units. The effect of the begin transmission event on each receiver varies according to the structure of each receiver and the set T. There are three different situations, corresponding to the sets I, J and K. Processes in I represent normal inputs. Since their environments are silent ($T|_{l_i,c}=\emptyset$), they become active inputs of the form $(x_i)^{\wr u\wr}_{[\![u]\!]}.P_i$ and start receiving data $[\![u]\!]$ for the next $\wr u\wr$ time units. By contrast, for processes in J, as they are currently reached by at least one other transmission ($T|_{l_j,c}\neq\emptyset$), they could not receive the begin transmission event clearly and stay idle. Finally, processes in K are active inputs, i.e., they are receiving another transmission, so the new begin transmission event causes interference, denoted by receiving symbol $\perp$.

Rule RST-SENDING deals with the time passing event for active processes. Initially, the active output process $\langle v\rangle^\delta.P$ requires δ time units to complete the date transmission. After a time interval, the remaining time would be $\delta-1$ units for both sender and receivers. Meanwhile rule RST-PASS-NA and RST-PASS-NULL handle the time passing event for non-active processes and empty networks respectively. No matter how time flies, they remain unchanged.

Rule RST-MOVE-AO, RST-MOVE-AI1, RST-MOVE-AI2, RST-MOVE-AI3, and RST-MOVE-NA are all used to describe node movements. In RST-MOVE-AO, an active transmitter moves from l to l'. Then for active receivers in set I, as they are always reachable no matter from l or l', they continue to receive data normally. As for active receivers in set J, since they are reachable from l but not from l', they get an error, represented by a special sign ε. Finally, active receivers in set K, which are reachable from l' but not from l, are receiving another transmission, so the newly joined transmitter will make them get interference. Rule RST-MOVE-AI1, RST-MOVE-AI2 and RST-MOVE-AI3 depict all the different movements of an active receiver. In RST-MOVE-AI1, the active receiver moves from l to l' which makes the original transmission no longer receivable, hence it gets an error. While in RST-MOVE-AI2, although the active receiver moves from l to l', it has always been in the transmitter's transmission cell and there is no more active transmitter in l', so the active receiver remains unchanged. On the contrary, in RST-MOVE-AI3, when the active receiver arrives at l', some other transmissions in l' interfere with its original one. As a result, the active receiver obtains an interference. Rule RST-MOVE-NA is straightforward, for

Table 3. Reduction Semantics - Node movement event

[RST-MOVE-AO]

$$\frac{\forall i{\in}I.d(l,l_i){\leq}r{\wedge}d(l',l_i){\leq}r \quad \forall j{\in}J.d(l,l_j){\leq}r{\wedge}d(l',l_j){>}r \quad \forall k{\in}K.d(l,l_k){>}r{\wedge}d(l',l_k){\leq}r}{T{\rhd}n[\langle v\rangle^{\delta}.P]^c_{l,r}|\prod_{h\in I\cup J} n_h[(x_h)^{\delta}_v.P_h]^c_{l_h,r_h}|\prod_{k\in K} n_k[(x_k)^{\delta_k}_{v_k}.P_k]^c_{l_k,r_k} \hookrightarrow^c_{l:l',r}}$$

$$n[\langle v\rangle^{\delta}.P]^c_{l',r}|\prod_{i\in I} n_i[(x_i)^{\delta}_v.P_i]^c_{l_i,r_i}|\prod_{j\in J} n_j[P_j\{\varepsilon/x_j\}]^c_{l_j,r_j}|\prod_{k\in K} n_k[P_k\{\perp/x_k\}]^c_{l_k,r_k}$$

[RST-MOVE-AI1]

$$\frac{d(l,l_i){\leq}r_i{\wedge}d(l',l_i){>}r_i}{T{\rhd}n[(x)^{\delta}_v.P]^c_{l,r}|n_i[\langle v\rangle^{\delta}.P]^c_{l_i,r_i}{\hookrightarrow}n[P\{\varepsilon/x\}]^c_{l',r}|n_i[\langle v\rangle^{\delta}.P]^c_{l_i,r_i}}$$

[RST-MOVE-AI2]

$$\frac{d(l,l_i){\leq}r_i{\wedge}d(l',l_i){\leq}r_i \quad T|_{l,c}{=}T|_{l',c}}{T{\rhd}n[(x)^{\delta}_v.P]^c_{l,r}|n_1[\langle v\rangle^{\delta}.P]^c_{l_i,r_i}{\hookrightarrow}n[(x)^{\delta}_v.P]^c_{l',r}|n_i[\langle v\rangle^{\delta}.P]^c_{l_i,r_i}}$$

[RST-MOVE-AI3]

$$\frac{d(l,l_i){\leq}r_i{\wedge}d(l',l_i){\leq}r_i \quad \forall j{\in}J.d(l,l_j){>}r_j{\wedge}d(l',l_j){\leq}r_j}{T{\rhd}n[(x)^{\delta}_v.P]^c_{l,r}|n_i[\langle v\rangle^{\delta}.P]^c_{l_i,r_i}|\prod_{j\in J} n_j[\langle v_j\rangle^{\delta_j}.P_j]^c_{l_j,r_j}{\hookrightarrow}n[P\{\perp/x\}]^c_{l',r}|n_i[\langle v\rangle^{\delta}.P]^c_{l_i,r_i}|\prod_{j\in J} n_j[\langle v_j\rangle^{\delta_j}.P_j]^c_{l_j,r_j}}$$

[RST-MOVE-NA] $\quad T \rhd n[P]^c_{l,r} \hookrightarrow n[P]^c_{l',r}$

[RST-CONT-MOVE] $\quad \dfrac{T{\rhd}N{\hookrightarrow}^c_{l:l',r}N' \quad (l,r,c)\not\Vdash_{ai}N''{\wedge}(l',r,c)\Vdash_{ai}N''}{T{\rhd}N|N''{\hookrightarrow}^c_{l:l',r}N'|N''}$

[RST-CONT-INT] $\quad \dfrac{T{\rhd}N{\hookrightarrow}N'}{T{\rhd}N|N''{\hookrightarrow}N'|N''}$

non-active nodes, their movement will not affect the environment, therefore they can move arbitrarily without any limitations and changes.

Rule RST-CONT, RST-CONT-PASS, RST-CONT-MOVE and RST-CONT-INT are closure rules with regard to different reduction forms ($\hookrightarrow^c_{l,r}$, $\hookrightarrow^{\sigma}$, $\hookrightarrow^c_{l:l',r}$, and $\hookrightarrow$). In RST-CONT, it provides a closure under contexts that do not contain receivers in the transmission cell of the transmitter. Similarly, rule RST-CONT-MOVE presents a closure under contexts that have never contained active receivers in the transmission cell of the transmitter when the transmitter moves from l to l'. Rule RST-CONT-INT is analogous to the previous two except that it concerns *internal events* (i.e., a node movement event from an active receiver or a non-active node). Rule RST-CONT-PASS is the time synchronization, it defines a closure under contexts that are also affected by the time passing event.

The last rule, RST-CONGR is a closure rule under structural congruence, where $\hookrightarrow^{\&}$ ranges over $\hookrightarrow^c_{l,r}$, $\hookrightarrow^{\sigma}$, $\hookrightarrow^c_{l:l',r}$ and $\hookrightarrow$ for some c, l, l' and r.

4 Labelled Transition Semantics

We divide our Labelled Transition Semantics (LTS) into two set of rules corresponding to the two-level structure of our language. Table 4 contains the rules for the processes, while Table 5 and 6 presents those for the networks.

In the process semantics, a transition has the form $Q \xrightarrow{\alpha} Q'$, where the grammar for α is:

$$\alpha \;:=\; !v:\delta \mid ?v:\delta \mid ?\perp \mid ?\varepsilon \mid \sigma$$

Table 4. Labelled Transitions for Processes

$$\frac{[\![u]\!]=v \quad \wr u \wr=\delta}{\mathsf{out}\langle u\rangle.P \xrightarrow{!v:\delta} \langle v\rangle^{\delta}.P}[\text{PS-OUT}_{begin}] \qquad \frac{\delta>0}{\langle v\rangle^{\delta}.P \xrightarrow{\sigma} \langle v\rangle^{\delta-1}.P}[\text{PS-OUT}_{send}]$$

$$\frac{-}{\mathsf{in}(x).P \xrightarrow{?v:\delta} (x)^{\delta}_{v}.P}[\text{PS-IN}_{begin}] \qquad \frac{\delta>0}{(x)^{\delta}_{v}.P \xrightarrow{\sigma} (x)^{\delta-1}_{v}.P}[\text{PS-IN}_{receive}]$$

$$\frac{-}{\mathsf{in}(x).P \xrightarrow{?\perp} \mathsf{in}(x).P}[\text{PS-IN}_{wait}] \qquad \frac{-}{(x)^{\delta}_{v}.P \xrightarrow{?\perp} P\{\perp/x\}}[\text{PS-IN}_{interfere}]$$

$$\frac{-}{(x)^{\delta}_{v}.P \xrightarrow{?\varepsilon} P\{\varepsilon/x\}}[\text{PS-IN}_{err}] \qquad \frac{-}{P \xrightarrow{\sigma} P}[\text{PS-PASS}] \qquad \frac{\alpha\in\{?v:\delta,?\varepsilon,?\perp\} \quad Q\notin IQ}{Q \xrightarrow{\alpha} Q}[\text{PS-NOIN}]$$

where IQ is the set of processes of the form $\mathsf{in}(x).P$ or $(x)^{\delta}_{v}.P$

Label $!v:\delta$ represents a begin transmission event (i.e., a transmission of value v in the following δ time units) is initiated by Q which then evolves into Q'; $?v:\delta$ indicates a begin transmission event reaches Q and makes the process transform into Q'; Analogously, $?\perp$ and $?\varepsilon$ stand for an interference or error arrives; finally, σ means a time passing event.

Explanations for the rules in Table 4 are as follows: in PS-OUT$_{begin}$, the output process calculates the value u and initiates the transmission of the result v in the next δ time units; in PS-IN$_{begin}$, the input process successfully becomes involved with the transmission of value v for the next δ instants of time; in PS-OUT$_{send}$ and PS-IN$_{receive}$, with the time passing by, the remaining transmission time is decreasing; in PS-IN$_{wait}$, the input process stays idle since it could not receive the begin transmission event clearly; in PS-IN$_{interfere}$ and PS-IN$_{err}$, an active input process encounters an interference or error in its reception, and hence stops receiving; Rule PS-PASS shows that the non-active process would never change as time goes by, and PS-NOIN demonstrates that the non-input processes would never respond to the reception of events.

Following are some useful mathematical symbols in LTS:

- $\mathsf{d}(l,l') \le r' \odot \mathsf{d}(l,l'') \le r' = (\mathsf{d}(l,l') \le r' \wedge \mathsf{d}(l,l'') \le r') \vee (\mathsf{d}(l,l') > r' \wedge \mathsf{d}(l,l'') > r')$.
- $T|_{l,c} - T|_{l',c}$ is the set of elements that are contained in $T|_{l,c}$ but not in $T|_{l',c}$.
- $T|_{l,c} \subset T|_{l',c}$ holds only if $T|_{l,c}$ is a proper subset of $T|_{l',c}$.

In the network semantics, transitions are of the form $T \rhd N \xrightarrow{\mu} N'$ where T is the same as in Section 3. Let's comment on the rules in Table 5 and 6. Rule NS-OUT, NS-IN$_1$, NS-IN$_2$ and NS-IN$_3$ concern the communication between a transmitter and its receivers. Rule NS-OUT shows that a node initiates a transmission. Then rule NS-IN$_1$ describes the behavior of a node that is within the transmission cell and could hear the begin transmission event clearly, whereas NS-IN$_2$ handles those that detect conflicts. Rule NS-IN$_3$ demonstrates that a node would not react to transmissions that are beyond its reception range or not in its listening channel.

Next rules NS-MOVE$_{ao}$, NS-MOVE$_{in1}$, NS-MOVE$_{in2}$, NS-MOVE$_{in3}$, NS-MOVE$_{ai1}$, NS-MOVE$_{ai2}$, NS-MOVE$_{ai3}$ and NS-MOVE$_{na}$ are all used to deal with the node movement events from different kinds of nodes. For example, rule NS-MOVE$_{ao}$ depicts that an active transmitter located at l moves to l' during its transmission over channel c with radius r. Then the behaviors of surrounding nodes can be divided

Table 5. Labelled Transitions for Networks - Begin transmission and time passing event

$$\frac{\mathsf{P}\xrightarrow{!v:\delta}\mathsf{A}}{\mathsf{T}\rhd\mathsf{n}[\mathsf{P}]^{c}_{l,r}\xrightarrow{c!v:\delta[l,r]}\mathsf{n}[\mathsf{A}]^{c}_{l,r}}[\text{NS-OUT}] \qquad \frac{\mathsf{Q}\xrightarrow{?v:\delta}\mathsf{Q}' \quad \mathsf{d}(l,l')\leq r' \quad \mathsf{T}|_{l,c}=\emptyset}{\mathsf{T}\rhd\mathsf{n}[\mathsf{Q}]^{c}_{l,r}\xrightarrow{c?v:\delta[l',r']}\mathsf{n}[\mathsf{Q}']^{c}_{l,r}}[\text{NS-IN}_1]$$

$$\frac{\mathsf{Q}\xrightarrow{?\perp}\mathsf{Q}' \quad \mathsf{d}(l,l')\leq r' \quad \mathsf{T}|_{l,c}\neq\emptyset}{\mathsf{T}\rhd\mathsf{n}[\mathsf{Q}]^{c}_{l,r}\xrightarrow{c?v:\delta[l',r']}\mathsf{n}[\mathsf{Q}']^{c}_{l,r}}[\text{NS-IN}_2] \qquad \frac{\mathsf{d}(l,l')>r'\vee c\neq c'}{\mathsf{T}\rhd\mathsf{n}[\mathsf{Q}]^{c}_{l,r}\xrightarrow{c'?v:\delta[l',r']}\mathsf{n}[\mathsf{Q}]^{c}_{l,r}}[\text{NS-IN}_3]$$

$$\frac{\mathsf{Q}\xrightarrow{\sigma}\mathsf{Q}'}{\mathsf{T}\rhd\mathsf{n}[\mathsf{Q}]^{c}_{l,r}\xrightarrow{\sigma}\mathsf{n}[\mathsf{Q}']^{c}_{l,r}}[\text{NS-PASS}] \qquad \frac{-}{\mathsf{T}\rhd 0\xrightarrow{c?v:\delta[l,r]}0}[\text{NS-NULL}_{in1}]$$

$$\frac{-}{\mathsf{T}\rhd 0\xrightarrow{\sigma}0}[\text{NS-NULL}_{pass}] \qquad \frac{\mathsf{T}\rhd\mathsf{N}_1\xrightarrow{c?v:\delta[l,r]}\mathsf{N}'_1 \quad \mathsf{T}\rhd\mathsf{N}_2\xrightarrow{c!v:\delta[l,r]}\mathsf{N}'_2}{\mathsf{T}\rhd\mathsf{N}_1|\mathsf{N}_2\xrightarrow{c!v:\delta[l,r]}\mathsf{N}'_1|\mathsf{N}'_2 \quad \mathsf{T}\rhd\mathsf{N}_2|\mathsf{N}_1\xrightarrow{c!v:\delta[l,r]}\mathsf{N}'_2|\mathsf{N}'_1}[\text{NS-COM}]$$

$$\frac{\mathsf{T}\rhd\mathsf{N}_1\xrightarrow{c?v:\delta[l,r]}\mathsf{N}'_1 \quad \mathsf{T}\rhd\mathsf{N}_2\xrightarrow{c?v:\delta[l,r]}\mathsf{N}'_2}{\mathsf{T}\rhd\mathsf{N}_1|\mathsf{N}_2\xrightarrow{c?v:\delta[l,r]}\mathsf{N}'_1|\mathsf{N}'_2}[\text{NS-COM}_{in}] \qquad \frac{\mathsf{T}\rhd\mathsf{N}_1\xrightarrow{\sigma}\mathsf{N}'_1 \quad \mathsf{T}\rhd\mathsf{N}_2\xrightarrow{\sigma}\mathsf{N}'_2}{\mathsf{T}\rhd\mathsf{N}_1|\mathsf{N}_2\xrightarrow{\sigma}\mathsf{N}'_1|\mathsf{N}'_2}[\text{NS-SYN}]$$

into three different cases corresponding to NS-MOVE$_{in1}$, NS-MOVE$_{in2}$ and NS-MOVE$_{in3}$ respectively. (1) For non-active nodes and active receivers that are receiving over other channels or that are always in or out of the transmission cell, they will remain unchanged. (2) For active receivers that are originally within the transmission cell, but later beyond it, they will receive an error. (3) For active receivers that are in the reverse situation, they will get interference. Analogously, rule NS-MOVE$_{ai1}$, NS-MOVE$_{ai3}$ and NS-MOVE$_{ai2}$ have described the possible scenarios of an active receiver that moves from l to l': (1) if the active receiver moves out of the transmission cell, it will obtain an error; (2) if the active receiver has always been within the transmission cell and there is no more transmission in l', it will continue to receive data normally; (3) if there are more transmissions in l' apart from its original one, the active receiver will get interference. Finally, we can see from NS-MOVE$_{na}$ that for non-active nodes, they can move arbitrarily without conditions and limitations.

Moreover, rule NS-PASS represents the responses of nodes as time goes by. Rule NS-NULL$_{in1}$, NS-NULL$_{in2}$ and NS-NULL$_{pass}$ allow the empty network to receive data and evolve with time. At last, the propagation of events through networks is portrayed by rule NS-COM, NS-MOVE, NS-INT, NS-COM$_{in}$, NS-MOVE$_{in}$, and NS-SYN. The first three denote that an event generated in a network is propagated to the parallel network; while the later ones indicate that two parallel networks receive the same event.

5 Harmony Theorem

The Harmony Theorem aims at proving that the LTS-based semantics coincides with the RST-based semantics. With this objective, the theorem has three parts. First, it shows that the structural congruence respects the LTS, i.e., application of structural congruence will not change the possible transitions. Then it demonstrates that the RST behaves the same as the LTS, i.e., each reduction in the RST has a corresponding transition in the LTS which makes the resulting networks structurally congruent. In the end, it testifies the converse also holds.

Before proving the theorem, there are some auxiliary lemmas that portray the shape of processes able to perform a particular labelled transition, and the shape of the derivative processes (see the Appendix).

Table 6. Labelled Transitions for Networks - Node movement event

$$\frac{-}{\mathsf{T}\rhd \mathsf{n}[\langle \mathsf{v}\rangle^{\delta}.\mathsf{P}]^{\mathsf{c}}_{\mathsf{l},\mathsf{r}} \xrightarrow{\mathsf{c}![(\mathsf{l}:\mathsf{l}'),\mathsf{r}]} \mathsf{n}[\langle \mathsf{v}\rangle^{\delta}.\mathsf{P}]^{\mathsf{c}}_{\mathsf{l}',\mathsf{r}}}\,[\text{NS-MOVE}_{ao}]$$

$$\frac{(\mathsf{Q}\notin \mathsf{AIQ})\vee(\mathsf{c}\neq \mathsf{c}')\vee(\mathsf{d}(\mathsf{l},\mathsf{l}')\leq \mathsf{r}'\odot \mathsf{d}(\mathsf{l},\mathsf{l}'')\leq \mathsf{r}')}{\mathsf{T}\rhd \mathsf{n}[\mathsf{Q}]^{\mathsf{c}}_{\mathsf{l},\mathsf{r}} \xrightarrow{\mathsf{c}'?[(\mathsf{l}':\mathsf{l}''),\mathsf{r}']} \mathsf{n}[\mathsf{Q}]^{\mathsf{c}}_{\mathsf{l},\mathsf{r}}}\,[\text{NS-MOVE}_{in1}]$$

$$\frac{\mathsf{Q}\in \mathsf{AIQ}\ \ \mathsf{Q}\xrightarrow{?\varepsilon}\mathsf{Q}'\ \ \mathsf{d}(\mathsf{l},\mathsf{l}')\leq \mathsf{r}'\wedge \mathsf{d}(\mathsf{l},\mathsf{l}'')>\mathsf{r}'}{\mathsf{T}\rhd \mathsf{n}[\mathsf{Q}]^{\mathsf{c}}_{\mathsf{l},\mathsf{r}} \xrightarrow{\mathsf{c}?[(\mathsf{l}':\mathsf{l}''),\mathsf{r}']} \mathsf{n}[\mathsf{Q}']^{\mathsf{c}}_{\mathsf{l},\mathsf{r}}}\,[\text{NS-MOVE}_{in2}]$$

$$\frac{\mathsf{Q}\in \mathsf{AIQ}\ \ \mathsf{Q}\xrightarrow{?\perp}\mathsf{Q}'\ \ \mathsf{d}(\mathsf{l},\mathsf{l}')>\mathsf{r}'\wedge \mathsf{d}(\mathsf{l},\mathsf{l}'')\leq \mathsf{r}'}{\mathsf{T}\rhd \mathsf{n}[\mathsf{Q}]^{\mathsf{c}}_{\mathsf{l},\mathsf{r}} \xrightarrow{\mathsf{c}?[(\mathsf{l}':\mathsf{l}''),\mathsf{r}']} \mathsf{n}[\mathsf{Q}']^{\mathsf{c}}_{\mathsf{l},\mathsf{r}}}\,[\text{NS-MOVE}_{in3}]$$

$$\frac{\mathsf{Q}\in \mathsf{AIQ}\ \ \mathsf{Q}\xrightarrow{?\varepsilon}\mathsf{Q}'\ \ \mathsf{T}|_{\mathsf{l},\mathsf{c}}-\mathsf{T}|_{\mathsf{l}',\mathsf{c}}=\mathsf{T}|_{\mathsf{l},\mathsf{c}}}{\mathsf{T}\rhd \mathsf{n}[\mathsf{Q}]^{\mathsf{c}}_{\mathsf{l},\mathsf{r}} \longrightarrow \mathsf{n}[\mathsf{Q}']^{\mathsf{c}}_{\mathsf{l}',\mathsf{r}}}\,[\text{NS-MOVE}_{ai1}]$$

$$\frac{\mathsf{Q}\in \mathsf{AIQ}\ \ \mathsf{Q}\xrightarrow{?\perp}\mathsf{Q}'\ \ \mathsf{T}|_{\mathsf{l},\mathsf{c}}\subset \mathsf{T}|_{\mathsf{l}',\mathsf{c}}}{\mathsf{T}\rhd \mathsf{n}[\mathsf{Q}]^{\mathsf{c}}_{\mathsf{l},\mathsf{r}} \longrightarrow \mathsf{n}[\mathsf{Q}']^{\mathsf{c}}_{\mathsf{l}',\mathsf{r}}}\,[\text{NS-MOVE}_{ai2}] \qquad \frac{\mathsf{Q}\in \mathsf{AIQ}\ \ \mathsf{T}|_{\mathsf{l},\mathsf{c}}=\mathsf{T}|_{\mathsf{l}',\mathsf{c}}}{\mathsf{T}\rhd \mathsf{n}[\mathsf{Q}]^{\mathsf{c}}_{\mathsf{l},\mathsf{r}} \longrightarrow \mathsf{n}[\mathsf{Q}]^{\mathsf{c}}_{\mathsf{l}',\mathsf{r}}}\,[\text{NS-MOVE}_{ai3}]$$

$$\frac{-}{\mathsf{T}\rhd \mathsf{n}[\mathsf{P}]^{\mathsf{c}}_{\mathsf{l},\mathsf{r}} \longrightarrow \mathsf{n}[\mathsf{P}]^{\mathsf{c}}_{\mathsf{l}',\mathsf{r}}}\,[\text{NS-MOVE}_{na}] \qquad \frac{-}{\mathsf{T}\rhd \mathsf{0} \xrightarrow{\mathsf{c}?[(\mathsf{l}:\mathsf{l}'),\mathsf{r}]} \mathsf{0}}\,[\text{NS-NULL}_{in2}]$$

$$\frac{\mathsf{T}\rhd \mathsf{N}_1 \xrightarrow{\mathsf{c}?[(\mathsf{l}:\mathsf{l}'),\mathsf{r}]} \mathsf{N}'_1\ \ \mathsf{T}\rhd \mathsf{N}_2 \xrightarrow{\mathsf{c}![(\mathsf{l}:\mathsf{l}'),\mathsf{r}]} \mathsf{N}'_2}{\mathsf{T}\rhd \mathsf{N}_1|\mathsf{N}_2 \xrightarrow{\mathsf{c}![(\mathsf{l}:\mathsf{l}'),\mathsf{r}]} \mathsf{N}'_1|\mathsf{N}'_2}\,[\text{NS-MOVE}] \qquad \frac{\mathsf{T}\rhd \mathsf{N}_1 \longrightarrow \mathsf{N}'_1}{\mathsf{T}\rhd \mathsf{N}_1|\mathsf{N}_2 \longrightarrow \mathsf{N}'_1|\mathsf{N}_2}\,[\text{NS-INT}]$$

$$\mathsf{T}\rhd \mathsf{N}_2|\mathsf{N}_1 \xrightarrow{\mathsf{c}![(\mathsf{l}:\mathsf{l}'),\mathsf{r}]} \mathsf{N}'_2|\mathsf{N}'_1 \qquad \mathsf{T}\rhd \mathsf{N}_2|\mathsf{N}_1 \to \mathsf{N}_2|\mathsf{N}'_1$$

$$\frac{\mathsf{T}\rhd \mathsf{N}_1 \xrightarrow{\mathsf{c}?[(\mathsf{l}:\mathsf{l}'),\mathsf{r}]} \mathsf{N}'_1\ \ \mathsf{T}\rhd \mathsf{N}_2 \xrightarrow{\mathsf{c}?[(\mathsf{l}:\mathsf{l}'),\mathsf{r}]} \mathsf{N}'_2}{\mathsf{T}\rhd \mathsf{N}_1|\mathsf{N}_2 \xrightarrow{\mathsf{c}?[(\mathsf{l}:\mathsf{l}'),\mathsf{r}]} \mathsf{N}'_1|\mathsf{N}'_2}\,[\text{NS-MOVE}_{in}]$$

where AIQ is the set of processes of the form $(\mathsf{x})^{\delta}_{\mathsf{v}}.\mathsf{P}$

Theorem 1 (*Harmony Theorem*). Let N be a network, and T a set of active transmitters.

(1) If $\mathsf{T}\rhd \mathsf{N}\xrightarrow{\mu}\mathsf{N}'$ and $\mathsf{N}\equiv \mathsf{N}_1$, then there exists N'_1 such that $\mathsf{T}\rhd \mathsf{N}_1\xrightarrow{\mu}\mathsf{N}'_1\equiv \mathsf{N}'$.

(2) (a) If $\mathsf{T}\rhd \mathsf{N}\hookrightarrow^{\mathsf{c}}_{\mathsf{l}:\mathsf{l}',\mathsf{r}}\mathsf{N}'$, then $\mathsf{T}\rhd \mathsf{N}\xrightarrow{\mathsf{c}![(\mathsf{l}:\mathsf{l}'),\mathsf{r}]}\mathsf{N}'_1\equiv \mathsf{N}'$.

(b) If $\mathsf{T}\rhd \mathsf{N}\hookrightarrow^{\mathsf{c}}_{\mathsf{l},\mathsf{r}}\mathsf{N}'$, then there are v and δ such that $\mathsf{T}\rhd \mathsf{N}\xrightarrow{\mathsf{c}!\mathsf{v}:\delta[\mathsf{l},\mathsf{r}]}\mathsf{N}'_1\equiv \mathsf{N}'$.

(c) If $\mathsf{T}\rhd \mathsf{N}\hookrightarrow^{\sigma}\mathsf{N}'$, then $\mathsf{T}\rhd \mathsf{N}\xrightarrow{\sigma}\mathsf{N}'_1\equiv \mathsf{N}'$.

(d) If $\mathsf{T}\rhd \mathsf{N}\hookrightarrow \mathsf{N}'$, then $\mathsf{T}\rhd \mathsf{N}\to \mathsf{N}'_1\equiv \mathsf{N}'$.

(3) For each item in (2), the reverse also holds.

Proof. Now we prove the three points in sequence.

(1) The equivalence is defined in terms of commutativity, associativity, and identity over the empty network. First commutativity is guaranteed since rule NS-COM, NS-MOVE and NS-INT are symmetric, and rule NS-COM$_{in}$, NS-MOVE$_{in}$ and NS-SYS are self-symmetric. Identity over the empty network conserves since, owing to rule NS-NULL$_{in1}$, NS-NULL$_{in2}$ and NS-NULL$_{pass}$, the

Table 7. Extended Syntax for Processes

$P \stackrel{\text{def}}{=} \ldots$	old processes	$\mid \lceil \mathsf{in}(x).P \rceil^{t} Q$	input with timeout
$\mid \sigma.P$	delay	$\mid \rhd c.P$	channel switch
$\mid [e]Q_1, Q_2$	choice	$\mid H(\overrightarrow{u})$	recursion
		where t is a positive integer greater than 0	

empty network can perform any labels of the form $\xrightarrow{c?v:\delta[l,r]}$, $\xrightarrow{c?[(l:l'),r]}$, and $\xrightarrow{\sigma}$, which serve as neutral element of parallel composition. Finally as the operations for parallel composition are associative and the network structure is always preserved, therefore associativity is also ensured.

(2) As proofs for the four statements are similar, we only take (a) as an example.
(a) The proof is by rule induction on the derivation of $T \rhd N \hookrightarrow^{c}_{l:l',r} N'$.

First we consider rule RST-MOVE-AO, the proof for this case is by induction on the size of $I \cup J \cup K$. The base case is $I \cup J \cup K = \emptyset$, using rule NS-MOVE$_{ao}$. In the inductive case, we randomly choose an element h from $I \cup J \cup K$. Remember that by the inductive hypothesis, we already have a transition with label $\xrightarrow{c![(l:l'),r]}$. Below are different cases according to which set h belongs to. Suppose $h \in I$, then we can apply rule NS-MOVE$_{in1}$ since $d(l,l_i) \leq r \wedge d(l',l_i) \leq r$ from the premise of rule RST-MOVE-AO. Thus the desired transition can be proved using rule NS-MOVE. Suppose now $h \in J$, we can use rule PS-IN$_{err}$ to derive $(x_j)^{\delta}_{v}.P_j \xrightarrow{?\varepsilon} P_j\{\varepsilon/x_j\}$, and then rule NS-MOVE$_{in2}$ to derive a transition with label $\xrightarrow{c?[(l:l'),r]}$. Hence the desired format can be arrived using rule NS-MOVE. Finally suppose $h \in K$, we can use rule PS-IN$_{interfere}$ to derive $(x_k)^{\delta_k}_{v_k}.P_k \xrightarrow{?\perp} P_k\{\perp/x_k\}$. This transition can be lifted up to the network level using rule NS-MOVE$_{in3}$ because $d(l,l_k) > r \wedge d(l',l_k) \leq r$ from the precondition of rule RST-MOVE-AO. Similarly, using rule NS-MOVE, we can get the desired transition.

The proof is analogous for rule RST-CONT-MOVE, since $(l,r,c) \not\Vdash_{ai} N'' \wedge$
$(l',r,c) \not\Vdash_{ai} N''$ ensures that all the active input nodes satisfy the conditions of NS-MOVE$_{in1}$. As for non-active input nodes, rule NS-MOVE$_{in1}$ can also be applied.

Rule RST-CONGR can be simulated by the first part of the theorem.

(3) The proof for (a) and (b) are based on Lemma 2 and Lemma 4 respectively. The proof for (d) is straightforward. Now we consider the proof for (c).
The proof is based on Lemma 5. If $I \cup J \cup K = \emptyset$, the desired reduction can be derived using rule RST-PASS-NULL and RST-CONGR. Otherwise, for each element i in I and for each element k in K, first apply rule RST-SENDING and RST-PASS-NA respectively, then employ rule RST-CONT-PASS and RST-CONGR to get the desired reduction. □

6 The Extended Language

So far we have considered the subset of TCMN with only the operators that are necessary for communication. Now we present some extensions: a series of processes are added in Table 7, while the syntax for others remains the same.

First of all, the input construct is replaced by the *input with timeout* construct in $\lceil \mathsf{in}(x).P \rceil^{t} Q$. This process is waiting for receiving a value, if the value arrives before the end of the t time units, the process evolves into an active receiver; otherwise, the process continues as Q. Process $\sigma.P$ stands for sleeping

Table 8. Extended Reduction Semantics

[RST-BEGIN]

$$\frac{\forall h\in I\cup J\cup K.d(I,I_h)\leq r \quad \forall i\in I.T|_{I_i,c}=\emptyset \quad \forall j\in J.T|_{I_j,c}\neq\emptyset}{T\rhd n[out\langle u\rangle.P]^c_{I,r}|\prod_{h\in I\cup J} n_h[\lceil in(x_h).P_h\rceil^{t_h}Q_h]^c_{I_h,r_h}|\prod_{k\in K} n_k[(x_k)^{\delta_k}_{v_k}.P_k]^c_{I_k,r_k}\hookrightarrow^c_{I,r}}$$

$$n[\langle\!|\!|\!|u|\!|\!|\!|\rangle^{\wr u\wr}.P]^c_{I,r}|\prod_{i\in I} n_i[(x_i)^{\wr u\wr}_{|\!|\!|u|\!|\!|}.P_i]^c_{I_i,r_i}|\prod_{j\in J} n_j[\lceil in(x_j).P_j\rceil^{t_j}Q_j]^c_{I_j,r_j}|\prod_{k\in K} n_k[P_k\{\perp/x_k\}]^c_{I_k,r_k}$$

[RST-INPUT-DELAY]

$$\frac{t>1}{T\rhd n[\lceil in(x).P\rceil^t Q]^c_{I,r}\hookrightarrow^\sigma n[\lceil in(x).P\rceil^{t-1}Q]^c_{I,r}}$$

[RST-TIMEOUT]

$$\frac{t=1}{T\rhd n[\lceil in(x).P\rceil^t Q]^c_{I,r}\hookrightarrow^\sigma n[Q]^c_{I,r}}$$

[RST-PASS-DELAY]

$$T\rhd n[\sigma.P]^c_{I,r}\hookrightarrow^\sigma n[P]^c_{I,r}$$

[RST-PASS-NA]

$$\frac{P\notin DIQ}{T\rhd n[P]^c_{I,r}\hookrightarrow^\sigma n[P]^c_{I,r}}$$

[RST-SWITCH]

$$T\rhd n[\rhd c'.P]^c_{I,r}\hookrightarrow n[P]^{c'}_{I,r}$$

[RST-IF-TRUE]

$$\frac{e=true}{T\rhd n[[e]Q_1,Q_2]^c_{I,r}\hookrightarrow n[Q_1]^c_{I,r}}$$

[RST-IF-FALSE]

$$\frac{e=false}{T\rhd n[[e]Q_1,Q_2]^c_{I,r}\hookrightarrow n[Q_2]^c_{I,r}}$$

where DIQ is the set of processes of the form $\sigma.P$ or $\lceil in(x).P\rceil^t Q$

for one time unit while $\rhd c.P$ represents a process that decides to switch its communication channel to c, and then continues as P. The construct $[e]Q_1,Q_2$ behaves as Q_1 if $e=true$ and as Q_2 otherwise. Here, e is a boolean value expression. Finally, $H(\overrightarrow{u})$ denotes a process defined via a (possibly recursive) definition $H(\overrightarrow{x})\stackrel{def}{=}Q$, with $|\overrightarrow{x}|=|\overrightarrow{u}|$, where $\overrightarrow{x}$ contains all free variables of Q.

We only provide the addition of the new operators to the RST semantics, since this is the simpler one and the one that we will use in Section 7. However the operators can be introduced in a similar way into the LTS semantics.

Before updating the RST semantics, a new structural congruence rule is appended:

$$n[H(\overrightarrow{u})]^c_{I,r}\equiv n[Q\{\overrightarrow{u}/\overrightarrow{x}\}]^c_{I,r} \text{ if } H(\overrightarrow{x})\stackrel{def}{=}Q\wedge|\overrightarrow{x}|=|\overrightarrow{u}|$$

The additional reduction rules are shown in Table 8. Rule RST-BEGIN is as before, except that input is substituted by input with timeout. In RST-TIMEOUT, a timeout fires if no reception has started before the end of the current instant of time. For processes of the form $\lceil in(x).P\rceil^t Q$ and $\sigma.P$, rule RST-INPUT-DELAY and RST-PASS-DELAY model the sleeping for one time unit respectively. Rule RST-PASS-NA is a modification of the former one: non-active processes other than those of the form $\lceil in(x).P\rceil^t Q$ and $\sigma.P$ have no reaction to the time passing event. Then the remaining rules are self-explanatory.

7 Case Study

We start this section by taking some MAC-layer protocols: CSMA and MACA/R-T as examples to show the expressiveness of our calculus.

7.1 Carrier Sense Multiple Access

The *Carrier Sense Multiple Access* (CSMA) scheme is a widely used MAC-layer protocol. In this protocol, each device senses the channel (physical carrier) before its transmission. If the channel is free, the sender starts the transmission immediately; otherwise the device keeps monitoring the channel until it becomes idle and then starts the transmission.

We can easily model the carrier sense action of CSMA scheme by the process defined below:

$$\mathsf{Send}(\mathsf{l},\mathsf{c},\mathsf{u}) \stackrel{\mathsf{def}}{=} [\mathsf{T}|_{\mathsf{l},\mathsf{c}} = \emptyset]\mathsf{out}\langle \mathsf{u}\rangle, \sigma.\mathsf{Send}(\mathsf{l},\mathsf{c},\mathsf{u})$$

Let us represent some reduction traces for a network where nodes adopt the CSMA protocol. These traces indicate that the CSMA protocol does not address the issue of node mobility. When an active transmitter moves to the reception range of an occupied receiver, any transmission of the intruding node may cause interference with the ongoing transmission. Similarly, if an active receiver moves in the transmission cell of another transmitter, the transmission of the new transmitter will also interfere with the original one. Further, interference may as well occur when different packages are targeted at the same receiver simultaneously.

Example 1 (*Interference*). This example represents an active transmitter n_3 moves to n_3' during its com-

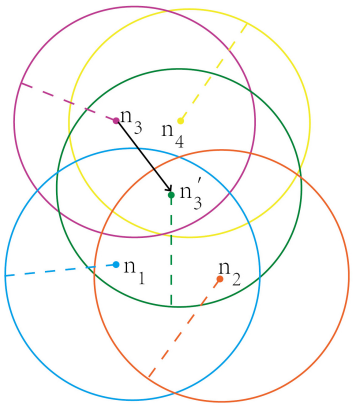


Figure 1: Network topology of Lemma 1-2 and Example 1-3

munication with node n_4. Due to this event, the active receiver n_2 which is receiving data from node n_1 gets an interference since it passively enters the transmission cell of n_3'. The network is:

$$\mathsf{N} \stackrel{\mathsf{def}}{=} \mathsf{n}_1[\langle \mathsf{v}_1\rangle^{\delta_1}]^{\mathsf{c}}_{\mathsf{l}_1,\mathsf{r}_1} \mid \mathsf{n}_2[(\mathsf{x}_2)^{\delta_1}_{\mathsf{v}_1}.\mathsf{P}_1]^{\mathsf{c}}_{\mathsf{l}_2,\mathsf{r}_2} \mid \mathsf{n}_3[\langle \mathsf{v}_3\rangle^{\delta_3}]^{\mathsf{c}}_{\mathsf{l}_3,\mathsf{r}_3} \mid \mathsf{n}_4[(\mathsf{x}_4)^{\delta_3}_{\mathsf{v}_3}.\mathsf{P}_4]^{\mathsf{c}}_{\mathsf{l}_4,\mathsf{r}_4}$$

where n_2 and n_3' are in the transmission cell of n_1, just as n_4 in n_3, n_1 and n_2 together with n_4 in n_3' (as shown in Fig.1).

We present a possible reduction trace, and it is easily understood.

$$\{(\mathsf{l}_1,\mathsf{r}_1,\mathsf{c}),(\mathsf{l}_3,\mathsf{r}_3,\mathsf{c})\} \rhd \mathsf{N} \hookrightarrow^{\mathsf{c}}_{\mathsf{l}_3:\mathsf{l}_3',\mathsf{r}_3} \mathsf{n}_1[\langle \mathsf{v}_1\rangle^{\delta_1}]^{\mathsf{c}}_{\mathsf{l}_1,\mathsf{r}_1} | \mathsf{n}_2[\mathsf{P}_2\{\perp/\mathsf{x}_2\}]^{\mathsf{c}}_{\mathsf{l}_2,\mathsf{r}_2} | \mathsf{n}_3[\langle \mathsf{v}_3\rangle^{\delta_3}]^{\mathsf{c}}_{\mathsf{l}_3',\mathsf{r}_3} | \mathsf{n}_4[(\mathsf{x}_4)^{\delta_3}_{\mathsf{v}_3}.\mathsf{P}_4]$$

Analogously, if n_3 is an active receiver and moves to n_3' during its reception from n_4. Then the transmission of n_1 to n_2 will also interfere with n_4 to n_3'. □

Example 2 (*Interference*). This example indicates interference caused by the simultaneous transmission of two different packages. Let us consider now a different network:

$$\mathsf{N} \stackrel{\mathsf{def}}{=} \mathsf{n}_1[\mathsf{Send}(\mathsf{l}_1,\mathsf{c},\mathsf{u}_1)]^{\mathsf{c}}_{\mathsf{l}_1,\mathsf{r}_1} \mid \mathsf{n}_2[\mathsf{Send}(\mathsf{l}_2,\mathsf{c},\mathsf{u}_2)]^{\mathsf{c}}_{\mathsf{l}_2,\mathsf{r}_2} \mid \mathsf{n}_3[\mathsf{in}(\mathsf{x}).\mathsf{P}]^{\mathsf{c}}_{\mathsf{l}_3',\mathsf{r}_3}$$

where n_2 and n_3' are in the transmission cell of n_1, just as n_1 and n_3' in n_2 (see Fig.1).

Table 9. MACA/R-T

$$
\begin{aligned}
\mathsf{SND}(\mathsf{sid},\mathsf{rid},\mathsf{u}) \stackrel{\mathsf{def}}{=} {} & \rhd \mathsf{c}_{\mathsf{r},\mathsf{rid}}.\mathsf{out}\langle\{\mathsf{sid},\mathsf{rid},\mathsf{rts},\wr\{\mathsf{sid},\mathsf{rid},\mathsf{end},\mathsf{u}\}\wr\}\rangle. \\
& \rhd \mathsf{c}_{\mathsf{s},\mathsf{rid}}.\lceil \mathsf{in}(\mathsf{x}). \\
& \qquad [\mathsf{fst}(\mathsf{x}) = \mathsf{rid} \wedge \mathsf{snd}(\mathsf{x}) = \mathsf{sid} \wedge \mathsf{trd}(\mathsf{x}) = \mathsf{cts}] \\
& \qquad \rhd \mathsf{c}_{\mathsf{s},\mathsf{sid}}.\mathsf{out}\langle\{\mathsf{sid},\mathsf{rid},\mathsf{end},\mathsf{u}\}\rangle. \rhd \mathsf{c}_{\mathsf{r},\mathsf{sid}}, \\
& \qquad \mathsf{SND}(\mathsf{sid},\mathsf{rid},\mathsf{u})\rceil^{\mathsf{t}} \\
& \quad \mathsf{SND}(\mathsf{sid},\mathsf{rid},\mathsf{u}) \\
\mathsf{RCV}(\mathsf{id},\mathsf{q}) \stackrel{\mathsf{def}}{=} {} & \lceil \mathsf{in}(\mathsf{x}). \\
& \quad [\mathsf{snd}(\mathsf{x}) = \mathsf{id} \wedge \mathsf{trd}(\mathsf{x}) = \mathsf{rts}] \\
& \quad \rhd \mathsf{c}_{\mathsf{s},\mathsf{id}}.\mathsf{out}\langle\{\mathsf{id},\mathsf{fst}(\mathsf{x}),\mathsf{cts},\mathsf{fth}(\mathsf{x})\}\rangle. \\
& \quad \rhd \mathsf{c}_{\mathsf{s},\mathsf{fst}(\mathsf{x})}.\lceil \mathsf{in}(\mathsf{y}). \\
& \qquad [\mathsf{snd}(\mathsf{y}) = \mathsf{id} \wedge \mathsf{fst}(\mathsf{y}) = \mathsf{fst}(\mathsf{x}) \wedge \mathsf{trd}(\mathsf{y}) = \mathsf{end}] \\
& \qquad \mathsf{RCV}(\mathsf{id},\mathsf{push}(\mathsf{q},\mathsf{fth}(\mathsf{y}))), \\
& \qquad \rhd \mathsf{c}_{\mathsf{r},\mathsf{id}}.\mathsf{RCV}(\mathsf{id},\mathsf{q})\rceil^{\mathsf{t}} \\
& \quad \rhd \mathsf{c}_{\mathsf{r},\mathsf{id}}.\mathsf{RCV}(\mathsf{id},\mathsf{q}), \\
& \mathsf{RCV}(\mathsf{id},\mathsf{q})\rceil^{\mathsf{t}} \\
& \mathsf{RCV}(\mathsf{id},\mathsf{q})
\end{aligned}
$$

A possible reduction trace is given:

$$\emptyset \rhd \mathsf{N} \hookrightarrow \mathsf{n}_1[\mathsf{out}\langle \mathsf{u}_1\rangle]^{\mathsf{c}}_{\mathsf{l}_1,\mathsf{r}_1} | \mathsf{n}_2[\mathsf{Send}(\mathsf{l}_2,\mathsf{c},\mathsf{u}_2)]^{\mathsf{c}}_{\mathsf{l}_2,\mathsf{r}_2} | \mathsf{n}_3[\mathsf{in}(\mathsf{x}).\mathsf{P}]^{\mathsf{c}}_{\mathsf{l}'_3,\mathsf{r}_3} \stackrel{\mathsf{def}}{=} \mathsf{N}_1$$

$$\emptyset \rhd \mathsf{N}_1 \hookrightarrow \mathsf{n}_1[\mathsf{out}\langle \mathsf{u}_1\rangle]^{\mathsf{c}}_{\mathsf{l}_1,\mathsf{r}_1} | \mathsf{n}_2[\mathsf{out}\langle \mathsf{u}_2\rangle]^{\mathsf{c}}_{\mathsf{l}_2,\mathsf{r}_2} | \mathsf{n}_3[\mathsf{in}(\mathsf{x}).\mathsf{P}]^{\mathsf{c}}_{\mathsf{l}'_3,\mathsf{r}_3} \stackrel{\mathsf{def}}{=} \mathsf{N}_2$$

Assign $\mathsf{v}_1 = [\![\mathsf{u}_1]\!]$ and $\delta_1 = \wr \mathsf{u}_1 \wr$, then

$$\emptyset \rhd \mathsf{N}_2 \hookrightarrow^{\mathsf{c}}_{\mathsf{l}_1,\mathsf{r}_1} \mathsf{n}_1[\langle \mathsf{v}_1\rangle^{\delta_1}]^{\mathsf{c}}_{\mathsf{l}_1,\mathsf{r}_1} | \mathsf{n}_2[\mathsf{out}\langle \mathsf{u}_2\rangle]^{\mathsf{c}}_{\mathsf{l}_2,\mathsf{r}_2} | \mathsf{n}_3[(\mathsf{x})^{\delta_1}_{\mathsf{v}_1}.\mathsf{P}]^{\mathsf{c}}_{\mathsf{l}'_3,\mathsf{r}_3} \stackrel{\mathsf{def}}{=} \mathsf{N}_3$$

Assign $\mathsf{v}_2 = [\![\mathsf{u}_2]\!]$ and $\delta_2 = \wr \mathsf{u}_2 \wr$, then

$$\{(\mathsf{l}_1,\mathsf{r}_1,\mathsf{c})\} \rhd \mathsf{N}_3 \hookrightarrow^{\mathsf{c}}_{\mathsf{l}_2,\mathsf{r}_2} \mathsf{n}_1[\langle \mathsf{v}_1\rangle^{\delta_1}]^{\mathsf{c}}_{\mathsf{l}_1,\mathsf{r}_1} | \mathsf{n}_2[\langle \mathsf{v}_2\rangle^{\delta_2}]^{\mathsf{c}}_{\mathsf{l}_2,\mathsf{r}_2} | \mathsf{n}_3[\mathsf{P}\{\bot/\mathsf{x}\}]^{\mathsf{c}}_{\mathsf{l}'_3,\mathsf{r}_3}$$

Here n_1 senses the channel free, and then almost at the same time, n_2 also finds the channel available, so they begin to transmit data successively. Therefore an interference is generated at n'_3. □

7.2 MACA/R-T

The *Receiver-Transmitter-Based Multiple Access with Collision Avoidance Protocol* (MACA/R-T) is a promising protocol used in MANETs. In MACA/R-T, all mobile nodes in the network agree to a set of pre-specified channels, e.g., a node id is assigned with $\mathsf{c}_{\mathsf{r},\mathsf{id}}$ and $\mathsf{c}_{\mathsf{s},\mathsf{id}}$ as its receiver and transmitter channels respectively. At the idle stage, all nodes will tune their receivers to their own receiver channel. When node sid wants to send a data package to node rid, node sid first sends a short control packet RST (request-to-send, which includes the sender id, the receiver id and the transmission duration of the data package) to node rid over channel $\mathsf{c}_{\mathsf{r},\mathsf{rid}}$ and then tunes its receiver to channel $\mathsf{c}_{\mathsf{s},\mathsf{rid}}$ to wait for a control packet CTS (clear-to-send, which includes the same duration information) from node rid. Upon receiving the RTS, node rid will send a CTS over channel $\mathsf{c}_{\mathsf{s},\mathsf{rid}}$ and tune its receiver to channel $\mathsf{c}_{\mathsf{s},\mathsf{sid}}$ for the data package. Finally, node sid receives the CTS and sends the data package to node rid over channel $\mathsf{c}_{\mathsf{s},\mathsf{sid}}$.

In Table 9, we provide an encoding of a sender and a receiver process in our TCMN with respect to the MACA/R-T protocol. We assume that the receiver has a queue to store the received packages, with

an operation push to insert an element. We also use four-tuples as values, with constructor $\{\_,\_,\_,\_\}$ and destructors fst, snd, trd and fth, retrieving the first, second, third and fourth component separately. We indicate with t the maximum time for a data package from the sender to arrive at the receiver.

The sender process $\mathsf{SND}(\mathsf{sid},\mathsf{rid},\mathsf{u})$ runs at node sid and intends to transmit the value u to node rid. The process first switches its transmitter channel to $\mathsf{c}_{\mathsf{r},\mathsf{rid}}$ and then sends a RTS packet. After that, it waits for the CTS packet. If the CTS packet is not received before the end of the t time units, the process will move to itself and restart the transmission. On the other hand, if the CTS packet is received before the timeout, the data package $\{\mathsf{sid},\mathsf{rid},\mathsf{end},\mathsf{u}\}$ is transmitted over channel $\mathsf{c}_{\mathsf{s},\mathsf{sid}}$ and the sender finishes the transmission.

The receiver process $\mathsf{RCV}(\mathsf{id},\mathsf{q})$ is supposed to run at node id waiting for a RTS packet. If the RTS packet, with destination id, arrives before the timeout, the receiver switches its transmitter channel to $\mathsf{c}_{\mathsf{s},\mathsf{id}}$ and then replies with a CTS packet as well as waits for the data package over channel $\mathsf{c}_{\mathsf{s},\mathsf{fst}(\mathsf{x})}$. Otherwise, the receiver aborts the current reception and resets to process $\mathsf{RCV}(\mathsf{id},\mathsf{q})$.

We show below that the MACA/R-T protocol is robust against node mobility, i.e., node movement will not give rise to communication interference. When an active transmitter moves to the reception range of an occupied receiver, the transmission of the intruding node will not interfere with the ongoing one. Besides, if an active receiver moves in the transmission cell of another transmitter, the transmission of the new transmitter will not interfere with the original one.

Lemma 1 Suppose when node n_1 is transmitting to node n_2 and node n_3 is transmitting to node n_4, n_3 moves to n_3'. The network topology is shown in Fig.1, n_2 and n_3' are in the transmission cell of n_1, just as n_4 in n_3, n_1 and n_2 together with n_4 in n_3'. Then the transmission of n_3' to n_4 will not interfere with that of n_1 to n_2.

Proof. Remember that only when an active receiver has received more than one transmission over the same channel, does the receiver get interference.

There are three kinds of packages in the MACA/R-T protocol: RTS, CTS and data, which are transmitted over channels $\mathsf{c}_{\mathsf{r},\mathsf{rid}}$, $\mathsf{c}_{\mathsf{s},\mathsf{rid}}$, and $\mathsf{c}_{\mathsf{s},\mathsf{sid}}$ respectively. According to the package types that n_1 and n_3 are sending, all the possible cases of the active transmitter n_3 moves to n_3' are listed in the table below.

$\mathsf{n}_3 \dashrightarrow \mathsf{n}_4$	$\mathsf{n}_3' \dashrightarrow \mathsf{n}_4$	$\mathsf{n}_1 \dashrightarrow \mathsf{n}_2$	Interference at n_2	Reasons
$\mathsf{c}_{\mathsf{r},\mathsf{n}_4}.\mathsf{RTS}$	$\mathsf{c}_{\mathsf{r},\mathsf{n}_4}.\mathsf{RTS}$	$\mathsf{c}_{\mathsf{r},\mathsf{n}_2}.\mathsf{RTS}$	No	Different channels
$\mathsf{c}_{\mathsf{r},\mathsf{n}_4}.\mathsf{RTS}$	$\mathsf{c}_{\mathsf{r},\mathsf{n}_4}.\mathsf{RTS}$	$\mathsf{c}_{\mathsf{s},\mathsf{n}_2}.\mathsf{CTS}$	No	Different channels
$\mathsf{c}_{\mathsf{r},\mathsf{n}_4}.\mathsf{RTS}$	$\mathsf{c}_{\mathsf{r},\mathsf{n}_4}.\mathsf{RTS}$	$\mathsf{c}_{\mathsf{s},\mathsf{n}_1}.\mathsf{data}$	No	Different channels
$\mathsf{c}_{\mathsf{s},\mathsf{n}_4}.\mathsf{CTS}$	$\mathsf{c}_{\mathsf{s},\mathsf{n}_4}.\mathsf{CTS}$	$\mathsf{c}_{\mathsf{r},\mathsf{n}_2}.\mathsf{RTS}$	No	Different channels
$\mathsf{c}_{\mathsf{s},\mathsf{n}_4}.\mathsf{CTS}$	$\mathsf{c}_{\mathsf{s},\mathsf{n}_4}.\mathsf{CTS}$	$\mathsf{c}_{\mathsf{s},\mathsf{n}_2}.\mathsf{CTS}$	No	Different channels
$\mathsf{c}_{\mathsf{s},\mathsf{n}_4}.\mathsf{CTS}$	$\mathsf{c}_{\mathsf{s},\mathsf{n}_4}.\mathsf{CTS}$	$\mathsf{c}_{\mathsf{s},\mathsf{n}_1}.\mathsf{data}$	No	Different channels
$\mathsf{c}_{\mathsf{s},\mathsf{n}_3}.\mathsf{data}$	$\mathsf{c}_{\mathsf{s},\mathsf{n}_3}.\mathsf{data}$	$\mathsf{c}_{\mathsf{r},\mathsf{n}_2}.\mathsf{RTS}$	No	Different channels
$\mathsf{c}_{\mathsf{s},\mathsf{n}_3}.\mathsf{data}$	$\mathsf{c}_{\mathsf{s},\mathsf{n}_3}.\mathsf{data}$	$\mathsf{c}_{\mathsf{s},\mathsf{n}_2}.\mathsf{CTS}$	No	Different channels
$\mathsf{c}_{\mathsf{s},\mathsf{n}_3}.\mathsf{data}$	$\mathsf{c}_{\mathsf{s},\mathsf{n}_3}.\mathsf{data}$	$\mathsf{c}_{\mathsf{s},\mathsf{n}_1}.\mathsf{data}$	No	Different channels

We can see that when an active transmitter (e.g., n_3) moves to the reception range of an occupied receiver (e.g., n_2), due to the different tranmission channels, the transmission of the intruding node will not interfere with the ongoing one. □

Lemma 2 Suppose when node n_1 is transmitting to node n_2 and node n_4 is transmitting to node n_3, n_3 moves to n_3'. As shown in Fig.1, n_2 and n_3' are in the transmission cell of n_1, just like n_3 and n_3' in n_4. Then the transmission of n_1 to n_2 will not interfere with that of n_4 to n_3'.

Proof. The proof is similar to the one for Lemma 1, and we conclude that if an active receiver (e.g., n_3) moves in the transmission cell of another transmitter (e.g., n_1), the transmission of the new transmitter will not interfere with the original one. □

Nevertheless, in the MACA/R-T protocol, interference may still occur when different RTS packets are targeted at the same receiver simultaneously.

Example 3 (*Interference*). Let's consider the network:

$$\mathsf{N} \stackrel{\mathsf{def}}{=} \mathsf{n}_1[\mathsf{SND}(\mathsf{n}_1,\mathsf{n}_3,\mathsf{u}_1)]^{\mathsf{c}_{\mathsf{r},\mathsf{n}_1}}_{\mathsf{l}_1,\mathsf{r}_1} \mid \mathsf{n}_2[\mathsf{SND}(\mathsf{n}_2,\mathsf{n}_3,\mathsf{u}_2)]^{\mathsf{c}_{\mathsf{r},\mathsf{n}_2}}_{\mathsf{l}_2,\mathsf{r}_2} \mid \mathsf{n}_3[\mathsf{RCV}(\mathsf{n}_3,[\,])]^{\mathsf{c}_{\mathsf{r},\mathsf{n}_3}}_{\mathsf{l}'_3,\mathsf{r}_3}$$

where n_2 and $\mathsf{n}_3^{'}$ are in the transmission cell of n_1, as n_1 and $\mathsf{n}_3^{'}$ in n_2 (see Fig.1).
Here we present a possible reduction trace:

$$\emptyset \rhd \mathsf{N} \hookrightarrow\hookrightarrow\hookrightarrow^{\mathsf{c}_{\mathsf{r},\mathsf{n}_3}}_{\mathsf{l}_1,\mathsf{r}_1} \hookrightarrow^{\mathsf{c}_{\mathsf{r},\mathsf{n}_3}}_{\mathsf{l}_2,\mathsf{r}_2}$$

Initially, n_1 tunes its receiver to channel $\mathsf{c}_{\mathsf{r},\mathsf{n}_3}$ and sends a RTS packet to $\mathsf{n}_3^{'}$. Almost at the same time, n_2 also tunes its receiver to channel $\mathsf{c}_{\mathsf{r},\mathsf{n}_3}$ and sends a RTS packet to $\mathsf{n}_3^{'}$ which unfortunately results in an interference at $\mathsf{n}_3^{'}$. □

8 Conclusions and Future Work

In this paper, we have proposed a timed calculus for mobile ad hoc networks paying particular attention to local broadcast, node mobility and communication interference. Then the operational semantics of our calculus is given both in terms of a Reduction Semantics and in terms of a Labelled Transition Systems. We have also proved that these two semantics coincide. Finally, we extend our core language by adding some new operators to model the CSMA and MACA/R-T protocol. And we have demonstrated that the former doesn't address the issue of node mobility while the latter is robust against node mobility.

In the future, a quantity of developments are possible. First, we would try to establish adequate Behavioral Equivalences which define when two terms have the same observable behavior. One possible approach is via UTP method, so as to investigate the denotational semantics for mobile ad hoc networks. Second, we would also like to study a set of algebraic laws, which can represent the features of mobile ad hoc networks.

References

[1] Imrich Chlamtac, Marco Conti and Jennifer J.-N. Liu, *Mobile ad hoc networking: imperatives and challenges*, Ad Hoc Networks 1 (1) (2003) 13-64.

[2] Jari Ahola, *Ambient Intelligence*, ERCIM (European Research Consortium for Information and Mathematics) NEWS, 47 (2001).

[3] Stefano Basagni, Imrich Chlamtac and Violet R. Syrotiuk, *Location aware one-to-many communication in mobile multi-hop wireless networks*, in: Proceedings of the IEEE Vehicular Technology (VTC), (2000).

[4] Thomas Guthrie Zimmerman, *Personal Area Networks (PAN): Near-field intrabody communication*, IBM Systems Journal 35 (3-4) (1996) 609-617.

[5] Mario Joa-Ng and I-Tai Lu, *Spread Spectrum Medium Access Protocol with Collision Avoidance in Mobile Ad-hoc Wireless Network*, in: Proceedings of the IEEE INFOCOM 99, (1999) 21-25.

[6] Sebastian Nanz and Chris Hankin, *A framework for security analysis of mobile wireless networks*, Theoretical Computer Science 367 (1-2) (2006) 203-227.

[7] Jens Chr. Godskesen, *A calculus for mobile ad hoc networks*, in: COORDINATION, in: Lecture Notes in Computer Science, 4467 (2007) 132-150.

[8] Fatemeh Ghassemi, Wan Fokkink and Ali Movaghar, *Restricted broadcast process theory*, in: SEFM, IEEE Computer Society, (2008) 345C354.

[9] Fatemeh Ghassemi, Wan Fokkink and Ali Movaghar, *Equational reasoning on ad hoc networks*, in: FSEN, in: Lecture Notes in Computer Science, 5961 (2009) 113C128.

[10] Jens Chr. Godskesen, *A calculus for mobile ad-hoc networks with static location binding*, Electr. Notes Theor. Comput. Sci. 242 (1) (2009) 161-183.

[11] Massimo Merro, *An observational theory for mobile ad hoc networks (full version)*, Information and Computation 207 (2) (2009) 194-208.

[12] Ivan Lanese and Davide Sangiorgi, *An operational semantics for a calculus for wireless systems*, Theoretical Computer Science 411 (19) (2010) 1928-1948.

[13] Anu Singh, C.R. Ramakrishnan and Scott A. Smolka, *A process calculus for mobile ad hoc networks*, Science of Computer Programming, 75 (6) (2010) 440-469.

[14] Massimo Merro, Francesco Ballardin and Eleonora Sibilio, *A Timed Calculus for Wireless Systems*, Theoretical Computer Science 412 (47) (2011) 6585-6611.

[15] Dimitrios Kouzapas and Anna Philippou, *A process calculus for dynamic networks*, in: FMOODS/FORTE, in: Lecture Notes in Computer Science, 6722 (2011) 213-227.

[16] Qun Li and Daniela Rus, *Global clock synchronization in sensor networks*, IEEE Transactions on Computers 55 (2) (2006) 214-226.

[17] Suyoung Yoon, Chanchai Veerarittiphan and Mihail L. Sichitiu, *Tiny-sync: tight time synchronization for wireless sensor networks*, ACM Transactions on Sensor Networks 3 (2) (2007) 81-118.

9 Appendix

Lemma 1. If $\mathsf{T} \rhd \mathsf{N} \xrightarrow{\mathsf{c?[(I:I'),r]}} \mathsf{N}'$, then

$$\mathsf{N} \equiv \prod_{\mathsf{i}\in\mathsf{I}} \mathsf{n_i}[(\mathsf{x_i})^{\delta}_{\mathsf{v}}.\mathsf{P_i}]^{\mathsf{c}}_{\mathsf{l_i},\mathsf{r_i}} | \prod_{\mathsf{j}\in\mathsf{J}} \mathsf{n_j}[(\mathsf{x_j})^{\delta}_{\mathsf{v}}.\mathsf{P_j}]^{\mathsf{c}}_{\mathsf{l_j},\mathsf{r_j}} | \prod_{\mathsf{k}\in\mathsf{K}} \mathsf{n_k}[(\mathsf{x_k})^{\delta_{\mathsf{k}}}_{\mathsf{v_k}}.\mathsf{P_k}]^{\mathsf{c}}_{\mathsf{l_k},\mathsf{r_k}} | \mathsf{N}''$$

where $\forall \mathsf{i} \in \mathsf{I}.\mathsf{d}(\mathsf{l},\mathsf{l_i}) \leq \mathsf{r} \wedge \mathsf{d}(\mathsf{l}',\mathsf{l_i}) \leq \mathsf{r},\ \ \forall \mathsf{j} \in \mathsf{J}.\mathsf{d}(\mathsf{l},\mathsf{l_j}) \leq \mathsf{r} \wedge \mathsf{d}(\mathsf{l}',\mathsf{l_j}) > \mathsf{r},\ \ \forall \mathsf{k} \in \mathsf{K}.\mathsf{d}(\mathsf{l},\mathsf{l_k}) > \mathsf{r} \wedge \mathsf{d}(\mathsf{l}',\mathsf{l_k}) \leq \mathsf{r}$

and $(\mathsf{l},\mathsf{r},\mathsf{c}) \not\Vdash_{\mathsf{ai}} \mathsf{N}'' \wedge (\mathsf{l}',\mathsf{r},\mathsf{c}) \not\Vdash_{\mathsf{ai}} \mathsf{N}''$. Furthermore

$$\mathsf{N}' \equiv \prod_{\mathsf{i}\in\mathsf{I}} \mathsf{n_i}[(\mathsf{x_i})^{\delta}_{\mathsf{v}}.\mathsf{P_i}]^{\mathsf{c}}_{\mathsf{l_i},\mathsf{r_i}} | \prod_{\mathsf{j}\in\mathsf{J}} \mathsf{n_j}[\mathsf{P_j}\{\varepsilon/\mathsf{x_j}\}]^{\mathsf{c}}_{\mathsf{l_j},\mathsf{r_j}} | \prod_{\mathsf{k}\in\mathsf{K}} \mathsf{n_k}[\mathsf{P_k}\{\perp/\mathsf{x_k}\}]^{\mathsf{c}}_{\mathsf{l_k},\mathsf{r_k}} | \mathsf{N}''$$

Proof. The proof is by rule induction on the derivation of $\mathsf{T} \rhd \mathsf{N} \xrightarrow{\mathsf{c?[(I:I'),r]}} \mathsf{N}'$.

Rule NS-MOVE$_{in1}$ From the premise of the rule, we know that either $\mathsf{n}[\mathsf{Q}]^{\mathsf{c}}_{\mathsf{l},\mathsf{r}}$ is a non-active input node or an active input node that always within or beyond the mobile transmitter's transmission range. In the first and third case, the corresponding node can be inserted into N'', while in the second case, $\mathsf{Q} = \mathsf{Q}' = (\mathsf{x_i})^{\delta}_{\mathsf{v}}.\mathsf{P_i}$, thus it follows that $\mathsf{I} = \{\mathsf{i}\}, \mathsf{J} = \mathsf{K} = \emptyset$, and $\mathsf{N}'' = \mathsf{0}$.

Rule NS-MOVE$_{in2}$ Here too we know that Q is an active input process and $\mathsf{Q} \xrightarrow{?\varepsilon} \mathsf{Q}'$, thus by inspection on the LTS for processes, we get one case, rule PS-IN$_{err}$. It corresponds to index Q with $\mathsf{j} \in \mathsf{J}$.

Rule NS-MOVE$_{in3}$ Similarly, when Q is an active input process, then $\mathsf{Q} \xrightarrow{?\perp} \mathsf{Q}'$ can only be derived using

rule RS-IN$_{interfere}$. Hence, Q is indexed with $\mathsf{k} \in \mathsf{K}$.

Rule NS-MOVE$_{in}$ This is the inductive case. It brings the corresponding sets of indices and the non-index part of the network in the previous premises to the desired form. □

Lemma 2. If $\mathsf{T} \rhd \mathsf{N} \xrightarrow{\mathsf{c}![(\mathsf{l}:\mathsf{l}'),\mathsf{r}]} \mathsf{N}'$, then

$$\mathsf{N} \equiv \mathsf{n}[\langle \mathsf{v} \rangle^{\delta}.\mathsf{P}]^{\mathsf{c}}_{\mathsf{l},\mathsf{r}} | \prod_{\mathsf{i}\in\mathsf{I}} \mathsf{n}_\mathsf{i}[(\mathsf{x}_\mathsf{i})^{\delta}_{\mathsf{v}}.\mathsf{P}_\mathsf{i}]^{\mathsf{c}}_{\mathsf{l}_\mathsf{i},\mathsf{r}_\mathsf{i}} | \prod_{\mathsf{j}\in\mathsf{J}} \mathsf{n}_\mathsf{j}[(\mathsf{x}_\mathsf{j})^{\delta}_{\mathsf{v}}.\mathsf{P}_\mathsf{j}]^{\mathsf{c}}_{\mathsf{l}_\mathsf{j},\mathsf{r}_\mathsf{j}} | \prod_{\mathsf{k}\in\mathsf{K}} \mathsf{n}_\mathsf{k}[(\mathsf{x}_\mathsf{k})^{\delta_\mathsf{k}}_{\mathsf{v}_\mathsf{k}}.\mathsf{P}_\mathsf{k}]^{\mathsf{c}}_{\mathsf{l}_\mathsf{k},\mathsf{r}_\mathsf{k}} | \mathsf{N}''$$

where $\forall \mathsf{i} \in \mathsf{I}.\mathsf{d}(\mathsf{l},\mathsf{l}_\mathsf{i}) \le \mathsf{r} \wedge \mathsf{d}(\mathsf{l}',\mathsf{l}_\mathsf{i}) \le \mathsf{r},\ \ \forall \mathsf{j} \in \mathsf{J}.\mathsf{d}(\mathsf{l},\mathsf{l}_\mathsf{j}) \le \mathsf{r} \wedge \mathsf{d}(\mathsf{l}',\mathsf{l}_\mathsf{j}) > \mathsf{r},\ \ \forall \mathsf{k} \in \mathsf{K}.\mathsf{d}(\mathsf{l},\mathsf{l}_\mathsf{k}) > \mathsf{r} \wedge \mathsf{d}(\mathsf{l}',\mathsf{l}_\mathsf{k}) \le \mathsf{r}$

and $(\mathsf{l},\mathsf{r},\mathsf{c}) \not\downarrow_{\mathsf{ai}} \mathsf{N}'' \wedge (\mathsf{l}',\mathsf{r},\mathsf{c}) \not\downarrow_{\mathsf{ai}} \mathsf{N}''$. Furthermore

$$\mathsf{N}' \equiv \mathsf{n}[\langle \mathsf{v} \rangle^{\delta}.\mathsf{P}]^{\mathsf{c}}_{\mathsf{l}',\mathsf{r}} | \prod_{\mathsf{i}\in\mathsf{I}} \mathsf{n}_\mathsf{i}[(\mathsf{x}_\mathsf{i})^{\delta}_{\mathsf{v}}.\mathsf{P}_\mathsf{i}]^{\mathsf{c}}_{\mathsf{l}_\mathsf{i},\mathsf{r}_\mathsf{i}} | \prod_{\mathsf{j}\in\mathsf{J}} \mathsf{n}_\mathsf{j}[\mathsf{P}_\mathsf{j}\{\varepsilon/\mathsf{x}_\mathsf{j}\}]^{\mathsf{c}}_{\mathsf{l}_\mathsf{j},\mathsf{r}_\mathsf{j}} | \prod_{\mathsf{k}\in\mathsf{K}} \mathsf{n}_\mathsf{k}[\mathsf{P}_\mathsf{k}\{\perp/\mathsf{x}_\mathsf{k}\}]^{\mathsf{c}}_{\mathsf{l}_\mathsf{k},\mathsf{r}_\mathsf{k}} | \mathsf{N}''$$

Proof. The proof is similar to the one for Lemma 1, and it uses Lemma 1 itself to handle premises which are input transitions. □

Lemma 3. If $\mathsf{T} \rhd \mathsf{N} \xrightarrow{\mathsf{c}?\mathsf{v}:\delta[\mathsf{l},\mathsf{r}]} \mathsf{N}'$, then

$$\mathsf{N} \equiv \prod_{\mathsf{i}\in\mathsf{I}} \mathsf{n}_\mathsf{i}[\mathsf{in}(\mathsf{x}_\mathsf{i}).\mathsf{P}_\mathsf{i}]^{\mathsf{c}}_{\mathsf{l}_\mathsf{i},\mathsf{r}_\mathsf{i}} | \prod_{\mathsf{j}\in\mathsf{J}} \mathsf{n}_\mathsf{j}[\mathsf{in}(\mathsf{x}_\mathsf{j}).\mathsf{P}_\mathsf{j}]^{\mathsf{c}}_{\mathsf{l}_\mathsf{j},\mathsf{r}_\mathsf{j}} | \prod_{\mathsf{k}\in\mathsf{K}} \mathsf{n}_\mathsf{k}[(\mathsf{x}_\mathsf{k})^{\delta_\mathsf{k}}_{\mathsf{v}_\mathsf{k}}.\mathsf{P}_\mathsf{k}]^{\mathsf{c}}_{\mathsf{l}_\mathsf{k},\mathsf{r}_\mathsf{k}} | \mathsf{N}''$$

where $\forall \mathsf{h} \in \mathsf{I}\cup\mathsf{J}\cup\mathsf{K}.\mathsf{d}(\mathsf{l},\mathsf{l}_\mathsf{h}) \le \mathsf{r},\ \ \forall \mathsf{i} \in \mathsf{I}.\mathsf{T}|_{\mathsf{l}_\mathsf{i},\mathsf{c}} = \emptyset,\ \ \forall \mathsf{j} \in \mathsf{J}.\mathsf{T}|_{\mathsf{l}_\mathsf{j},\mathsf{c}} \neq \emptyset$ and $(\mathsf{l},\mathsf{r},\mathsf{c}) \not\downarrow_{\mathsf{i}} \mathsf{N}''$. Furthermore

$$\mathsf{N}' \equiv \prod_{\mathsf{i}\in\mathsf{I}} \mathsf{n}_\mathsf{i}[(\mathsf{x}_\mathsf{i})^{\delta}_{\mathsf{v}}.\mathsf{P}_\mathsf{i}]^{\mathsf{c}}_{\mathsf{l}_\mathsf{i},\mathsf{r}_\mathsf{i}} | \prod_{\mathsf{j}\in\mathsf{J}} \mathsf{n}_\mathsf{j}[\mathsf{in}(\mathsf{x}_\mathsf{j}).\mathsf{P}_\mathsf{j}]^{\mathsf{c}}_{\mathsf{l}_\mathsf{j},\mathsf{r}_\mathsf{j}} | \prod_{\mathsf{k}\in\mathsf{K}} \mathsf{n}_\mathsf{k}[\mathsf{P}_\mathsf{k}\{\perp/\mathsf{x}_\mathsf{k}\}]^{\mathsf{c}}_{\mathsf{l}_\mathsf{k},\mathsf{r}_\mathsf{k}} | \mathsf{N}''$$

Proof. The proof is similar to the one for Lemma 1, using rules for begin transmission event. □

Lemma 4. If $\mathsf{T} \rhd \mathsf{N} \xrightarrow{\mathsf{c}!\mathsf{v}:\delta[\mathsf{l},\mathsf{r}]} \mathsf{N}'$, then

$$\mathsf{N} \equiv \mathsf{n}[\mathsf{out}\langle \mathsf{u} \rangle.\mathsf{P}]^{\mathsf{c}}_{\mathsf{l},\mathsf{r}} | \prod_{\mathsf{i}\in\mathsf{I}} \mathsf{n}_\mathsf{i}[\mathsf{in}(\mathsf{x}_\mathsf{i}).\mathsf{P}_\mathsf{i}]^{\mathsf{c}}_{\mathsf{l}_\mathsf{i},\mathsf{r}_\mathsf{i}} | \prod_{\mathsf{j}\in\mathsf{J}} \mathsf{n}_\mathsf{j}[\mathsf{in}(\mathsf{x}_\mathsf{j}).\mathsf{P}_\mathsf{j}]^{\mathsf{c}}_{\mathsf{l}_\mathsf{j},\mathsf{r}_\mathsf{j}} | \prod_{\mathsf{k}\in\mathsf{K}} \mathsf{n}_\mathsf{k}[(\mathsf{x}_\mathsf{k})^{\delta_\mathsf{k}}_{\mathsf{v}_\mathsf{k}}.\mathsf{P}_\mathsf{k}]^{\mathsf{c}}_{\mathsf{l}_\mathsf{k},\mathsf{r}_\mathsf{k}} | \mathsf{N}''$$

where $\forall \mathsf{h} \in \mathsf{I}\cup\mathsf{J}\cup\mathsf{K}.\mathsf{d}(\mathsf{l},\mathsf{l}_\mathsf{h}) \le \mathsf{r},\ \ \forall \mathsf{i} \in \mathsf{I}.\mathsf{T}|_{\mathsf{l}_\mathsf{i},\mathsf{c}} = \emptyset,\ \ \forall \mathsf{j} \in \mathsf{J}.\mathsf{T}|_{\mathsf{l}_\mathsf{j},\mathsf{c}} \neq \emptyset$ and $(\mathsf{l},\mathsf{r},\mathsf{c}) \not\downarrow_{\mathsf{i}} \mathsf{N}''$. Furthermore if $[\![\mathsf{u}]\!] = \mathsf{v}$ and $\wr \mathsf{u} \wr = \delta$ then

$$\mathsf{N}' \equiv \mathsf{n}[\langle \mathsf{v} \rangle^{\delta}.\mathsf{P}]^{\mathsf{c}}_{\mathsf{l},\mathsf{r}} | \prod_{\mathsf{i}\in\mathsf{I}} \mathsf{n}_\mathsf{i}[(\mathsf{x}_\mathsf{i})^{\delta}_{\mathsf{v}}.\mathsf{P}_\mathsf{i}]^{\mathsf{c}}_{\mathsf{l}_\mathsf{i},\mathsf{r}_\mathsf{i}} | \prod_{\mathsf{j}\in\mathsf{J}} \mathsf{n}_\mathsf{j}[\mathsf{in}(\mathsf{x}_\mathsf{j}).\mathsf{P}_\mathsf{j}]^{\mathsf{c}}_{\mathsf{l}_\mathsf{j},\mathsf{r}_\mathsf{j}} | \prod_{\mathsf{k}\in\mathsf{K}} \mathsf{n}_\mathsf{k}[\mathsf{P}_\mathsf{k}\{\perp/\mathsf{x}_\mathsf{k}\}]^{\mathsf{c}}_{\mathsf{l}_\mathsf{k},\mathsf{r}_\mathsf{k}} | \mathsf{N}''$$

Proof. The proof is similar to the one for Lemma 2, and it uses Lemma 3 itself to handle premises which are input transitions. □

Lemma 5. If $\mathsf{T} \rhd \mathsf{N} \xrightarrow{\sigma} \mathsf{N}'$, then

$$\mathsf{N} \equiv \prod_{\mathsf{i}\in\mathsf{I}} \mathsf{n}_\mathsf{i}[\langle \mathsf{v}_\mathsf{i} \rangle^{\delta_\mathsf{i}}.\mathsf{P}_\mathsf{i}]^{\mathsf{c}}_{\mathsf{l}_\mathsf{i},\mathsf{r}_\mathsf{i}} | \prod_{\mathsf{j}\in\mathsf{J}} \mathsf{n}_\mathsf{j}[(\mathsf{x}_\mathsf{j})^{\delta_\mathsf{j}}_{\mathsf{v}_\mathsf{j}}.\mathsf{P}_\mathsf{j}]^{\mathsf{c}}_{\mathsf{l}_\mathsf{j},\mathsf{r}_\mathsf{j}} | \prod_{\mathsf{k}\in\mathsf{K}} \mathsf{n}_\mathsf{k}[\mathsf{P}_\mathsf{k}]^{\mathsf{c}}_{\mathsf{l}_\mathsf{k},\mathsf{r}_\mathsf{k}}$$

where $\forall \mathsf{i} \in \mathsf{I}.\delta_\mathsf{i} > 0$ and $\forall \mathsf{j} \in \mathsf{J}.\delta_\mathsf{j} > 0$. Furthermore

$$\mathsf{N}' \equiv \prod_{\mathsf{i}\in\mathsf{I}} \mathsf{n}_\mathsf{i}[\langle \mathsf{v}_\mathsf{i} \rangle^{\delta_\mathsf{i}-1}.\mathsf{P}_\mathsf{i}]^{\mathsf{c}}_{\mathsf{l}_\mathsf{i},\mathsf{r}_\mathsf{i}} | \prod_{\mathsf{j}\in\mathsf{J}} \mathsf{n}_\mathsf{j}[(\mathsf{x}_\mathsf{j})^{\delta_\mathsf{j}-1}_{\mathsf{v}_\mathsf{j}}.\mathsf{P}_\mathsf{j}]^{\mathsf{c}}_{\mathsf{l}_\mathsf{j},\mathsf{r}_\mathsf{j}} | \prod_{\mathsf{k}\in\mathsf{K}} \mathsf{n}_\mathsf{k}[\mathsf{P}_\mathsf{k}]^{\mathsf{c}}_{\mathsf{l}_\mathsf{k},\mathsf{r}_\mathsf{k}}$$

Proof. The proof is by induction on the size of $\mathsf{I}\cup\mathsf{J}\cup\mathsf{K}$, and then for each case it is similar to the one for Lemma 1, using rules for time passing event. □

PALS-Based Analysis of an Airplane Multirate Control System in Real-Time Maude

Kyungmin Bae Joshua Krisiloff José Meseguer Peter Csaba Ölveczky

Department of Computer Science
University of Illinois at Urbana-Champaign

Department of Informatics
University of Oslo

Distributed cyber-physical systems (DCPS) are pervasive in areas such as aeronautics and ground transportation systems, including the case of distributed hybrid systems. DCPS design and verification is quite challenging because of asynchronous communication, network delays, and clock skews. Furthermore, their model checking verification typically becomes unfeasible due to the huge state space explosion caused by the system's concurrency. The PALS ("physically asynchronous, logically synchronous") methodology has been proposed to reduce the design and verification of a DCPS to the much simpler task of designing and verifying its underlying synchronous version. The original PALS methodology assumes a single logical period, but Multirate PALS extends it to deal with multirate DCPS in which components may operate with different logical periods. This paper shows how Multirate PALS can be applied to formally verify a nontrivial multirate DCPS. We use Real-Time Maude to formally specify a multirate distributed hybrid system consisting of an airplane maneuvered by a pilot who turns the airplane according to a specified angle through a distributed control system. Our formal analysis revealed that the original design was ineffective in achieving a smooth turning maneuver, and led to a redesign of the system that satisfies the desired correctness properties. This shows that the Multirate PALS methodology is not only effective for formal DCPS verification, but can also be used effectively in the DCPS *design* process, even before properties are verified.

1 Introduction

Distributed cyber-physical systems (DCPS) are pervasive in areas such as aeronautics, ground transportation systems, medical systems, and so on; they include, in particular, the case of distributed hybrid systems, whose continuous dynamics is governed by differential equations. DCPS design and verification is quite challenging, because to the usual complexity of a non-distributed CPS one has to add the additional complexities of asynchronous communication, network delays, and clock skews, which can easily lead a DCPS into inconsistent states. In particular, any hopes of applying model checking verification techniques in a direct manner to a DCPS look rather dim, due to the typically huge state space explosion caused by the system's concurrency.

For these reasons, we and other colleagues at UIUC and Rockwell-Collins Corporation have been developing the *physically asynchronous but logically synchronous* PALS methodology [9, 11], which can drastically reduce the system complexity of a DCPS so as to make it amenable to model checking verification (for a comparison of PALS with related methodologies see [13]). The PALS methodology applies to the frequently occurring case of a DCPS whose implementation must be asynchronous due to physical constraints and for fault-tolerance reasons, but whose *logical* design requires that the system components should act together in a virtually synchronous way. For example, distributed control systems are typically of this nature. The key idea of PALS is to reduce the design and verification of a DCPS of this kind to the much simpler task[1] of designing and verifying the idealized synchronous systems that

[1] For a simple avionics case study in [10], the number of system states for their simplest possible distributed version with perfect clocks and no network delays was 3,047,832, but the PALS pattern reduced the number of states to a mere 185.

Preliminary Report. Final version to appear in:
FTSCS 2012

should be realized in a distributed and asynchronous way. This is achieved by a *model transformation* $\mathscr{E} \mapsto \mathscr{A}(\mathscr{E},\Gamma)$ that maps a synchronous design $\mathscr{E}$ to a distributed implementation $\mathscr{A}(\mathscr{E},\Gamma)$ which is *correct-by construction* and that, as shown in [9, 10], is *bisimilar* to the synchronous system $\mathscr{E}$. This bisimilarity is the essential feature allowing the desired, drastic reduction in system complexity and making model checking verification feasible: since bisimilar systems satisfy the exact same temporal logic properties, we can verify that the asynchronous system $\mathscr{A}(\mathscr{E},\Gamma)$ satisfies a temporal logic property φ (which typically would be impossible to model check directly on $\mathscr{A}(\mathscr{E},\Gamma)$) by verifying the same property φ on the vastly simpler synchronous system $\mathscr{E}$.

The original PALS methodology presented in [9, 11] assumes a *single logical period*, at which all components of the DCPS must communicate with each other and transition to their next states. However, a DCPS such as a distributed control system may have components that, for physical reasons, must operate with different periods, even though those periods may all divide an overall longer period. That is, many such systems, although still having to be virtually synchronous for their correct behavior, are in fact *multirate* systems, with some components running at a faster rate than others. An interesting challenge is how to extend PALS to multirate DCPS. This challenge has been given two different answers. On the one hand, an engineering solution for Multirate PALS based on the AADL modeling language has been proposed by Al-Nayeem et al. in [1]. On the other hand, three of us have defined in [4] a mathematical model of a multirate synchronous system $\mathscr{E}$, and have formally defined a model transformation

$$\mathscr{E} \mapsto \mathscr{MA}(\mathscr{E},T,\Gamma)$$

that generalizes to multirate systems the original single-rate PALS transformation defined in [9, 10]. As before, we have proved in [4] that $\mathscr{MA}(\mathscr{E},T,\Gamma)$ is a correct-by-construction implementation of $\mathscr{E}$, and that $\mathscr{E}$ and $\mathscr{MA}(\mathscr{E},T,\Gamma)$ are *bisimilar*, making it possible to verify temporal logic properties about $\mathscr{MA}(\mathscr{E},T,\Gamma)$ on the much simpler system $\mathscr{E}$.

But how *effective* is Multirate PALS in practice? Can it be applied to formally verify important properties of a nontrivial multirate CPS such as a distributed hybrid system? The main goal of this paper is to show that the answer is an emphatic *yes*. We use Real-Time Maude [12] to formally specify in detail a multirate distributed hybrid system consisting of an airplane maneuvered by a pilot, who turns the airplane according to a specified angle α through a distributed control system with effectors located in the airplane's wings and rudder. Our formal analysis revealed that the original design had control laws that were ineffective in achieving a smooth turning maneuver. This led to a redesign of the system with new control laws which, as verified in Real-Time Maude by model checking, satisfies the desired correctness properties. This shows that the Multirate PALS methodology is not only effective for formal DCPS verification, but, when used together with a tool like Real-Time Maude, can also be used effectively in the DCPS *design* process, even before properties are verified. To the best of our knowledge, this is the first time that the Multirate PALS methodology has been applied to the model checking verification of a DCPS. In this sense, this paper complements our companion paper [4], where the mathematical foundations of Multirate PALS were developed in detail, but where only a brief summary of some of the results presented here was given.

This paper is organized as follows. Section 2 explains Multirate PALS and Real-Time Maude. Section 3 describes a simple model of an airplane turning control system whose continuous dynamics is governed by differential equations. Section 4 presents a modeling framework for multirate ensembles in Real-Time Maude, and Section 5 then formally specifies the airplane turning control system using the ensemble framework. Section 6 illustrates Real-Time Maude-based verification of the airplane turning control system. Finally, Section 7 gives some concluding remarks.

2 Preliminaries on Multirate PALS and Real-Time Maude

2.1 Multirate PALS

In many distributed real-time systems, such as automotive and avionics systems, the system design is essentially a *synchronous design* that must be realized in a distributed setting. Both design and verification of such *virtually synchronous* distributed real-time systems is very hard because of asynchronous communication, network delays, clock skews, and because the state space explosion caused by the system's concurrency can make it unfeasible to apply model checking to verify required properties. The (single-rate) PALS ("physically asynchronous logically synchronous") formal design pattern [9, 11] reduces the design and verification of a virtually synchronous distributed real-time system to the much simpler task of designing and verifying its synchronous version, provided that the network infrastructure can guarantee bounds on the messaging delays and the skews of the local clocks.

We have recently developed Multirate PALS [4], which extends PALS to hierarchical *multirate* systems in which controllers with the same rate communicate with each other and with a number of faster components. As is common for hierarchical control systems [1], we assume that the period of the higher-level controllers is a multiple of the period of each fast component. Given a multirate synchronous design $\mathfrak{E}$, bounds Γ on the network transmission times and clock skews, and function T assigning to each distributed component its period, Multirate PALS defines a transformation $(\mathfrak{E},T,\Gamma) \mapsto \mathscr{MA}(\mathfrak{E},T,\Gamma)$ mapping each synchronous design $\mathfrak{E}$ to a specification $\mathscr{MA}(\mathfrak{E},T,\Gamma)$ of the corresponding distributed multirate real-time system. In [4] we formalize Multirate PALS and show that the synchronous design $\mathfrak{E}$ and the asynchronous distributed model $\mathscr{MA}(\mathfrak{E},T,\Gamma)$ satisfy the same temporal logic properties.

Multirate Synchronous Models. In Multirate PALS, the synchronous design is formalized as the synchronous composition of a collection of *typed machines*, an *environment*, and a *wiring diagram* that connects the machines. A *typed machine* M is a tuple $M = (\mathscr{D}_i, S, \mathscr{D}_o, \delta_M)$, where $\mathscr{D}_i = D_{i_1} \times \cdots \times D_{i_n}$ is an *input set*, S is a set of *states*, $\mathscr{D}_o = D_{o_1} \times \cdots \times D_{o_m}$ is an *output set*, and $\delta_M \subseteq (D_i \times S) \times (S \times D_o)$ is a *transition relation*. Such a machine M has n input ports and m output ports.

To compose a collection of machines with different rates into a synchronous system in which all components perform one transition in lock-step in each iteration of the system, we "slow down" the faster components so that all components run at the slow rate in the synchronous composition. A fast machine that is slowed, or *decelerated*, by a factor k *performs* k *internal transitions* in one synchronous step. Since the fast machine consumes an input and produces an output in each of these internal steps, the decelerated machine consumes (resp. produces) k-tuples of inputs (resp. outputs) in each synchronous step. A k-tuple output from the fast machine must therefore be *adapted* so that it can be read by the slow component. That is, the k-tuple must be transformed to a single value (e.g., the average of the k values, the last value, or any other function of the k values); this transformation is formalized as an *input adaptor*. Likewise, the single output from a slow component must be transformed to a k-tuple of inputs to the fast machine; this is also done by input adaptors which may, for example, transform an input d to a k-tuple $(d, \bot, \ldots, \bot)$ for some "don't care" value $\bot$. Formally, an input adaptor α_M for a typed machine $M = (\mathscr{D}_i, S, \mathscr{D}_o, \delta_M)$ is a family of functions $\alpha_M = \{\alpha_k : D'_k \to D_{i_k}\}_{k \in \{1,\ldots,n\}}$, each of which determines a desired value from an output D'_k of another typed machine.

Typed machines (with rate assignments and input adaptors) can be "wired together" into *multirate machine ensembles* as shown in Figure 1, where "local" fast environments are integrated with their corresponding fast machines. A multirate machine ensemble is a tuple

$$\mathfrak{E} = (J_S \cup J_F \cup \{e\}, \{M_j\}_{j \in J_s \cup J_F}, E, src, rate, adaptor)$$

where: (i) J_S (resp., J_F) is a finite set of *indices* for controller components (resp., fast components), and $e \notin J_S \cup J_F$ is the *environment index*, (ii) $\{M_j\}_{j \in J_S \cup J_F}$ is a family of typed machines, (iii) the *environment* is a pair $E = (\mathscr{D}_i^e, \mathscr{D}_o^e)$, with $\mathscr{D}_i^e$ the environment's *input set* and $\mathscr{D}_o^e$ its *output set*, (iv) *src* is a function that assigns to each input port (j,n) (input port n of machine j) its "source," such that there are no connections between fast machines, (v) *rate* is a function assigning to each fast machine a value denoting how many times faster the machine runs compared to the controller machines, and (vi) *adaptor* is a function that assigns an input adaptor to each $l \in J_F \cup J_S$.

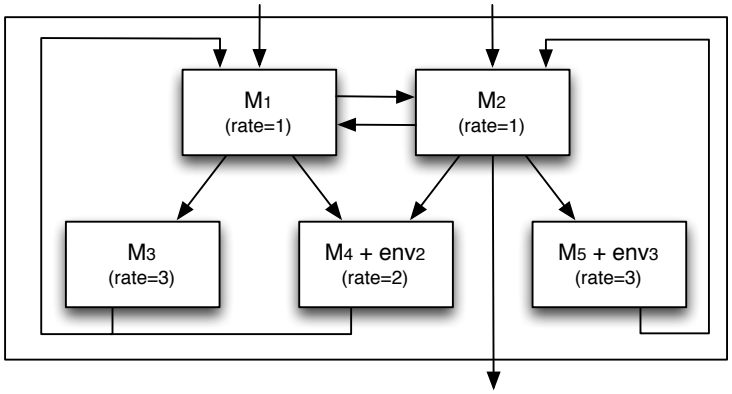


Figure 1: A multirate machine ensemble. M_1 and M_2 are controller machines, and env_2 and env_3 are local environments with faster rates, hidden from high-level controllers.

The transitions of all machines in an ensemble are performed simultaneously, where each fast machine with rate k performs k "internal transitions" in one synchronous transition step. If a machine has a feedback wire to itself and/or to another machine, then the output becomes an input at the *next* instant. The *synchronous composition* of a multirate ensemble $\mathfrak{E}$ is therefore equivalent to a *single machine* $M_{\mathfrak{E}}$, where each state of $M_{\mathfrak{E}}$ consists of the states of its subcomponents and of the contents in the feedback outputs, as explained in [4, 10]. The synchronous composition of the ensemble in Figure 1 is the machine given by the outer box. Since $M_{\mathfrak{E}}$ is itself a typed machine which can appear as a component in another multirate ensemble, we can easily define hierarchical multirate systems.

2.2 Real-Time Maude

Real-Time Maude [12] extends the Maude language and tool [6] to support the formal modeling analysis of real-time systems in rewriting logic. In Real-Time Maude, the data types of the system are defined by a *membership equational logic* [6] theory (Σ, E) with Σ a signature[2] and E a set of *confluent and terminating conditional equations*; the system's *instantaneous* (i.e., zero-time) local transitions are defined by (possibly conditional) *labeled instantaneous rewrite rules*[3]; and time elapse is modeled explicitly by *tick rewrite rules* of the form `crl [`l`] : {`u`} => {`v`} in time` τ `if` *cond*, which specifies a transition with duration τ from an instance of the term $\{u\}$ to the corresponding instance of the term $\{v\}$.

The Real-Time Maude syntax is fairly intuitive. A function symbol f is declared with the syntax `op` f `:` $s_1 \ldots s_n$ `->` s, where $s_1 \ldots s_n$ are the sorts of its arguments, and s is its (value) *sort*. Equations are written with syntax `eq` u `=` v, or `ceq` u `=` v `if` *cond* for conditional equations. We refer to [6] for more details on the syntax of Real-Time Maude.

[2] i.e., Σ is a set of declarations of *sorts*, *subsorts*, and *function symbols*.

[3] $E = E' \cup A$, where A is a set of axioms such as associativity and commutativity, so that deduction is performed *modulo* A.

A *class* declaration `class` C `|` att_1 `:` s_1 `, ... ,` att_n `:` s_n declares a class C with attributes att_1 to att_n of sorts s_1 to s_n. An *object* of class C is represented as a term $\langle O : C \mid att_1 : val_1, ..., att_n : val_n \rangle$ where O is the object's *identifier*, and where val_1 to val_n are the current values of the attributes att_1 to att_n. The global state has the form `{`t`}`, where t is a term of sort `Configuration` that has the structure of a *multiset* of objects and messages, with multiset union denoted by a juxtaposition operator that is declared associative and commutative. A *subclass* inherits all the attributes and rules of its superclasses.

The dynamic behavior of concurrent object systems is axiomatized by specifying each of its transition patterns by a rewrite rule. For example, the rule

```
rl [l] : m(O,w)  < O : C | a1 : x, a2 : O', a3 : z >   =>
                 < O : C | a1 : x + w, a2 : O', a3 : z >  dly(m'(O'),x) .
```

defines a parametrized family of transitions in which a message `m`, with parameters `O` and `w`, is read and consumed by an object `O` of class `C`. The transitions change the attribute `a1` of the object `O` and send a new message `m'(O')` *with delay* `x`. "Irrelevant" attributes (such as `a3`) need not be mentioned in a rule.

A Real-Time Maude specification is *executable*, and the tool provides a variety of formal analysis methods. The *rewrite* command `(trew` t `in time <=` τ `.)` simulates *one* behavior of the system within time τ, starting with a given initial state t. The *search* command

`(tsearch [`n`]` t `=>*` *pattern* `such that` *cond* `in time <=` τ `.)`

uses a breadth-first strategy to find n states reachable from the initial state t within time τ, which match a *pattern* and satisfy a *condition*. The Real-Time Maude's *LTL model checker* checks whether each behavior from an initial state, possibly up to a time bound, satisfies a linear temporal logic formula. *State propositions*, possibly parametrized, are operators of sort `Prop`. A temporal logic *formula* is constructed by state propositions and temporal logic operators such as `True`, `~` (negation), `/\`, `\/`, `->` (implication), `[]` ("always"), `<>` ("eventually"), `U` ("until"), and `O` ("next"). The command `(mc` t `|=u` φ `.)` then checks whether the temporal logic formula φ holds in all behaviors starting from the initial state t.

3 The Airplane Turning Control System

This section presents a simple model of an avionics control system to turn an aircraft. In general, the direction of an aircraft is maneuvered by two components in the wings: the ailerons and the rudder. As shown in Figure 2, an aileron is a flap attached to the end of the left or the right wing, and a rudder is a flap attached to the vertical tail (the aircraft figures in this section are borrowed from [7]).

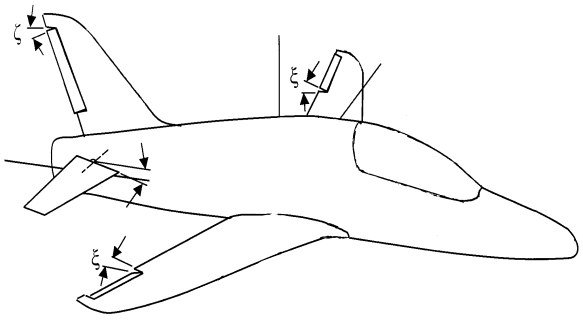


Figure 2: The ailerons and the rudder of an aircraft.

When an aircraft makes a turn, the aircraft rolls towards the desired direction of the turn, so that the lift force caused by the two wings acts as the centripetal force and the aircraft moves in a circular motion. The turning rate $d\psi$ can be given as a function of the aircraft's roll angle ϕ:

$$d\psi = (g/v) * \tan\phi \tag{1}$$

where ψ is the direction of the aircraft, g is the gravity constant, and v is the velocity of the aircraft [7]. The ailerons are used to control the rolling angle ϕ of the aircraft by generating different amounts of lift force in the left and the right wings. Figure 3 describes such a banked turn using the ailerons.

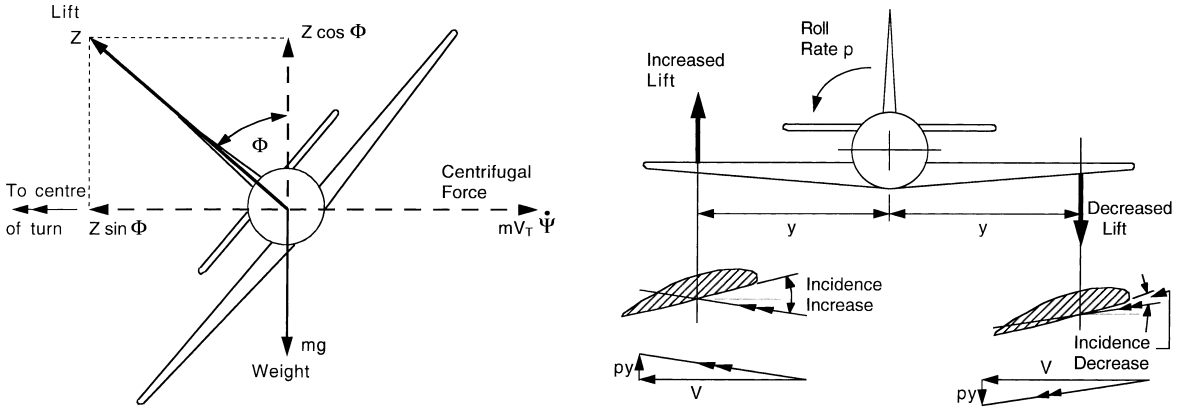


Figure 3: Forces acting in a turn of an aircraft with ϕ a roll angle and ρ a roll rate.

However, the rolling of the aircraft causes the difference in drag on the left and the right wings, which produces a yawing moment in the opposite direction to the roll, called adverse yaw. The adverse yaw makes the aircraft sideslip in a wrong direction with the amount of the yaw angle β, as described in Figure 4. This undesirable side effect is usually countered by using the aircraft's rudder, which generates the side lift force on the vertical tail that opposes the adverse yaw. To turn an aircraft safely and effectively, the roll angle ϕ of the aircraft should be increased for the desired direction while the yaw angle β stays at 0.

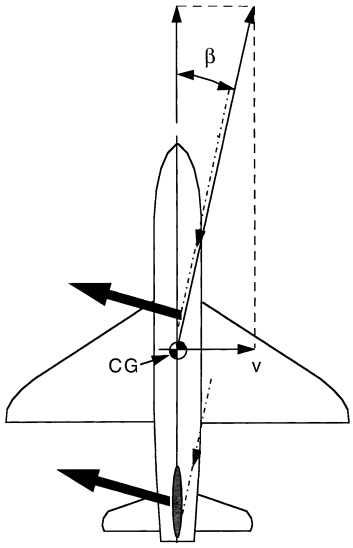


Figure 4: Adverse yaw.

Such a roll and yaw can be modeled by simple mathematical equations under some simplifying assumptions, including: (i) the aircraft's wings are flat with respect to the horizontal axis of the aircraft, (ii) the altitude of the aircraft does not change, which can be separately controlled by using the aircraft's elevator (a flap attached to the horizontal tail of the aircraft), (iii) the aircraft maintains a constant speed by separately controlling the thrust power of the aircraft, and (iv) there are no external influences such as wind or turbulence. Then, the roll angle ϕ and the yaw angle β can be modeled by the following equations [2]:

$$d\phi^2 = (\mathit{Lift\ Right} - \mathit{Lift\ Left}) \,/\, (\mathit{Weight} * \mathit{Length\ of\ Wing}) \tag{2}$$

$$d\beta^2 = \mathit{Drag\ Ratio} * (\mathit{Lift\ Right} - \mathit{Lift\ Left}) \,/\, (\mathit{Weight} * \mathit{Length\ of\ Wing}) + \mathit{Lift\ Vertical} \,/\, (\mathit{Weight} * \mathit{Length\ of\ Aircraft}) \tag{3}$$

The lift force from the left, the right, or the vertical wing is given by the following linear equation:

$$\mathit{Lift} = \mathit{Lift\ constant} * \mathit{Angle} \tag{4}$$

where, for *Lift Right* and *Lift Left*, *Angle* is the angle of the aileron, and for *Lift Vertical*, *Angle* is the angle of the rudder. The lift constant depends on the geometry of the corresponding wing, and the drag ratio is given by the size and the shape of the entire aircraft.

We model the airplane turning control system as a multirate ensemble with 4 typed machines: the main controller, the left wing controller, the right wing controller, and the rudder controller. The environment for the airplane turning control system is given by the pilot console, which is modeled as another typed machine and is connected to the main controller on the outside of the control system. Each sub-controller moves the surface of the wing towards the goal angle specified by the main controller, which sends the desired angles to the sub-controllers to make a coordinated turn whose goal direction is specified from the pilot console. The main controller also models position sensors that measure the roll, the yaw, and the direction by using the aeronautics equations above. In this case study, we assume that the main controller has period 60 ms, the left and the right wing controllers have period 15 ms (rate 4), and the rudder controller has period 20 ms (rate 3). Our model is a two-level hierarchical multirate ensemble, since the airplane turning control system itself forms a single typed machine with period 60 ms (rate 10) and then is connected to the pilot console with period 600 ms, as illustrated in Figure 5.

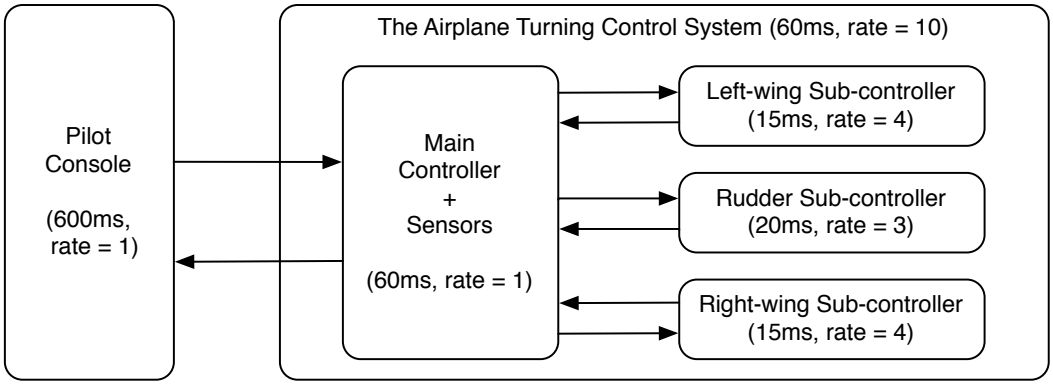


Figure 5: The architecture of our airplane turning control system.

The behavior of this system is defined in the following sections by the multi-rate ensemble PALS modeling framework in Real-Time Maude. That is, we will specify the multirate ensemble $\mathfrak{E}$ corresponding to the above airplane control system, which is much simpler than its asynchronous realization $\mathscr{MA}(\mathfrak{E},T,\Gamma)$. Then, we will exploit the bisimulation $\mathfrak{E} \simeq \mathscr{MA}(\mathfrak{E},T,\Gamma)$ (see [4]) to verify properties about $\mathscr{MA}(\mathfrak{E},T,\Gamma)$ by model checking them on the much simpler system $\mathfrak{E}$.

4 Multirate Synchronous Ensembles in Real-Time Maude

We have defined a framework for formally modeling and analyzing multirate ensembles in Real-Time Maude. Given a specification of *deterministic* single typed machines, their periods, input adaptors, a wiring diagram, and one top-level nondeterministic environment, our framework gives an executable Real-Time Maude model of the synchronous composition of the ensemble. It is natural to assume that controller components are deterministic; however, if we have a nondeterministic component, our framework could be modified so that local transitions are modeled by rewrite rules instead of equationally defined functions. If the system has "local" fast environments, they should be dealt with by the corresponding fast machines.[4]

This section gives a brief overview of this framework and of how the user should specify the ensemble; the entire definition of our framework is given in our longer report [3].

Representing Multirate Ensembles in Real-Time Maude. The multirate ensemble can be naturally specified in an object-oriented style, where its machines and the ensemble can be modeled as objects. A typed machine is represented as an object instance of a subclass of the base class `Component`, which has the common attributes `rate`, `period`, and `ports`.

```
class Component | rate : NzNat,   period : Time,   ports : Configuration .
```

The `period` denotes the period of the typed machine, and `rate` denotes its rate in a multirate ensemble. The `ports` attribute contains the input/output "ports" of a typed machine, represented as a multiset of `Port` objects, whose `content` attribute contains the data content as a list of values of the supersort `Data`:

```
class Port | content : List{Data} .   class InPort .   class OutPort .
subclass InPort OutPort < Port .
```

For each typed machine, the user must define an appropriate subclass of the class `Component`, the function `delta` defining its transition function, and the `adaptor` function for each input port:

```
op delta : Object -> Object .
op adaptor : ComponentId PortId NeList{Data} -> NeList{Data} .
```

where sort `NeList{Data}` denotes a non-empty list of data.

A multirate machine ensemble is modeled as an object instance of the class `Ensemble`. We support the definition of hierarchical ensembles by letting an ensemble be a `Component`, which also contains the wiring diagram (`connections`) and the `machines` in the ensemble. In this case, the `ports` attribute represents the environment ports of the ensemble:

```
class Ensemble | machines : Configuration,   connections : Set{Connection} .
subclass Ensemble < Component .
```

A wiring diagram is modeled as a set of connections. A connection from an output port P_1 of a component C_1 to an input port P_2 of a component C_2 is represented as a term $C_1 \,.\, P_1$ `-->` $C_2 \,.\, P_2$. Similarly, a connection between an environment port P_1 and a machine port P_2 of a subcomponent C_2 is represented as a term P `-->` $C_2 \,.\, P_2$ (for an environment input) or a term $C_2 \,.\, P_2$ `-->` P (for an environment output).

[4] An environment can be viewed as a nondeterministic typed machine [10]. Therefore, a faster machine's environment and the fast machine itself form a nondeterministic 2-machine ensemble.

Defining Synchronous Compositions of Multirate Ensembles. Given these definitions of an ensemble, our framework defines its synchronous composition as follows. The function `delta` of a multi-rate machine ensemble $\mathfrak{E}$ defines the transitions in the synchronous composition of $\mathfrak{E}$.

```
eq delta(< C : Ensemble | >)
 =
   transferResults(  execute(  transferInputs(< C : Ensemble | >))) .
```

In the above equation, the function `transferInputs` first transfers to each input port a value in the corresponding environment output port or the feedback output port. The `execute` function below applies the appropriate input adaptor to each sub-component, and then performs the function `delta` of each component as many times as its deceleration rate. Finally, the new outputs in sub-components are transferred to the environment ports by the function `transferResults`.

```
eq execute(< C : Ensemble | machines : COMPS >)
 = < C : Ensemble | machines : executeSub(COMPS) > .
eq executeSub(< C : Component | > COMPS)
 = k-delta(applyAdaptors(< C : Component | >)) executeSub(COMPS) .
eq executeSub(none) = none .
```

where `k-delta` applies `delta` as many times as the rate of a given typed machine:

```
eq k-delta(< C : Component | rate : N >) = k-delta(N, < C : Component | >) .
eq k-delta(s N, OBJECT) = k-delta(N, delta(OBJECT)) .
eq k-delta(  0, OBJECT) = OBJECT .
```

The rates and periods should be consistent in an ensemble; that is, if some component has rate k and period p, then any component with rate $k \cdot t$ in the same ensemble should have period p/t.

Formalizing the Synchronous Steps. When each typed machine is deterministic and the system contains one (nondeterministic) top-level environment, the dynamics of the entire system given by a multirate machine ensemble is specified by the following tick rewrite rule that simulates each synchronous step of the composed system:

```
crl [step] : {< C : Component | period : T >}
           =>
             {delta(envOutput(ENVOUTPUT, clearOutputs(< C : Component | >)))}
            in time T
 if possibleEnvOutput => ENVOUTPUT .
```

In the condition of the rule, *any* possible environment output can be nondeterministically assigned to the variable `ENVOUTPUT`, since the constant `possibleEnvOutput` can be rewritten to any possible environment output by user-defined rewriting rules of the form

```
rl possibleEnvOutput => (PortId1 = Value1, ..., PortIdn = Valuen) .
```

These values are inserted into the appropriate component input port by the `envOutput` function, after clearing the outputs generated in the previous round by the `clearOutputs` function.

```
eq envOutput((P = DL, ENVASSIGNS),  < C : Component | ports : < P : InPort | > PORTS >)
 = envOutput(ENVASSIGNS,  < C : Component | ports : < P : InPort | content : DL > PORTS >) .
eq envOutput(empty, < C : Component | >) = < C : Component | > .
```

The `delta` function is finally applied to perform the transition of the component. Since an ensemble is an instance of the `Component` class and the `delta` function is also given for the `Ensemble` class, this tick rewrite rule can also be applied to a hierarchical multirate ensemble. After each synchronous step, the input ports of the component contain the environment input given by the `possibleEnvOutput`, and the output ports will have the resulting environment output generated by the `delta` function. Note that the `period` does not have any effect on the structure of the transition system, but can be useful to verify *timed* properties, such as time-bounded LTL properties and metric CTL properties [8].

5 Modeling the Airplane Turning Control System

In this section we formally specify the airplane turning control system in Section 3 using the ensemble framework in Real-Time Maude described in Section 4. The entire specification is available in our report [3]. We consider the following parameters chosen to be representative of a small general aviation aircraft. The speed of the aircraft is assumed to be $50\,m/s$, and the gravity constants is $g = 9.80555\,m/s^2$.

```
eq planeSize     = 4.0 .    eq weight        = 1000.0 .     eq wingSize  = 2.0 .
eq virtLiftConst = 0.6 .    eq horzLiftConst = 0.4 .        eq dragRatio = 0.05 .
```

Subcontroller. The subcontrollers for the ailerons and the rudder are modeled as object instances of the following class `SubController`:

```
class SubController | curr-angle : Float,   goal-angle : Float,   diff-angle : Float .
subclass SubController < Component .
```

A subcontroller increases/decreases the `curr-angle` toward the `goal-angle` in each round, but the difference in a single (fast) round should be less than or equal to the maximal angle `diff-angle`. The transition function `delta` of a subcontroller is then defined by the following equation:

```
ceq delta(< C : SubController | ports : < input : InPort | content : D LI >
                                        < output : OutPort | content : LO >,
                                curr-angle : CA, goal-angle : GA, diff-angle : DA >)
  =
      < C : SubController | ports : < input : InPort | content : LI >
                                    < output : OutPort | content : LO d(CA') >,
                            curr-angle : CA', goal-angle : GA' >
  if CA' := angle(moveAngle(CA,GA,DA))
  /\ GA' := angle(if D == bot then GA else float(D) fi) .
```

where `moveAngle(CA,GA,DA)` equals the angle that is increased or decreased from the current angle `CA` to the goal angle `GA` up to the maximum angle difference `DA`:

```
eq moveAngle(CA,GA,DA) = if abs(GA - CA) > DA then CA + DA * sign(GA - CA) else GA fi .
```

The `angle` function keeps the angle value between -180° and 180°. The `delta` function updates the goal angle to the input from the main controller, and keeps the previous goal if it receives `bot` (i.e., $\perp$).

Main Controller. The main controller for the aircraft turning control system is represented as the object instances of the following class `MainController`:

```
class MainController | velocity : Float,   goal-dir : Float,
                       curr-yaw : Float,   curr-rol : Float,    curr-dir : Float .
subclass MainController < Component .
```

The velocity attribute denotes the speed of the aircraft that is assumed to be constant in this specification. The curr-yaw, curr-roll, and curr-dir indicate the current position status of the yaw angle β, the roll angle ϕ, and the direction Ψ, respectively, which model the position sensors of the aircraft. The goal-dir keeps the goal direction given by the pilot's input from the environment.

For each round of the main controller, the three attributes curr-yaw, curr-roll, and curr-dir are updated[5] using the angles of the wings in the input ports that are transferred from the subcontrollers. The goal-dir is also updated if a new goal direction arrives to the input ports. Based on the new current position status and the goal direction, the new angles of the wings are evaluated and sent back to the subcontrollers. The transition function delta of the main controller is then defined as follows:

```
ceq delta(< C : MainController |
             ports : < input : InPort | content : IN PI >    < output : OutPort | content : PO >
                     < inLW : InPort | content : d(LA) LI >   < outLW : OutPort | content : LO >
                     < inRW : InPort | content : d(RA) RI >   < outRW : OutPort | content : RO >
                     < inTW : InPort | content : d(TA) TI >   < outTW : OutPort | content : TO >,
             velocity : VEL, period : T,
             curr-yaw : CY, curr-rol : CR,
             curr-dir : CD, goal-dir : GD >)
  =
      < C : MainController |
          ports : < input : InPort | content : PI >  < output : OutPort | content : PO OUT >
                  < inLW : InPort | content : LI >   < outLW : OutPort | content : LO d(- RA') >
                  < inRW : InPort | content : RI >   < outRW : OutPort | content : RO d(RA') >
                  < inTW : InPort | content : TI >   < outTW : OutPort | content : TO d(TA') >,
          curr-yaw : CY', curr-rol : CR',
          curr-dir : CD', goal-dir : GD' >
    if CY' := angle( CY + dBeta(LA,RA,TA) * float(T) )
    /\ CR' := angle( CR + dPhi(LA,RA) * float(T) )
    /\ CD' := angle( CD + dPsi(CR,VEL) * float(T) )
    /\ GD' := angle( if IN == bot then GD else GD + float(IN) fi )
    /\ RA' := angle( horizWingAngle(CR', goalRollAngle(CR', CD', GD')) )
    /\ TA' := angle( tailWingAngle(CY') )
    /\ OUT := dir: CD' roll: CR' yaw: CY' goal: GD' .
```

The first four lines in the condition compute new values for curr-yaw, curr-roll, curr-dir, and goal-dir, based on values in the input ports. A non-$\perp$ value in the port input is added to goal-dir. The variables RA' and TA' denote new angles of the ailerons and the rudder, computed by the control functions explained below. Such new angles are queued in the corresponding output ports, and will be transferred to the related subcontrollers at the next synchronous step since they are feedback outputs. The last line in the condition gives the output for the current step, the new position information of the aircraft, which will be transferred to its container ensemble at the end of the current synchronous step.

The new angles of the ailerons and the rudder are computed by the *control functions*. The function horizWingAngle evaluates the new angles for the aileron in the right wing, based on the current roll angle and the goal roll angle. The angle of the aileron in the left wing is always exactly opposite to the one of the right wing. The function goalRollAngle computes the desired roll angle ϕ to make a turn, based on the current roll angle and the difference between the goal direction and the current direction. Finally, in order to achieve a coordinated turn where the yaw angle is always 0, the function tailWingAngle calculates the new rudder angle based on the current yaw angle. We define all three control functions by simple linear equations as follows, where CR is a current roll angle and CY is a current yaw angle:

[5] We currently use the simple Euler's method to compute such position values given by the differential aeronautical equations (1-3), but more precise methods can be easily applied.

```
eq goalRollAngle(CR,CD,GD) = sign(angle(GD - CD)) * min(abs(angle(GD - CD)) * 0.3, 20.0) .
eq horizWingAngle(CR,GR)   = sign(angle(GR - CR)) * min(abs(angle(GR - CR)) * 0.3, 45.0) .
eq tailWingAngle(CY)       = sign(angle(- CY)) * min(abs(angle(- CY)) * 0.8, 30.0) .
```

That is, the goal roll angle is proportional to the difference GD − CD between the goal and current directions with the maximum 20°. The horizontal wing (aileron) angles are also proportional to the difference GR − CR between the goal and current roll angles with the maximum 45°. Similarly, the rudder angle is proportional to the difference − CY between the goal and current yaw angles with the maximum 30°, where the goal yaw angle is always 0°.

Pilot Console. The pilot console, the environment for the aircraft turning control system, is represented as the object instances of the following class PilotConsole:

```
class PilotConsole | scenario : List{Data} .
subclass PilotConsole < Component .
```

The attribute scenario contains a list of goal angles that are transmitted to the main controller. The transition function delta of the pilot console keeps sending goal angles in the scenario to its output port until no more data remains in the scenario. In addition, the pilot console has an extra input port to receive an outer environment input to generate *nondeterministic* goal directions. A non-⊥ value in the input port is added to the output given by the scenario.

```
ceq delta(< C : PilotConsole | ports : < input : InPort | content : IN LI >
                                       < output : OutPort | content : LO >,
                               scenario : d(F) SL >)
  =
    < C : PilotConsole | ports : < input : InPort | content : LI >
                                 < output : OutPort | content : LO OUT >,
                         scenario : SL > .
 if OUT := d(if IN == bot then F else angle(F + float(IN)) fi) .

eq delta(< C : PilotConsole | ports : < input : InPort | content : IN LI >
                                      < output : OutPort | content : LO >,
                              scenario : nil >)
 =
   < C : PilotConsole | ports : < input : InPort | content : LI >
                                < output : OutPort | content : LO bot >  > .
```

Note that if the scenario is empty, i.e., nil, the outer environment input is ignored, and the pilot console will keep sending ⊥ to its output port.

Airplane System. The entire architecture of the airplane turning control system in Figure 5, including the environment (i.e., the pilot console), is then represented as an ensemble object with subcomponents as follows (some parts of the specification are replaced by '. . .'):

```
< airplane : Ensemble |
   rate : 1, period : 600,
   ports :  < input : InPort | content : nil >  < output : OutPort | content : nil >,
   connections :
      input --> pilot . input ;
      pilot . output --> csystem . input ;  csystem . output --> output,
   machines :
      (< pilot : PilotConsole | rate : 1, period : 600,... >
```

```
< csystem : Ensemble |
   rate :10, period : 60,
   ports : < input : InPort | content : nil >
           < output : OutPort | content : nil >,
   connections :
      input --> main . input ;  main . output --> output ;
      left . output --> main . inLW ;  main . outLW --> left . input ; ...,
   machines : (< main : MainController | rate : 1, period : 60,... >
               < left : SubController | rate : 4, period : 15,... >
               < right : SubController | rate : 4, period : 15,... >
               < rudder : SubController | rate : 3, period : 20,... >) >) >
```

The top ensemble `airplane` includes the pilot console `pilot` and the ensemble `csystem` for the airplane turning control system. The `airplane` has one input port for the pilot console to generate nondeterministic goals, and one output port to display the result.

In the ensemble `csystem`, the input adaptors for the subcontrollers generate a vector with extra $\perp$'s and the adaptor for the main controller selects the last value of the input vector.

```
eq adaptor(left, input, D) = D bots(3) .   eq adaptor(rudder, input, D) = D bots(2) .
eq adaptor(right, input, D) = D bots(3) .  eq adaptor(main, P, LI D) = D .
```

The function `bots(n)` generates n $\perp$-constants. Similarly, in the top ensemble `airplane`, the input adaptor for the ensemble `csystem` generates a vector with extra $\perp$'s, and the adaptor for the pilot console selects the last value of the input vector.

```
eq adaptor(csystem, input, D) = D bots(9)
eq adaptor(pilot, input, LI D) = D .
```

6 Formal Analysis of the Airplane Turning Control System

This section explains how we have formally analyzed the above Real-Time Maude model of the airplane turning control system as a multi-rate synchronous design, and how the ensemble design itself has been improved as a result of the analysis. Multirate PALS then ensures that the corresponding properties also hold in a distributed multi-rate realization of the design. There are two important requirements that the airplane turning control system should satisfy:

- *Liveness*: the airplane should reach the goal direction within a reasonable time with a stable status in which both the roll angle and the yaw angle are close to 0.
- *Safety*: during a turn, the yaw angle should always be close to 0.

We first analyze deterministic behaviors when the airplane turns $+60^\circ$ to the right.[6] In this case, there are no outer environment inputs to the pilot console, so that the pilot console sends the goal angles in the scenario to the main controller in turn. We consider the following three scenarios:

1. The pilot gradually increases the goal direction in 6 seconds, $+10^\circ$ for each second.
2. The pilot sets the goal direction to $+60^\circ$ at the first step.
3. The goal direction is at first -30°, and then it is suddenly set to $+60^\circ$ in one second.

Figure 6 shows the simulation results for these three scenarios up to 6 seconds, obtained by using the Real-Time Maude simulation command `(trew {model(`*Scenario*`)} in time <= 6000 .)`, where

[6] In our model, a turn of positive degrees is a right turn, and one of negative degrees a left turn.

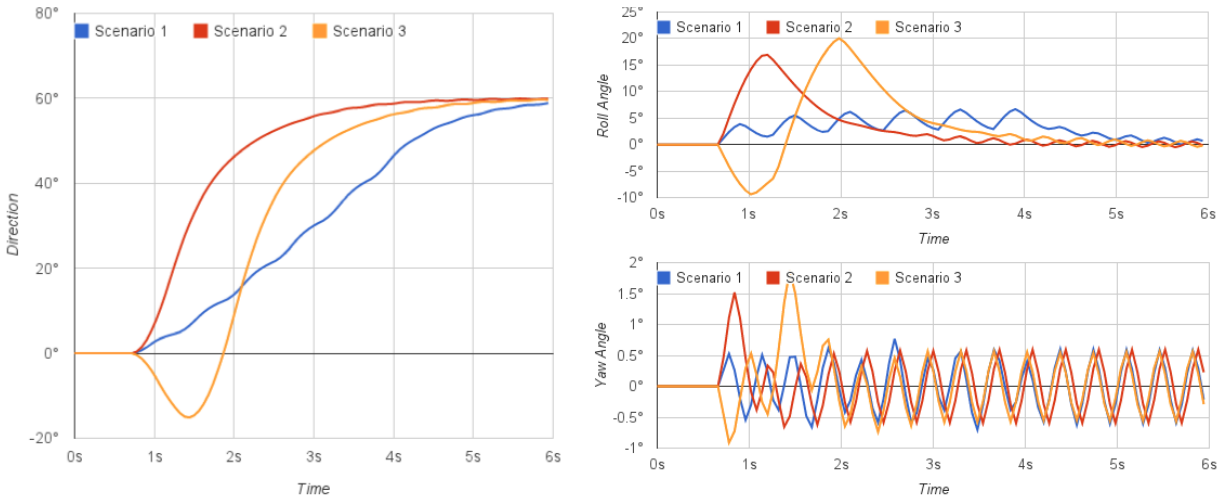


Figure 6: Simulation results for turning scenarios: the directions of aircraft (left), the roll angles (top right), and the yaw angles (bottom right)

`model(`*Scenario*`)` gives the initial state of the system with the scenario *Scenario* for the pilot console. For example, the scenario 3 is represented as the term "`d(-30.0) d(90.0)`" for a list of directions. As we can see in the graph, the airplane reaches the goal direction 60° in a fairly short time, and the roll angle also goes to a stable status. However, the yaw angle seems to be quite unstable.

There are basically two reasons why the yaw angle is not sufficiently close to 0 during a turn. First, since all control functions are linear, the new angles for the ailerons and the rudder are not small enough when the yaw angle is near 0. Second, the roll angle is sometimes changing too fast, so that the rudder cannot effectively counter the adverse yaw. Therefore, we modify the control functions as follows:

```
ceq horizWingAngle(CR, GR)
  = sign(FR) * (if abs(FR) > 1.0 then min(abs(FR) * 0.3, 45.0) else FR ^ 2.0 * 0.3 fi)
 if FR := angle(GR - CR) .
ceq tailWingAngle(CY)
  = sign(FY) * (if abs(FY) > 1.0 then min(abs(FY) * 0.8, 30.0) else FY ^ 2.0 * 0.8 fi)
 if FY := angle(- CY) .
ceq goalRollAngle(CR, CD, GD)
  = if abs(FD * 0.32 - CR) > 1.5 then CR + sign(FD * 0.32 - CR) * 1.5 else FD * 0.32 fi
 if FD := angle(GD - CD) .
```

When the difference between the goal and the current angles (i.e., `FR` or `FY`) is less than or equal to 1, the functions `horizWingAngle` and `tailWingAngle` are now *not* proportional to the difference, but proportional to the *square* of it. Further, the goal roll angle can be changed at most 1.5° at a time, so that there is no more abrupt rolling.

In order to check if the new control functions are safe, we define some auxiliary functions. The function *PortId* `?=` *Component* returns the content of the corresponding port in the *Component*:

```
eq P ?= < C : Component | ports : < P : Port | content : DL > PORTS >  =  DL .
```

and the function `safeYawAll(`*OutputDataList*`)` checks whether every output data in the given list has a safe yaw angle, namely, an angle less than 1.0°.

```
eq safeYawAll(DO LI) = abs(yaw(DO)) < 1.0 and safeYawAll(LI) .
eq safeYawAll(nil)   = true .
```

Then, we can verify, using the Real-Time Maude search command, that there exists no dangerous yaw angle within sufficient time bound; e.g., for the scenario 3:

```
Maude> (tsearch [1] {model(d(-30.0) d(90.0))} =>* {SYSTEM}
        such that not safeYawAll(output ?= SYSTEM) in time <= 27000 .)

No solution
```

Although each state of the transition system captures only the slow steps for the top ensemble (i.e., every 600ms), `safeYawAll` also checks all fast steps for the main controller (every 60ms), since it accesses the *history* of the main controller's status in the output port of the top ensemble, which the main controller sends to the top ensemble for each fast step of it.

Moreover, we can use Real-Time Maude's LTL model checking to verify liveness as well as safety at the same time. This combined property is that the airplane reaches the desired direction with a stable status while keeping the yaw angle close to 0, which is formalized as the LTL formula:

$$\Box(\neg\texttt{stable} \rightarrow (\texttt{safeYaw}\ \mathbf{U}\ (\texttt{reach} \wedge \texttt{stable})))$$

where the atomic propositions `safeYaw`, `stable`, and `reach` are defined as follows:

```
 eq {SYSTEM} |= safeYaw = safeYawAll(output ?= SYSTEM) .
 eq {SYSTEM} |= stable  = stableAll(output ?= SYSTEM) .
ceq {SYSTEM} |= reach   = abs(angle(goal(DO) - dir(DO))) < 0.5
 if DO := last(output ?= SYSTEM) .
```

and the function `stableAll(`*OutputDataList*`)` returns `true` only if both the yaw angle and the roll angle are less than 0.5° for every output data in the *OutputDataList*, defined in a similar way as the `safeYawAll`. We have verified that all three scenarios now satisfy the above LTL property with the new control functions, using the time-bounded LTL model checking command of Real-Time Maude:

```
Maude> (mc {model(d(-30.0) d(90.0))} |=t [] (~stable -> (safeYaw U reach /\ stable))
           in time <= 7200 .)

Result Bool : true
```

Finally, we have verified nondeterministic behaviors in which the pilot sends one of the turning angles -60.0°, -10.0°, 0°, 10°, and 60.0° to the main controller for 6 seconds. Such nondeterministic behaviors can be defined by adding the following five rewrite rules, which nondeterministically assign one of these values to the input port of the pilot console:

```
rl possibleEnvOutput => input = d(0.0) .
rl possibleEnvOutput => input = d(10.0) .   rl possibleEnvOutput => input = d(-10.0) .
rl possibleEnvOutput => input = d(60.0) .   rl possibleEnvOutput => input = d(-60.0) .
```

The following model checking command then shows that our redesigned system, with the new control functions, satisfies the same LTL property within 18 seconds, where one of the above five angles is nondeterministically chosen and added to the angle 0° in the scenario for each step of the pilot console:

```
Maude> (mc {model(d(0.0) d(0.0) d(0.0) d(0.0) d(0.0) d(0.0))}
             |=t
           [] (~ stable -> (safeYaw U (reach /\ stable))) in time <= 18000 .)

Result Bool : true
```

The number of states explored in this model checking analysis is 246,785,[7] which is a huge state space reduction compared to the distributed asynchronous model since: (i) asynchronous behaviors are eliminated thanks to PALS, and (ii) any intermediate fast steps for the sub-components are merged into a single-step of the system's top-level ensemble.

7 Conclusions

The present work can be seen from different perspectives. First, from the perspective of research on the PALS methodology, its main contribution is to demonstrate that Multirate PALS, when used in combination with a tool like Real-Time Maude, can be effectively applied to the formal verification of nontrivial DCPS designs, and even to the process of refining a DCPS design before it is verified. Second, from the perspective of the formal specification and verification of distributed hybrid systems, it also shows that Real-Time Maude is an effective tool for specifying and verifying such systems.

Much work remains ahead. On the one hand, more case studies like this one should be developed. On the other hand, our work in [5], which applies PALS to a synchronous fragment of the AADL CPS modeling language in the single rate case, should be extended to the Multirate PALS case. Finally, the hybrid system applications of Real-Time Maude should be further developed independently of PALS. Several such applications have been developed in the past; but many more are possible, and a richer experience will be gained.

Acknowledgments. We thank the anonymous reviewers for their very useful comments that helped us improve our paper. This work was partially supported by Boeing Corporation Grant C8088 and NSF Grant CCF 09-05584.

References

[1] A. Al-Nayeem, L. Sha, D. D. Cofer & S. M. Miller (2012): *Pattern-Based Composition and Analysis of Virtually Synchronized Real-Time Distributed Systems*. In: *Proc. ICCPS'12*, IEEE, pp. 65–74, doi:10.1109/ICCPS.2012.15.

[2] J.D. Anderson (2005): *Introduction to flight*. McGraw-Hill.

[3] K. Bae, J. Krisiloff, J. Meseguer & P. C. Ölveczky (2012): *PALS-Based Analysis of an Airplane Multirate Control System in Real-Time Maude (Extended Version)*. Manuscript available at `http://csaba.olveczky.se/ftscs12-techrep.pdf`.

[4] K. Bae, J. Meseguer & P. C. Ölveczky (2012): *Formal Patterns for Multi-Rate Distributed Real-Time Systems*. In: *Proc. FACS'12*, LNCS, Springer. To appear.

[5] K. Bae, P. C. Ölveczky, A. Al-Nayeem & J. Meseguer (2011): *Synchronous AADL and its Formal Analysis in Real-Time Maude*. In: *Proc. ICFEM'11*, *LNCS* 6991, Springer, pp. 651–667, doi:10.1007/978-3-642-24559-6_43.

[6] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer & C. Talcott (2007): *All About Maude – A High-Performance Logical Framework*. *LNCS* 4350, Springer, doi:10.1007/978-3-540-71999-1.

[7] R. P. G. Collinson (1996): *Introduction to avionics*. Chapman & Hall.

[8] D. Lepri, E. Ábrahám & P.C. Olveczky (2012): *Timed CTL Model Checking in Real-Time Maude*. In: *Proc. WRLA'12*, *LNCS* 7571, Springer, pp. 182–200, doi:10.1007/978-3-642-34005-5_10.

[7] This analysis took almost 9 hours on Intel Core i5 2.4 GHz with 4 GB memory.

[9] J. Meseguer & P. C. Ölveczky (2010): *Formalization and Correctness of the PALS Architectural Pattern for Distributed Real-Time Systems*. In: *Proc. ICFEM'10*, *LNCS* 6447, Springer, pp. 303–320, doi:10.1007/978-3-642-16901-4_21.

[10] J. Meseguer & P. C. Ölveczky (2012): *Formalization and correctness of the PALS architectural pattern for distributed real-time systems*. *Theor. Comp. Sci.* 451, pp. 1–37, doi:10.1016/j.tcs.2012.05.040.

[11] S. P. Miller, D. D. Cofer, L. Sha, J. Meseguer & A. Al-Nayeem (2009): *Implementing Logical Synchrony in Integrated Modular Avionics*. In: *Proc. DASC'09*, IEEE, doi:10.1109/DASC.2009.5347579.

[12] P. C. Ölveczky & J. Meseguer (2007): *Semantics and Pragmatics of Real-Time Maude*. *Higher-Order and Symbolic Computation* 20(1-2), pp. 161–196, doi:10.1007/s10990-007-9001-5.

[13] W. Steiner & J. Rushby (2011): *TTA and PALS: Formally verified design patterns for distributed cyber-physical systems*. In: *Proc. DASC'11*, IEEE, doi:10.1109/DASC.2011.6096120.

A Formal Model For Real-Time Parallel Computation

Peter Hui Satish Chikkagoudar
Pacific Northwest National Laboratory
Washington, USA
peter.hui@pnnl.gov satish.chikkagoudar@pnnl.gov

The imposition of real-time constraints on a parallel computing environment— specifically high-performance, cluster-computing systems— introduces a variety of challenges with respect to the formal verification of the system's timing properties. In this paper, we briefly motivate the need for such a system, and we introduce an automaton-based method for performing such formal verification. We define the concept of a consistent parallel timing system: a hybrid system consisting of a set of timed automata (specifically, timed Büchi automata as well as a timed variant of standard finite automata), intended to model the timing properties of a well-behaved real-time parallel system. Finally, we give a brief case study to demonstrate the concepts in the paper: a parallel matrix multiplication kernel which operates within provable upper time bounds. We give the algorithm used, a corresponding consistent parallel timing system, and empirical results showing that the system operates under the specified timing constraints.

1 Introduction

Real-time computing has traditionally been considered largely in the context of single-processor and embedded systems, and indeed, the terms real-time computing, embedded systems, and control systems are often mentioned in closely related contexts. However, real-time computing in the context of multinode systems, specifically high-performance, cluster-computing systems, remains relatively unexplored. It can be argued that one reason for the relative dearth of work in this area is the lack of scenarios to date which would require such a system. Previously [12, 13], we have motivated the emerging need for such an infrastructure, giving a specific scenario related to the next generation North American electrical grid. In that work, we described the changes and challenges in the power grid driving the need for much higher levels of *computational resources* for power grid operations. To briefly summarize (and to provide some motivational context for the current work), many of these computations— particularly floating-point intensive simulations and optimization calculations ([2, 3, 8, 10, 11])—can be more effectively done in a centralized manner, and the amount and scale of such data is estimated by some [12] to be on the order of terabytes per day of streaming sensor data (e.g. Phasor Measurement Units (PMUs)), with the need to analyze the data within a strict cyclical window (every 30ms) presumably with the aid of high-performance, parallel computing infrastructures. With this in mind, the current work is part of a larger research effort at Pacific Northwest National Laboratory aimed at developing the necessary infrastructure to support an HPC cluster environment capable of processing vast amounts of streaming sensor data under hard real-time constraints.

While verifying the temporal properties of a more traditional (e.g. embedded) real-time system poses complex questions in its own right, imposing real-time constraints on a parallel (cluster) computing environment introduces an entirely new set of challenges not seen in these more traditional environments. For example, in addition to standard real-time concepts such as *worst-case execution time* (WCET), real-time parallel computation introduces the necessity of considering *worst case transmission time* when

communicating over the network between nodes, as well as the need to ensure that timing properties of one process do not invalidate those of the entire parallel process as a whole.

These are but two examples of the many questions which must be addressed in a real-time parallel computing system; certainly there are many more questions than can be addressed in a single paper. To this end, we introduce a simple, event driven, automata-based model of computation intended to model the temporal properties of a specific class of parallel programs. Namely, we consider SPMD (Single Program, Multiple Data), parent-child type programs, in part because in practice, many parallel programs— including many prototypical MPI-based [14] programs— fall into this category. We give an example of such a program in Section 3. This model is typified by the existence of a cyclic *master* or *parent* process, and a set of noncyclic *child* or *slave* processes amongst which work is divided. With this characterization, a very natural correspondence emerges between the processes and the automata which model them: the cyclic *parent* process is very naturally modeled by an ω-automaton, and the *child* processes by a standard finite automaton. Our main contribution of this paper, then, is twofold: first, a formal method of modeling the respective processes in this manner, combining these into a single hybrid system of parallel automata, and secondly, a simple case study demonstrating a practical application of this system. We should note that the notion of parallel finite automata is not a new one; variants have been studied before (e.g. [6, 15]). We take the novel approach of combining *timed* variants ([1, 5]) of finite automata into a single hybrid model which captures the timing properties of the various component processes of a parallel system.

The rest of the paper proceeds as follows: Section 2 defines the automaton models used by our system: Timed Finite Automata in Section 2.1, Timed Büchi Automata in Section 2.2, and a hybrid system combining these two models in Section 2.3. Section 3 gives a case study in the form of an example real-time matrix multiplication kernel, running on a small, four-node real-time parallel cluster. Section 4 concludes.

2 Formalisms

In this section, we give formal definitions for the machinery used in our hybrid system of automata. The definitions given in Sections 2.1 and 2.2 are not new [1]. However, it is still important that we state their definitions here, as they are used later on, in Section 2.3.

2.1 Timed Finite Automata

In this section, we define a simple timed extension of traditional finite state automata and the words they accept. We will use these in later sections to model the timing properties of child processes in a real-time cluster system.

Timed strings take the form $(\bar{\sigma}, \bar{\tau})$, where $\bar{\sigma}$ is a string of symbols, and $\bar{\tau}$ is a monotonically increasing sequence of reals (timestamps). τ_x denotes the timestamp at which symbol σ_x occurs. We also use the notation (σ_x, τ_x) to denote a particular symbol/timestamp pair. For instance, the timed string $((abc), (1, 10, 11))$ is equivalent to the sequence $(a, 1)(b, 10)(c, 11)$, and both represent the case where 'a' occurs at time 1, 'b' at time 10, and 'c' at time 11.

Correspondingly, we extend traditional finite automata to include a set of *timers*, which impose temporal restrictions along state transitions. A timer can be *initialized* along a transition, setting its value to 0 when the transition is taken, and it can be *used* along a transition, indicating that the transition can only be taken if the value of the timer satisfies the specified constraint. Formally, we associate with each

automaton a set of timer variables $\bar{T}$, and following the nomenclature of [1], an *interpretation* ν for this set of timers is an assignment of a real value to each of the timers in $\bar{T}$. We write $\nu[T \mapsto 0]$ to denote the interpretation ν with the value of timer T reset to 0. Clock constraints consist of conjunctions of upper bounds:

Definition 1. *For a set $\bar{T}$ of clock variables, the set $X(\bar{T})$ of clock constraints χ is defined inductively as*

$$\chi := (T < c)? \mid \chi_1 \wedge \chi_2$$

where T is a clock in $\bar{T}$ and c is a constant in $\mathbb{R}^+$.

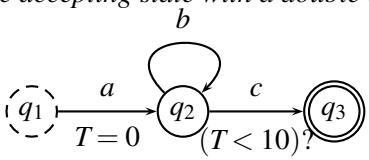
While this definition may seem overly restrictive compared to some other treatments (e.g. [1]), we believe it to be acceptable in this early work for a couple of reasons. First, while simple, this sole syntactic form remains expressive enough to capture an interesting, non-trivial set of use cases (e.g. Section 3). Secondly, the timing analysis in subsequent sections of the paper becomes rather complex, even when timers are limited to this single form. Restricting the syntax in this manner simplifies this analysis to a more manageable level. We leave more complex formulations and the corresponding analysis for future work.

Definition 2 (Timed Finite Automaton (TFA))**.** *A Timed Finite Automaton (TFA) is a tuple $\langle \Sigma, Q, s, q_f, \bar{T}, \delta, \gamma, \eta \rangle$, where*

- Σ *is a finite alphabet,*
- Q *is a finite set of states,*
- $s \in Q$ *is the start state,*
- $q_f \in Q$ *is the accepting state,*
- $\bar{T}$ *is a set of clocks,*
- $\delta \subseteq Q \times Q \times \Sigma$ *is the state transition relation,*
- $\gamma \subseteq \delta \times 2^{\bar{T}}$ *is the clock initialization relation, and*
- $\eta \subseteq \delta \times X(\bar{T})$ *is the constraint relation.*

A tuple $\langle q_i, q_j, \sigma \rangle \in \delta$ indicates that a symbol σ yields a transition from state q_i to state q_j, subject to the restrictions specified by the timer constraints in η. A tuple $\langle q_i, q_j, \sigma, \{T_1, ..., T_n\} \rangle \in \gamma$ indicates that on the transition on symbol σ from q_i to q_j, all of the specified timers are to be initialized to 0. Finally, a tuple $\langle q_i, q_j, \sigma, X(\bar{T}) \rangle \in \eta$ indicates that the transition on σ from q_i to q_j can only be taken if the constraint $X(\bar{T})$ evaluates to true under the current timer interpretation.

Example 3. *The following TFA accepts the timed language $\{(ab^*c, \tau_1 ... \tau_n) \mid \tau_n - \tau_1 < 10\}$ (i.e., the set of all strings consisting of an 'a', followed by an arbitrary number of 'b's, followed by a 'c', such that the elapsed time between the first and last symbols is no greater than 10 time units). The start state is denoted with a dashed circle, and the accepting state with a double line.*



Paths and runs are defined in the standard way:

Definition 4 (Path)**.** *Let* **A** *be a TFA with state set Q and transition relation δ. Then $(q_1, ..., q_n)$ is a* path *over* **A** *if, for all $1 \leq i < n$, $\exists \sigma . \langle q_i, q_{i+1}, \sigma \rangle \in \delta$.*

Definition 5 (Run)**.** *A run r of a TFA* $\langle \Sigma, Q, q_0, q_f, \bar{T}, \delta, \gamma, \eta \rangle$ *over a timed word* $(\bar{\sigma}, \bar{\tau})$*, is a sequence of the form*

$$r : (q_0, \nu_0) \xrightarrow[\tau_1]{\sigma_1} (q_1, \nu_1) \xrightarrow[\tau_2]{\sigma_2} (q_2, \nu_2) \xrightarrow[\tau_3]{\sigma_3} \ldots \xrightarrow[\tau_n]{\sigma_n} (q_n, \nu_n)$$

satisfying the following requirements:

- *Initialization:* $\nu_0(k) = 0, \forall k \in \bar{T}$
- *Consecution: For all* $i \geq 0$*:*
  - $\delta \ni \langle q_i, q_{i+1}, \sigma_i \rangle$,
  - $(\nu_{i-1} + \tau_i - \tau_{i-1})$ *satisfies* χ_i*, where* $\eta \ni \langle q_{i-1}, q_i, \sigma_i, \chi_i \rangle$*, and*
  - $\nu_i = (\nu_{i-1} + \tau_i - \tau_{i-1})[T \mapsto 0]$, $\forall T \in \bar{T}$*, where* $\gamma \ni \langle q_i, q_j, \sigma_i, \bar{T} \rangle$

r is an accepting *run if* $q_n = q_f$.

A TFA **A** *accepts* a timed string $s = (\bar{\sigma}, \tau_1 \ldots \tau_n)$ if there is an accepting run of s over **A**, and $\tau_n - \tau_1$ is called the *duration* of the string.

Note (Well-Formedness)**.** *We introduce a restriction on how timers can be used in a TFA, thus defining what it means for a TFA to be* well-formed*. Namely, we restrict timers to be used only once along a path; this is to simplify somewhat the timing analysis in subsequent sections. In particular, we say that a TFA* **A** *is well-formed if, for all pairs of states* (q_x, q_y)*, all timers T, and all paths from* q_x *to* q_y*, T is used no more than once. For example, the TFAs shown in Figure 1 are not well-formed, since in both cases, timers can potentially be used more than once— in the first case (*$\mathbf{A}_1$*), along the self-loop on* q_2*, and in the second case (*$\mathbf{A}_2$*), along two separate transitions along the path. At first, this may appear to be overly restrictive, but as it turns out, many of these cases can easily be rewritten equivalently to conform to the single-use restriction, as shown in Figure 2.*

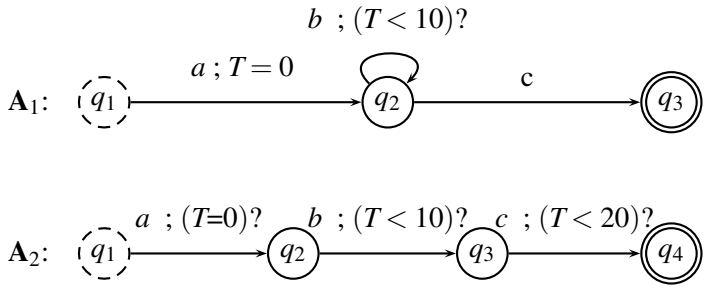


Figure 1: Malformed TFAs. Start states are denoted with a dashed circle, and accepting states with a double line. The intent of $\mathbf{A}_1$ is to allow strings of the form a, followed by arbitrarily many bs, as long as they all occur less than 10 units after the a, followed by a c. The intent of $\mathbf{A}_2$ is to allows strings of the form abc, where the elapsed time between the a and b is less than 10, and that between the a and c is less than 20. Both of these can be rewritten using conforming automata, as shown in Figure 2.

2.1.1 Bounding Maximum Delay

An important notion throughout the remainder of the paper is that of computing bounds on the allowable delays along all possible paths through a TFA. Specifically, we are interested in doing so to be able to reason formally about the maximum execution time for a child process, with the end goal of being able to bound the execution time of the system— parent and all child processes— as a whole.

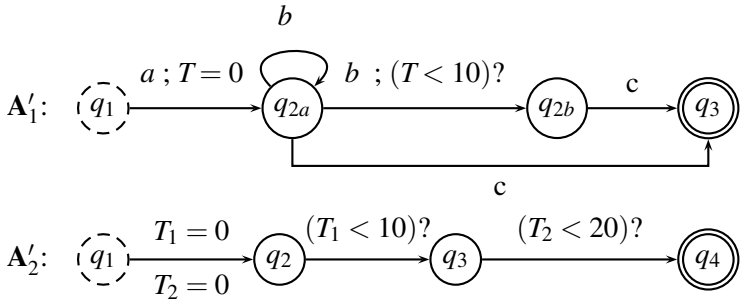


Figure 2: Equivalent, well-formed versions of automata from Figure 1.

The idea is that we will ultimately use TFAs to represent the timing properties of a child process. Paths through the automaton from its start state to an accepting state correspond to possible execution paths of the child process' code. Certainly, proving a tight upper bound on the delay between two arbitrary points along an execution path remains a very difficult problem, but to be clear, this is not our goal. Rather, our approach involves modeling an execution path through a child process (and, by extension, its corresponding timed automaton) using an *event-based* model, in which selected system events are modeled by transitions in the automaton, and we rely on timing properties of the process to be guaranteed by the underlying RTOS process scheduler. The problem of computing the worst-case delay through the automaton equates to that of computing the maximum delay over all possible paths through the automaton from its start state to its accepting state:

$$\Delta_{\mathbf{A}} = \max_{p \in \mathbf{paths}(\mathbf{A})} \Delta(p)$$

where

- $\mathbf{A} = \langle \Sigma, Q, q_0, q_f, \bar{T}, \delta, \gamma, \eta \rangle$ is the TFA
- $\mathbf{paths}(\mathbf{A})$ denotes the set of all paths in $\mathbf{A}$ from its start state q_0 to accepting state q_f, and
- $\Delta(p)$, for path $p = (q_0, ..., q_f)$, denotes the maximum delay from q_0 to q_f. That is, the maximum duration of any timed string $(\bar{\sigma}, \bar{\tau})$ such that $(q_0 ... q_f, \bar{\nu})$ is a run of the string over $\mathbf{A}$ (for some $\bar{\nu}$).

This problem can thus be formulated in the following manner: given a timed finite automaton $\mathbf{A}$ and an integer n, is there a timed word of duration $d \geq n$ that is accepted by $\mathbf{A}$? While simple cases, such as those presented in this paper, can be computed by observation and enumeration, the complexity of the general problem remains an open question, although we highly suspect it to be intractable— Courcoubetis and Yannakakis give exponential-time algorithms for this and related problems, and have shown a strictly more difficult variant of the problem to be **PSPACE**-complete [4]. Furthermore, expanding the timer constraint syntax to a more expressive variant (c.f. [1]) can only complicate matters in terms of complexity. We must be cautious, then, to ensure that we do not impose an inordinately large number of timers on a child process.

2.2 Timed Büchi Automata

Whereas we model the timing properties of the child processes of a cluster system using the timed finite automata of the previous section, we model these properties of the parent using a timed variant of ω-automata, specifically Timed Büchi Automata. We assume a basic familiarity with these; due to space constraints, we give only brief overview here. To review briefly, ω-automata, like standard finite automata, also consist of a finite number of states, but instead operate over words of infinite length.

Classes of ω-automata are distinguished by their acceptance criteria. Büchi automata, which we consider in this paper, are defined to accept their input if and only if a run over the input string visits an accepting state infinitely often. Other classes of ω-automata exist as well. For example, Muller automata are more stringent, specifying their acceptance criteria as a *set* of acceptance sets; a Muller automaton accepts its input if and only if the set of states visited infinitely often is specified as an acceptance set. More detailed specifics can be found elsewhere— for example, [1].

A *Timed Büchi Automaton* (TBA) is a tuple $\langle \Sigma, Q, q_0, q_f, \bar{T}, \delta, \gamma, \eta \rangle$, where

- Σ is a finite alphabet,
- Q is a finite set of states,
- $q_0 \in Q$ is the start state,
- $F \subseteq Q$ is a set of accepting states,
- $\bar{T}$ is a set of clocks,
- $\delta \subseteq Q \times Q \times \Sigma$ is the state transition relation,
- $\gamma \subseteq \delta \times 2^{\bar{T}}$ is the clock initialization relation, and
- $\eta \subseteq \delta \times X(\bar{T})$ is the constraint relation.

A tuple $\langle q_i, q_j, \sigma \rangle \in \delta$ indicates that a symbol σ yields a transition from state q_i to state q_j, subject to the restrictions specified by the clock constraints in η. A tuple $\langle q_i, q_j, \sigma, \bar{T} \rangle \in \gamma$ indicates that on the transition on symbol σ from q_i to q_j, all clocks in $\bar{T}$ are to be initialized to 0. Finally, a tuple $\langle q_i, q_j, \sigma, X(\bar{T}) \rangle \in \eta$ indicates that the transition on σ from q_i to q_j can only be taken if the constraint $X(\bar{T})$ evaluates to true under the values of the current timer interpretation.

We define *paths*, *runs*, and *subruns* over a TBA analagously to those over a TFA:

Definition 6 (Path (TBA))**.** *Let $\mathscr{A}$ be a TBA with state set Q and transition relation δ. $(q_1, ..., q_n)$ is a* path *over* **A** *if, for all $1 \leq i < n$, $\exists \sigma . \langle q_i, q_{i+1}, \sigma \rangle \in \delta$.*

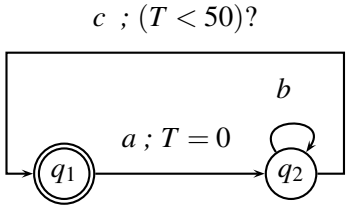
Definition 7 (Run, Subrun (TBA))**.** *A run (subrun) r, denoted by $(\bar{q}, \bar{\nu})$, of a Timed Büchi Automaton $\langle \Sigma, Q, q_0, q_f, \bar{T}, \delta, \gamma, \eta \rangle$ over a timed word $(\bar{\sigma}, \bar{\tau})$, is an infinite (finite) sequence of the form*

$$r : (q_0, \nu_0) \xrightarrow[\tau_1]{\sigma_1} (q_1, \nu_1) \xrightarrow[\tau_2]{\sigma_2} (q_2, \nu_2) \xrightarrow[\tau_3]{\sigma_3} ...$$

satisfying the same requirements as given in Definition 5.

For a run r, the set $inf(r)$ denotes the set of states which are visited infinitely many times. A TBA $\mathscr{A}$ with final states F accepts a timed word $w = (\bar{\sigma}, \bar{\tau})$ if $inf(r) \bigcap F \neq \emptyset$, where r is the run of w on $\mathscr{A}$. That is, a TBA accepts its input if any of the states from F repeat an infinite number of times in r.

Example 8. *Consider the following TBA $\mathscr{A}_1$, with start state q_1 and accept states $F = \{q_1\}$:*



This TBA accepts the ω-language $L_1 = \{((ab^*c)^\omega, \tau) \mid \forall x.\exists i,j.\forall k.\phi\}$ where ϕ is the boolean formula

$$\tau_i < \tau_k < \tau_j \implies (\sigma_i = a) \wedge (\sigma_k = b) \wedge (\sigma_j = c) \wedge (\tau_j - \tau_i < 50)$$

Lastly, we take the concept of maximum delay, introduced in the previous section with respect to Timed Finite Automata, and extend it to apply to Timed Büchi Automata. Doing so first requires the following definition, which allows us to restrict the timing analysis for TBAs to finite subwords:

Definition 9 (Subword over $\bar{q}$)**.** *Let $\mathscr{A}$ be a TBA, and let $\bar{q} = (q_m...q_n)$ be a finite path over $\mathscr{A}$. A finite timed word $w = ((\sigma_m...\sigma_n),(\tau_m...\tau_n))$ is a* subword over $\bar{q}$ *iff $\exists q_0,...,q_{m-1},\sigma_0,...,\sigma_{m-1},\tau_0,...,\tau_{m-1}$ such that $(q_0...q_{m-1}q_m...q_n,\bar{\nu})$ is a subrun of $((\sigma_0...\sigma_{m-1}\sigma_m...\sigma_n),(\tau_0...\tau_{m-1}\tau_m...\tau_n))$ over $\mathscr{A}$ for some $\bar{\nu}$.*

Definition 9 is a technicality which is necessary to support the following definition of the maximum delay between states of a TBA:

Definition 10. *Let $\mathscr{A}$ be a TBA, and let $\bar{q}$ be a finite path over $\mathscr{A}$. Then $\Delta_{\mathscr{A}}(\bar{q})$ is the maximum duration of any subword over $\bar{q}$.*

Example 11. *Consider $\mathscr{A}_1$ from Example 8. Then $\Delta_{\mathscr{A}_1}(q_1q_2q_2q_1) = 50$.*

Algorithmically computing $\Delta_{\mathscr{A}}(\bar{q})$ for a TBA $\mathscr{A}$ is analogous to the case for TFAs; in small cases (i.e., relatively few timers with small time constraints), the analysis is relatively simple, while we conjecture the problem for more complex cases to be intractable; we leave more detailed analysis for future work.

2.3 Parallel Timing Systems

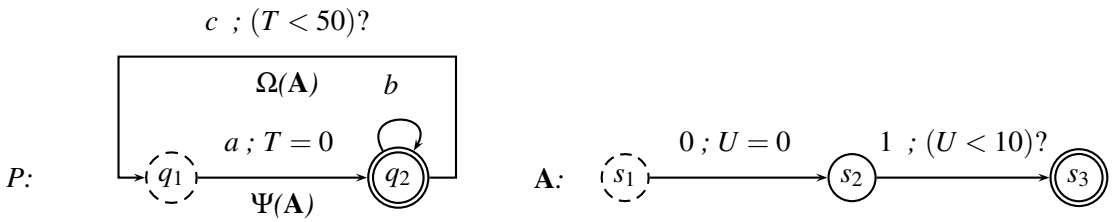
Next, we model the timing properties of a SPMD-type parallel system as a whole by combining the two models of Sections 2.1 and 2.2 into a single *parallel timing system*. A *parallel timing system* (PTS) is a tuple $\langle P, \bar{\mathbf{A}}, \psi, \varphi \rangle$, where

- $P = \langle \Sigma, Q, q_0, q_f, \bar{T}, \delta, \gamma, \eta \rangle$ is a TBA (used to model the timing properties of the parent process)
- $\bar{\mathbf{A}}$ is a set $\{\mathbf{A}_1,...,\mathbf{A}_n\}$ of TFAs (used to model the timing properties of the child processes)
- $\psi \subseteq \delta \times \bar{\mathbf{A}}$ is a *fork* relation (used to model the spawning of child processes)
- $\varphi \subseteq \delta \times \bar{\mathbf{A}}$ is a *join* relation (used to model barriers (joins))

A tuple $\langle q_i, q_j, \sigma, \mathbf{A} \rangle$ in ψ, with $\mathbf{A} \in \bar{\mathbf{A}}$, indicates that an instance of $\mathbf{A}$ is to be "forked" on the transition from q_i to q_j on symbol σ, and this "fork" is denoted graphically as $q_i \xrightarrow{\Psi(\mathbf{A})} q_j$, modeling the spawning of a child process along the transition. Similarly, a tuple $\langle q_i, q_j, \sigma, \mathbf{A} \rangle$ in φ indicates that a previously forked instance of $\mathbf{A}$ is to be "joined" on the transition from q_i to q_j on symbol σ. This "join" is denoted graphically as $q_i \xrightarrow{\Omega(\mathbf{A})} q_j$, modeling the joining along the transition with a previously spawned child process[1].

Example 12. *Consider the following timing system $S_1 = \langle P, \{\mathbf{A}\}, \psi, \varphi \rangle$:*

[1] Ψ was chosen as the symbol for 'fork', as it graphically resembles a "fork"; Ω was chosen as that for 'join', as it connotes "ending" or "finality".



P is the parent TBA with initial state q_1 and final state set $F = \{q_2\}$. P accepts the ω-language L_1 (see p. 7), and **A** *is a TFA which accepts the timed language $\{(01, \tau_1\tau_2) \mid \tau_2 - \tau_1 < 10\}$. In addition, the* fork *and* join *relations ψ and φ dictate that on the transition from q_1 to q_2, an instance of* **A** *is forked ($\Psi(\mathbf{A})$), and that the transition from q_2 to q_1 can only proceed once that instance of* **A** *has completed ($\Omega(\mathbf{A})$).*

Conceptually, this system models a parent process (P) which exhibits periodic behavior, accepting an infinite number of substrings of the form ab^*c, in which the initial 'a' triggers a child process **A** *which must be completed prior to the end of the sequence, marked by the following 'c'. In addition, the 'c' must occur no more than 50 time units after the initial 'a'. The child process is modeled by* **A***, which accepts strings of the form* 01*, in which the* 1 *must occur no more than ten time units after the initial* 0*.*

In theory, child processes could spawn children of their own (e.g. recursion). For now, however, we disallow this possibility, as it somewhat complicates the analysis in the following section without adding significantly to the expressive power of the model. The model can be expanded later to allow for arbitrarily nested children of children with the appropriate modifications; specifically, TBAs would need to be extended to include their own ψ and φ relations, as would the definition of Δ for TBAs.

Before proceeding, it is important to note that a PTS $S = \langle P, \bar{\mathbf{A}}, \psi, \varphi \rangle$ is not itself interpreted as an automaton. In particular, we do not ever define a language accepted by S. Indeed, it is not entirely clear what such a language would be, as we never specify the input to any of the children in **A**. Rather, the sole intent in specifying such a system S is to specify the *timing behavior* of the overall system, rather than any particular language that would be accepted by it.

2.3.1 Consistency

With this said, we note that in Example 12, **A** is in some sense "consistent" with its usage in P. Specifically, since the maximum duration of any string accepted by **A** is 10, we are guaranteed that any instance of **A** forked on the $q_1 \xrightarrow{a} q_2$ transition will have completed in time for the 'join' along the $q_2 \xrightarrow{c} q_1$ transition and hence, the timer $(T < 50)?$ on this transition would be respected in all cases. In this sense, all $(\Psi(\mathbf{A}), \Omega(\mathbf{A}))$ pairs are consistent with timer T. However, such consistency is not always the case. Consider, for instance, the parallel timing system S_2 shown in Figure 3. In this case, there are two child

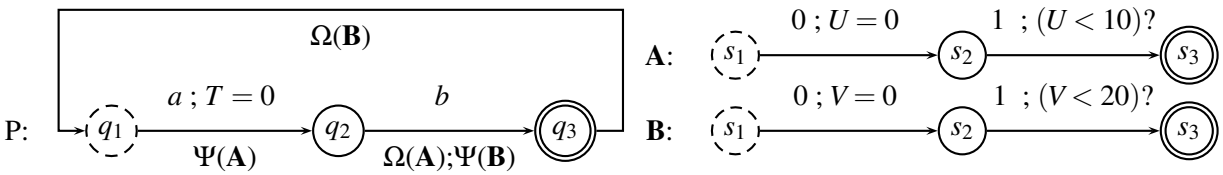


Figure 3: An inconsistent parallel timing system S_2.

processes: **A** and **B**. The maximum duration of a timed word accepted by **A** is 10, and that of **B** is 20.

Supposing that an 'a' occurs (and **A** forked) at time 0, it is thus possible that the **A** will not complete until time $10-\varepsilon_1$, at which time the 'b' and fork of **B** can proceed. It is therefore possible that **B** will not complete until time $30-\varepsilon_1-\varepsilon_2$ (for small $\varepsilon_1, \varepsilon_2$). This would then violate the $(T<25)$? constraint, corresponding to a case in which a child process could take longer to complete than is allowable, given the timing constraints of the parent process. It is precisely this type of interference which we must disallow in order for a timing system to be considered consistent with itself.

To this end, we propose a method of defining *consistency* within a timing system. Informally, we take the approach of deriving a new set of conditions from the timing constraints of the child processes, so that checking *consistency* reduces to the process of verifying that these conditions respect the timing constraints of the master process.

First, we replace $\bar{\mathbf{A}}, \psi$, and φ from the parallel timing system with a new set of *derived* timers, one for each $\mathbf{A} \in \bar{\mathbf{A}}$, defining the possible "worst case" behavior of the child processes. Each such timer $T_{\mathbf{A}}$ is initialized on the transition along which the corresponding **A** is forked, and is used along (constrains) any transitions along which **A** is joined. Each such use ensures that the timer is less than $\Delta_{\mathbf{A}}$, representing the fact that the elapsed time between the forking and joining of a child process is bounded in the worst case by $\Delta_{\mathbf{A}}$— the longest possible duration for the child process. As an example, "flattening" the timing system S_1 of Example 12 results in a single new timer $T_{\mathbf{A}}$, initialized along the $q_1 \xrightarrow{a} q_2$ transition, and used along the $q_2 \xrightarrow{c} q_1$ transition with the constraint $(T_{\mathbf{A}} < 10)$?. We then check that none of these new derived timers invalidate the timing constraints of the parent process.

Formally, we define two relations. The first of these is *flattening*, which takes a parallel timing system $\langle P, \bar{\mathbf{A}}, \psi, \varphi \rangle$ and yields a new pair of relations (γ, η). Intuitively, γ defines the edges along which each of the derived timers are initialized, and η defines the edges along which each of the derived timers are used:

Definition 13. *Let $S = \langle P, \bar{\mathbf{A}}, \psi, \varphi \rangle$ be a parallel timing system. Then* **flatten**$(S) = (\gamma, \eta)$*, where*

$$\gamma = \{\langle q_i, q_j, \sigma, \{T_{\mathbf{A}}\}\rangle \mid \langle q_i, q_j, \sigma, \mathbf{A}\rangle \in \psi\}$$
$$\eta = \{\langle q_i, q_j, \sigma, X\rangle \mid \langle q_i, q_j, \sigma, \mathbf{A}\rangle \in \varphi\}$$

and

$$X = \bigwedge_{\langle q_i, q_j, \sigma, \mathbf{A}\rangle \in \varphi} (T_{\mathbf{A}} < \Delta_{\mathbf{A}})$$

The second relation takes ψ and φ as inputs and extracts a set of edge pairs, defined such that each such pair (e_1, e_2) specifies when a derived timer is initialized (e_1) and used (e_2).

Definition 14. *Let $S = \langle P, \bar{\mathbf{A}}, \psi, \varphi \rangle$ be a parallel timing system, with $\mathbf{A} \in \bar{\mathbf{A}}$. Then the set of all use pairs of* **A** *in S is defined as* **pairs**$(\mathbf{A}, S) = \{((q_x, q_y), (q_m, q_n)) \mid (\langle q_x, q_y, \sigma_1, \mathbf{A}\rangle \in \psi) \wedge (\langle q_m, q_n, \sigma_2, \mathbf{A}\rangle \in \varphi)\}$ *for some σ_1, σ_2. Furthermore,*

$$\textbf{pairs}(S) = \bigcup_{\mathbf{A} \in \bar{\mathbf{A}}} \textbf{pairs}(\mathbf{A}, S)$$

Example 15. *Consider parallel timing system S_3 shown in Figure 4. Observe that $\Delta_{\mathbf{A}} = 25$ and $\Delta_{\mathbf{B}} = 11$. Then:* **flatten**$(S_3) = (\gamma, \eta)$*, where*

$$\gamma = \{\langle q_1, q_2, a, \{T_{\mathbf{A}}\}\rangle, \langle q_2, q_3, b, \{T_{\mathbf{B}}\}\rangle\}$$
$$\eta = \{\langle q_3, q_1, c, X\rangle\} \text{ , where } X = (T_{\mathbf{A}} < 25) \wedge (T_{\mathbf{B}} < 11)$$

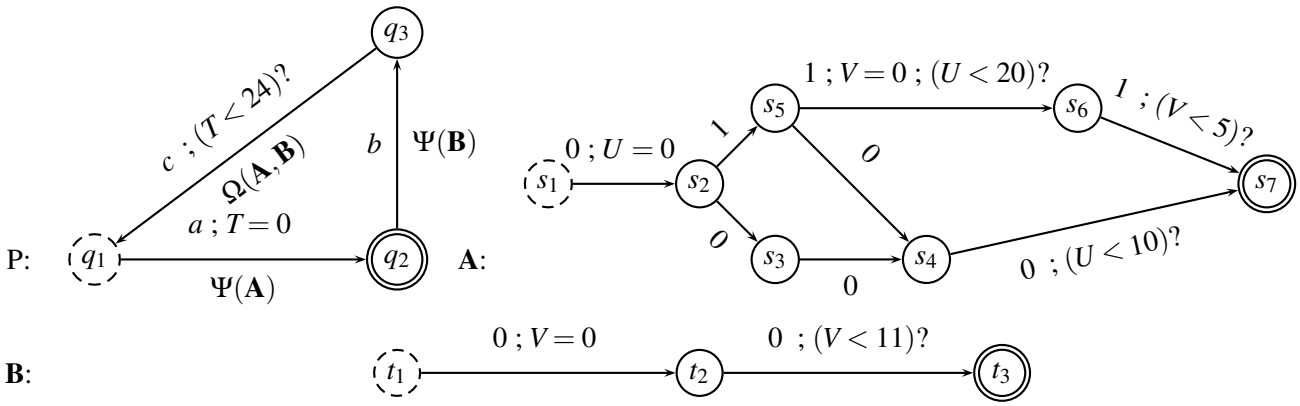


Figure 4: Parallel timing system S_3. $\Delta_{\mathbf{A}} = 25, \Delta_{\mathbf{B}} = 11$.

shown graphically in Figure 5, and

$$\begin{aligned}\mathbf{pairs}(S) &= \mathbf{pairs}(\mathbf{A},S) \cup \mathbf{pairs}(\mathbf{B},S) \\ &= \{((q_1,q_2),(q_3,q_1))\} \cup \{((q_2,q_3),(q_3,q_1))\} \\ &= \{((q_1,q_2),(q_3,q_1)),((q_2,q_3),(q_3,q_1))\}\end{aligned}$$

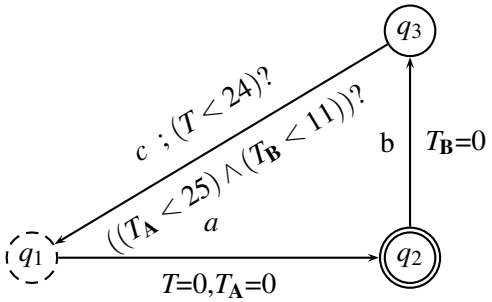


Figure 5: The result of flattening S_3: forks and joins of **A** and **B** are shown along with derived timers $T_{\mathbf{A}}$ and $T_{\mathbf{B}}$. Compare with Figure 4.

We can now proceed with a formal definition of consistency for a parallel timing system. Recall that intuitively, such a system is consistent if the worst case timing scenarios over all child processes will not invalidate the timing constraints of the parent process— in other words, if the maximum delay between two states allowed by the child processes never exceeds the corresponding maximum delay allowed by the timers in the parent process.

Definition 16 (Consistency)**.** *Let $S = \langle \mathscr{A}, \psi, \varphi, \bar{\mathbf{A}} \rangle$ be a PTS, where*

- *$\mathscr{A} = \langle \Sigma, Q, q_0, F, \bar{T}, \delta, \gamma, \eta \rangle$ is a TBA*
- **flatten**$(S) = (\gamma', \eta')$
- *$\mathscr{A}' = \langle \Sigma, Q, q_0, F, \bar{T}, \delta, \gamma', \eta' \rangle$*

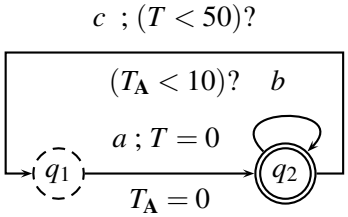
Then S is consistent *if for all edge pairs $((q_x,q_y),(q_m,q_n)) \in$ **pairs**(S) and all paths $p = q_x q_y ... q_m q_n$ through $\mathscr{A}$,*

$$\Delta_{\mathscr{A}'}(p) \leq \Delta_{\mathscr{A}}(p) \tag{16.1}$$

We conclude this section with a few simple examples, which should help to clarify Definition 16; the following section gives a more realistic example.

Example 17. *S_1 is consistent.*

Proof. P', the result of flattening S_1, is shown below, with $T_{\mathbf{A}}$ being the derived timer corresponding to **A**:



Furthermore, $\mathbf{pairs}(P') = \{((q_1,q_2),(q_2,q_1))\}$, and by observation, all paths through P' beginning with the edge (q_1,q_2) and ending with the edge (q_1,q_2) take the form $q_1(q_2)^*q_1$. All such paths p satisfy inequality 16.1, and thus by definition, S_1 is consistent. □

Example 18. *S_3 is not consistent.*

Proof. P', the result of flattening S_3, is shown in Figure 5. Furthermore,

$$\mathbf{pairs}(P') = \{((q_1,q_2),(q_3,q_1)),((q_2,q_3),(q_3,q_1))\}$$

There are thus two paths against which we need to test inequality 16.1: $(q_1q_2q_3q_1)$, and $(q_2q_3q_1)$; the first of these fails the test: $\Delta_{P'}(q_1q_2q_3q_1) = 25$,and $\Delta_P(q_1q_2q_3q_1) = 24$.

□

3 Case Study: Matrix Multiplication

We now turn our attention to a practical application of the concepts discussed so far. Namely, we demonstrate the use of the formal validation concepts on a simple parallel, MPI-style [14] matrix multiplication kernel, extracted from the larger power-grid analysis application described in [12, 13]. Our kernel implements a variant of Fox's algorithm for matrix multiplication [7]. For simplicity, we assume square matrices, and that the number of columns, rows, and processors are all perfect squares. The algorithm distributes the task of multiplying two matrices amongst all processors in the system.

We give a simple distributed algorithm for matrix multiplication, and a consistent parallel timing system for that algorithm. We conclude the section with empirical results— timing measurements taken on a small, four-node real-time cluster, each node consisting of dual quad-core 2.66Ghz Xeon X5660 processors running the Xenomai RTOS [9] with 48GB RAM. The timing measurements of the PTS, along with the usual restrictions associated with real-time computation (e.g. no virtual memory or paging, process scheduling, ensuring minimal variance in execution timings, etc.), are bounded by virtue of Xenomai's real-time process scheduler. The result is a matrix multiplication kernel which provably runs in under 9 ms per cycle for 128×128 double-precision matrices. We emphasize that we are not claiming the speed of the operation to be a groundbreaking result— obviously, this is a relatively small matrix size, but was so chosen as this is the order of the size required by our targeted application kernel. Rather, we give these numbers, as well as the PTS, to illustrate the *process* by which we analyze the temporal interactions between processes, thus showing this delay to be a provable upper bound.

3.1 Algorithm

Algorithm 1 MatrixMultiply: Compute $\mathbf{C} = \mathbf{A} \times \mathbf{B}$

p : Number of processors
N : Rank of matrices

1: $q \leftarrow \sqrt{p}$
2: **while true do**
3: $dest \leftarrow 1$
4: **if** self $== 0$ **then** {Master process}
5: **for** $i = 0$ to $q-1$ **do**
6: **for** $j = 0$ to $q-1$ **do**
7: $w \leftarrow i\frac{N}{q}, x \leftarrow (i+1)\frac{N}{q}$
8: $y \leftarrow j\frac{N}{q}, z \leftarrow (j+1)\frac{N}{q}$
9: $\bar{X} \leftarrow \mathbf{A}[w:x][0:N]$
10: $\bar{Y} \leftarrow \mathbf{B}[0:N][y:z]$
11: **if** $i \neq 0$ **and** $j \neq 0$ **then** {Master already has these chunks}
12: **send**$(\bar{X}, dest)$
13: **send**$(\bar{Y}, dest)$
14: $dest \leftarrow dest + 1$
15: **end if**
16: **end for**
17: **end for**
18: **else** {Child processes}
19: $\bar{X} \leftarrow$ **recv**(0)
20: $\bar{Y} \leftarrow$ **recv**(0)
21: **end if**
22: $\bar{Z} \leftarrow$ **locMM**$(\bar{X}, \bar{Y})$
23: **reduce**$(\bar{Z}, \mathbf{C})$
24: **end while**

The pseudocode for the algorithm is given in Algorithm 1. Conceptually, to multiply two $N \times N$ matrices $\mathbf{A}$ and $\mathbf{B}$ using a p processor cluster, each matrix is divided into segments, which are then distributed in round-robin fashion amongst the processors of the cluster. Each processor then performs a local matrix multiplication on its own local submatrices, and the results of these local operations are aggregated (reduced) to form the matrix product $\mathbf{A} \times \mathbf{B}$.

Due to space constraints, we will not describe the partitioning in detail; Figure 6 shows the partitioning and distribution of work by Algorithm 1 for a four-processor cluster. In this figure, $\mathbf{A}, \mathbf{B}$, and $\mathbf{C}$ are all $N \times N$ matrices. $\mathbf{A}$ is partitioned into 2 sets of $\frac{N}{2}$ rows each, and $\mathbf{B}$ is partitioned into 2 sets of $\frac{N}{2}$ columns each. The master process, p_0, computes the local product $\mathbf{A}_1 \times \mathbf{B}_1$, and writes the result to $\mathbf{C}_1$. p_0 then sends submatrices $\mathbf{A}_1$ and $\mathbf{B}_2$ to p_1, who then computes their product, writing the result to $\mathbf{C}_2$. Similarly, p_2 receives and computes $\mathbf{C}_3 = \mathbf{A}_2 \times \mathbf{B}_1$, and p_3 receives and computes $\mathbf{C}_4 = \mathbf{A}_2 \times \mathbf{B}_2$.

The algorithm proceeds as follows: the master process executes lines 5 through 17, which partition $\mathbf{A}$ and $\mathbf{B}$ into submatrices (lines 7–10), and send these parts out to the respective child processes (lines 12–13). Conversely, the child processes execute lines 19–20, which receive the submatrices assigned by the master process. Lines 22–23 are run by all processes, including the master process (which, in

this case, participates in the task of matrix multiplication as well). Line 22 performs the local operation, line 23 writes the local result to the appropriate location in **C**.The entire process then repeats indefinitely, as given by the **while** loop (lines 2 and 24).

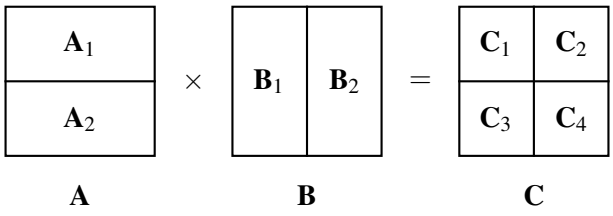


Figure 6: Partitioning and distribution of matrix multiplication by Algorithm 1 across a four processor cluster.

3.2 Parallel Timing System

Figure 7 shows a parallel timing system for Algorithm 1 across a four processor cluster, consisting of the TBA P_{MM}, which models the master process, and a child TFA $\mathbf{A}_{MM}$, modeling instances of the child processes. Specific events have been elided from the diagram in this case, since events in this case always represent transitions between statements.

3.2.1 Parent

States in the parent automaton P are prefixed with a 'P', followed by the line number as given in Algorithm 1. For example, $P3$ corresponds to the state of the parent process as it is executing line 3.

Additionally, lines 12 and 13 each beget three separate states— parameterized on the values of the loop induction variables i and j— and are labeled accordingly. As is commonly the case in WCET analysis, unrolling the loop nest in this fashion is necessary in order to obtain a strict upper bound on the number of iterations and, consequently, the total execution time, of the loop nest.

P forms, in this case, a simple cycle. The cycle starts at state $P3$, and steps sequentially through the steps (states) of the algorithm. Namely, the parent process starts at line 3 (i.e., state $P3$), and proceeds sequentially through lines 4 (state $P4$), and eventually to line 12 ($P12^{i=0}_{j=1}$). The delay between the initialization (state $P3$) and the first send ($P12^{i=0}_{j=1}$) is bounded by a timer, T_{setup1} (the idea being that this is the delay incurred by the time to "set up" the first send). Execution then proceeds to line 13 ($P13^{i=0}_{j=1}$); the delay along this transition represents the time to send the first chunk to the respective child process, and is bounded by timer T_{send1}. At this point, execution proceeds to line 14 ($P14^{i=0}_{j=1}$). Along this transition, there are two items to note: first, the time to process the second send is bounded by the timer T_{send2}, and second, the child process has now been sent the data it needs, and consequently, $\mathbf{A}_1$ is forked. Execution proceeds similarly through the next six states, representing the unwound iterations of the loop nest. Child process $\mathbf{A}_2$ is similarly forked on the transition from $P13^{i=1}_{j=0}$ to $P14^{i=1}_{j=0}$, and $\mathbf{A}_3$ on the transition from $P13^{i=1}_{j=1}$ to $P14^{i=1}_{j=1}$. Execution then proceeds through lines 22 (state $P22$) and 23 ($P23$). The duration of the local matrix multiplication operation (line 22) is bounded by the timer T_{MM}, and that through the reduce operation (line 23) by the timer T_{reduce}. Additionally, the transition from $P23$ back to $P3$ waits for (joins with) all child processes to complete before proceeding.

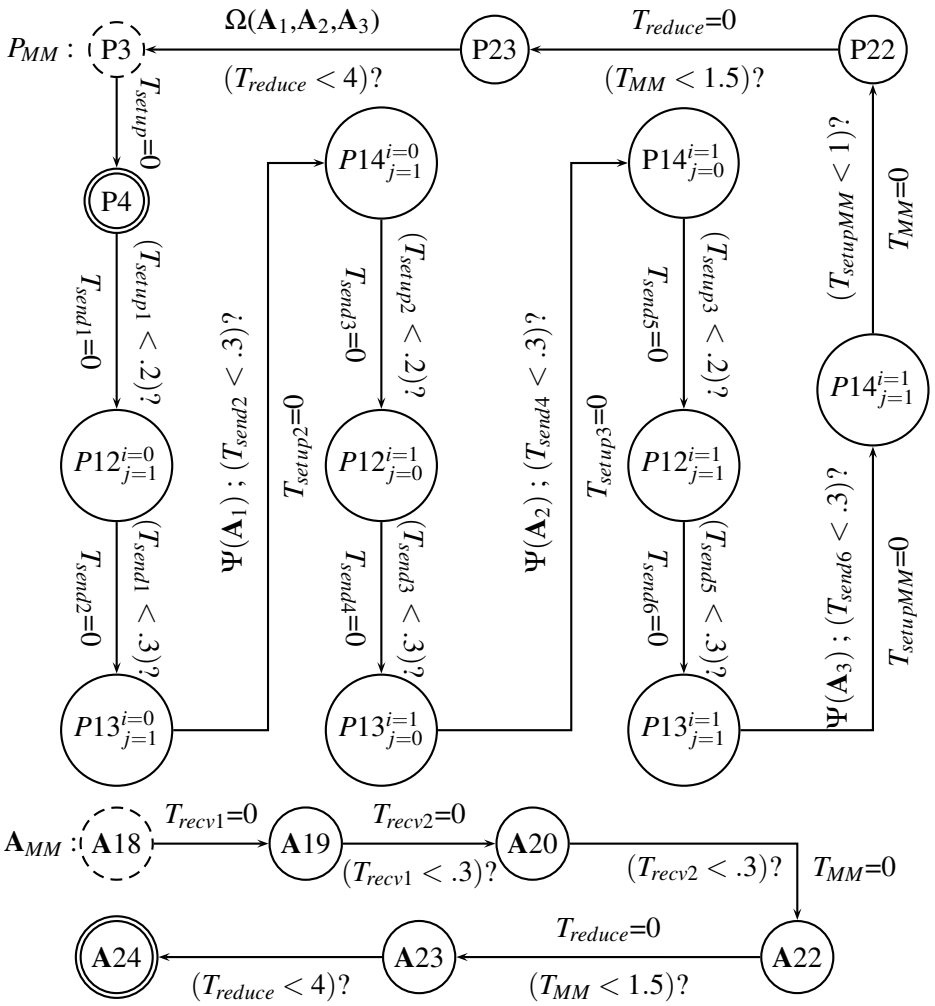


Figure 7: Parallel timing system S_{MM} for Algorithm 1 across a four processor cluster. Events have been elided for the sake of clarity. Upper bounds on timer constraints correspond to delay measurements taken over our implementation; times are given in milliseconds. Minimal variance from these bounds is ensured to the extent provided by the underlying RTOS.

3.2.2 Child

In this case, the child processes are modeled by the TFA **A**. Nomenclature is analogous to that of *P*: states in **A** are prefixed with an **A**, followed by the corresponding line number from Algorithm 1.

The child process starts at line 18 (state **A**18). The process then proceeds to receive the first block of data (line 19, state **A**19). The time to process the receive is bounded by timer T_{recv1}. Execution proceeds to receive the second block of data (line 20, state **A**20). The time to process this second receive is bounded by T_{recv2}. Execution proceeds next to the local matrix multiplication (line 22, state **A**22); the time spent on this operation is bounded by timer T_{locMM}. Finally, execution proceeds to the data writeback (line 23, state **A**23); the time spent on this operation is bounded by timer T_{reduce}.

Theorem 19. *S_{MM} is consistent.*

Proof. Let $\mathbf{flatten}(S_{MM}) = (\gamma', \eta')$, with $P'_{MM} = \langle \Sigma, Q, q_0, F, \bar{T}, \delta, \gamma', \eta' \rangle$. By definition, $\mathbf{pairs}(S_{MM}) =$

$\{(e_1,e_4),(e_2,e_4),(e_3,e_4)\}$, where

$$e_1 = (P13_{j=1}^{i=0}, P14_{j=1}^{i=0}) \qquad e_2 = (P13_{j=0}^{i=1}, P14_{j=0}^{i=1})$$
$$e_3 = (P13_{j=1}^{i=1}, P14_{j=1}^{i=1}) \qquad e_4 = (P23, P3)$$

By observation, there are three paths which we must consider:

$$p_1 = (P13_{j=1}^{i=0} P14_{j=1}^{i=0} ... P23P3)$$
$$p_2 = (P13_{j=0}^{i=1} P14_{j=0}^{i=1} ... P23P3)$$
$$p_3 = (P13_{j=1}^{i=1} P14_{j=1}^{i=1} ... P23P3)$$

The rest of the proof follows by enumeration:

$$(\Delta_{P'_{MM}}(p_1) = 6.1) < (\Delta_{P_{MM}}(p_1) = 8.1)$$
$$(\Delta_{P'_{MM}}(p_2) = 6.1) < (\Delta_{P_{MM}}(p_2) = 7.3)$$
$$(\Delta_{P'_{MM}}(p_3) = 6.1) < (\Delta_{P_{MM}}(p_3) = 6.5)$$

□

Finally, we note that the worst case delay along one iteration of the algorithm is 8.9 ms. This follows from the observation that the parent automaton P takes the form of a simple cycle with no unbound segments (i.e., subpaths which are not constrained by any timer). Specifically, P consists of consecutive pairs of segments, each constrained by pairs of timers. Consequently, we can derive an upper bound for a single iteration of the algorithm by summing the bounds of all of the timers, yielding the specified upper bound. Combined with Theorem 19, which ensures that the timing of the child processes does not invalidate this bound, we are left with a cyclic, parallel, time-bounded matrix multiplication kernel.

4 Concluding Remarks

We conclude with a few closing remarks. We have presented a formal system for modeling the temporal properties of a restricted class of real-time parallel systems, with a simple example of an application kernel. As is usually the case with real-time systems, loops need to be unrolled, bounding the number of iterations, in order to obtain an upper bound on the total execution time of the loop. Algorithm 1 (intentionally) distills to a relatively simple PTS, due to the basic structure of the control flow graph of both the parent and child processes; more complex examples are of obvious interest for future work. Similarly, the model in Figure 7 in our case was derived manually— in this case, a relatively simple task. More complex examples can certainly prove to be more of a challenge, and automated tools for this task are desirable. One possible approach for such automation would be compiler-driven, whereby users could specify to the compiler (via `#pragma`s, for instance), events of interest, and the compiler could proceed to output the appropriate annotated control flow graph.

We assume timing behavior is consistent across all child processes, although if there were to be significant variance across child processes (e.g. heterogeneous or NUMA architectures) we could account for such behavior using different child TFAs.

Additionally, we have laid out several interesting open questions which arise out of the analysis of our relatively straightforward formulation: what is the complexity of computing the worst case delay

along a single path of a TFA (TBA), and through a TFA (TBA) in general? Up to this point, we have only considered conjunctions of maximum constraints; how does this change in the presence of a more generalized constraint syntax (c.f. [1])?

We have largely been working with the SPMD execution model paradigmatic of many MPI-type programs. It would be interesting to investigate temporal models for other parallel models (e.g. OpenMP) as well. Lastly, our application kernel distills to a relatively simple set of automata. More complex examples are certainly of interest, and are on the horizon for future work.

References

[1] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[2] Fang Chen, Xueshan Han, Zhiyuan Pan, and Li Han. State Estimation Model and Algorithm Including PMU. In *Electric Utility Deregulation and Restructuring and Power Technologies, 2008. DRPT 2008. Third International Conference on*, pages 1097 –1102, april 2008.

[3] Yousu Chen, Zhenyu Huang, and D. Chavarria-Miranda. Performance evaluation of counter-based dynamic load balancing schemes for massive contingency analysis with different computing environments. In *Power and Energy Society General Meeting, 2010 IEEE*, pages 1 –6, july 2010.

[4] Costas Courcoubetis and Mihalis Yannakakis. Minimum and maximum delay problems in real-time systems. *Form. Methods Syst. Des.*, 1(4):385–415, December 1992.

[5] Alexandre David, Kim G. Larsen, Axel Legay, Ulrik Nyman, and Andrzej Wasowski. Timed I/O automata: a complete specification theory for real-time systems. In *Proceedings of the 13th ACM international conference on Hybrid systems: computation and control*, HSCC '10, pages 91–100, New York, NY, USA, 2010. ACM.

[6] Khaled El-Fakih, Nina Yevtushenko, Sergey Buffalov, and Gregor v. Bochmann. Progressive solutions to a parallel automata equation. *Theoretical Computer Science*, 362(13):17 – 32, 2006.

[7] G.C Fox, S.W Otto, and A.J.G Hey. Matrix algorithms on a hypercube I: Matrix multiplication. *Parallel Computing*, 4(1):17 – 31, 1987.

[8] Wenzhong Gao and Shaobu Wang. On-line dynamic state estimation of power systems. In *North American Power Symposium (NAPS), 2010*, pages 1 –6, sept. 2010.

[9] P. Gerum. The Xenomai Project. Implementing a RTOS emulation framework on GNU/Linux. In *Third Real-Time Linux Workshop*, 2001.

[10] I. Gorton, Zhenyu Huang, Yousu Chen, B. Kalahar, Shuangshuang Jin, D. Chavarria-Miranda, D. Baxter, and J. Feo. A high-performance hybrid computing approach to massive contingency analysis in the power grid. In *e-Science, 2009. e-Science '09. Fifth IEEE International Conference on*, pages 277 –283, dec. 2009.

[11] Zhenyu Huang, Yousu Chen, and J. Nieplocha. Massive contingency analysis with high performance computing. In *Power Energy Society General Meeting, 2009. PES '09. IEEE*, pages 1 –8, july 2009.

[12] Peter Hui, Satish Chikkagoudar, and Daniel Chavarría-Miranda. Towards a real-time cluster computing infrastructure. *IEEE Real-Time Systems Symposium (Work-in-Progress Session)*, 2011.

[13] Peter Hui, Barry Lee, and Satish Chikkagoudar. Towards real-time high performance computing for power grid analysis. In *Proceedings of the Second International Workshop on High Performance Computing, Networking and Analytics for the Power Grid*, HiPCNA-PG '12, Washington, DC, USA, 2012. IEEE.

[14] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI-The Complete Reference, Volume 1: The MPI Core*. MIT Press, Cambridge, MA, USA, 2nd. (revised) edition, 1998.

[15] P. David Stotts and William Pugh. Parallel finite automata for modeling concurrent software systems. *Journal of Systems and Software*, 27:27–43, 1994.

Generating Property-Directed Potential Invariants By Backward Analysis

Adrien Champion
Onera, The French Aerospace Lab
Toulouse, France
Rockwell Collins France
Blagnac, France
adrien.champion@onera.fr

Rémi Delmas
Onera, The French Aerospace Lab
Toulouse, France
remi.delmas@onera.fr

Michael Dierkes
Rockwell Collins France
Blagnac, France
mdierkes@rockwellcollins.com

This paper addresses the issue of lemma generation in a *k*-induction-based formal analysis of transition systems, in the linear real/integer arithmetic fragment. A backward analysis, powered by quantifier elimination, is used to output preimages of the negation of the proof objective, viewed as unauthorized states, or *gray states*. Two heuristics are proposed to take advantage of this source of information. First, a thorough exploration of the possible partitionings of the gray state space discovers new relations between state variables, representing potential invariants. Second, an inexact exploration regroups and over-approximates disjoint areas of the gray state space, also to discover new relations between state variables. *k*-induction is used to isolate the invariants and check if they strengthen the proof objective. These heuristics can be used on the first preimage of the backward exploration, and each time a new one is output, refining the information on the gray states. In our context of critical avionics embedded systems, we show that our approach is able to outperform other academic or commercial tools on examples of interest in our application field. The method is introduced and motivated through two main examples, one of which was provided by Rockwell Collins, in a collaborative formal verification framework.

1 Introduction

The recent DO-178C and its formal methods supplement DO-333 published by RTCA[1] acknowledge the use of formal methods for the verification and validation of safety critical flight control software and allow their use in development processes. Successful examples of industrial scale formal methods applications exist, such as the verification by Astrée [4] of the run-time safety of the Airbus A380 flight control software C code. However, the verification of general functional properties at model level, *i.e.* on Lustre [7] or MATLAB Simulink© programs, from which the embedded code is generated, still requires expert human intervention to succeed on common avionics software design patterns, preventing industrial designers from using formal verification on a larger scale. Formal verification at model level is important, since it helps raising the confidence in the correctness of the design at early stages of the development process. Also, the formal properties and lemmas discovered at model level can be forwarded to the generated code, in order to facilitate the final design verification and its acceptance by certification authorities [12]. Our work addresses some of the issues encountered when attempting formal verification of properties of synchronous data flow models written in Lustre. We propose a property-directed lemma generation approach, together with a prototype implementation. The proposed approach aims at reducing the amount of human intervention usually needed to achieve *k-induction* proofs, possibly using *abstract*

[1] http://www.rtca.org/

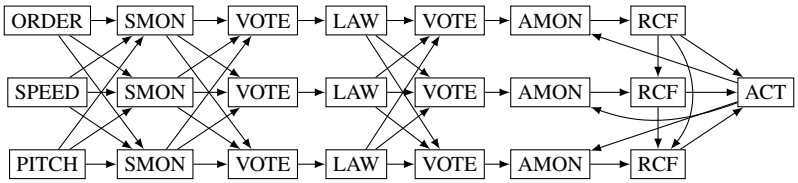


Figure 1: Shuffled, triple channel architecture

interpretation technique in cooperation. Briefly outlined, the approach consists first in an abstract interpretation pass to discover coarse bounds on the numerical state variables of the system; a k-induction engine and our lemma generation techniques are then ran in parallel to search for potential invariants in order to strengthen the property. We insist on the fact that the primary goal of the proposed method is discovering missing information needed to prove properties the verification of which is either very expensive or impossible with currently available methods and tools, rather than improving the performance of the verification of properties which are already relatively easily provable.

The paper is structured as follows: Section 2 describes the embedded software architectures targeted by our work. Related work and tools are discussed in Section 3 before notations and vocabulary are given in Section 4. A description of the underlying k-induction engine assumed in this paper follows in Section 5. We introduce and motivate our approach in Section 6 and detail the lemma generation techniques in Section 7. The proposed approach is then illustrated on a reconfiguration logic example and on Rockwell Collins industrial triplex sensor voter in Section 8. Implementation is briefly discussed in Section 9, before concluding in Section 10.

2 Fault Tolerant Avionics Architectures

We consider embedded reactive software functions which contribute to the safe operation of assemblies of hardware sensors, networked computers, actuators, moving surfaces, *etc.* called *functional chains*. A functional chain can for instance be in charge of "controlling the aircraft pitch angle", and must meet both qualitative and quantitative safety requirements depending on the effects of its failure. Effects are ranked from MIN (minor effect) to CAT (catastrophic effect, with casualties). For instance, the failure of a pitch control function is ranked CAT, and the function shall be robust to at least a double failure and have an average failure rate of at most 10^{-9} per flight hour. In order to meet these requirements, engineers must introduce hardware and software redundancy and implement several fault detection and reconfiguration mechanisms in software.

A frequently encountered architectural design pattern, triplication with shuffle, is depicted in Fig. 1. It allows to recover from single failures and to detect double failures. Data sources, data processing hardware and functions are triplicated to obtain three *channels*. The actuator is not replicated. Data is locally monitored right after acquisition/production, but is also broadcast across channels to be checked using triplex voting functions, in order to detect complex error situations. Last, in each channel, depending on the fault state of the channel and the observable behavior of other channels, the *reconfiguration logic* decides whether the channel in question must take control of the actuator or on the contrary mute itself. Being *healthy* for a channel means that no fault has occurred for a sufficient amount of consecutive time steps to become *confirmed*.

Previous work by our team addresses the formal verification of numerical stability of control laws [24], yet ensuring proper behavior of *voting functions* and *reconfiguration logic* as introduced here is equally important, for these building blocks and design patterns are ubiquitous in fault tolerant avionics soft-

ware. For the voting logic, we focus on BIBO properties, "bounded input implies bounded output", the verification of which is detailed in Section 8.2. For reconfiguration logic, which makes an extensive use of integer timers and discrete logic, we focus on bounded liveness properties such as "assuming at most two sensor, network or CPU faults, the actuator must never remain idle for more than N consecutive time units", the verification of which is addressed in Section 8.1.

3 Related Work and Tools

In this section we review the state of the art of verification tools relevant in our application domain, *i.e.*, of currently available tools and techniques allowing to address synchronous data flow models written in Lustre. We distinguish two main families of verification approaches.

First, approaches based on abstract interpretation (AI [9]). The tool NBac [15] for instance allows to analyze properties of Lustre models by using a combination of forward and backward fixpoint computation using AI. AI tends to need expert tuning for the choice of abstract domains, partitioning, *etc.* to behave correctly on the systems we consider. NBac proposes a heuristic selection of AI parameter tuning, which dynamically refines domains and partitionings to try to obtain a better precision without falling in a combinatorial blowup.

Second, the family of k-induction [25] based approaches, with the commercial tool Scade Design Verifier[2], or the academic tool Kind [17]. Kind is the most recently introduced tool, and wraps the k-induction core in an automatic counter example guided abstraction refinement loop whereas the Scade Design Verifier does not. k-induction is an exact technique, in which little or no abstraction is performed (the concrete semantics of the program is analyzed). Experiments show that it does not scale up out of the box on the systems encountered in our application field. Proving proof obligations on such systems often requires to unroll the system's transition relation to the reoccurrence diameter of the model which can be very large in practice (hundred or thousands of transitions). For such proof obligations, which are either k-inductive for a k too large to be reached in practice, or even non-inductive at all, numerical lemmas are needed to help better characterize the reachable state space and facilitate the inductive step of the reasoning.

In order to address this common issue with k-induction, automatic lemma generation techniques have been studied. Two main approaches can be distinguished. First, property agnostic approaches, such as [16], in which template formulas are instantiated in a brute force manner on combinations of the system state variables to obtain a set of potential invariants. They are then analyzed alongside the PO using the main k-induction engine. Second, property directed approaches, such as [6, 5], in which the negation of root states of counterexamples are used as strengthening lemmas, with or without generalization, or are used to guide template instantiation. Also worth mentioning, interpolation [18] yields very interesting results in lemma generation but unfortunately to our knowledge no interpolation tool analyzing Lustre code exists.

We consider a lemma generation pass successful when the generated potential invariants allow to prove the original proof objective with a k-induction run with a small k. Once the right lemmas are found, the proof can be easily re-run and checked by third party k-induction tools, an important criterion for industrials and certification organisms. As we will see in the rest of the paper, the lemma generation approach proposed in this paper takes inspiration from all the aforementioned techniques : while somehow brute force in its exploration of the gray state space partitionings, our approach discovers relevant lemmas thanks to its property-directed nature.

[2] http://www.esterel-technologies.com/products/scade-suite/add-on-modules/design-verifier

4 Notations

Let us now define several notions used throughout this paper. First, a *transition system* is represented as a tuple $\langle v, D, I(v), T(v,v')\rangle$ where v is a vector of state variables, D specifies the domain of each state variable, either boolean, integer or real valued, I is the initial state predicate, and T is the transition predicate in which v' represents next state variables. The logic used to express predicates is Linear Integer or Real Arithmetic with Booleans. The usual notions of trace semantics and reachability are used. Given a formula $PO(v)$ representing a Proof Objective (PO), we say that the PO holds if no state s such that $\neg PO(s)$ can be reached from I through repeated application of T. Lustre or Scade programs can be cast into this representation using adequate compilers.

An *atom* is a Boolean or its negation, or a linear equality or inequality in LRA or LIA. A *polyhedron* is a conjunction of atoms. More precisely, we will say polyhedron for not necessarily closed polyhedron, meaning that we do not impose restrictions on the form of the inequalities besides linearity. The *convex hull* of two polyhedra p_1 and p_2 is the smallest polyhedron such that it contains p_1 and p_2. We will say that the convex hull h of two polyhedra p_1 and p_2 is *exact* if and only if $h = p_1 \cup p_2$, and call it the *Exact Convex Hull* (or ECH) of p_1 and p_2 if it exists. For the sake of clarity, convex hulls that are not necessarily exact will be called *Inexact Convex Hulls* (or ICH). Note that for integer variables, the uniqueness of the convex hull is not guaranteed if non-integer values for the coefficients are not forbidden. We ban them in the rest of this paper; in our implementation, it is prevented by the type system. Still, there are several ways to represent the same inequality, *e.g.* $n > 0$ and $n \geq 1$. Despite their difference in representation, these polyhedra enclose the same (integer) points geometrically speaking, so this does not hinder our approach. Convex hull comparison in this paper does not rely on their syntax nor semantics, but rather on the *source* of the hull, *i.e.* the original polyhedra used to create them. This will be discussed in Section 7 during the explanation of our main contribution, the *hullification* algorithm.

5 Proofs by Temporal Induction

The Stuff framework provides an SMT-based k-induction module. Performing a k-induction analysis of a potential state invariant P on a transition system $\langle I, T\rangle$ consists in checking the satisfiability of the $Base_k(I,T,P)$ and $Step_k(T,P)$ formulas, defined in (1), for increasing values of k, starting from a user specified $k > 1$.

$$\begin{aligned} Base_k(I,T,P) &\equiv \overbrace{I(s_0)}^{\text{Initial state}} \wedge \overbrace{\bigwedge_{i\in[0,k-2]} T(s_i,s_{i+1})}^{\text{trace of k-1 transitions}} \wedge \overbrace{\bigvee_{i\in[0,k-1]} \neg P(s_i)}^{P \text{ falsified on some state}} \\ Step_k(T,P) &\equiv \underbrace{\bigwedge_{i\in[0,k-1]} T(s_i,s_{i+1})}_{\text{trace of } k \text{ transitions}} \wedge \underbrace{\bigwedge_{i\in[0,k-1]} P(s_i)}_{P \text{ satisfied on first } k \text{ states}} \wedge \underbrace{\neg P(s_k)}_{P \text{ falsified by last state}} \end{aligned} \tag{1}$$

The base and step instances are analysed, until either a base model has been found, in which case the proof objective is falsified, a user specified upper bound for k has been reached for base and step, in which case the status of the proof objective is still undefined, or a k value has been discovered so that both formulas are unsatisfiable, which proves the validity of the objective.

In addition, this k-induction engine allows, for any n, to partition a given set of proof objectives $P = \{P_j\}$, viewed as a conjunction $P = \bigwedge_j P_j$, in three maximal subsets F_n, U_n and V_n, such that:

- elements $P \in F_n$ are such that $Base_n(I,T,P)$ is satisfiable: they are *Falsified*;
- elements $P \in U_n$ are such that $Base_n(I,T,P)$ is unsatisfiable and $Step_n(T,P)$ is satisfiable: they are *Undefined* because neither falsifiable nor n-inductive;
- elements of V_n are such that $Base_n(I,T,\bigwedge_{P\in V_n} P)$ is unsatisfiable and $Step_n(T,\bigwedge_{P\in V_n} P)$ is unsatisfiable: they are mutually n-inductive, *i.e. Valid* on the transition system.

6 Approach Overview: Combining Backward Exploration and Hull Computation

Our lemma generation heuristic builds on a backward property-directed reachability analysis. We use Quantifier Elimination (QE [19, 22, 3]) to compute successive preimages of the negation of the PO, in the spirit of [21, 10]. In our approach, the states characterized by the preimages are generated in a way such that *(i)* they satisfy the PO and *(ii)* from them, it is possible to reach a state violating the PO if certain transitions are taken. Such states will be referred to as *gray states*. This can be achieved by calculating the preimages as follows:

$$\begin{aligned} preimage_1 &= QE(s', PO(s) \wedge T(s,s') \wedge \neg PO(s')) \\ preimage_i &= QE(s', PO(s) \wedge T(s,s') \wedge preimage_{i-1}[s'/s]) \quad (\text{for } i > 1) \end{aligned} \tag{2}$$

where $QE(\vec{v}, F)$ returns a quantifier-free formula equisatisfiable to $\exists \vec{v},\ F$ and such that $FV(QE(v,F)) = FV(F) \setminus v$. The preimages themselves are assumed to be in DNF, by using [19] as a QE engine for instance.

From these preimages we extract information using two search heuristics introduced and motivated in the rest of this section and detailled in Section 7. These heuristics run in parallel, alongside the backward analysis computing the next preimage and a k-induction engine. The backward analysis is not run to a fixed point before proceeding further, it is rather meant to probe the gray state space around the negation of the PO, and feeding the potential lemma generation with the preimages as soon as they are produced.

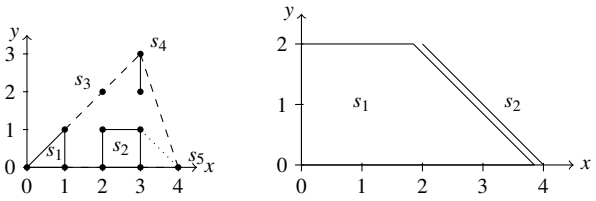
To extract information out of the preimages, at any point of the backward exploration, their disjunction is considered: it represents the gray states found so far as a union of polyhedra. The main idea underlying the potential lemma generation is to explore the ways in which those polyhedra can be grouped using convex hull calculation, thus discovering linear relations over state variables representing boundaries between convex regions of the gray state space. Since these convex boundaries enclose unauthorized states, they are negated before being sent to our k-induction engine to check their validity and try to stenghten the PO.

The PO is successfully strengthened by a set of lemmas when the set V_k of valid POs, produced by the k-induction analysis detailled in Section 5, contains the main PO at the end of a run. If the original PO is not strengthened by the potential invariants extracted from the currently available preimages, a new preimage is calculated, bringing more information. Yet, when the PO is strengthened and proved valid, it can be the case that not all elements of the valid subset V_k are needed to entail the original PO. A minimization pass inspects them one by one, discarding $l \neq PO$ from V_k if $\bigwedge(V \setminus l)$ remains k-inductive, to obtain a relatively small and readable set of lemmas.

Note that, in the backward exploration, the choice of which variables to eliminate by QE and which to keep is important. Eliminating the next state variables and keeping the current state variables is not satisfactory in the general case, as on large scale systems, many state variables might not be relevant

for the PO under investigation, and might hinder the performance of the convex hull calculation or k-induction. Therefore, the only state variables that are **not** eliminated are the ones found in the cone of influence of the PO, in their *current state* version. In particular, the system inputs are eliminated since they do not provide more information from a backward analysis point of view.

Before going into the details of the potential lemma generation algorithm, let us illustrate how computing ICHs and ECHs can actually make new numerical relations appear, using the examples given in Figure 2a and Figure 2b.



(a) ECH on integers (b) ECH on Reals

Figure 2: New relations with hulls

In Figure 2a, the gray state space of a system with two integer state variables is represented. States are represented as dots, polyhedron s_1 contains three states, polyhedra s_3 and s_5 only contain one state *etc*.

Computing exact convex hulls over these base polyhedra in the LIA fragment yields (at least) two new borders, *i.e.* potential relational invariants, pictured as dashed lines. An example of merging order is to merge s_1 with s_2, s_3 with s_4, $\{s_1,s_2\}$ with $\{s_3,s_4\}$, and $\{s_1,s_2,s_3,s_4\}$ with s_5 (1).

On a system with real valued state variables however, as shown in Figure 2b, the only case in which we will *discover* a new border by computing exact convex hulls is when one is the limit of another, as illustrated on Figure 2b. Here s_1 is made of $0 \leq x$, $0 \leq y \leq 2$ and $y+x-4<0$; s_2 is made of $0 \leq y \leq 2$ and $y+x-4=0$, so the resulting hull will be $0 \leq x$, $0 \leq y \leq 2$ and $y+x-4 \leq 0$. The information learned this way has little chance of strengthening the PO.

As will be seen in the next sections, when trying to discover new relations, ECH-based techniques work best for integer valued systems, while ICH can be beneficial for both real or integer valued systems.

6.1 A First Example

We consider a simple example called the double counter[3] with two integer state variables x and y and three boolean inputs a, b and c. Variables x and y are initialized to 0, and are both incremented by one when a is true or keep their current value when a is false. The variable x is reset if $b \vee c$ is true, and saturates at n_x. The variable y is reset when c is true and saturates at n_y, hence y cannot be reset without resetting x, and $n_x > n_y$. The proof objective is $x = n_x \Rightarrow y = n_y$. Here is a possible transition relation for such a system:

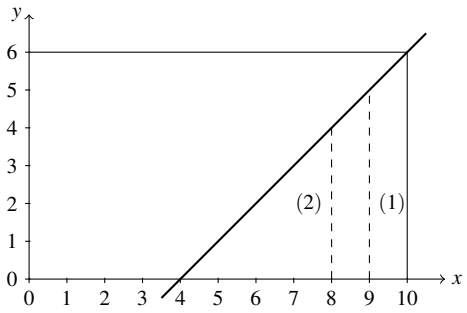
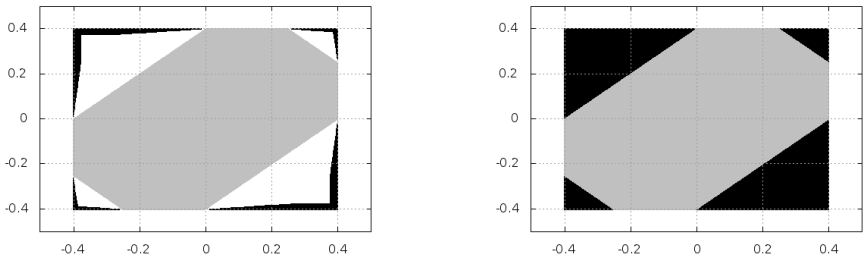


Figure 3: ECH calculation on the double counter

$$T(s,s') = \begin{array}{ll} & \big(\text{if } (b \vee c) \quad \text{then } x' = 0 \text{ else if } (a \wedge x < n_x) \quad \text{then } x' = x+1 \quad \text{else } x' = x\big) \\ \wedge & \big(\text{if } (c) \quad \text{then } y' = 0 \text{ else if } (a \wedge y < n_y) \quad \text{then } y' = y+1 \quad \text{else } y' = y\big). \end{array}$$

Let us see now how the proposed approach performs on this system when fixing $n_x = 10$ and $n_y = 6$ for instance. First, using the abstract interpretation tool presented in [23], bounds on x and y are easily discovered: $0 \leq x \leq n_x = 10$ and $0 \leq y \leq n_y = 6$, yet the PO cannot be proved with AI without

[3] Code available at http://www.onera.fr/staff-en/adrien-champion/.



(a) Two inputs voter, first preimage (b) Two inputs voter with ICH

Figure 4: Simple voting logic.

further manual intervention. So, using these range properties once k-induction has confirmed them, we start the backward property-directed analysis, which outputs a first preimage: $x = 9 \wedge 0 \leq y < 5$ (1). Unsurprisingly, it is too weak to conclude, *i.e.* its negation is not k-inductive for a small k. The next preimage is $x = 8 \wedge 0 \leq y < 4 \vee x = 9 \wedge 0 \leq y < 5$ (2) which does not allow to conclude either for the same reason. Instead of iterating until a fixed point is found, consider the graph on Figure 3. It shows the two first preimages as dashed lines which seem to suggest a relation between x and y, pictured as a bold line. This relation can be made explicit by calculating the convex hull of the disjunction of the first two preimages – since this particular system can stutter, it is the same as (2). This yields $8 \leq x \leq 9 \wedge 0 \leq y < x-4$. Note that this convex hull is an ECH, since both x and y are integers. The four inequalities are negated – they characterize gray states – and are sent to the k-induction engine. Potential invariants $\neg 8 \leq x$, $\neg x \leq 9$ and $\neg 0 \leq y$ are falsified, and the PO in conjunction with lemma $\neg y < x-4$ is found to be 1-inductive.

In fact, this PO could also be proved correct by k-induction given the bounds found by AI only, by unrolling the transition relation to the reoccurrence diameter of the system. In practice, even on such a simple system it is not possible for large values of n_x and n_y (hundreds or thousands of transitions). The performance of our technique on the other hand is not sensitive to the actual value of numerical constants: it will always derive the strengthening lemma from the first two preimages. Obviously, the time needed to compute the preimages is not impacted by changing the constants values either.
For more complex systems with preimages made of more than two polyhedra, simply merging them in arbitrary order using convex hull calculation is not robust since the resulting convex hulls would depend on the merging order, and interesting polyhedra could be missed. This idea of an exhaustive enumeration of the intermediary ECHs that can appear when merging a set of polyhedra is explored in Section 7.1.

6.2 A Second Example

Let us now consider briefly a two input, real valued voting logic system derived from the Rockwell Collins triplex voter. We will not discuss the system itself since the triplex voter is detailled in Section 8.2. It simply allows us to represent graphically the state space in a plan. The PO here is that two of the state variables, $Equalization_1$ and $Equalization_2$, range between -0.4 and 0.4. Figure 4 depicts the corresponding square. On Figure 4a we can see the first preimage calculated by our backward reachability analysis as black triangles, and the strengthening lemmas found by hand in [11] transposed to the two input system as a gray octagon. Calculating ECH on this first preimage does not allow to conclude.

A more relevant approach would be to calculate ICH. Yet, since the ICH of all the preimage polyhedra

is the $[-0.4, 0.4]^2$ square, we need to be more subtle and introduce a criterion for ICH to be actually computed between two polyhedra: they have to intersect. Intersection can be checked by a simple satisfiability test performed using a SMT solver. This check allows us to identify overlapping areas of the gray state space and to over-approximate them, while not merging disjoint areas in the gray state space explored so far. This approximation obtained through ICH resembles widening techniques used in abstract interpretation [9] in the sense that it allows to *jump* forward in the analysis iterations, yet it differs in the sense that, contrary to widening, it does not ensure termination. The only goal here is to generate potential invariants for the PO, and Figure 4 shows that the ICH yields exactly the dual, in the $[-0.4, 0.4]^2$ square, of the octagon invariant found by hand in [11]. This second idea of using ICHs to perform overapproximations will be discussed in Section 7.4.

7 Generating Potential Lemmas Through Hull Computation

We now detail two heuristics which use the preimages output by the backward analysis. The first one follows the example from Section 6.1 and consists in a thorough, exact exploration of the partitionings of the gray state space. After explaining the basic algorithm in Section 7.1, optimizations are developed in Section 7.2. A small example illustrates the method in Section 7.3. The second heuristic over-approximates areas of the gray state space in the spirit of the discussion in Section 6.2, and is discussed in Section 7.4. Both aim at discovering new relations between the state variables which once negated become potential invariants. Figure 6 provides a high level view of the different components and the way they interact internally and with the exterior.

7.1 Hullification Algorithm

The algorithm presented in this section, called *hullification*, calculates all the convex hulls that can be created by iterating the convex hull calculation on a given set of polyhedra, called the *source* polyhedra. In this algorithm we will calculate ECH as opposed to ICH to avoid both losing precision in the process and the potential combinatorial blow up – ICH are used in a different approach in Section 7.4. The difficulty here is to not miss any of the ECH that can be possibly calculated from the source polyhedra. Indeed, back to the example on Figure 2a the merging order (1) misses the ECH of s_2 and s_5 (represented as a dotted line), and consequently the potential relational lemma $y \leq -x+4$, which could have strengthened the PO.

Imperative and slightly object-oriented pseudo-code is provided on Algorithm 1. The purpose of *generatorSetMemory* is related to optimizations, discussed in Section 7.2. Please note that for the sake of clarity, the function called on line 20 is detailed separately on Algorithm 2. The hullification algorithm iterates on a set of pairs called the *generatorSet*: the first component of each of these pairs is a convex hull called the *pivot*. The second one is a set of convex hulls the pivot will be tried to be exactly merged with, called the pivot *seeds*. Note that since the ECHs are calculated by merging polyhedra two by two, our hullification algorithm cannot find convex hulls that require to merge more than two polyhedra at the same time to be exact.

The *generatorSet* is initialized such that for any couple (i, j) such that $0 \leq i \leq n$ and $i < j \leq n$, p_i is a pivot and p_j is one of its seeds, line 3. A *newgeneratorSet* is initialized with the same pivots as the *generatorSet* but without any seeds (line 8). At each iteration (line 6), a first loop enumerates the pairs of pivot and seeds of the generation set (line 9). Embedded in the first one, a second loop iterates on the seeds (line 11) and tries to calculate the ECH of the pivot and the seed (line 14) as described in

Algorithm 1 Hullification Algorithm:
$hullification(\{p_i | 0 \leq i \leq n\})$.

```
1: generatorSetMemory = {{p_i}|0 ≤ i ≤ n}
2: sourceMap = {p_i → {p_i}|0 ≤ i ≤ n}
3: generatorSet = {(p_i, S_i)|0 ≤ i ≤ n ∧ S_i = {p_k|i < k ≤ n}}
4: generatorSetMemory = generatorSetMemory ∪ {{p_i, p_j}|0 ≤ i ≤ n, i < j ≤ n}
5: fixedPoint = false
6: while (¬fixedPoint) do
7:     fixedPoint = true
8:     newGeneratorSet = {(p_i, {})|∃S, (p_i, S) ∈ generatorSet}
9:     for all ((pivot, seeds) ∈ generatorSet) do
10:        sourcePivot = sourceMap.get(pivot)
11:        for all (seed ∈ seeds) do
12:            sourceSeed = sourceMap.get(seed)
13:            source = sourcePivot ∪ sourceSeed
14:            hull = computeHull(pivot, seed)
15:            newGeneratorSet.update(pivot, newGeneratorSet.get(pivot) − seed)
16:            if (hull ≠ false) then
17:                fixedPoint = false
18:                sourceMap.add(hull → source)
19:                newGeneratorSet =
20:                    updateGenSet(hull, source, pivot, seed, newGeneratorSet)
21:            end if
22:        end for
23:    end for
24:    generatorSet = newGeneratorSet
25:    // Communication.
26: end while
27: return {p_i|∃S, (p_i, S) ∈ generatorSet}
```

Algorithm 2 Updating the *newGeneratorSet*:
$updateGenSet(hull, source, newGeneratorSet)$.

```
1: result = {}
2: for all ((pivotAux, seedsAux) ∈ newGeneratorSet) do
3:     sourceAux = sourceMap.get(pivotAux)
4:     shallAdd = (sourceAux ∪ source) ∉ generatorSetMemory &&
5:                (sourceAux ⊄ source)
6:     if (shallAdd) then
7:         result.update((pivotAux, seedsAux ∪ {hull}))
8:         generatorSetMemory.add(sourceAux ∪ source)
9:     else
10:        result.add((pivot, seeds))
11:    end if
12: end for
13: result.add((hull, {}))
14: return result
```

Section 4. If the exact merge was successful, the new ECH is added to the seeds of the pivots of the *newGeneratorSet* (line 20, detailled below) and as a new pivot with no seeds. Once the elements of the *generatorSet* have all been inspected and if new ECH(s) have been found, a new iteration begins with the *newGeneratorSet*. When no new convex hulls are discovered during an iteration, the algorithm returns all the ECHs found so far (line 27).

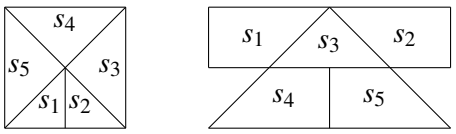
7.2 Optimizing Hullification

The hullification algorithm is highly combinatorial, and this section presents optimizations that improve its scalability.

In the hullification algorithm, the number of merge attemps increases dramatically depending on the number of elements added in the *generatorSet* at each iteration. With hullification as is, in many cases, elements of this set can be redundant, in the sense that the new hulls derived from them, if any, would be the same even though the elements are different. The key idea to reducing redundancy is to keep a link between any ECH calculated and the source polyhedra merged to create it, thereafter called the ECH source, and use this information to skip redundant ECH calculation attempts.

Consider for example Figure 5a. If we already tried to merge the ECH of source $\{s_1, s_2, s_3\}$ with the one of source $\{s_4, s_5\}$ then it is not necessary to consider trying to merge say the ECH of source $\{s_3, s_4\}$ with the one of source $\{s_1, s_2, s_5\}$. The result would be the same, *i.e.* the same ECH or a failure to merge the convex hulls exactly (the same ECH here). Note that since we are generating all the existing ECHs from the source polyhedra, this case happens every time an ECH can be calculated by merging its source in strictly more than one order, that is to say **very** often. More generally, we do not want to attempt merges of different hulls deriving from the same set of source polyhedra.

Another source of redundancy is that, when a seed is added to a pivot during the *generatorSet* update, it represents a potential merge of the union of the pivot source and the seed source. Even if this merge has not yet been considered, a potential merge of the same source might have already been added to the *generatorSet* through a different seed added to a different pivot. In this case we do not want the seed to be added. So, in order to prevent redundant elements from being added to the *generatorSet*, we introduce a memory called *generatorSetMemory*, and control how new hulls are added to the *newGeneratorSet*. For a new hull to be added to a pivot as a seed, $source(pivot) \cup source(hull) \notin generatorSetMemory$ must hold (Algorithm 2 line 4); if the hull is indeed added to the seeds of the pivot, then $generatorSetMemory += source(pivot) \cup source(hull)$ (Algorithm 2 line 8). Informally, this memory contains the sources of all the potential merges added to the generator set. This ensures that the merge of a source will never be considered more than once, and that the merges we did not consider were not reachable by successive pair-wise ECH calculation.



(a) Square example (b) Hat example

Figure 5: Hullification redundancy issues

Also, we forbid adding a seed to a pivot's seeds if the source of the latter is a subset of the former, since the result would necessarily be the seed itself (we call this (1)). Another improvement deals with *when* hullification interacts with the the rest of the framework. Since our goal is to generate potential invariants, we do not need to wait for the hullification algorithm to terminate to communicate the potential invariants already found so far. They are therefore communicated, typically to k-induction, after each *big iteration* of the algorithm (loop on Algorithm 1 line 25). This has the added benefit of launching k-induction on smaller potential invariant sets.

There is a drawback in comparing hulls using their sources: assume that two of the input (source) polyhedra p_i and p_j are such that $p_i \Rightarrow p_j$. Then the exact merge of p_i and p_j succeeds and yields the hull of source $\{p_i, p_j\}$, which is really p_j. As a consequence, the pivots of the *generatorSet* are redundant, as are their seeds and in the end the merge attempts. To avoid this, we first check the set of input polyhedra and discard redundant ones.

Last but not least, merges are also memorized in between calls to the algorithm so that we do not call the merge algorithm when considering two polyhedra we already merged during a previous call. Since hullification is called on the ever-growing disjunction of all preimages found so far, each new disjunction contains the previous one and this represents a significant improvement.

In the next subsection we illustrate hullification on a small example before introducing another po-

tential invariant generation algorithm in Section 7.4. Hullification will be illustrated on a reconfiguration logic system in Section 8.1

7.3 Hullification Example

Let us now unroll the algorithm on a simple example depicted on Figure 5b. For the sake of concision a source $\{s_1, s_2, \cdots, s_n\}$ will be written $12\cdots n$. We write generator sets in the following fashion: $\{(pivot, [seeds])\}$.
With this convention, the initial *generatorSet* is $\{(1,[2,3,4,5]),(2,[3,4,5]),(3,[4,5]),(4,[5]),(5,[])\}$. The *newGeneratorSet* for the first *big step* iteration trace is as follows:

$1,[]$	$2,[]$	$3,[]$	$4,[]$	$5,[]$			
$1,[]$	$2,[13]$	$3,[]$	$4,[13]$	$5,[13]$	$13,[]$		
$1,[]$	$2,[13]$	$3,[]$	$4,[13,23]$	$5,[13,23]$	$13,[]$	$23,[]$	
$1,[45]$	$2,[13,45]$	$3,[45]$	$4,[13,23]$	$5,[13,23]$	$13,[45]$	$23,[45]$	$45,[]$

At first *newGeneratorSet* is the same as *generatorSet* without seeds (first line of the trace). We first consider 1 as a pivot. The merge of 1 and 3 works while the other ones fail, leading to the second line of the trace. Note that 13 is not added to 1 nor 3 since $1 \subseteq 13$ and $3 \subseteq 13$ by (1). With this pivot we add three sources to the *generatorSetMemory*: 213, 413 and 513 (2). The next pivot is 2 which is merged with 3 while the merges with the other seeds fail. After the *newGeneratorSet* update we obtain the third line of the trace. Note that 23 is not added to the seeds of 1 since source 213 has already been added to the *generatorSetMemory* at (2) so $23 \cup 1 \in$ *generatorSetMemory*. Similarily, it is not added to the seeds of 13 either. Next pivot 3 cannot be merged with any of its seeds. Pivot 4 can be merged with 5 producing the fourth line of the generator trace. A new big step iteration begins during which 2 will be merged with 13 and 3 with 45 while all the other merges will fail. At the beginning of the third big step iteration the *generatorSet* is

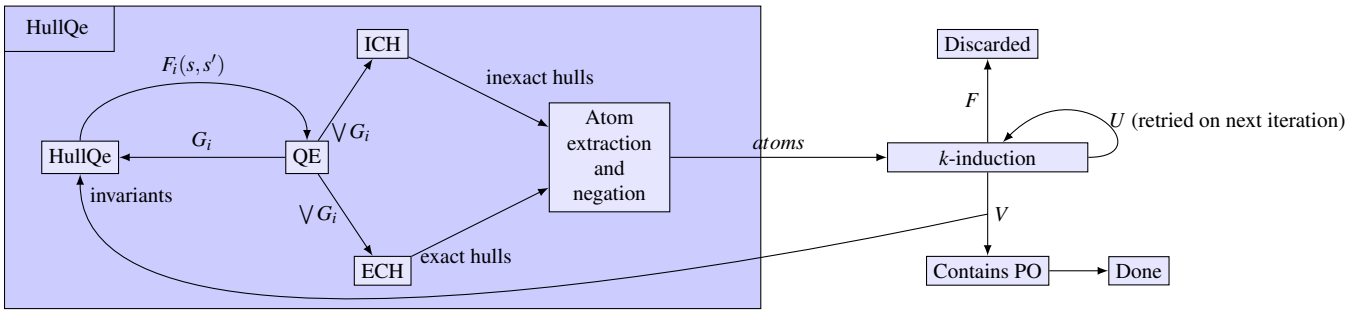
$$\{(1,[345]),(2,[345]),(3,[]),(4,[123]),(5,[123]),(13,[345]),(23,[345]),(45,[123]),(123,[345]),(345,[])\}.$$

No new hull is found and the algorithm detects that a fixed point has been reached.

7.4 Another Way to Generate Potential Invariants: ICHs

As mentioned before in Section 7.1, ECH calculation cannot do much for real state variables. We therefore propose a second approach based on Inexact Convex Hull (ICH) calculation modulo intersection as mentioned in Section 6.2, simply called ICH calculation in the rest of this paper. That is, two polyhedra will be inexactly merged if and only if their intersection is not empty. This regroups areas of the gray state space that are not disjoint and over-approximates them to make new numerical relations appear. An efficient way to check for intersection is to check the satisfiability of the conjunction of the constraints describing the two polyhedra using an SMT solver. Note that this technique is also of interest in the integer case.

For a given set of polyhedra more ICHs than ECHs can be created, in practice often a lot more. The hullification algorithm using ICHs thus tends to choke. We propose the following algorithm, only briefly described for the sake of concision.
Select a pivot in the input polyhedra set and try to find an ICH with the other ones. If an ICH with another polyhedra (source) exists, both the pivot and the source are discarded, and the ICH becomes the new pivot. Once all the merges have been tried, the pivot is put aside and a new pivot is selected in the



$$F_i(s,s') = \begin{array}{ll} PO(s) \wedge invs(s) \wedge T(s,s') \wedge invs(s') \wedge \neg PO(s') & \text{if } i = 1 \\ PO(s) \wedge invs(s) \wedge T(s,s') \wedge invs(s') \wedge G_{i-1}(s') & \text{if } i > 1 \end{array}$$

Figure 6: High level sequential description

remaining polyhedra set. When the algorithm runs out of polyhedra, it starts again on the polyhedra put aside if at least one new hull was found. If not, a fixed point has been reached and the algorithm stops. Although the intermediary ICHs computed in this algorithm depends on the order in which the pivots are selected and merged with the other polyhedra, its result does not. Indeed, the fact that two polyhedra have a non-empty intersection will stay true even if one or both of them are merged with other polyhedra. This result, as depicted in Section 6.2, is an over-approximation of disjoint areas of the gray state space.

In practice, both the ECH based hullification and the ICH calculation heuristics run in parallel, and the sets of potential invariants they output are merged before being sent to the k-induction. This allows us to combine the precision of ECHs with the over-approximation effect of ICHs. A high level view of our approach is available on Figure 6. The next section will present two examples taken from a functional chain as presented in Section 2 each illustrating the ideas introduced in this section: a reconfiguration logic system and a voting logic system.

8 Applications

In this section we discuss the results of the proposed approach on two real world examples: a reconfiguration logic and the triplex voter of Rockwell Collins.

8.1 Reconfiguration Logic

Distributed reconfiguration logic as presented in Section 2 would be best described as a distributed priority mechanism. In each redundant channel, the reconfiguration logic comes last and monitors the warning flags raised by the monitoring logic implemented earlier in the data flow. Integer timers and latches are used to confirm warnings over a number of consecutive time steps and trigger a reconfiguration. The duration of the various confirmations can vary from a few steps to hundreds or thousands of steps and are tuned by system designers to be not overly sensitive to transient perturbations, which would unnecessarily trigger reconfigurations of otherwise healthy channels, while being fast enough to ensure safety. Assuming at most two sensors, network or CPU faults, the following generic property is expected to hold for the reconfiguration mechanism: “No unhealthy channel shall be in control for more than N steps”. This property can be decomposed and instantiated per channel. However, a property such as “No more than one channel shall be in command at any time”, or “The actuator must never stay idle for more

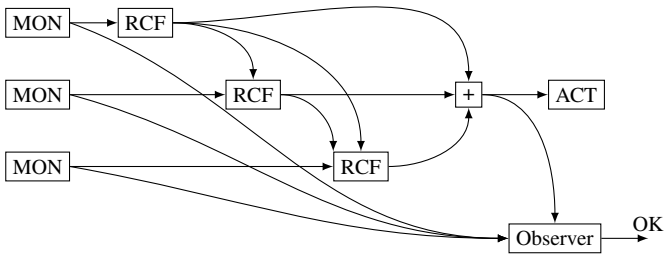


Figure 7: Reconfiguration subsystem with observer.

than m_4 steps" are more challenging because they cover all three channels simultaneously and drag many state variables in their cone of influence. For instance, the formal verification of the second property is done by assembling a model of the distributed system and by using the synchronous observer technique as shown in Figure 7. The observer uses a timer and is coded so that its output becomes true as soon as the absence of control of the actuator has been confirmed for the requested amount of m_4 consecutive steps. The proof objective on the system/observer composition is to show that the output of this observer can never be true.

The timer logic found in this system is similar to that of the toy example developed in Section 6.1, and instantiated several times, indeed a channel becoming corrupt triggers several timers with different bounds, running into each other or in parallel. Let us now see how hullification performs on this system. The first preimage does not contain enough information, since hullification generates no potential lemma which either strengthens the PO or is k-inductive by itself. The union of the first and second preimages however allows hullification to generate about 200 potential invariants. Once they are negated, k-induction invalidates most of them and indicates the PO was found (1-)inductive conjoined with about 50 lemmas after about 30 seconds of computation.

After the minimization phase described in Section 6, it turns out that only three lemmas are required. If we call $timer_i$ the integer variable used to count the time channel i is not in command for $1 \leq i \leq 3$, and $timer_o$ the timer used by the observer, the lemmas are: $\neg(timer_o - timer_i \geq m_4 - m_i - 1)$ where $1 \leq i \leq 3$. These lemmas are found no matter the values of the m_i for $1 \leq i \leq 4$. We insist on the interest of hullification here. Merging polyhedra in some single arbitrary order is too coarse and the resulting hull cannot strengthen the PO, whereas the thorough exploration generates useful lemmas.

The reconfiguration logic was also analyzed using NBac, Scade Design Verifier and Tinelli's Kind. NBac did not succeed in proving the property after 1 hour of computation. Both the Scade Design Verifier and Kind kept on incrementing the induction depth without finding a proof after 30 minutes of run time.

The invariant generation of Kind was also run on this system, and yielded a number of small theorems, but obviously not property directed and unfortunately not sufficient to strengthen the PO and prove it.

In conclusion, the proposed combination of backward analysis, hullification and k-induction allows us to complete a proof in a few seconds on a widely used avionics design pattern, where other state of the art tools fail. In addition, we see two very interesting points worth highlighting about hullification: *(i)* The PO is made (1-)inductive, implying the proof can easily and quickly be re-run and checked by any existing induction tool; *(ii)* the time needed to complete the proof does not depend on the numerical values of the system –about thirty seconds on a decent machine in practice for this system[4]. This is very

[4]Using our prototype implementation in Scala.

important for critical embedded systems manufacturers as point *(i)* means that the proofs are trustworthy, both for the industrials themselves and the certification organisms. On the other hand, point *(ii)* implies that strengthening lemmas can be very quickly generated for similar design patterns with altered numerical values, easing the integration of formal verification in the development process. Indeed, it avoids the need for an expert to manually transpose the lemmas on the new system, as can be the case for complicated and resource/time consuming proofs.

8.2 The Triplex Voter

Let us now turn to the Rockwell Collins triplex sensor voter, an industrial example of voting logic as introduced in Section 2, implementing redundancy management for three sensor input values. This voter does not compute an average value, but uses the $middleValue(x,y,z)$ function, which returns the input value, bounded by the minimum and the maximum input values (*i.e.* z if $y < z < x$). Other voter algorithms which use a (possibly weighted) average value are more sensitive to one of the input values being out of the normal bounds. The values considered for voting are *equalized* by subtracting equalization values from the inputs. The following recursive equations describe the behaviour of the voter with $X \in \{A,B,C\}$:

$$\begin{aligned}
Equalization X_0 &= 0.0 \\
Equalized X_t &= Input X_t - Equalization X_t \\
Equalization X_{t+1} &= 0.9 * Equalization X_t + \\
&\quad 0.05 * (Input X_t + ((Equalization X_t - VoterOutput_t) - Centering_t)) \\
Centering_t &= middleValue(Equalization A_t, Equalization B_t, \\
&\qquad Equalization C_t) \\
VoterOutput_t &= middleValue(Equalized A_t, Equalized B_t, Equalized C_t)
\end{aligned}$$

The role of the equalization values is to compensate offset errors of the sensors, assuming that the middle value gives the most accurate measurement.

We are interested in proving Bounded-Input Bounded-Output (BIBO) stability of the voter, which is a fundamental requirement for filtering and signal processing systems, ensuring that the system output cannot grow indefinitely as long as the system input stays within a certain range. In general, it is necessary to identify and prove auxiliary system invariants in order to prove BIBO stability.

So, we want to prove the stability of the system, *i.e.* we want to prove that the voter output is bounded as long as the input values differ by at most the maximal authorized deviation $MaxDev$ from the true value of the measured physical quantity represented by the variable $TrueValue$. In our analysis, we fixed the maximal sensor deviation to 0.2, a value that domain experts gave us as typical value in practical applications. It is staightforward to prove that the system is stable if the equalization values are bounded.

When applied to Rockwell Collins **tri**plex sensor voter, our prototype implementation manages to prove the PO in less than 10 seconds by discovering that $-0.9 \leq \sum_{i=1}^{3} Equalization_i \leq 0.9$ is a strengthening lemma, using ICH calculation. Again, the time taken to complete the proof does not depend on the system numerical constants, and the strengthened PO is (1-)inductive. We insist on the importance of these characteristics for both industrials and certification organisms: the proof is trustworthy and can be redone easily for similar, slightly altered designs.

The stability of the system without fault detection nor reset was already proven in [11], but the necessary lemmas had to be found by hand after the Scade Design Verifier, Kind as well as Astrée

(which was run on C-Code generated from the Lustre source) failed at automatically verifying the BIBO property.

9 Framework and Implementation

Our actor oriented collaborative verification framework [8] called *Stuff* is composed of several elements: the k-induction engine, the abstract interpreter, the backward analysis, ICH calculation and ECH hullification. They can all evolve in parallel and communicate.

Stuff is written in Scala except for the abstract interpreter, written in OCaml. We implemented the backward analysis and the two heuristics presented in this paper using the QE algorithm from [19] modified to handle integers or reals whith booleans. The underlying projections of [19] are performed by the Parma Polyhedra Library [1], also used for convex hull computation. Stuff can use any SMT lib 2.0 [2] compliant solver thanks to the Assumptio[5] actor oriented SMT solver wrapper. In practice, the backward analysis and the heuristics use Microsoft Research Z3 [20] and MathSat 5 [14] by the University of Trento.

A run of the framework in the default configuration begins by a preprocessing phase using abstract interpretation with intervals as abstract domains in order to infer bounds on the state variables. This provides an over-approximation of the reachable state space which once verified by k-induction is propagated to all the other elements of the framework. The rest of the analysis follows the approach discussed in Section 6 with the backward analysis feeding preimages to both ECH hullification and ICH calculation. They in turn feed potential invariants to the k-induction engine which detects real invariants and check if they strengthen the property as described in Section 6. In this setting, even if our approach does not consider the initial states, it benefits from the over-approximation of the AI preprocessing phase, which takes into account the initial states but not the PO. The AI results also enhance the quality of the output and the overall performance of the incremental k-induction engine.

10 Conclusion

In this paper, the authors presented two automatic and property directed lemma generation heuristics, which operate on preimages of the negation of the proof objective obtained by a backward exploration, itself powered by quantifier elimination.

The first heuristic originality lies in the thorough exploration of a set of possible convex partitionings of the gray state space by exact convex hull calculations. This exploration, called *hullification*, is performed incrementally, as soon as new preimages containing new information about the gray state space, are computed by the backward analysis. As illustrated on the reconfiguration logic example, the blowup inherent to the exploration of the partitionings is avoided thanks to the optimizations discussed in this paper and far outperforms other available tools.

The second heuristic over-approximates disjoint areas of the gray state space by accepting inexact hulls when the candidate polyhedra intersect. It performs very well in the Rockwell Collins Triplex Sensor Voter experiments, allowing to conclude a proof none of the other state of the art tools could conclude.

These results, obtained with the prototype implementation of the proposed method, are of interest in our application field. Indeed, they allow to discover strengthening lemmas, in reasonable time, for

[5] https://cavale.enseeiht.fr/redmine/projects/assumptio

essential safety properties of widely used fault tolerance design patterns at model level, a task which has proved difficult to achieve using other techniques such as AI or k-induction with manual analysis of failed proofs.

Future work include further reflexion on systems mixing integers and reals and on heuristics using preimages from the backward analysis. Also, the authors think that when hullification cannot find strengthening lemmas, it can still provide interesting starting points for template based techniques and experiments have been started in this direction. Outside of the proposed approach, the authors believe in a multi method approach and will continue to experiment in this direction: work on an implementation of PDR [5, 13] adapted to numerical systems is in progress. It was observed that PDR is able to discover range lemmas similar to those found using interval based AI, while being able to conclude inductive proofs, and the cooperation of hullification and PDR is being studied. The long term goal is to refine and bridge the verification techniques developed for precise parts of the functional chains (voting, reconfiguration logic and numerical stability for control laws) to obtain a methodology and tool support suitable for end-to-end verification of avionics software at model level.

References

[1] R. Bagnara, P. M. Hill & E. Zaffanella (2008): *The Parma Polyhedra Library: Toward a Complete Set of Numerical Abstractions for the Analysis and Verification of Hardware and Software Systems*. *Science of Computer Programming* 72(1–2), pp. 3–21. Available at http://doi.acm.org/10.1016/j.scico.2007.08.001.

[2] C. Barrett, A. Stump, C. Tinelli, S. Boehme, D. Cok, D. Deharbe, B. Dutertre, P. Fontaine, V. Ganesh, A. Griggio, J. Grundy, P. Jackson, A. Oliveras, S. Krstić, M. Moskal, L. de Moura, R Sebastiani, T. D. Cok & J. Hoenicke (2010): *C.: The SMT-LIB Standard: Version 2.0*.

[3] N. Bjørner (2010): *Linear Quantifier Elimination as an Abstract Decision Procedure*. In J"urgen Giesl & Reiner H"ahnle, editors: *Automated Reasoning*, *Lecture Notes in Computer Science* 6173, Springer Berlin / Heidelberg, pp. 316–330. Available at http://dx.doi.org/10.1007/978-3-642-14203-1_27.

[4] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux & X. Rival (2003): *A Static Analyzer for Large Safety-Critical Software*. In: *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI'03)*, ACM Press, San Diego, California, USA, pp. 196–207. Available at http://dl.acm.org/citation.cfm?id=781153.

[5] A. R. Bradley (2011): *SAT-Based Model Checking without Unrolling*. In: *VMCAI*, pp. 70–87. Available at http://dx.doi.org/10.1007/978-3-642-18275-4_7.

[6] A. R. Bradley & Z. Manna (2006): *Verification Constraint Problems with Strengthening*. In: *ICTAC*, pp. 35–49. Available at http://dx.doi.org/10.1007/11921240_3.

[7] P. Caspi, D. Pilaud, N. Halbwachs & J. Plaice (1987): *Lustre: A Declarative Language for Programming Synchronous Systems*. In: *POPL*, pp. 178–188. Available at http://doi.acm.org/10.1145/41625.41641.

[8] A. Champion, R. Delmas, P.L. Garoche & P. Roux (2011): *Towards Cooperation of Formal Methods for the Analysis of Critical Control Systems*. In: *SAE Int. J. Aerosp.*, pp. 850–858. Available at http://dx.doi.org/10.4271/2011-01-2558.

[9] P. Cousot & R. Cousot (1977): *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints*. In: *POPL*, pp. 238–252. Available at http://doi.acm.org/10.1145/512950.512973.

[10] Werner Damm, Stefan Disch, Hardi Hungar, Swen Jacobs, Jun Pang, Florian Pigorsch, Christoph Scholl, Uwe Waldmann & Boris Wirtz (2007): *Exact State Set Representations in the Verification of Linear Hybr id*

Systems with Large Discrete State Space. In: *ATVA*, pp. 425–440. Available at http://dx.doi.org/10.1007/978-3-540-75596-8_30.

[11] M. Dierkes (2011): *Formal Analysis of a Triplex Sensor Voter in an Industrial Context*. In G. Salaün & B. Schätz, editors: *Proceedings of the 16th edition of FMICS*, *LNCS* 6959, Springer, pp. 102–116. Available at http://dx.doi.org/10.1007/978-3-642-24431-5_9.

[12] M. Dierkes & D. Kästner (2012): *Transferring Stability Proof Obligations from Model Level to Code Level*. Available at http://www.erts2012.org/Site/0P2RUC89/5C-1.pdf.

[13] N. Een, A. Mishchenko & R. Brayton (2011): *Efficient Implementation of Property Directed Reachability*. Available at http://dl.acm.org/citation.cfm?id=2157675.

[14] A. Griggio (2012): *A Practical Approach to Satisfiability Modulo Linear Integer Arithmetic*. *JSAT* 8, pp. 1–27. Available at http://jsat.ewi.tudelft.nl/content/volume8/JSAT8_1_Griggio.pdf.

[15] B. Jeannet (2003): *Dynamic Partitioning in Linear Relation Analysis: Application to the Verification of Reactive Systems*. *Formal Methods in System Design* 23(1), pp. 5–37. Available at http://dx.doi.org/10.1023/A:1024480913162.

[16] T. Kahsai, Y. Ge & C. Tinelli (2011): *Instantiation-Based Invariant Discovery*. In M. Bobaru, K. Havelund and G. Holzmann & R. Joshi, editors: *Proceedings of the 3rd NASA Formal Methods Symposium (Pasadena, CA, USA)*, *Lecture Notes in Computer Science* 6617, Springer, pp. 192–207. Available at http://dl.acm.org/citation.cfm?id=1986326.

[17] T. Kahsai & C. Tinelli (2011): *PKind: A parallel k-induction based model checker*. In: *PDMC*, pp. 55–62. Available at http://dx.doi.org/10.4204/EPTCS.72.6.

[18] Kenneth L. McMillan (2008): *Quantified Invariant Generation Using an Interpolating Saturation Prover*. In: *TACAS*, pp. 413–427. Available at http://dx.doi.org/10.1007/978-3-540-78800-3_31.

[19] D. Monniaux (2008): *A Quantifier Elimination Algorithm for Linear Real Arithmetic*. In: *LPAR*, pp. 243–257. Available at http://dx.doi.org/10.1007/978-3-540-89439-1_18.

[20] L. M. de Moura & N. Bjørner (2008): *Z3: An Efficient SMT Solver*. In: *TACAS*, pp. 337–340. Available at http://dx.doi.org/10.1007/978-3-540-78800-3_24.

[21] L. M. de Moura, H. Rueß & M. Sorea (2003): *Bounded Model Checking and Induction: From Refutation to Verification (Extended Abstract, Category A)*. In: *CAV*, pp. 14–26. Available at http://doi.acm.org/10.1007/978-3-540-45069-6_2.

[22] T. Nipkow (2010): *Linear Quantifier Elimination*. *J. Autom. Reasoning* 45(2), pp. 189–212. Available at http://dx.doi.org/10.1007/s10817-010-9183-0.

[23] P. Roux, R. Delmas & P.L. Garoche (2010): *SMT-AI: an Abstract Interpreter as Oracle for k-induction*. *Electr. Notes Theor. Comput. Sci.* 267(2). Available at http://dx.doi.org/10.1016/j.entcs.2010.09.018.

[24] P. Roux, R. Jobredeaux, P.L. Garoche & E. Féron (2012): *A generic ellipsoid abstract domain for linear time invariant systems*. In: *Proceedings of the 15th ACM international conference on Hybrid Systems: Computation and Control*, pp. 105–114. Available at http://doi.acm.org/10.1145/2185632.2185651.

[25] M. Sheeran, S. Singh & G. Stålmarck (2000): *Checking Safety Properties Using Induction and a SAT-Solver*. In: *FMCAD*, pp. 108–125. Available at http://dx.doi.org/10.1007/3-540-40922-X_8.

Property-based Code Slicing for Efficient Verification of OSEK/VDX Operating Systems*

Mingyu Park Taejoon Byun Yunja Choi
School of Computer Science and Engineering
Kyungpook National University
Deagu, Korea
pqrk8805@gmail.com bntejn@gmail.com yuchoi76@knu.ac.kr

Testing is a de-facto verification technique in industry, but insufficient for identifying subtle issues due to its optimistic incompleteness. On the other hand, model checking is a powerful technique that supports comprehensiveness, and is thus suitable for the verification of safety-critical systems. However, it generally requires more knowledge and cost more than testing. This work attempts to take advantage of both techniques to achieve integrated and efficient verification of OSEK/VDX-based automotive operating systems. We propose property-based environment generation and model extraction techniques using static code analysis, which can be applied to both model checking and testing. The technique is automated and applied to an OSEK/VDX-based automotive operating system, Trampoline. Comparative experiments using random testing and model checking for the verification of assertions in the Trampoline kernel code show how our environment generation and abstraction approach can be utilized for efficient fault-detection.

1 Introduction

The operating system is the core part of automotive control software; any malfunction can cause critical errors in the automotive system, which in turn may result in loss of lives and assets. Testing has been widely used as a systematic and cost-effective safety analysis/assurance method [6, 18], but its optimistic incompleteness often misses critical problems and cannot guarantee the "absence of wrong behavior". As an alternative and complimentary technique, model checking [10, 16] has been drawing attention from both academia and industry.

Model checking is a comprehensive formal verification technique, suitable for functional safety analysis. It can effectively identify subtle issues, such as process dead lock, illegal behavior, and starvation, but may require more resources and domain knowledge. In particular, the use of model checking faces the following challenges:

1. The size of model/code to be verified needs to be minimized to avoid state-space explosion.
2. Modeling of the environment, such as user tasks and hardware environment, is necessary and critical for embedded software.

Since an operating system is a reactive system responding to environmental stimuli, the correctness of its behavior needs to be analyzed with respect to the behavior of its environments. A non-deterministic environment is typically used to over-approximate actual behavior, but it is often too expensive in model

*This work was partially supported by the IT R&D program of MKE/KEIT [10041145, Self-Organized Software-platform(SOS) for welfare devices] and the National Research Foundation of Korea Grant funded by Korean Government (2012R1A1A4A01011788).

checking. The difficulty and importance of defining a *good* environment model has been addressed in a number of previous works [22, 23, 12, 25, 17, 20].

We note that these two problems apply to both model checking and testing. Though the level of comprehensiveness differs, both techniques rely on automated search techniques that are initiated by environmental stimuli. This is called environment model in model checking and test scenario in testing. This work anticipates that the efficiency of automated verification techniques depends on the modeling of the environment and proposes an application of property-based code slicing [24] for automatically generating an environment model using the data/function dependency analyzed from the operating system kernels. The goal is to construct a valid and comprehensive usage model of the operating system with minimal dependency on the kernel code.

Our approach extracts functions that have a direct dependency on a given property to be verified and generates non-deterministic function-call sequences by imposing (1) external constraints from the OSEK/VDX standard [1] for automotive operating systems, and (2) internal constraints identified from the function call structure of the operating system kernel. The external constraints are manually identified from the specifications of the standard and are imposed on the initially random sequence of function calls. The internal constraints are imposed by identifying the top-level functions using backward slicing from a given property and by computing the cone-of-influence from each top-level function using forward slicing. The Environment model is defined as an arbitrary sequence of calls of those extracted functions. The operating system kernel is also abstracted as a collection of extracted functions and its relevant code required for pre-processing them. This procedure reduces the size of the verification target and minimizes the behavior of the environment model. The extraction and model construction process is automated with the aid of the static analysis tool Understand [4].

The approach and the tool are applied to the verification of safety properties of the Trampoline operating system [3], which is an open source automotive operating system compliant with OSEK/VDX. Environment models are generated using the assertions identified from the kernel code, and the kernel code itself is reduced by including only those extracted functions and their relevant code. The environment model is used to model-check/test the abstract code using CBMC [9] and random testing. We compare their fault-detection capability, their comprehensiveness in terms of code coverage, and their efficiency in terms of resource consumption.

The remainder of this paper is organized as follows. Section 2 briefly discusses related work and Section 3 provides the motivation for our work. Section 4 provides an overview of our approach and Section 5 presents the methods and the process for the automated environment generation technique. Section 6 explains the environment settings for the collaborative verification, followed by experimental results and the evaluation using Trampoline OS as a case example in Section 7. We conclude in Section 8.

2 Related Work

Environment modeling for efficient model checking has been an active research issue [22, 23, 12, 25, 17, 20]. Reference [20] is one of the earliest works concerning environment assumptions in verification. It introduced the *observer* concept to represent assumptions about the environment. The approaches for assumption generation were developed further in [12, 23, 13, 19]. Reference [23] automatically generates the environment of Java programs from the specifications written by a user. [19, 13] are concerned about automatic partitioning, learning, or minimizing assumptions for compositional verification. None of them considers environment generation for both model checking and testing.

Several specification-based environment generation methods exist: [21] uses ADL to define proto-

cols of Java components and constructs an environment for the ADL specification. [11] describes environmental assumptions in LTL and uses them to filter a universal environment, which is adopted in our approach to constrain the non-deterministic initial task model. Reference [25] automatically generates scripts in PROMELA from environment models for OSEK/VDX-based operating systems that are modeled in UML diagrams. Their approach, however, models all basic objects in OSEK/VDX using UML class diagrams and state diagrams, from which all combinations of deterministic environments are generated and verified individually. The models are then used to automatically generate exhaustive test cases for the conformance testing of OSEK/VDX-compliant operating systems [8]. Their approach assures the exhaustiveness of test cases, but the scalability issue remains, as the number of test cases may increase exponentially.

Program slicing [24] has been a popular technique for reducing verification complexity for both model checking and testing. References [5] and [14] use slicing algorithms to explicitly detect $def - use$ associations that are affected by a program change for efficient regression testing. Reference [7] performs program slicing for C programs with respect to the alarms generated from value analysis. [15] integrates aggressive program slicing and a proof-based abstraction-refinement strategy for wireless cognitive radio systems. It is a representative example of using program slicing and bounded model checking for embedded software, but the slicing is integrated into the model checking process, and is thus not suitable for application in testing.

3 Background

3.1 OSEK/VDX

OSEK/VDX is a joint project of the automotive industry, which aims at establishing an industry standard for an open-ended architecture for distributed control units in vehicles [1]. The aim of OSEK/VDX is to provide standard interfaces independent of application, hardware, and network, and ultimately, to save the development costs for non-application related aspects of control software. It is specialized for automotive control systems, removing all undesired complexities such as dynamic memory allocation, circular waiting for resources, multi-threading, and so on. Since its target system is safety-critical, it strictly prohibits uncontrolled dynamic behavior of the system.

Conformation testing is a standard verification method for the certification of OSEK/VDX-based operating systems. However, conformation testing suites are typically insufficient to identify safety problems. As OSEK/VDX explicitly specifies more than 26 basic APIs, thorough conformation testing would require at least $26 \times 2 \times 3$ test cases even if we assume two arguments per API and only boundary values for the arguments are chosen. The possible number of execution sequences for these $26 \times 2 \times 3$ test cases would rise to 156 factorials, a large number to be tested in practice.

3.2 Trampoline

Trampoline [3] is an open source, real-time operating system compliant with OSEK/VDX version 2.2.3. It is developed in ANSI C and can be ported to various hardware platforms such as Arm, POSIX, PPC, AVR, HCS12, C166, etc. Since it also supports POSIX, it can be test-run on a UNIX/Linux environment before being ported to an actual operational environment. As its target platform varies, its platform-dependent part is clearly structured in a separate module that combines with the kernel module at compile time. Access to the hardware-specific part is abstracted using *extern* variables and macros so that the main control logic does not need to be aware of the specific hardware feature. As illustrated in

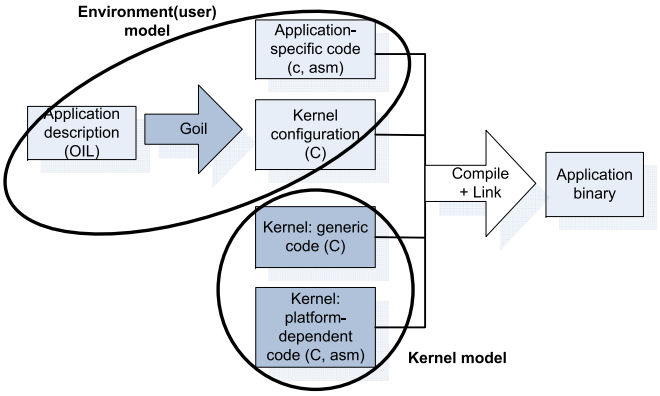


Figure 1: Components of Trampoline

Figure 1, development of an automotive software using Trampoline requires four components; (1) application source code, (2) kernel configuration generated from configuration description written in OIL (OSEK Implementation Language) using the Goil compiler, (3) generic OS kernel code compliant with the OSEK/VDX standard, and (4) platform-dependent kernel code. The generic OS kernel code implements services for task management, resource management, interrupt handling, and event/counter/alarm management, providing corresponding APIs.

3.3 Model checking using CBMC

Formal verification methods based on model checking [10] are an effective technique for identifying subtle issues in software safety which is particulary important for embedded systems. Current technological advances in model checking enable engineers to directly apply the technique to program source code, removing the manual model construction process. CBMC [9] is one of these model checking tools, which is capable of verifying almost full ANSI C. It can be used to verify buffer overflows, pointer safety, exceptions and user-specified assertions. Furthermore, it can check ANSI C and C++ for consistency with other languages, such as Verilog. The main advantage is that it is completely automated and generates counterexample traces when a property in question is refuted.

As with any other model checking tool, CBMC also suffers from the problem of scalability. When applied to the Trampoline kernel as a whole with an arbitrary sequence of API calls, for example, it ran out of memory for checking one assertion on a PC with 3GB of memory.

4 Overall Approach

Comprehensive verification, required by functional safety analysis, is too costly to be applied in practice. Reducing the cost while maintaining comprehensiveness is a challenging, but crucial task. Our approach attempts to achieve this goal with the following three strategies:

1. Property-based environment generation: An environment of the operating system kernel is automatically generated using static code analysis for a given safety property.
2. Property-based abstraction: The operating system kernel is abstracted by extracting only the code relevant to a given property.

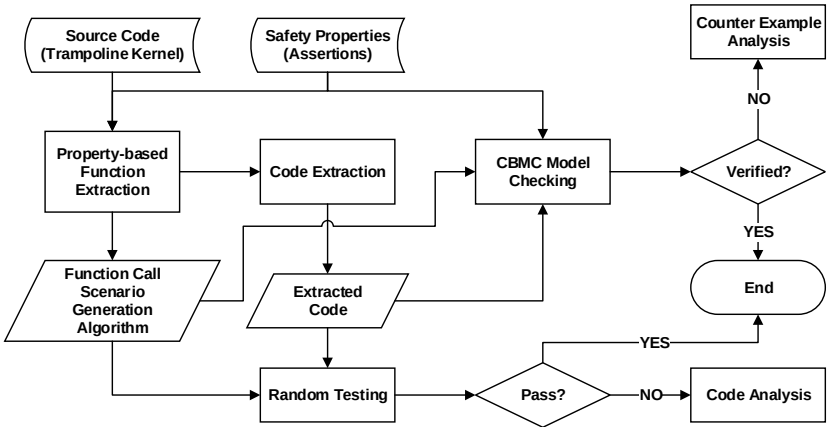


Figure 2: Collaborative verification approach

3. Collaborative verification using model checking and testing: Both model checking and testing are used complementarily for the verification of the abstract kernel code under the generated environment model.

Cost reduction is achieved through property-based environment generation and code abstraction. The efficiency of verification is increased by taking advantage of both verification techniques. Figure 2 is an overview of the suggested collaborative verification approach. Our approach uses both model checking and testing to complimentarily utilize their different capabilities when only limited resources are available.

5 Environment Generation

A straightforward way to include all possible task interactions with the operating system is to model the task with strongly connected states, where each state represents an API call to the kernel and each transition between states is not guarded. However, this includes too many spurious and/or impossible behaviors and increases the cost for verification as well as counterexample analysis; if 26 APIs are provided by the operating system, the task model would have at least 26 strongly connected states. Our approach tries to minimize unnecessary verification cost by using property-based extraction of dependent functions.

5.1 Abstraction through static code analysis

Given a property, we first extract the variables specified in the property, which is called *Verification Target Variables*, and identify all the variables that are used to define the *Verification Target Variable*, called *Extended Verification Target Variable*. Then, functions modifying those *Extended Verification Target Variables*, called $End\_Level\_Functions$, are extracted. The prototypes of the $End\_Level\_Functions$ are used to construct an end-level environment model. The corresponding end-level abstract kernel code consists of all the $End\_Level\_Functions$ and their dependent code. The $Root\_Level\_Functions$ are identified by performing backward reachability analysis from each *End_Level_Function*. The prototypes of the $Root\_Level\_Functions$ are used to construct a root-level environment model. Its corresponding abstract kernel code is identified by performing forward reachability analysis from each $Root\_Level\_Function$.

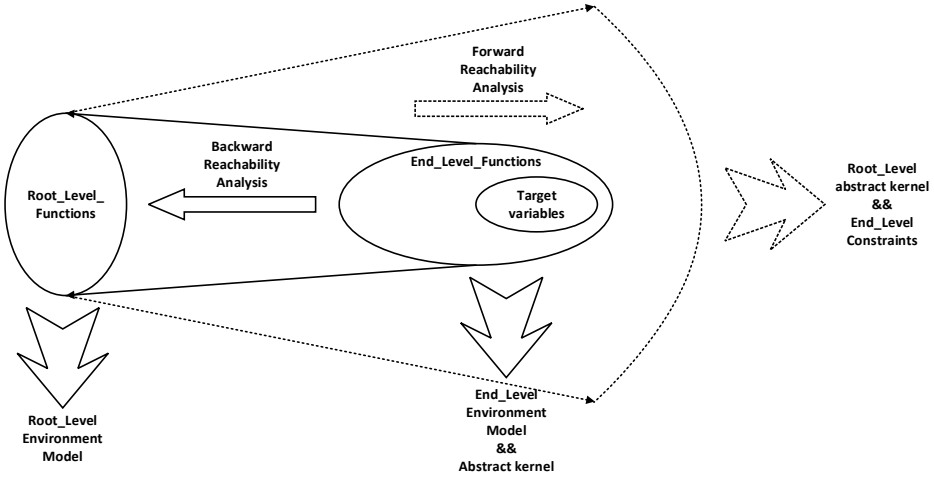


Figure 3: Backward and forward reachability analysis for environment generation

The result of forward reachability analysis is also used to identify constraints for the end-level environment model.

Definition 1 *Property-related variables:*

1. *A Verification Target Variable is a variable that appears in the property specification.*
2. *An Extended Verification Target Variable is a variable that a Verification Target Variable depends on.*

Definition 2 *Classification of functions:*

1. *An End_Level_Function is a function that directly modifies, sets, or uses an Extended Verification Target Variable.*
2. *A Root_Level_Function is an API that is a terminal node of the called-by graph of an End_Level_Function.*

From this process, we extract two types of functions for constructing different levels of environment models: (1) functions for root-level environments, and (2) functions for end-level environments. Figure 3 shows the conceptual diagram for the whole process.

For a simple example, if a property in question is

$$Property_1 : \ assert(tpl\_fifo\_rw[tpl\_h\_prio].size > 0),$$

then we first identify *Extended Verification Target Variables* and *End_Level_Functions* for *tpl_fifo_rw* and *tpl_h_prio*. The identified set of $End\_Level\_Functions$ for the variable tpl_h_prio is $\{$ tpl_get_proc, $tpl\_put\_preempted\_proc$, $tpl\_put\_new\_proc$, $tpl\_schedule\_from\_running$ $\}$ in the Trampoline kernel. An end-level environment model is constructed as non-deterministic calls to those end-level functions and its corresponding abstract kernel encompasses all the identified $End\_Level\_Functions$ and their dependent code. We then identify *Root_level_Functions* for each of the *End_Level_Functions*; For examples, $\{ReleaseResource$, $Schedule$, $ActivateTask$, $SetEvent$, $TerminateTask$, $ChainTask$, $WaitEvent$, $StartOS\}$ are $Root\_Level\_Functions$ for tpl_get_proc, which are identified from its called-by graph. A root-level environment model consists of non-deterministic calls to those API functions and its corresponding abstract kernel encompasses all the identified root level functions and their dependent code.

5.2 Implementation

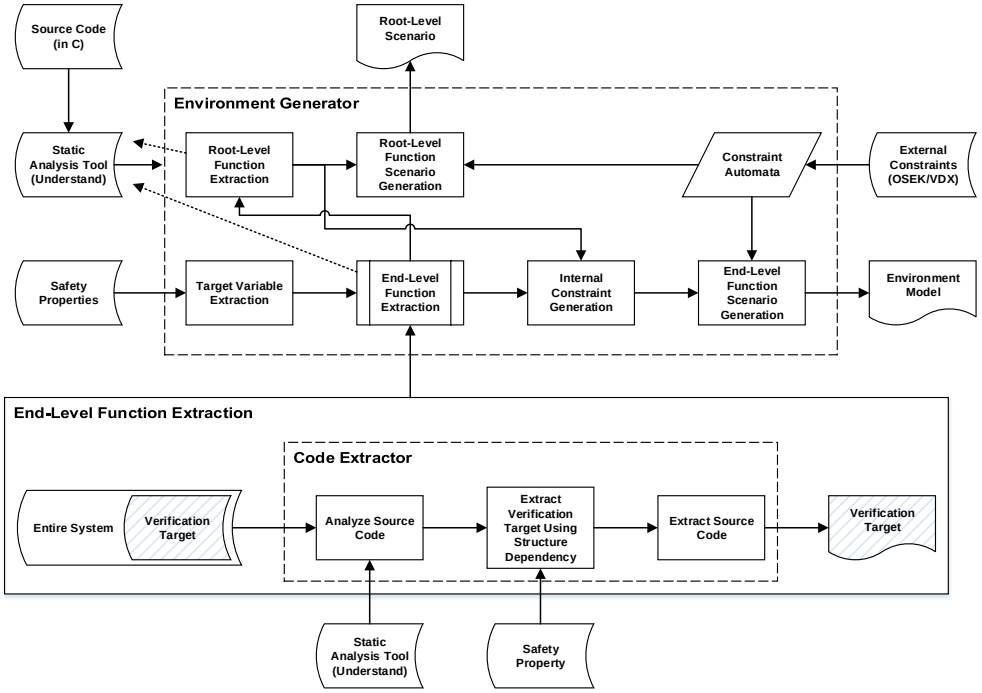


Figure 4: Environment Generation

The suggested approach was implemented and fully automated. Figure 4 shows the overall structure of the automation. The source code of the Trampoline OS is analyzed by the static analysis tool Understand [4], which creates a data repository of the analysis results from which information on variable/function dependencies can be extracted using a C plug-in. The environment generator first extracts target variables from the properties and then extracts Extended Target Variables and End-Level Functions by analyzing dependency relations among variables and functions. Root-Level Functions are then extracted from the called-by graph for each End-Level Function. A Root-Level Function Scenario is generated as an arbitrary sequence of function calls of Root-Level Functions, which complies with the external constraints from the OSEK/VDX standard. Finally, the environment model is generated as an arbitrary sequence of End-Level Functions that complies with both the external and the internal constraints. The internal constraints are partial-order relations among End-Level Functions, which are generated from each Root-Level Function.

The last step is the property-based abstraction of the original code. Since the environment generation step identifies all necessary End-Level Functions modifying the Verification Target Variables together with the ordering relation among them in call sequences, verification requires only the source code of those End-Level Functions plus codes for preprocessing them. Therefore, for each safety property, verification is performed using the environment model generated with the tool and the property-based abstract code.

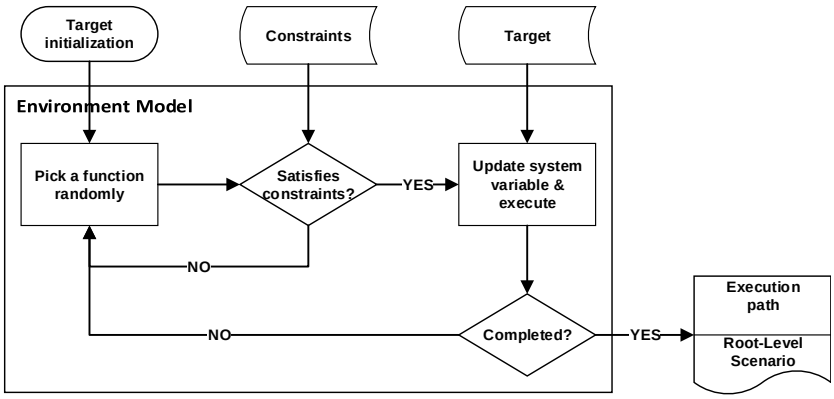


Figure 5: Scenario generation process for random testing

6 Setting up Environments for Collaborative Verification

CBMC and testing require different settings for their verification environments even with the same set of *End_Level_Functions*; CBMC requires only the algorithm of non-deterministic function calls. Random testing (both root-level and end-level), however, requires explicit function call sequences generated from a given environment model. Figure 5 shows the process for generating such function call sequences, which can be applied to both End-Level and Root-Level environment models. It repeats the selection and checking process by arbitrarily selecting a function and checking whether the selected function satisfied the constraints or not. We have implemented an OSEK/VDX simulator to check the external constraints in the process of root-level sequence generation. Internal constraints are identified by call graph analysis. Details are described in the following two sub-sections.

We note that the checking the constraints is not necessary to ensure the correctness of the scenario generation, but it is essential for making the verification efficient, since otherwise the verification produces too many false errors.

6.1 Root-Level Scenario Generation

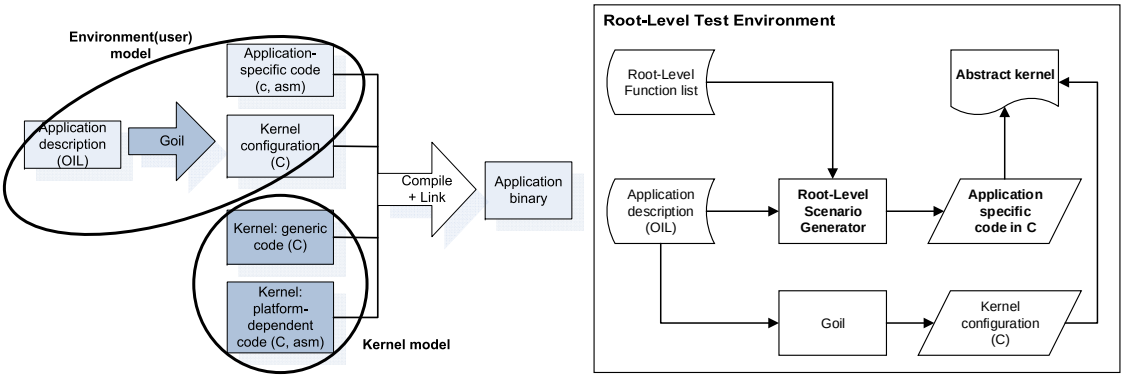


Figure 6: Root-Level scenario and the testing environment

Root-Level scenario generation is based on the scenario generation process illustrated in Figure 5.

A Root-Level Function Scenario is an arbitrary sequence of function calls of Root-Level Functions, and it complies with the external constraints from the OSEK/VDX standard. Figure 6 shows the Root-Level test environment for the Trampoline OS. Since an OIL file is required for testing, an OIL file is specified as an input, which is compiled with the extracted kernel code and the random sequence of Root-Level Functions generated from our Root-Level scenario generator.

There are two things to be done before implementing a scenario generator. First, every constraint specified in the OSEK/VDX standard should be identified. Second, an OSEK/VDX Simulator - an abstract OSEK/VDX model - should be implemented in order to trace all the changes and fully observe constraints.

6.1.1 Identification of external constraints

The OSEK/VDX standard explicitly specifies constraints among the APIs. The *description* column of Figure 7 lists some of the constraints manually identified from the standard. These constraints are represented as pre-conditions with respect to other APIs. For example, the API function *TerminateTask* can be called only if the task has been activated either by *ActivateTask* or *ChainTask*. Therefore, we set $\{ActivateTask, ChainTask\}$ as preconditions of *TerminateTask*. Figure 7 shows a couple of preconditions of other API functions.

API function	Pre-condition	Description
StartOS	!StartOS	*StartOS* is executed only once at the beginning.
WaitEvent	!WaitEvent || SetEvent	Since the state of calling task is set to *WAITING* after calling *WaitEvent*, it cannot take any action until *SetEvent* is called. So *WaitEvent* can only be called when it has first appeared, or every waiting task has been set by *SetEvent*.
ReleaseResource	GetResource	*ReleaseResource* can be called when the resource is already held.
TerminateTask	ActivateTask || ChainTask	*TerminateTask* can be called only when there is running task, which is activated by *ActivateTask* or *ChainTask*.
Schedule	!GetResource	*Schedule* can be called only when there is held resources, which is held by *GetResource*
ChainTask	!GetResource	*ChainTask* can be called only when there is held resources, which is held by *GetResource*

Figure 7: Constrant list extrected from OSEK/VDX spec

Identified constraints are then imposed in the Root-Level scenario generation process illustrated in Figure 5. Figure 8 is an example algorithm of the constraint checking.

```
bool canReleaseResource( Resource r1 ){
        if ( running task doesn't exist )
                return false;
        else if ( running task is not holding resource r1 )
                return false;
        else    // when running task is holding the resource r1
                return true;
}
```

Figure 8: ReleaseResource constraint checker

6.1.2 OSEK/VDX Simulator

To fully consider all the identified constraints, it is necessary to trace changes that previous function calls have made. For example, if *ActivateTask(t1)* is chosen as the first Root-Level Function in a scenario, task *t1* should be marked as *READY* task, for further scenario validation. This process is fully automated by implementing an OSEK/VDX simulator.

The OSEK/VDX simulator traces run-time information such as list of resources, list of events, list of task models, reference to running task, ready queue (priority queue), and waiting queue. It provides Root-Level Function calls just like OSEK/VDX APIs. When one of these procedures is called, it simulates the behavior of OSEK/VDX. The simulator includes task model, task scheduler, event management, and resource management. Figure 9 shows the overall process of the OSEK/VDX simulator.

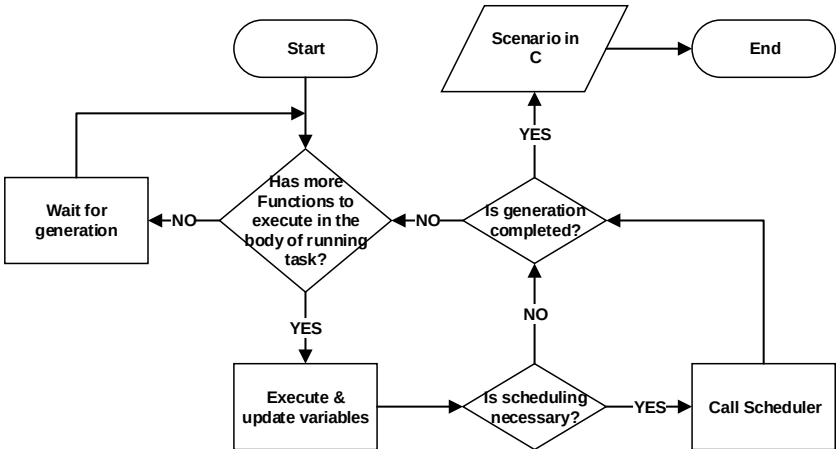


Figure 9: OSEK/VDX simulator

A randomly chosen function is added to the body of the running task by a module called Scenario Generator. When there is a function in the body of the running task that has not been executed by the simulator, it executes the function. If scheduling is necessary, it calls the scheduler and executes all the functions that already exist in the body of the preempted task, when preemption occurs. This process is repeated until there is no function left to execute, and the Scenario Generator is requested to generate another function randomly.

Whether scenario generation has been completed or not can be determined by checking all tasks that are initially generated from the given OIL file. If every task has completed its execution with *TerminateTask* or *ChainTask*, it can be determined that the scenario generation is completed.

6.2 End-Level Environment Model

The End-Level scenario generation process is also based on the scenario generation process illustrated in Figure 5. It executes an arbitrary sequence of *End_Level_Functions*, which is chosen from among the list of *End_Level_Functions* serving as an environment of CBMC model checking and End-Level random testing.

This process considers internal constraints among End-Level functions; since End-Level functions can only be called by API-Level functions, the pre-conditional constraints of Root-Level functions identified from the OSEK/VDX constraints must be implicitly obeyed by the End-Level functions. These

implicit constraints can be identified by analyzing call-graphs of Root-Level functions and their preconditional relations. We call such implicit constraints internal constraints.

API	End-Level Function
StartOS	tpl_put_new_proc, tpl_put_new_proc, tpl_get_proc
WaitEvent	tpl_get_proc
SetEvent	tpl_put_new_proc, tpl_schedule_from_running
ReleaseResource	tpl_schedule_from_running
ActivateTask	tpl_put_new_proc, tpl_schedule_from_running
TerminateTask	tpl_get_proc
Schedule	tpl_schedule_from_running
ChainTask	tpl_put_new_proc, tpl_get_proc

Figure 10: End-Level Functions called by each API-Level Function

For example, if we consider the Root-Level APIs and their corresponding End-Level functions, the external constraint *WaitEvent can be called after SetEvent is called* can be re-interpreted as tpl_get_proc *can be called after* $tpl\_put\_new\_proc$ *and* $tpl\_schedule\_from\_running$ *are called*, which can be written in a regular expression $(tpl\_put\_new\_proc\ tpl\_schedule\_from\_running\ (End\_Level\_Function - tpl\_get\_proc)^*\ tpl\_get\_proc)^*$.

Checking constraints of this kind cannot be done using the OSEK/VDX simulator because the End-Level functions include implementation-specific function names that cannot be modeled from the standard. Instead, the internal constraints are simplified using the characteristic that *a function cannot be called more times than its preceding functions in the partial order relation*. The example internal constraint can be simplified as *The number of* tpl_get_proc *calls cannot exceed the one of either* $tpl\_put\_new\_proc$ *or* $tpl\_schedule\_from\_running$. The constraint checker keeps track of the number of each End-Level functions calls and checks $(\#\,tpl\_get\_proc < \#\,tpl\_put\_new\_proc)$ && $(\#\,tpl\_get\_proc < \#\,tpl\_schedule\_from\_running)$.

7 Experiments

We have conducted a series of experiments to show the impact of our approach using CBMC model checking, End-Level random testing, and Root-Level random testing. The target verification properties are three functional safety properties from the Trampoline kernel:

$$assert(tpl\_h\_prio \neq -1) \tag{1}$$
$$assert(tpl\_kern \neq NULL) \tag{2}$$
$$assert(tpl\_kern \rightarrow state == RUNNING) \tag{3}$$

tpl_h_prio is the value of the highest-priority task in the ready queue in the Trampoline kernel. $tpl\_h\_prio \neq -1$ is supposed to be true whenever rescheduling is necessary. tpl_kern stores the key information of the currently running task. $tpl\_kern \neq NULL$ and $tpl\_kern \rightarrow state == RUNNING$ checks if the state of the running task is *RUNNING* when the scheduler is called.

We performed verification of these three assertions using the model checker CBMC, Root-Level random testing, and End-Level random testing. The verification cost in terms of the number of verification

conditions and the resource requirements was measured for CBMC verification. Branch coverage was measured using the Squish Coco code coverage tool [2] for random testing. All experiments were performed on Linux Fedora 16 OS, with Intel Xeon 3.4GHz e3-1270 processor and 32GB of 1333MHz DDR3 RAM.

Property / Unwind	**tpl_h_prio != -1** code size : 437 lines *3 End-Level Functions / 19 Functions in total*			**tpl_kern != NULL** **tpl_kern->state == RUNNING** code size : 787 lines *7 End-Level Functions / 32 Functions in total*		
Unwind	**VCC**	**Time(s)**	**Memory(MB)**	**VCC**	**Time(s)**	**Memory(MB)**
Unwind 3	15	2	55.72	53	15	200.94
Unwind 7	43	91	297.14	157	4,100	1288.90
Unwind 10	64	2,379	923.67	235	42,241	1942.54
Unwind 15	99	30,748	2693.75	365	> 6 days	> 7366.89

Figure 11: Time and memory space to verify with CBMC

Figure 11 shows the time and memory space it took to verify the Trampoline operating system with the End-Level Environment Model using the model checker CBMC. Time and memory space increase exponentially as the length of the End-Level function calls (unwind value) increases. CBMC verifies the assertions by searching through every possible scenario with the length of the unwind value, making it a powerful method. With the given resources, CBMC reported no counter examples up to the unwind value 10, but was not able to finish its verification process for the unwind value 15 after 6 days. We do not report the result of CBMC model checking using Root-Level environment models since it is too costly to perform even with the value of unwind option 10.

Property	**tpl_h_prio != -1** code size : 1337 lines *(8 Root-Level, 3 End-Level Functions) / 50 Functions*					**tpl_kern != NULL** **tpl_kern->state == RUNNING** code size : 1378 lines *(9 Root-Level, 7 End-Level Functions) / 52 Functions*				
Length of the Scenario	**14**	**20**	**22**	**32**	**34**	**14**	**20**	**22**	**32**	**34**
tpl_schedule_from_running	100%(3/3)	100%(3/3)	100%(3/3)	100%(3/3)	100%(3/3)	100%(3/3)	100%(3/3)	100%(3/3)	100%(3/3)	100%(3/3)
tpl_schedule_from_dying	-	-	-	-	-	80%(4/5)	80%(4/5)	60%(3/5)	80%(4/5)	100%(5/5)
tpl_schedule_from_waiting	-	-	-	-	-	0%(0/3)	100%(3/3)	66.67%(2/3)	66.67%(2/3)	66.67%(2/3)
tpl_start_scheduling	-	-	-	-	-	100%(1/1)	100%(1/1)	100%(1/1)	100%(1/1)	100%(1/1)
tpl_wait_event_service	-	-	-	-	-	0%(0/5)	60%(3/5)	60%(3/5)	60%(3/5)	60%(3/5)
tpl_activate_task	-	-	-	-	-	100%(4/4)	100%(4/4)	100%(4/4)	100%(4/4)	100%(4/4)
tpl_set_event	-	-	-	-	-	0%(0/5)	80%(4/5)	80%(4/5)	80%(4/5)	80%(4/5)
tpl_get_proc	100%(3/3)	100%(3/3)	100%(3/3)	100%(3/3)	100%(3/3)	-	-	-	-	-
tpl_put_new_proc	66.67%(2/3)	66.67%(2/3)	66.67%(2/3)	66.67%(3/3)	66.67%(2/3)	-	-	-	-	-
Time(s)	-	-	-	-	-	-	-	-	-	-
Memory(MB)	2.64	2.64	2.64	2.64	2.64	2.64	2.64	2.64	2.64	2.64

Figure 12: Coverage, time, and memory space to verify with Root-Level Random Testing

There are a few dozens of extracted functions in random testing environments, including End-Level Functions, as illustrated in Figures 12 and 13. Due to a lack of space, only the test results of End-Level Functions are described. Root-Level Random Testing (Figure 12) is much faster (less than 1/100 seconds), consumes little memory (up to 2.64MB of memory), and achieves a certain level of test coverage quickly, but the coverage does not improve after test sequences of length 34. In End-Level Random Testing (Figure 13), a test sequence of length 100 achieves a certain level of coverage both for tpl_h_prio and

Property	**tpl_h_prio != -1** code size : 437 lines *3 End-Level Functions / 19 Functions in total*					**tpl_kern != NULL** **tpl_kern->state == RUNNING** code size : 787 lines *7 End-Level Functions / 32 Functions in total*				
Length of the Scenario	**10**	**50**	**100**	**1000**	**10000**	**10**	**50**	**100**	**1000**	**10000**
tpl_schedule_from_running	66.67%(2/3)	66.67%(2/3)	66.67%(2/3)	66.67%(2/3)	66.67%(2/3)	33.33%(1/3)	100%(3/3)	100%(3/3)	100%(3/3)	100%(3/3)
tpl_schedule_from_dying	-	-	-	-	-	60%(3/5)	60%(3/5)	100%(5/5)	100%(5/5)	100%(5/5)
tpl_schedule_from_waiting	-	-	-	-	-	0%(0/3)	66.67%(2/3)	66.67%(2/3)	66.67%(2/3)	66.67%(2/3)
tpl_start_scheduling	-	-	-	-	-	100%(1/1)	100%(1/1)	100%(1/1)	100%(1/1)	100%(1/1)
tpl_wait_event_service	-	-	-	-	-	0%(0/5)	60%(3/5)	60%(3/5)	60%(3/5)	60%(3/5)
tpl_activate_task	-	-	-	-	-	100%(4/4)	100%(4/4)	100%(4/4)	100%(4/4)	100%(4/4)
tpl_set_event	-	-	-	-	-	0%(0/5)	80%(4/5)	80%(4/5)	100%(5/5)	100%(5/5)
tpl_get_proc	100%(3/3)	100%(3/3)	100%(3/3)	100%(3/3)	100%(3/3)	-	-	-	-	-
tpl_put_new_proc	66.67%(2/3)	66.67%(2/3)	100%(3/3)	100%(3/3)	100%(3/3)	-	-	-	-	-
Time(s)	-	-	-	-	0.04	-	-	-	-	0.07
Memory(MB)	1.06	1.06	1.06	1.06	1.06	1.07	1.07	1.07	1.07	1.07

Figure 13: Coverage, time, and memory space to verify with End-Level Random Testing

tpl_kern. The coverage stays the same afterwards. End-Level Random Testing required around 1.06 1.07 MBytes of memory.

For many cases, the coverage did not increase even with lengthier test cases. There can be two reasons why some part of the code are unreachable. The first reason is exception handling; parts of the code for exception handling are never reached unless an exceptional situation occurs. The second reason is that some variables in conditional statements are not included in the Extended Verification Target Variable. So the behavior of updating these variables might not be fully extracted, which can make some conditional statements fixed. An example of this case is illustrated in Figure 14. This conditional statement is only

```
void tpl_schedule_from_running(void){
        ……
        // READY_AND_NEW
        if (tpl_kern.running->state == READY_AND_NEW)
        {
            tpl_init_proc(tpl_kern.running_id);
        }
        ……
}
```

Figure 14: Example of uncovered conditional statement

executed when *tpl_schedule_from_dying* or *tpl_activate_task* is executed before this code. But these two functions are out of boundary in this model because the model is generated only with regard to the property *"tpl_h_prio != 0"*. Thus coverage cannot be increased, and testing terminates.

In terms of comprehensiveness, CBMC is the most powerful method, verifying every possible scenario within the same length, but it is limited by the length. As shown in Figure 11, the verification cost increases exponentially as unwinding depth increases. Therefore, CBMC cannot detect potential faults that can be identified only in long task scenarios. Unlike verification with CBMC, the length of the scenarios is not limited in random testing since the cost is much cheaper than CBMC, as shown in Figure 13. Though it cannot be comprehensive, it can be more effective in stress testing, since the length of the test sequences can be sufficiently long.

End-Level Random Testing did in fact, catch some overflow errors in the Trampoline kernel as illustrated in Figure 15. These errors have occurred because the size of the variables saving the activation

```
tpl_get_proc              -> tpl_h_prio :  2, taskNum : 4, activationCount(T1, T2, T3) : 1, 255, 2
tpl_put_new_proc          -> tpl_h_prio :  2, taskNum : 5, activationCount(T1, T2, T3) : 1, 255, 1
---put_new_proc : 1
tpl_get_proc              -> tpl_h_prio :  2, taskNum : 4, activationCount(T1, T2, T3) : 1, 0, 1
tpl_get_proc              -> tpl_h_prio :  0, taskNum : 3, activationCount(T1, T2, T3) : 1, 0, 0
tpl_schedule_from_running -> tpl_h_prio : -1, taskNum : 3, activationCount(T1, T2, T3) : 0, 0, 0
RandomTest: RandomTest.c:505: tpl_schedule_from_running: Assertion `tpl_h_prio != -1' failed.
```

Figure 15: Error caused by overflow

count is limited to 8 bits; the second line of Figure 15 shows that *Task*2 has the activation count 255, but adding another activation changed its value to 0. So Trampoline has changed the value of *tpl_h_prio* to -1, meaning that the process table has no activated task available. The variable size is implementation-specific and is not constrained to 8 bits, neither in the OIL specification nor in the OSEK/VDX specification. This problem could be addressed by constraining the size to 8 bits in the OIL specification.

A Model checker could find this type of potential faults if we can set the value of the unwind option larger than 255, but our experiments could not identify it due to resource limitations. End-Level Random Testing is appropriate for finding this kind of errors because the cost does not increase much even with lengthy test scenarios.

We could not identify the same fault using Root-Level Random Testing, either. The main reason is that Root-Level Random Testing is coupled with a pre-defined OIL configuration file. The OIL file specifies the typical system configuration, and thus, activating a task over 255 times is not likely to happen unless we specifically aim at stress testing. End-Level Random Testing is more effective in stress testing, since it is not constrained by the system configuration and can test abnormal cases.

API-Level Random Testing, however, is beneficial in that it is not necessary to do additional API-Level analysis when testing identifies faults, which is necessary in model checking and End-Level Random Testing.

8 Conclusion

This paper presented methods and tools for environment generation and code abstraction to improve the efficiency of verification using model checking and testing. The effect of using the suggested approach was demonstrated through a series of experiments using the Trampoline operating system as a case example. The benefit of property-based environment generation is two-fold: (1) it reduces verification cost by reducing the target code and by limiting its environment to the task interaction scenario relevant to the verification property, and (2) it simplifies the analysis process and localizes the verification activity by focusing on the points of interest.

The experiments revealed relative pros and cons of the three verification methods and identified potential safety faults, which suggests the following collaborative use of model checking and testing;

1. Apply End-Level Random Testing first for stress testing.
2. Apply Root-Level Random Testing to conform the errors identified through End-Level Random Testing.
3. Apply model checking using CBMC last for comprehensive verification within a limited scope.

Our tool still needs some improvements. First, conditional dependencies need to be considered so that test coverage can be improved. Second, Root-Level scenario generation currently assumes a fixed OIL configuration. We would like to relax the condition so that an arbitrary OIL can be handled by the tool.

References

[1] *OSEK/VDX Portal*. Http://portal.osek-vdx.org.

[2] *Squish Coco Code Coverage*.

[3] *Trampoline – OpenSource RTOS project*. Http://trampoline.rts-software.org.

[4] *Understand: Source Code Analysis and Metrics*. Http://www.scitools.com/.

[5] David Binkley (1999): *The Application of Program Slicing to Regression Testing*. Information and Software Technology, pp. 583–594.

[6] Manfred Broy (2006): *Challenges in automotive software engineering*. In: Proceedings of the 28th International Conference on Software Engineering.

[7] Omar Chebaro, Nikolai Kosmatov, Alain Giorgetti & Jacques Julliand (2012): *Program slicing enhances a verification technique combining static and dynamic analysis*. In: Proceedings of the 27th Annual ACM Symposium on Applied Computing, pp. 1284–1291.

[8] Jiang Chen & Toshiaki Aoki (2011): *Conformance Testing for OSEK/VDX Operating System Using Model Checking*. In: 18th Asia-Pacific Software Engineering Conference.

[9] Edmund Clarke, Daniel Kroening & Flavio Lerda (2004): *A Tool for Checking ANSI-C Programs*. In: 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems.

[10] Edmund M. Clarke, Orna Grumberg & Doron Peled (1999): *Model Checking*. MIT Press.

[11] Matthew Dwyer & Corina Pasareanu (1998): *Filter-Based Model Checking of Partial Systems*. In: 6th ACM SIGSOFT International symposium on Foundations of Software Engineering, pp. 189–202.

[12] Dimitra Giannakopoulou, Corina S. Pasareanu & Howard Barringer (2002): *Assumption Generation for Software Component Verification*. In: 17th IEEE International Conference on Automated Software Engineering, pp. 3–12.

[13] A. Gupta, K.L.McMillan & Z. Fu (2008): *Automated Assumption Generation for Compositional Verification*. Formal Methods in System Design 32, pp. 285–301.

[14] Rajiv Gupta, Mary Jean Harrold & Mary Lou Soffa (1992): *An Approach to Regression Testing using Slicing*. In: Proceedings of the Conference on Software Maintenance, pp. 299–308.

[15] Nannan He & Michael S. Hsiao (2007): *Bounded Model Checking of Embedded Software in Wireless Cognitive Radio Systems*. In: 25th International Conference on Computer Design, pp. 19–24.

[16] Gerard J. Holzmann (2003): *The* SPIN *Model Checker: Primer and Reference Manual*. Addison-Wesley Publishing Company.

[17] Natalia Ioustinova & Natalia Sidorova (2002): *Abstraction and Flow Analysis for Model Checking Open Asynchronous Systems*. In: Proceedings of the Ninth Asia-Pacific Software Engineering Conference.

[18] Jürgen Mössinger (2010): *Software in Automotive Systems*. IEEE Software 27(2), pp. 92–94.

[19] Wonhong Nam, P. Madhusudan & Rajeev Alur (2008): *Automatic symbolic compositional verification by learning assumptions*. Formal Methods in System Design.

[20] Pascal Raymond Nicolas Halbwachs, Fabienne Lagnier (1993): *Synchronous Observers and the Verification of Reactive Systems*. In: Third International Conference on Algebraic Methodology and Software Technology, AMAST'93, Workshops in Computing.

[21] Pavel Parizek & Frantisek Plasil (2007): *Partial Verification of Software Components: Heuristics for Environment Construction*. In: *17th International SPIN Conference on Software Model Checking*.

[22] John Penix, Willem Visser, Seungjoon Park, Corina Pasareanu, Eric Engstrom, Aaron Larson & Nicholas Weininger (2005): *Verifying Time Partitioning in the DEOS Scheduling Kernel*. *Formal Methods in Systems Design Journal* 26(2), pp. 103–135.

[23] O. Tkachuk, M.B. Dwyer & C.S. Pasareanu (2003): *Automated Environment Generation for Software Model Checking*. In: *18th IEEE International Conference on Automated Software Engineering*, pp. 116–129.

[24] M. Weiser (1984): *Program Slicing*. *IEEE Transactions on Software Engineering* SE-10(4), pp. 352–357.

[25] Kenro Yatake & Toshiaki Aoki (2010): *Automatic Generation of Model Checking Scripts based on Environment Modeling*. In: *17th International SPIN Conference on Software Model Checking*.

Model Checking with Program Slicing Based on Variable Dependence Graph

Masahiro Matsubara Kohei Sakurai Fumio Narisawa
Green Mobility Research Department
Hitachi Research Laboratory, Hitachi, Ltd.
Japan
[masahiro.matsubara.td|kohei.sakurai.cp|fumio.narisawa.ks]@hitachi.com

Masushi Enshoiwa Yoshio Yamane
Engineering Department 6, Image & Information Division
Hitachi Advanced Digital, Inc.
Japan

Hisamitsu Yamanaka
System Development Engineering Department
Hitachi Automotive Systems, Ltd.
Japan

In embedded control systems, the potential risks of software defects have been increasing because of software complexity which leads to, for example, timing related problems. These defects are rarely found by tests or simulations. To detect such defects, we propose a modeling method which can generate software models for model checking with a program slicing technique based on a variable dependence graph. We have applied the proposed method to one case in automotive control software and demonstrated the effectiveness of the method. Furthermore, we developed a software tool to automate model generation and achieved a 35% decrease in total verification time on model checking.

1 Introduction

In embedded control systems, potential risks to system safety have been increasing because programs are getting larger due to electrification, enhancement of diagnosis, etc. Functional safety standards such as IEC 61508 or ISO 26262 have been established to ensure these systems do not fall into dangerous situations. However, the problem of test coverage still remains. Furthermore, there are corner cases which are difficult to find with usual tests or simulations.

For example, hardware malfunctions could cause software faults, so it is not sufficient to test the software only. Inspection techniques to test combinations of hardware and software such as HILS (Hardware in-the-Loop Simulation) have been in practical use, but failures caused by hardware malfunctions or timing problems caused by software interruptions are difficult to detect because there could be large number of test cases.

To solve these problems, model checking is applied. In model checking, all of the state transitions of the system are fully searched, and model checking is also effective against timing problems. Implementation bugs can also be found if models are made out of the source code. However, the problem in applying model checking is also well known: a state explosion that verification does not complete. To avoid a state explosion, the scope of verification has to be limited. However, too limited a model misses some causes of malfunctions, so that the part of the source code to be verified has to be determined appropriately.

We are trying to adopt model checking to the development of automotive control software, using it to find software defects which occur rarely on real systems by means of modeling both the software from the source code and the hardware. To solve the state explosion problem mentioned above, we introduced a program slicing technique [10] based on a variable dependence graph which enables an

Submitted to:
FTSCS 2012

adjustment of the boundaries between the source code to be modeled as software model in detail and the external environment to the software model. Also we introduced external environment models which include unsteady behavior or faults in them to detect malfunctions caused by interactions of hardware and software.

The contributions of this paper are as follows:

- A practical modeling method to verify a large scale embedded control program. The main point of the method is the slicing technique based on a variable dependence graph.
- The algorithm for analysis of variable dependence graph, which could be adapted to the analysis of system dependence graph.
- An experimental evaluation of the tool which automates the above method.

The rest of this paper is structured as follows: Section 2 presents the modeling method and the slicing technique. Section 3 presents an example case where the method is applied, to show the usefulness of the method. Section 4 presents the tool which automates the method. Section 5 presents the related work. Finally, Section 6 concludes this paper by summarizing its main points.

2 Modeling Method

2.1 System Modeling

The way of modeling an embedded control system is shown in Figure 1. To find defects in implementations, the software of the target system is converted from the source code, which is related to the variables selected by a verifier in accordance with a verification point. This code is sliced out from the whole program to avoid a state explosion, using a slicing technique based on a variable dependence graph (see 2.2).

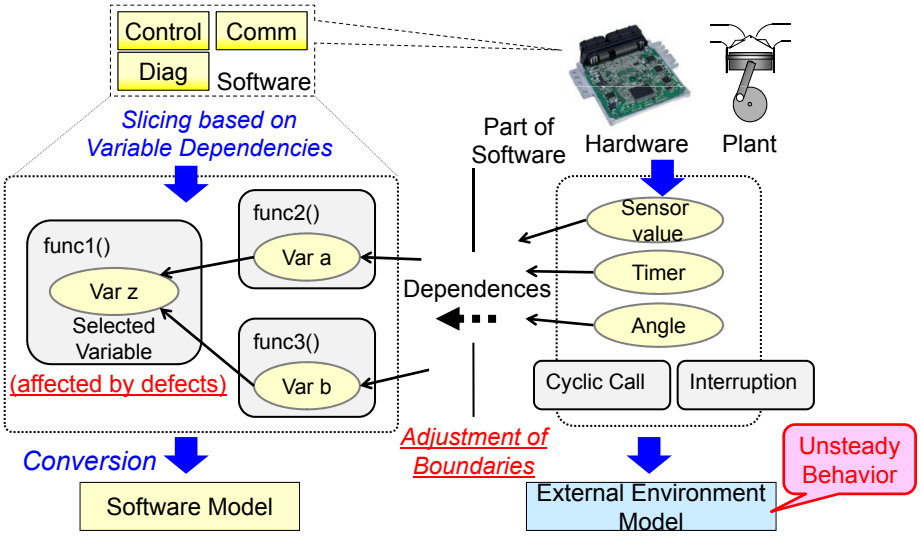


Figure 1: Modeling Method for Embedded Control System

On the other hand, the hardware, controlled instruments (if needed for verification), and a part of the

software are modeled as external environments. These external environment models can have unsteady behavior or faults if they are concerned with the content to be verified.

The slicing technique based on the variable dependence graph makes it easy to adjust the boundaries between the software models and the external environment models. The reason that such adjustments are needed is that the appropriate size of the software model differs depending on the verification points. Too large a model leads to a state explosion, and too small a model can miss the cause of a malfunction.

An example of a boundary adjustment is shown in Figure 2. There is a communication driver between the software and the hardware, and this driver has complex processing. It is better to omit the driver to reduce the state number of the software model, if the driver has no relation to a malfunction. In this case, it is appropriate to treat the driver as a part of the external environment, and simplify its processing. This method makes the modeling highly automated but not fully automated, and the use of design knowledge is expected for modeling. The scope of the software model has to be changed according to the verification point.

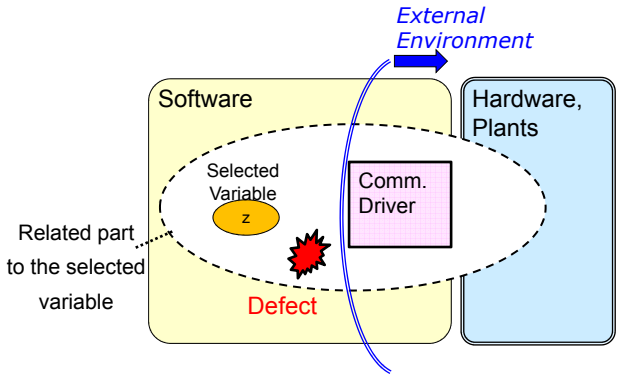


Figure 2: Abstraction of Boundary between Software and Hardware

2.2 Slicing Technique

The data to be used for slicing is usually a program dependence graph (PDG) or a system dependence graph (SDG) [7]. The criteria for slicing include a start statement, an end statement, and variables in the end statement.

In this method, the variable dependence graph (VDG) in Figure 3 is used as data for slicing, which is a kind of data flow to connect dependence among the variables. VDG is equivalent to PDG (or SDG) except for the data unit (a statement in PDG), but it is more suitable for expressing dependencies in a tree format. Tree format is better to pursue related variables and code sequentially, and to adjust the boundaries between code converted to the software model and code treated as external environments. In PDG, data dependence and control dependence are used as an edge. On the other hand, in VDG, dependence brought by an assignment which is similar to data dependence is also used as an edge. Nodes in VDG are variables distinct with positions and execution paths in the source code.

There are two directions in VDG trees. In one, the root node is the goal of the dependencies in the tree (we call it the goal tree): in the other, the root node is the start (the start tree). These root nodes are one side of the slicing criteria, and leaf nodes including those which are on a boundary adjusted by a verifier are another side.

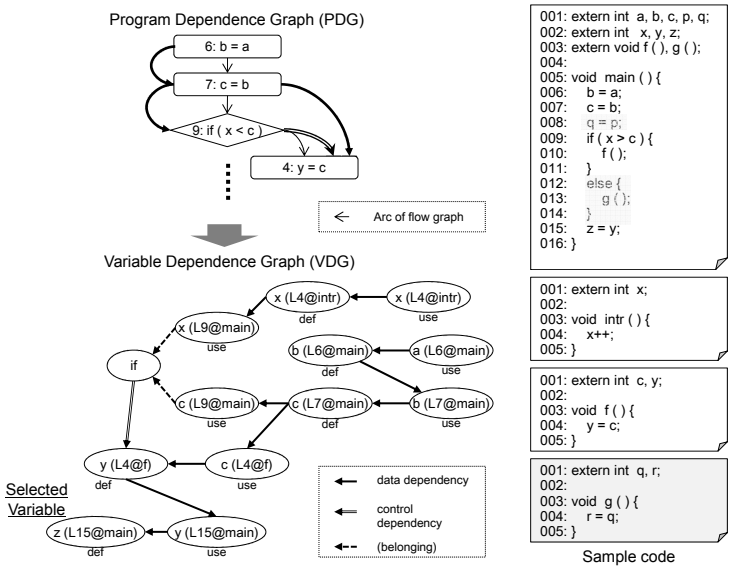


Figure 3: Variable Dependence Graph of Sample Code

The steps of slicing are as follows:

1. Select variables which are related to the verification point in the source code of the target system. These variables are those which could be affected by defects, and are written in properties or assertions.
2. Analyze and extract the VDG goal / start trees from the source code, where the root nodes are the variables selected in step 1.
3. Adjust the boundaries in the VDG trees. (Figure 4)
4. Execute slicing to extract code which has a relation to the variables in the VDG trees extracted in Step 2 and adjusted in Step 3.

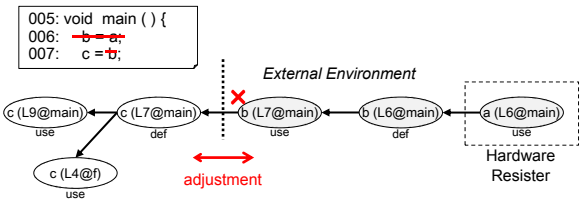


Figure 4: Adjustment Boundary on VDG

In step 3, a variable on a boundary becomes a interface between the software model and external environment models. It is possible to set a boundary between functions by setting the boundary between variables which belong to different functions.

In step 4, code to be extracted are not only statements which include a variable in VDG directly, but also declarations, functions and control statements which include statements above using control flow.

The limitation of this technique is that statements which do not include a variable except function calls cannot be extracted, so that these statements have to be picked up on control flow if needed.

The benefits of this slicing technique are as follows:

- High accuracy of slicing, equal to PDG-based slicing.
- Easy to adjust boundaries between software models and external environments, especially in the tree expression of variable dependencies.
- Interface variables between software models and external environments are clear. Information on the interface is important to make or select external environment models.
- Variables whose range has to be changed are clear when input / output values have data mapping.

The sliced code is converted into software models in the language of the model checker, such as Promela of SPIN [5]. This conversion is done via AST (Abstract Syntax Tree) according to the predetermined rules.

2.3 External Environment Models

External environment models are necessary to detect malfunctions caused by interactions between the hardware and software. Specifically, they set every possible combination of input values to software models, or limit those combinations to reduce the state number with abstraction. They also simulate the behavior outside of the software so that it becomes possible to judge whether a property for the external environment is held or not.

External environment models are written manually. When a verifier makes a new model or selects one from the existing models of the external environment, information about the interface is useful, as is shown in VDG. Usually, input values to a software model or determinants of them are set with randomness. Processes are added to external environment models.

Furthermore, a verifier can put unsteady behavior or faults into the external environment models which occur at random realized with indecisive constructions or concurrency. If there is a malfunction when an external environment model has unsteady behavior, and vice versa, that behavior seems to be one of the factors which cause the malfunction, so that the verifier could analyze a counter example from that point of view.

3 Case Study

In this section, we applied the modeling method described in section 2 to show its usefulness.

3.1 Example Case

The target to be verified is automotive software for a diagnosis of a power IC in a controller, as shown in Figure 5. The purpose of the verification was to investigate the reason for a malfunction. The malfunction was an incorrect error detection in the diagnosis on a long-term test, and the reason was unknown. Using a standard test methodology was not enough to find the defect and to clarify the cause of the malfunction, so model checking was applied.

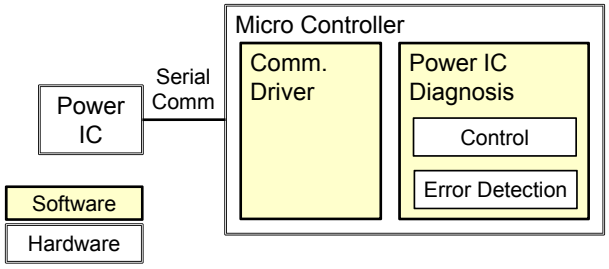


Figure 5: System Structure of Sample Case

The verification was done in two steps. In the first step, we focused on the serial communication driver between the micro controller and the power IC, but no defects were found. In the second step, we modeled the software for the diagnosis, and the power IC instead of the communication driver which was simplified as an external environment model.

The modeling method was applied in the second step. Figure 6 shows the model. For the slicing, the variable which shows the result of the diagnosis was selected as the objective variable. The boundaries were adjusted so that the communication driver was omitted, and several inputs were given by external environment models. One of the inputs is the power IC state to be diagnosed, and another is the controller state which is decided in software depending on the power supply. In the transition of the controller state, a path was added that transits to a power-off state suddenly and abnormally from states with the power on. This path is intended to express the effect of an instantaneous drop in the power voltage.

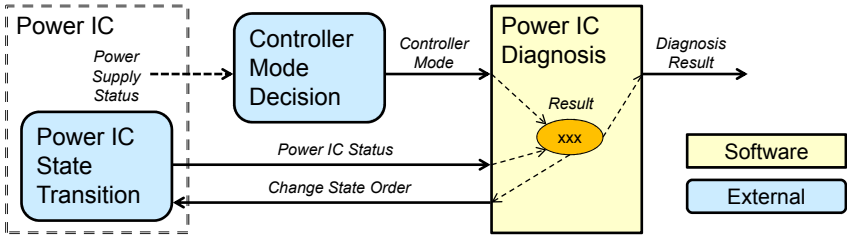


Figure 6: Modeling of Sample Case

In model checking, it is important to reduce the number of concurrent processes in order to reduce the state number. Although the micro controller and power IC run concurrently, the diagnosis is a cyclic task, so the external environment models are set to change states before the diagnosis in the same process of the software model. If no malfunction was detected in this model, the external environment models could be separated into other processes. Moreover, data mapping was applied to the power IC state to be diagnosed.

The result was that one software defect was detected which causes a malfunction when the controller state turns to the power-off state suddenly while a specific procedure is executed. It is confirmed that this defect could cause the same malfunction in an actual controller. The sizes of the source code and models are shown in Table 1.

The mechanism of the malfunction is as follows. Although the diagnosis is stopped by power off,

Table 1: Sizes of Source code and Model

Item	Size (approx.)
Whole source code (.c, .h)	130 k steps
Software model (Promela)	600 LOC
System model (Promela)	750 LOC

the power IC continues to change its state while the power is off. After the power supply is restored, an inconsistency between the actual power IC state and the recognition by diagnosis for that state occurs. This phenomenon happens only in a specific situation, so this is a timing problem. It is difficult to detect this defect with usual tests or simulations like HILS because a verifier can hardly set conditions to make this malfunction occur.

This case shows that model checking is effective with timing problems, and the modeling method can treat large-scale control software.

An obstacle to applying this modeling method to a development process is that it is time consuming. To improve the efficiency of the method, we developed a software tool to automate the method of generating verification models.

4 Tool

4.1 Functionalities

The tool we developed has two main functions as shown in Figure 7. One is slicing C language code based on a variable dependence graph; another is conversion from C language code to Promela.

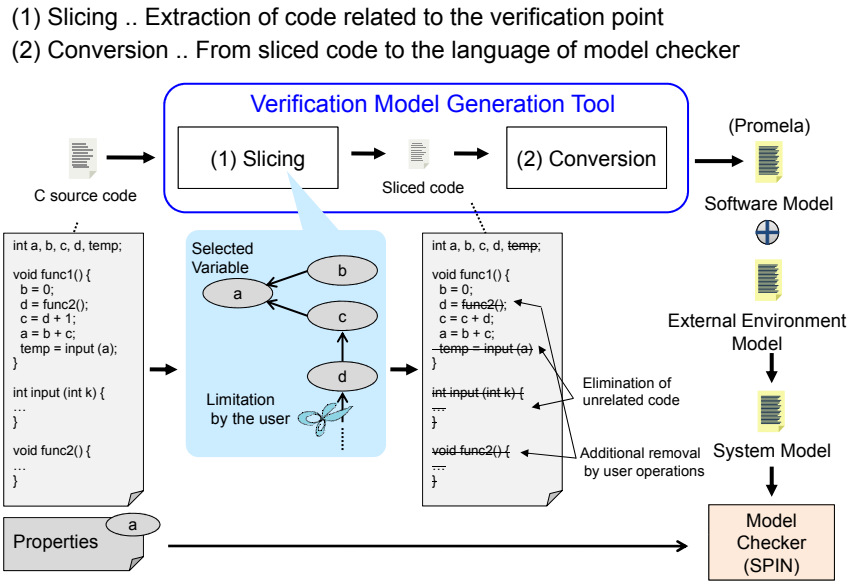


Figure 7: Verification Model Generation Tool

When a user indicates one or more functions as start points of execution, the tool will analyze the

variable dependencies. A variable dependence graph in a tree format will be extracted and displayed according to the variables that the user has selected. The tool can extract variable dependencies which extend other execution paths in consideration of interruptions or other tasks.

In the screen of the tool, variable dependence trees, function call trees, and source code files are displayed in association with each other, as shown in Figure 8. The user can limit the variable dependencies to adjust the boundaries between the software models and external environment models. Such an operation is reflected as a color classification of the source code.

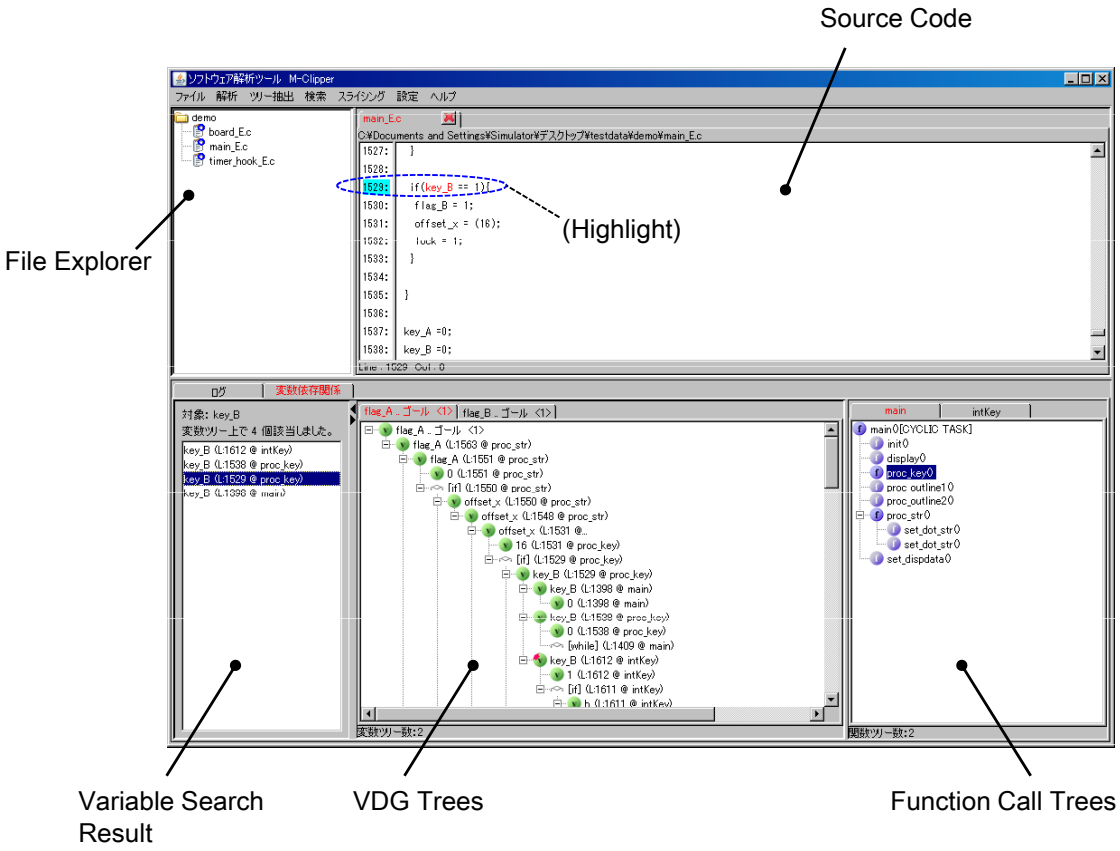


Figure 8: Screen Shot

In a tree format, the VDG could become larger than in a network format, so there are some refinements in the projection of the tree. Redundant tree paths (the same partial trees as other parts) are replaced with a node which indicates redundancy. Every node can be gathered to the function node on which the variable is written. Also, the tool has search functions for variables on trees or execution paths.

At present, the limitations of the tool are as follows. Source code has to be pre-processed. Pointers can be treated, but only one variable is pointed by a pointer, and pointed variables given by loop controls are ignored. Function calls by function pointers are also ignored. Slicing and model conversion are separated into different tools, model conversion is done for each C code file. This tool is not open to the public.

4.2 Algorithm for Analysis of VDG

The algorithm to analyze VDG used in this tool is shown in Figure 9. Dependencies are connected one by one along control flow graph. Every node, which is a variable distinguished with the position in code and the execution path, is assigned information about the stack trace and associated to the control flow graph (CFG). It is judged whether the nodes connects or not using that information of the node to connect, the node to be connected, and nodes that have been already connected to the target node. Candidate nodes to connect in A and C can be limited in consideration of the variable substance (memory area) to be connected and the execution context. Different from [7] [8], initialization vertexes or finalization vertexes are not used. The advantage of this algorithm we think is scalability. Actually, the analysis of code which size is over 350k steps were succeeded.

```
A .. nodes already acquired
C .. nodes already acquired, in conditional statements

construct CFG
for all start point p of execution paths on CFG
  statement s ← p
  while s exists do
    T ← stack trace of S
    get variable nodes V corresponding to S, and assign T to V
    connect use in V to def in V
    for all node vi in V do
      for all def d in A do
        if d is connectable to vi then
          connect d to vi
        end if
      end for
      for all use u in C do
        if u is connectable to vi then
          connect u to vi
        end if
      end for
    end for
    add V to A (and C)
    s ← next statement on CFG
  end while
end for
```

Figure 9: Algorithm for Analysis of VDG

The process of the analysis for the sample code is shown in Figure 10, where nodes in a conditional statement belongs a “if” node for the simplicity of appearance. There are no parameters and arguments in the sample code, but they can be treated in the same way.

After the analysis, a VDG for an execution path is obtained. This VDG becomes a network if there is a loop statement. VDG goal /start trees are extracted from VDG networks, considering shared variables between different execution paths.

4.3 Evaluation

Figure 11 shows the verification process with model checking we employed. If a false positive / false negative error was detected in the verification, the step “Model Revision” is taken additionally to refine the model, and another verification is done. Besides the slicing tool and the model converter, the verification assist tool was used which actuates checking on more than one property simultaneously, and shows a variable value table of counter examples to help the user understand the situation more easily.

Table 2 shows the condition of the experiments. The target is automotive control software, and it has never been verified with model checking. Verifications are done without the tools first and with the tools

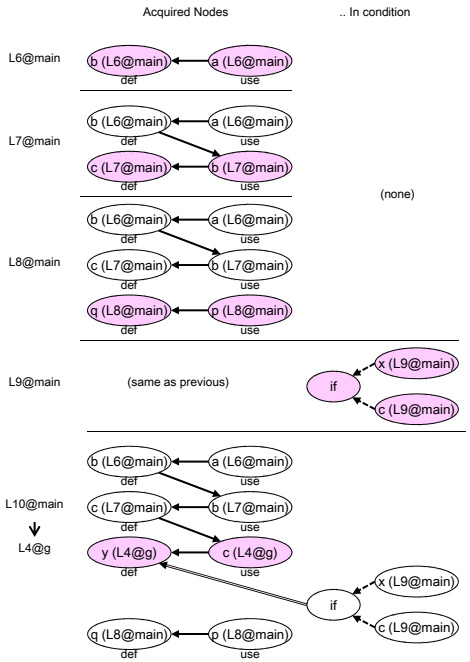


Figure 10: VDG Analysis Process of Sample Code

second. The time in the first try is applied for some steps of the second try which should have no effect of the tools, because the verifier could work faster the second time.

Table 2: Condition of Experiment

Item	Description
Experience of verifier	Approx. 1 year
Target	Diagnosis of input signal
Source code size (.c, .h)	94 k steps
Number of properties	2

Table 3 shows measured time, and those values are normalized. Highlighted rows are the steps to which the tools were applied. The step ”Model Revision” was taken because there was a false positive error at first.

When tools are used, the whole verification time on model checking is reduced 35% compared to checking without the tools. Although no tool is used in the step “Simulation Check”, the time is reduced because the conversion is precise. There is no reduction of time in the step “Verification” because the number of properties is only two in this case. It is expected that more reduction will be achieved when th tools are improved.

Except for user operations, the analysis and the slicing were completed within 5 minutes. (PC Spec: Intel(R) Core(TM)2 Duo E6550 @2.33GHz)

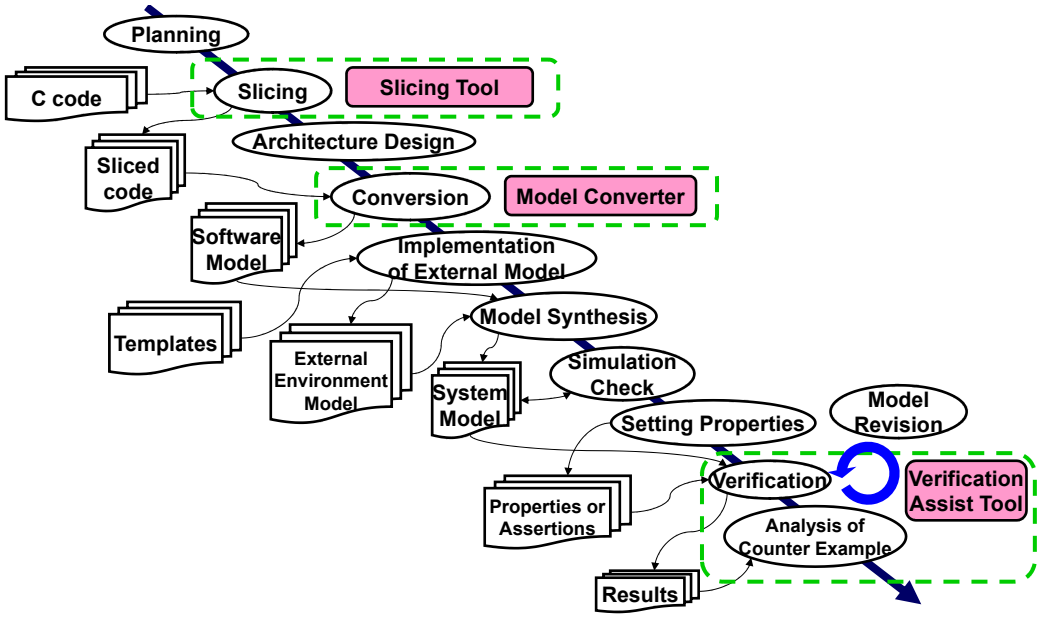


Figure 11: Verification Steps

Table 3: Result of Evaluation

#	Step	(1) without tools	(2) with tools	Difference	Decrease	Remarks
1	Planning	0.032	0.032	0.00	0.0%	same as (1)
2	Slicing	0.193	0.098	0.09	49.2%	
3	Architecture Desigin	0.148	0.148	0.00	0.0%	same as (1)
4	Model Conversion	0.131	0.030	0.10	77.4%	
5	Impl. External Model	0.102	0.102	0.00	0.0%	same as (1)
6	Model Synthesis	0.006	0.006	0.00	0.0%	same as (1)
7	Simulation Check	0.181	0.074	0.11	59.1%	
8	Setting Properties	0.008	0.008	0.00	0.0%	same as (1)
9	Verification	0.002	0.002	0.00	0.0%	2 properties
10	Analysis of CE	0.119	0.074	0.05	38.0%	2 counter examples
11	Model Revision	0.078	0.078	0.00	0.0%	same as (1)
	Sum	1.000	0.652	0.35	34.8%	

5 Related Work

Bandera [4] slices Java code and converts them into languages for model checker such as SPIN. However, the major language in embedded control systems is C. Modex/Feaver [6] can convert C code to Promela with detailed rules using a convert table and a test harness, but this tool doesnft provide the functionality to slice large code. The Wisconsin Program-Slicing Tool [8] [9] and its commercial counterpart can analyze PDG, but provide no functionality to adjust boundaries on PDG. CBMC [3], BLAST [2] and SLAM [1] input source code directly for model checking. SLAM verifies programs with software stubs as external environments. Such a static analyzer can detect defects automatically, but verification items are fixed.

6 Conclusion

In embedded control systems, the potential risks of software defects have been increasing because of increasing software complexity. To detect software defects which are difficult to find with usual tests or simulations, we proposed a modeling method which can generate software models from source code for model checking, with a program slicing technique based on a variable dependence graph. We applied the proposed method to one case in automotive control software and demonstrated the effectiveness of the method. Furthermore, we developed a software tool to automate the method and achieved a 35% decrease in total verification time on model checking. It is expected that more reduction will be achieved when the tool is improved.

References

[1] T. Ball, V. Levin & S. K. Rajamani (2011): *A Decade of Software Model Checking with SLAM*. In: *Communications of the ACM*, 54, pp. 68–76.

[2] D. Beyer, T. A. Henzinger, R. Jhala & R. Majumdar (2007): *The software model checker BLAST*. In: *Journal on Software Tools for Technology Transfer*, pp. 505–525.

[3] E. Clarke, D. Kroening & F. Lerda (2004): *A tool for checking ANSI-C programs*. In: *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems*, pp. 168–176.

[4] J. C. Corbett & M. B. Dwyer et al. (2000): *Bandera: Extracting Finite-state Models from Java Source Code*. In: *Proceedings of the 22nd International Conference on Software Engineering*, pp. 439–448.

[5] G. J. Holzmann (2003): *The SPIN Model Checker*. Addison-Wesley Pub.

[6] G. J. Holzmann & T. C. Ruys (2005): *Effective Bug Hunting with SPIN and Modex*. In: *Proceedings of the 12th International SPIN Workshop*.

[7] S. Horwitz, T. Reps & D. Binkley (1988): *Interprocedural slicing using dependence graphs*. In: *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pp. 35–46.

[8] T. Reps, S. Horwitz, M. Sagiv & G. Rosay (1994): *Speeding up Slicing*. In: *Proceedings of the 2nd ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 11–20.

[9] T. Reps & G. Rosay (1995): *Precise interprocedural chopping*. In: *Proceedings of the 3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 41–52.

[10] Mark Weiser (1981): *Program Slicing*. In: *Proceedings of the 5th international conference on Software engineering*, pp. 439–449.

A Framework for Analysing Driver Interactions with Semi-Autonomous Vehicles

Siraj Ahmed Shaikh
Digital Security and Forensics (SaFe) Research Group,
Department of Computing, Faculty of Engineering and Computing,
Coventry University, Coventry, UK

s.shaikh@coventry.ac.uk

Padmanabhan Krishnan
Centre for Software Assurance,
Bond University, Gold Coast, QLD 4229, Australia

pkrishna@bond.edu.au

Semi-autonomous vehicles are increasingly serving critical functions in various settings from mining to logistics to defence. A key characteristic of such systems is the presence of the human (drivers) in the control loop. To ensure safety, both the driver needs to be aware of the autonomous aspects of the vehicle and the automated features of the vehicle built to enable safer control. In this paper we propose a framework to combine empirical models describing human behaviour with the environment and system models. We then analyse, via model checking, interaction between the models for desired safety properties. The aim is to analyse the design for safe vehicle-driver interaction. We demonstrate the applicability of our approach using a case study involving semi-autonomous vehicles where the driver fatigue are factors critical to a safe journey.

1 Introduction

Human failure is often a cause of accidents. Increasing the level of automation while useful in many cases, does not necessarily reduce the number of human failure related accidents. Standards such as ISO 26262 describe functional safety but do not explicitly describe the role of the human user. Most of the research related to ISO 26262 focusses only on the reliability of the electronics and the human user is often ignored. Reliable electronics is not sufficient to guarantee safety.

For such automation to be successful the human user must be aware of the automation and react to it appropriately. In some cases it is not possible to fully automate the behaviour and the system has to rely on humans exhibiting the right behaviour. Examples of such systems include Unmanned Aerial Vehicle (UAV) guidance [9], health care especially patient safety [7] and computer security [11]. Thus it is important to understand the role of humans in such systems and analyse the "human in the loop" behaviour [11].

Recent research effort in this direction has seen the choice of various formal methods to model and verify correctness with a view to also modelling human behaviour as it interacts with the system interface. This is largely achieved for isolated behaviours and cognitive errors including mode confusion [28], post-completion errors [12], error recovery [30] and automatic behaviour [8].

The modelling and analysis of such systems however is non-trivial. For instance, to analyse driver interactions with a road vehicle requires knowledge of the vehicle (to model for vehicle dynamics), the

environment (such as road conditions, weather, terrain and traffic flow) and a model of the human factors that affect the interaction with the vehicle such as levels of stress, expertise, attention and fatigue.

In this paper we present a modelling and analysis framework to model human behaviour and analyse the interaction to determine if any safety conditions are violated. We model a semi-autonomous system where the driver is part of the control system. Our approach allows us to incorporate human factors in the controllability analysis of the system. Our focus is on the interaction between the user and the system; we do not necessarily focus on the reliability of the system.

1.1 Rest of this paper

The rest of this paper is organised as follows. Section 2 provides motivation behind this work with a view to limitations of some existing work in this area. Section 3 presents our approach and describes how it serves to combine the individual strands of effort from Section 2 to provide for a basis for further analysis. Section 3.1 is an attempt to describe our approach in brief formal terms.

Section 4 describes a case study that we use to demonstrate our contribution. The problem of driver fatigue is well understood. Recent developments in driver assistance and adaptive systems means that the safety risks that emerge as a result of interaction between drivers and increasingly semi-autonomous vehicles is yet to be fully explored. System, behaviour and environment models are described in Sections 4.1, 4.2 and 4.3 respectively. The verification process is described in Section 4.4.

Section 5 presents the results of the application of our approach to this problem, with notable results discussed in Sections 5.1 and 5.2. Section 6 concludes the paper with a brief comment on the contribution made by this paper.

2 Motivation

There are potential economic, health and safety benefits of semi-autonomous vehicles in various industrial applications (e.g., mining [21, 13]). Although the level of automation in mining is more advanced than many other domains, human oversight and control is still necessary given various factors such as legacy equipment, interoperability of hardware, and the ability to handle unforeseen circumstances. It is essential to use virtual engineering environments to model the vehicle and environment which can then be used to train drivers [21].

In addition to the known challenges, such as mode error where the driver cannot recall what state the system is in, there are particular challenges posed by semi-autonomous vehicles that merit attention including

- *handover* between manual and automated control during a task [16, 37], which is critical as the driver needs to be able to judge when to reclaim control or otherwise,
- *inadequate feedback* from the vehicle to the driver [22], with the consequence that the system fails on drivers' expectations during a task and ultimately maximum benefit of the technology is not derived, and
- a fundamental change of task for the driver as it changes from monitoring the *situation* to monitoring the *situation and automation* [17].

Most of the work done so far in this area has addressed such challenges in isolation and at a high abstract level [28, 12, 30, 8], has studied vehicle sensor data [37], driver feedback [17, 20, 36, 24, 38, 14] (in a real or simulated environment) or performed physiological assessments [31, 32]. The latter two

strands of work entirely focusses on driver perception and experience; moreover borrowing from separate traditions of cognitive and physiological science.

Another approach to understand the interaction between these various systems is to conduct empirical studies [5]. It is not possible to conduct an empirical study that includes all these parameters. So most empirical psychological and ergonomic studies focus on a few parameters [9, 3, 26]. It is hard to visualise a holistic model from such results. As full system interactive behaviour depends on a number of parameters, it is important to have an analysis method that allows specification of all relevant parameters.

Hence the need to allow for more sophisticated models of driver behaviour and vehicle dynamics, on the one hand, and derive such models from multiple disciplines, on the other, to capture the true nature of the problem. The recognition that effective analysis could only be achieved as such is the main motivation behind our work.

More recent work has acknowledged this. Oppenheim and Shinar [23] develop an approach to model these various aspects that can enable such analyses. They identify a number of parameters for each model that are important, which is addressed later in Section 5.

3 Approach

We adapt the standard discrete-event simulation [25] where we have inputs and outputs and the state of the computation. We generalise the separation between plant and controller, and augment the controller with a human operator allowing us to model systems that explicitly have human in the loop [11]. This allows us to analyse the interaction between the human operator and the control system to achieve safe behaviours. We use model-checking (and reachability analysis [2]) to determine the safety of the entire system.

We propose a *system model* to represent the features and behaviours of the system that we deal with. The importance of this model is argued by Bass et al. [4] as divergence between the state as described by the system model and the user's mental model is often the cause of unsafe behaviour.

This paper has a focus on semi-autonomous vehicles, and our representation allows for specifying relevant driver control (acceleration and braking) and engine control unit features (odometer and service meters) and chassis control data (vehicle handling and steering, and braking and stability sensors). Aspects of autonomy (adaptive cruise control, lane discipline and navigation) can be factored in as part of the system model.

A *behaviour model* is then used to demonstrate user actions driven by cognitive and emotional stimuli. The behaviour modelled is an abstraction of the user's mental model and associated actions relevant to the interaction with the system. Traditionally such models have been derived from cognitive science [28, 12, 30, 8]. However it is increasingly feasible to look to human physiology to sense for driver perception, stress and comfort given advances in sensors [31]. Recent work carried out quantitative physiological assessment of human stress in response to vehicle interface design (ranging from touch display, voice to multimodal control) [32]. Such an approach provides for an objective assessment of the human condition and further possibility of system adaptation for refined interaction.

Ultimately, a model of behaviour could then be drawn from both branches of science to inform the analysis. Our approach permits the integration of different models. We discuss this issue in our case study section. It is important to note that we do not validate the behaviour model. We only check if the joint behaviour of the system – the control system and the human operator – is safe. One can view the behavioural model as documenting the assumptions we make about the human user.

We also propose an *environment model* to account for operational factors external to the system

and the user yet of influence. Such factors are strictly beyond the control of the system or the user, and remain unaffected by any interaction that results. This provides for a clear separation to study the potential impact on the driver and the system as they interact with the environment individually or while interaction.

The various factors influencing each of the models are represented as a set of parameters. Some of these parameters would be derived from the cognitive or physiological model underpinning the behaviour model while others would be directly measured. Assuming there are no circular dependencies, based on the inputs and the current state of the system the outputs and the new state of the system is computed.

A schematic description of our approach is given in Figure 1 and the behaviour of the entire system is expressed using the control loop shown in Figure 2. In our approach we assume that within one iteration the environment can affect the system and the environment and system can affect the user's behaviour.

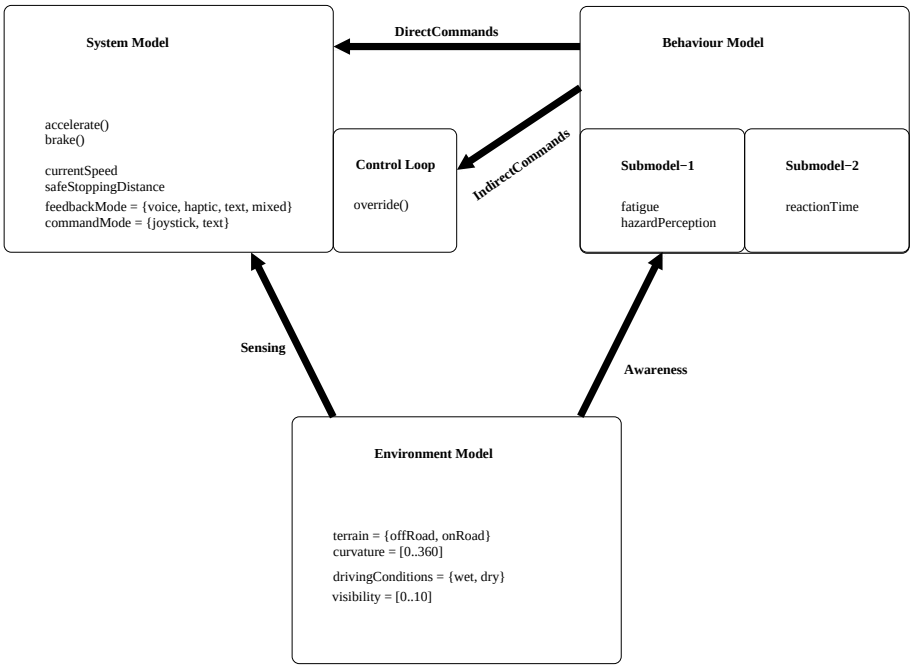


Figure 1: Schematic Representation of Approach.

The entire system is developed in C and we use *CBMC (Bounded Model Checker)* [10] to validate the safety properties. Given that it is a bounded model-checker we either have to specify the number of iterations that need to be explored or have to set up the system such that it can calculate the bound. Thus when CBMC indicates that a system is safe, it is safe only within the bounds specified for the verification. However, if it finds an error (that is, a counterexample that violates the assertion) we can be sure that the system is unsafe.

Our approach allows us to explore various scenarios where safe interaction between the driver and vehicle is critical. Given behaviour, vehicle and environment models, overall system states that lead to safety violations would be explored. Specific scenarios can be generated if the functions that represent the calculation of the model values return a unique value. We also use the CBMC feature that allows a function to return non-deterministic values which effectively allows us to analyse a class of behaviours.

```
loop {
   calculate values from the environmental model
   calculate values from the system model
   calculate values from the behavioural model
   calculate the outputs
   update the state of the system
   update the state of the user
   assert(properties)
   generate the_outputs
}
```

Figure 2: Control Loop.

3.1 Formal Structure

In this section we present some of the formal details associated with our models and the verification process. While the exact details will depend on the system being studied, there are some general principles that underly our approach. We focus on the behavioural and environment models. The system model is straightforward – we assume there is a linear ordering of the variables which is used by the control loop.

The environment model is to used to explore different scenarios. We can either set the variables in the model to specific values or choose (non-deterministically) from a collection of possibilities.

The empirical results that we use are represented as tables. Thus the basic values in the behaviour model are obtained via a table lookup. If there are multiple models that determine a particular value, one has to define a suitable function that combines the values. Choosing this function non-deterministically allows exploration of situations where empirical results are not available.

The environment model consists of variables E_I, E_O and E_S, the system model consists of variables S_I, S_O and S_S while the behaviour model consists of variables B_I, B_O and B_S. We use the subscripts O, I and S to denote output, input and state respectively.

There are two types of assignment statements. The first (of the form $x = e$) evaluates the expression e and assigns the value to the variable x. The second (of the form $x = nondet(P)$) chooses a value from the set of possibilities (P) and assigns to x. As we are using the CBMC model-checker to verify behaviours this is equivalent to checking each and every value from P.

The control loop (shown in Figure 2 for the system consists of the following steps.

1. $E'_I = f_{EI}(S_O, E_S)$ where f_{EI} is a function that calculates the new input values for the environment and E'_I denotes the new set of input values. The user can influence the environment only via the system and hence the outputs of the behaviour model are not used in this calculation. The update function may involve non-deterministic choices.
2. $S'_I = f_{SI}(B_O, E_O, S_S)$ and
3. $B'_I = f_{BI}(B_S, E_O, S_O)$ are similar to the first step. The only restriction is that the system update function is deterministic.
4. The next three steps calculate the outputs of the three models, viz.,
   $E'_O = f_{EO}(E'_I, E_S)$,
   $S'_O = f_{SO}(S'_I, S_S)$,
   $B'_O = f_{BO}(B'_I, B_S)$.

5. The next three steps update the local state of the models. They are,
   $E'_S = f_{ES}(E'_I, E_S)$,
   $S'_S = f_{SS}(S'_I, S_S)$,
   $B'_S = f_{BS}(B'_I, B_S)$.
6. Execute the assertions that encode the safety property.

The various functions shown above essentially capture the three models in our system.

4 Case Study: Driver Fatigue

Enhancing the driver experience through increasing autonomy has been of interest for well over a decade now. Reduction in drive stress, freeing up limited attentional resources and improving road safety have been the major goals of this effort. However, autonomy brings with it a variety of other challenges that potentially risk road safety [33]. This could be due to sensor limitations, system design faults, error inducing design, or inadequate driver training; these certainly are some of the lessons learned from the introduction of autonomy in the aviation domain.

Of interest here is driver fatigue, which results in drowsiness and hypovigilance, particularly after prolonged periods of driving and monotonous roadside experience through motorways [36]. This has been confirmed by a number of studies [20, 18]. This is a major cause of accidents across the world, with around 10-35% of all road accidents in the USA and several European countries estimated to be fatigue and sleep related [1]. While manual strategies adopted by drivers to cope with this problem are recognised [14], fatigue serves to be one of the main factors for increasing autonomy in vehicles [23].

We view each vehicle as a safety critical system where one should avoid accidents as well as complete the mission. Even if the vehicle is unoccupied, a human will be involved in its control. The vehicles are travelling in a convoy and it is important to maintain safe stopping distance. Otherwise, unexpected environment events (like explosions) can cause the driver or human controller to accelerate and run into the vehicle in front of the convoy. The case study studies the effect of driver ability and input modes on the desired vehicle separation for safety given a specific route.

The example we present here focusses on using different actual empirical results that are incorporated into our behavioural models. For instance, the parameters for hazard perception are chosen from [6, 9]. Similarly, the parameters for the control mode are chosen from [3, 6] while the desired separation and safe stopping distance are chosen from [6]. The function that calculates the reaction time from factors such as fatigue is derived from [15, 3, 5]. Other factors such as speed and route are generated (either manually or non-deterministically) as part of the scenario exploration experiment.

The use of sub-models requires us to experiment with hypothetical integration to achieve a single behavioural model. We have to consider hypothetical integration due to lack of suitable models that cover all aspects covered in the sub-models. This separation of actual models and hypothetical integration documents the assumptions under which our analysis is valid. It also points to areas where more precise cognitive models are required.

4.1 Structure of the ACC System Model

The system model has the key aspects of the ACC mainly the control system to maintain speed of the various vehicles, the separation between them and a notion of safe stopping distance to prevent accidents. The environment model has the terrain and hazards that are non-deterministically generated. The

behaviour model we consider is that of fatigue which is calculated from the duration the driver has been in the vehicle and the complexity of the terrain. The driver's cognitive ability is influenced by fatigue and hazard perception. This when combined with the mode to issue the commands determines the reaction time. The current speed of the vehicle and the reaction time of the driver will determine the safe stopping distance.

Adaptive cruise control (ACC) is one of the mechanisms introduced to provide safe distance control from lead vehicle in front: once engaged, the vehicle operates in a typical cruise controlled fashion with the added feature of sensing the vehicle in front to adapt speed (if it slows down or speeds up) ensuring a minimal safe distance at all times. Figure 3 below shows an algorithm for a simple implementation of ACC. The `ACC` algorithm uses the vehicle's on-board sensors to read in the gap from the vehicle in

```
void function ACC(dist_veh_in_front)
{
   loop{
        input current_dist_veh_in_front;
        IF dist_veh_in_front == current_dist_veh_in_front
         CALL Maintain_host_veh_speed;
        END IF
        IF dist_veh_in_front > current_dist_veh_in_front
         CALL Decelerate_host_veh_speed;
        END IF
        IF dist_veh_in_front < current_dist_veh_in_front
         CALL Accelerate_host_veh_speed;
        END IF
   }
}
```

Figure 3: Adaptive Cruise Control (ACC) algorithm in pseudocode

front. This is preset by the driver and passed onto the algorithm as a parameter. Once it enters the loop, it strives to maintain this distance by continuing to maintain the vehicle speed if the gap to the vehicle in front is the same, decelerating if the distance gets narrower than desired or accelerating if the gap is wider.

Studies have demonstrated that ACC has the potential of causing delayed driver reaction [38], and awkward handover and mode confusion with up to a third of drivers having forgotten at some stage whether ACC was engaged or otherwise [17]. This has serious road safety risks and raises a question whether the design of such mechanisms would ultimately serve to be detrimental to the intended goal. In addition to the time on task effects, road conditions and terrain also significantly affect driver experience, and contribute to fatigue [24]; difficult terrains require more frequent driver interventions [29] in semi-autonomous vehicles.

The above points to a clear need to analyse the relationship between driver control and vehicles offering semi-autonomous features such as ACC. The case study proposed to demonstrate the framework presented in this paper revolves around how driver behaviour and perception is affected by fatigue which itself is influenced by factors such as journey time. The behaviour model is influenced by parameters including over time on task, fatigue and perception of ACC status. The system model is parameterised to represent a vehicle equipped with ACC, ranging over speed, acceleration, distance to vehicle in front,

and ACC status.

4.2 Structure of Behaviour Models

The sub-models associated with the behaviour model are now described. The fatigue is calculated using the variables that are identified by Oppenheim and Shinar [23]. Although they do not present empirical results they summarise results from other papers. These results are not specific to the military situation. They are more from generic driving conditions. However, as that is the best data available we use it. Similarly, the role of terrain is identified as an important factor [5, 27]. The role of terrain is explained only in qualitative terms and neither present any concrete model of how the terrain actually affects fatigue. Therefore, there is no obvious way to create a unified behaviour model from the empirical results.

Thus we explicitly define an integrator function that will combine the variables and values indicated in prior work [23, 5, 27]. As the behaviour of this function is not available, we encode various candidate functions. All candidate functions are automatically evaluated in the verification process. Technically, this is achieved using non-deterministic choice in CBMC. So if our analysis indicates that safe stopping distance is always maintained, we can conclude that safety is independent of the candidate functions.

The reaction time calculation is from Baber et al. [3]. The reaction time depends on the mode of communication between the driver and the vehicle control system as well as the derived value for the fatigue factor. Baber et al. [3] present empirical results for mode of communication and reaction time. Here again we rely on a non-deterministic choice of integrator functions to combine the model from Baber et al. [3] and the calculated fatigue value. For the case study the interaction mode is a factor where research shows that in certain circumstances the use of speech is better than using text while the use of joystick or gamepad to control the system is often better than issuing text based commands.

For instance, the empirical results from Baber et al. [3] are encoded to calculate the reaction time is shown in Figure 4. They function is `setReactionTime` which calculates a qualitative reaction time based on the input mode and then invokes `changeReactionTime` with the fatigue level to calculate a quantitative value.

In general we represent the parameters in the various models as global variables and represent the actual calculation of the the values as functions. For instance, `timeDriven` and `terrain` are global integer values. As there are discrete levels of fatigue we use enums to represent the possible values the global variable `driverFatigue` can take. The function `setDriverFatigue` assigns a value to the variable `driverFatigue` based on the behaviour model.

The hazard perception is based directly on Chen [9]. As we are using only a single model there is no need to define an integrator for hazard perception. This ability includes the size of the hazard. In general larger the hazard the easier it is to perceive. We use the variable `hazardPerception` to denote the user's ability to perceive hazards. This can be derived from `driverFatigue`. It is also possible for one to use CBMC's non-determinism to generate a value. An example is the following statement

```
hazardPerception = hpFunction(nondet_int());
```

where the function `hpFunction` converts an integer to a suitable hazard perception value. The function `hpFunction` is a part of f_{BI} in the formal model.

The structure of a non-deterministic integrator function is shown in Figure 5. The variable `opChoice` represents the function to combine the values from the different models and its value is calculated non-deterministically. The actual integration is performed by the function `choiceFunction` which applies the chosen function to the values from the sub-models to return the actual value for the joint model and represents a specific instance of f_{BS} (i.e., the function that calculates the state associated with the user's behaviour).

```
void setReactionTime(mode iM, fatigueLevel df)
/* mode is the input mode used by the driver and
   fatigueLevel is a indication of the driver's  tiredness
   we use symbolic values rather than concrete values for the reactionTime

*/
{
  switch (iM) {
    case GamePad: reactionTime = okay;
    case Speech: reactionTime = slow;
    case MultiModal: reactionTime = fast;
  };
  changeReactionTime(df);
}

void changeReactionTime(fatigueLevel df)
/* this function alters the reaction time based on the
   level of tiredness. These factors can either be chosen from
   empirical results  or can be  set non-deterministically
*/
{
   switch (df) {
     case Exhausted : reactionTime = reactionTime * eFactor;
     case Tired     : reactionTime = reactionTime * tFactor;
     case Normal    : reactionTime = reactionTime * nFactor;
    };
}
```

Figure 4: Encoding of Human Factors.

4.3 Structure of Environmental Models

The environment model allows for variability in scenario to help analysis, including road conditions, terrain or obstacles that result in manual driver interventions) and journey time. The travel scenario is expressed as a sequence (i.e., an array) of route points where each route point has the relevant information (and represented as a struct). A simple example of the structure of the route is shown in Figure 6.

The cause and effect relations, informed by the literature above, are explored to analyse for relative safety thresholds that arise from the interaction of the three models. Potential safety violations in terms of dangerous levels of fatigue and increased likelihood of mode confusion are studied.

4.4 Verification Process

In the context of our framework (shown in Figure 1) the ACC is part of the control loop associated with the system model. The user's commands are, normally, filtered via the control loop except when the user can override the ACC and issue commands directly to the system. This type of behaviour is typically seen when the user is fatigued or encounters a hazardous situation.

```
v1 = model1(p1); /* p1 represents the parameters used by
                    the first model */
v2 = model2(p2); /* p2 represents the parameters used by
                    the second model */
opChoice = nondet_uint() % choices;
return choiceFunction(opChoice,v1,v2);
```

Figure 5: Non-Deterministic Integrator Function

```
struct routePoints{
int obstacle; /* represents difficulty to overcome */
int distance; /* time to travel is calculated using the speed */
int terrain; /* represents offRoad, onRoad */
int curvature;
};

struct routePoints route[5];  /* there are 5 sections to travel */
```

Figure 6: Example of Route.

The behaviour of the system (i.e., the vehicle and the human operator) was analysed for under various conditions (e.g., route choices, travelling speed). We were able to show that a small separation between vehicles was safe if the driver was not too tired and the interaction mode involved game pads. We were also able to show that even a large separation was unsafe if the driver was tired and the interaction mode involved only speech. The safety conditions are written as assertions such as

```
assert(isOkay(driverFatigue,hazardPerception,safeStoppingDistance));
```

where `isOkay` determines if the current separation is safe given the parameters from the behaviour model. These are specific to the application being analysed.

In the function `main` we invoke the function associated with the control loop (shown in Figure 2) with the various scenarios. By varying the variables that are assigned non-deterministic values, one can explore the safety of different scenarios within a given model. We also use CBMC to calculate values where failures occur. For instance, the following program fragment determines if there are any values of `driverFatigue` that can lead to accidents.

```
int driverFatigue = nondet_uint()%numFatigueValues;
assert(isOkay(driverFatigue,hazardPerception,safeStoppingDistance));
```

CBMC will calculate a value for `driverFatigue` that will violate the assertion. If CBMC cannot find such a violation, we are sure that the composite behaviour represented by the models is safe.

5 Results of Case Study

We evaluate our approach to show how effective it is in capturing the interaction between the various inputs provided to the sub-models. The resulting analysis should allow us to check whether the composite system demonstrates safe behaviours. We consider three scenarios for analysis, which are essentially

an exploration into the relationship between fatigue and the various parameters that can be influenced. Evidence suggests that prolonged periods of driving contribute to an increase in fatigue [36, 20, 18] and mechanisms such as ACC available to address such problems need to be evaluated for their effectiveness and safety.

We recall a few select inputs and variables of the system of interest to us here. *Route* denotes the distance travelled by the vehicle over a given journey, and $speed$ is the speed of the vehicle travels at. The *control mode* is used to signal whether the vehicle is driven manually or with ACC enabled. The $safe\ stopping\ distance$ is the estimated safe distance between the host vehicle and the vehicle in front under the conditions of fatigue. This is related to $desired\ separation$ which is understood to be the minimum safe distance (ranging over given vehicle speed). We represent the driver state using a combination of variables including $fatigue$ to show the level of driver tiredness, *hazard perception* to show the driver's ability to perceive hazards encountered enroute, and *reaction time* as a measure of the time taken for the driver to respond. The variables are defined in relative terms for symbolic representation.

We manipulate the above parameters in different analyses by configuring them as constants where we intend them to be fixed for the purpose of the scenario. To range over values, we use non-deterministic assignment. In cases where values are determined by the respective models, we leave the variables assigned accordingly, either as a result of a fixed model (where assignments are straightforward) or as a result of sub-models combined using a variety of functions chosen non-deterministically.

Note that our implementation of the ACC is adapted to be sensitive to driver fatigue over the course of a journey. Implicit here is that the speed of the vehicle is allowed to be adjusted and the desired separation distance is also left for calculation accordingly. The rule is that as the journeys progresses, the desired separation is extended to account for the increase in fatigue, which in turn contributes to weakened hazard perception.

In all cases we analyse for two properties. First, that the desired separation is always maintained for the given control mode and level of fatigue. Secondly, that the use of ACC does not result in an actual increase of driver fatigue.

5.1 Lowered speed and increased fatigue

Our first scenario deals with an unexpected side-effect of the ACC operation whereby the ability to adapt vehicle speed results in an increase in journey time, and hence fatigue.

We configure the system for a non-deterministic route, and leave the level of driver fatigue to be derived by the system output. We want to check whether the driver fatigue goes past a threshold. The control mode remains enabled for ACC, and the safe stopping and desired separation distances also remain fixed. We let the speed of the vehicle adjusted as per the operation of the ACC.

Our analysis reveals that the scenario fails to satisfy the second property. To maintain the desired separation distance the vehicle speed is reduced by the duration that increases. This leads to increased driver fatigue. At the next point on the route the safe separation may need to be increased owing to increased fatigue. This in turn further reduces the speed. After a few iterations the vehicle speed is slowed to such a level that fatigue due to time on road becomes unacceptable.

5.2 Manual override and variable speed

An alternative scenario is where the driver's ability to override ACC at any stage of the journey is acknowledged. This is essentially to model for cases where driver actions may have undesired conse-

quences.

We consider a fixed route, and the safe stopping and desired separation distances also remain fixed. We allow for the control mode to be non-deterministic and have no control over the choice operated. Driver fatigue is then a calculation based on various inputs from the behaviour and system models. Speed is also dependent on the choice of control mode and ultimately the driver, and so are the rest of the variables.

Our analysis reveals that the first property is violated. Counter examples are due to a case where the driver is able to manually override ACC and increase vehicle speed, which results in unsafe distance from the vehicle in front. A different possibility is where the driver switches over to manual mode and ultimately reaches an unsafe state (due to fatigue for example). In one sense such a possibility is difficult to avoid, unless speed or proximity alerts are modelled.

5.3 The ideal scenario

We consider a final scenario where we control the parameters for a best case scenario: the route and control mode (ACC) are both fixed. Driver fatigue is calculated as influenced by a combination of system, behaviour and environmental models. All other parameters are calculated from the relevant model respectively.

We are able to confirm that both properties are satisfied. The fixed length of route means that journey time is ultimately limited, even if speed is adapted (slowed) in response to driver fatigue as discussed in Section 5.1. The fixed control mode helps to avoid any driver-led errors as in Section 5.2.

5.4 Discussion

For all scenarios, CBMC is able to verify the safety property or calculate a counter example in less than five seconds on a low end machine (an Intel U1400 processor running at 1.2GHz with 1.0GB RAM running Linux). Our typical loop unwinding parameter is 100 which is enough to explore all behaviours. Unfortunately, given the calculations involved CBMC is unable to prove the unwinding assertion.

There are two main limitations of this case study. The first is getting an appropriate system model as such models are largely proprietary. We have constructed this model from various published sources.

The second is the choice of “sensible” ways to combine sub-models While our framework can explore any function, it is not clear what classes of functions are close to reality. We have explored only simple arithmetic operators for the two integrator functions mentioned earlier, namely, the safe stopping distance and fatigue. These operators can be chosen non-deterministically. This is a clear limitation in our case study. Technically the integrator functions need empirical validation but owing to the number of parameters such validation studies are hard to perform.

6 Conclusion

The main contribution of this work is an approach to modelling and design of human in the loop systems. The approach takes into account real systems as well as cognitive models that are supported by empirical studies.

By expressing the models in the C programming language we are able to encode all empirical models without compromising on numerical accuracy. We have had informal discussions with practising engineers from the automotive domain and the general feed back is that the approach is simple enough to be used by them. Their models are expressed as code fragments that can be translated quite easily to C.

We also avoid concurrency related issues using a standard sense-control cycle approach. This simplification (which reduces the state space that needs to be explored) when combined with CBMC enables us to handle larger realistic models. The only care required is the sequence in which the variables are updated. Using C is an advantage of our approach over other approaches that use specific modelling languages [12, 4].

This approach supports automatic verification of safety properties as well as systematic scenario exploration. The non-deterministic choices of the various functions in the behaviour sub-models document the assumptions that we make on the interaction between different aspects. These interactions are not supported by empirical studies but occur in real systems.

The case study was chosen to demonstrate our approach. Further work beyond this early exploration would look to both adopt more mature models of behaviour with respect to semi-autonomous vehicles [34, 35], and draw parallels to research that has addressed intricate problems in this area from other domains [19] which have traditionally relied on driver experience and feedback to evaluate such vehicles.

We are also currently exploring the encoding of more specific models, for instance, where the adaptive cruise control system is actually available (and hence we do not need to model it) or automobiles with in-wheel motors where the dynamics are more complex. The aim is to integrate the formal analysis of user interaction with relevant standards for reliability including the automotive safety integrity level requirements.

Acknowledgement

We are grateful to Graham Shelton-Rayner and Damian Harty for useful discussions. Padmanabhan Krishnan was partially supported by a grant from Coventry University, UK.

References

[1] (2001): *Driver Fatigue and Road Accidents*. Technical Report BAE Research Report 12.1, The Royal Society for the Prevention of Accidents.

[2] M. Althoff & J. M. Dolan (2011): *Set-based computation of vehicle behaviors for the online verification of autonomous vehicles*. In: *14th International IEEE Conference on Intelligent Transportation Systems (ITSC)*, IEEE, pp. 1162–1167.

[3] C. Baber, C. Morin, M. Parekh, M. Cahillane & R.J. Houghton (2011): *Multimodal control of sensors on multiple simulated unmanned vehicles*. *Ergonomics* 54(9), pp. 792–805.

[4] E. J. Bass, E. Gunter K. M. Feigh & J. Rushby (2011): *Formal Modeling and Analysis for Interactive Hybrid Systems*. In: *Proceedings of the Fourth International Workshop on Formal Methods for Interactive Systems (FMIS), Electronic Communications of the EASST (ECEASST)*, 45, EASST.

[5] M. A. Cahillane, C. Baber & C. Morin (2012): *Human Factors in UAVs*. In P. Angelov, editor: *Sense and Avoid in UAVs: Research and Applications*, John Wiley and Sons.

[6] Y. Cao, A. Mahr, S. Castronovo, M. Theune, C. Sthal & C. Müller (2010): *Local danger warnings for drivers: The effect of modality and level of assistance on driver reaction*. In: *The International Conference on Intelligent User Interfaces (IUI)*, ACM, Hong Kong, China, pp. 239–248.

[7] P. Carayon, editor (2011): *Human Factors and Ergonomics in Health Care and Patient Safety*, second edition. CRC Press.

[8] A. Cerone (2011): *Closure and Attention Activation in Human Automatic Behaviour: A Framework for the Formal Analysis of Interactive Systems*. In: *Proceedings of the Fourth International Workshop on Formal Methods for Interactive Systems (FMIS), Electronic Communications of the EASST (ECEASST)*, 45.

[9] J. Y. C. Chen (2010): *UAV-guided Navigation for Ground Robot Tele-operation in a Military Reconnaissance Environment*. *Ergonomics* 53(8), pp. 940–950.

[10] E. Clarke, D. Kroening & F. Lerda (2004): *A tool for checking ANSI-C programs*. In K. Jensen & A. Podelski, editors: *TACAS*, *LNCS* 2988, pp. 168–176.

[11] L. Cranor (2008): *A Framework for Reasoning About the Human in the Loop*. In: *Proceedindgs of the 1st Conference on Usability, Psychology and Security*, USENIX, pp. 1–15.

[12] P. Curzon, R. Ruksenas & A. Blandford (2007): *An Approach to Formal Verification of Human-Computer Interaction*. *Formal Aspects of Computing* 19(4), pp. 513–550.

[13] B. S. Fisher & S. Schnittger (2012): *Autonomous and Remote Operation Technologies in the Mining Industry: Benefits and Costs*. Technical Report BAE Research Report 12.1, BAEconomics.

[14] P. Gershon, D. Shinar, T. Oron-Gilad, Y. Parmet & A. Ronen (2011): *Usage and perceived effectiveness of fatigue countermeasures for professional and nonprofessional drivers*. *Accident Analysis & Prevention* 43(3), pp. 797–803.

[15] C. D. Griffith & S. Mahadevan (2011): *Inclusion of fatigue effects in human reliability analysis*. *Reliability Engineering and System Safety*, pp. 1437–1447.

[16] Annika F.L. Larsson (2010): *Issues in Reclaiming Control from Advanced Driver Assistance Systems*. In: *European Conference on Human Centred Design for Intelligent Transport Systems*, 2, pp. 557–564.

[17] Annika F.L. Larsson (2012): *Driver usage and understanding of adaptive cruise control*. *Applied Ergonomics* 43(3), pp. 501 – 506.

[18] J.M. Lyznicki, T.C. Doege, R.M. Davis & M.A. Williams (1998): *Sleepiness, Driving and Motor Vehicle Crashes*. *Journal of the American Medical Association* 279(23), pp. 1908–13.

[19] Greg Marsden, Mike McDonald & Mark Brackstone (2001): *Towards an understanding of adaptive cruise control*. *Transportation Research Part C: Emerging Technologies* 9(1), pp. 33 – 51.

[20] S. Milosevic (1997): *Driver Fatigue Studies*. *Ergonomics* 40(3), pp. 381–389.

[21] P. Moore, S. Ujvari, J. Pu, J-O. Lundgren & C. Xie (1999): *Intelligent semi-autonomous vehicles in materials handling*. *Mechatronics* 9(7), pp. 881–892.

[22] D.A. Norman, editor (2007): *The Design of Future Things: Author of The Design of Everyday Things*, first edition. Basic Books.

[23] I. Oppenheim & D. Shinar (2011): *A context-sensitive model of driving behaviour and its implications for in-vehicle safety systems*. *Cognition, Technology and Work*, pp. 1–21.

[24] T. Oron-Gilad & A. Ronen (2007): *Road Characteristics and Driver Fatigue: A Simulator Study*. *Traffic Injury Prevention* 8(3), pp. 281–289.

[25] P. J. G. Ramadge & W. M. Wonham (1989): *The Control of Discrete Event Systems*. *Proceedings of the IEEE* 77(1), pp. 81–98.

[26] A. Rauschert, C. Meitinger & A. Schulte (2008): *Experimentally Discovered Operator Assistance Needs in the Guidance of Cognitive and Cooperative UAVs*. In: *Conference on Humans Operating Unmanned Systems (HUMOUS08)*.

[27] M. M. Rohde, V. E. Perlin, K. D. Iagnemma, R. M. Lupa, S. M. Rohde, J. Overhold & G. Fiorani (2008): *PointCom: Semi-Autonomous UGV Control with Intuitive Interface. In Unmanned Systems Technology*. In: *SPIE*.

[28] John Rushby (2002): *Using model checking to help discover mode confusions and other automation surprises*. *Reliability Engineering & System Safety* 75(2), pp. 167 – 177.

[29] J. Scholtz, B. Antonishek & J. Young (2004): *Operator interventions in autonomous off-road driving: effects of terrain*. In: *International Conference on Systems, Man and Cybernetics*, 3, IEEE, pp. 2797–2802.

[30] Siraj Ahmed Shaikh, Padmanabhan Krishnan & Antonio Cerone (2007): *A Formal Approach to Human Error Recovery*. In Paul Curzon & Antonio Cerone, editors: *Preproceedings of the 2nd International Workshop on Formal Methods for Interactive Systems (FMIS 2007), Lancaster, UK, September 4, 2007 (as a satellite event of HCI 2007)*, Technical Report RR-07-08, Department of Computer Science, Queen Mary, University of London, pp. 101–115.

[31] G. K. Shelton-Rayner (2009): *Quantifying exposure to psychological and physiological stress and automotive design*. Ph.D. thesis, Coventry University.

[32] G. K. Shelton-Rayner, R. Mian, S. Chandler, D. Robertson & David W. Macdonald (2011): *Quantitative Physiological Assessment of Stress Via Altered Immune Functioning Following Interaction With Differing Automotive Interface Technologies*. *International Journal of Human-Computer Interaction* 27(9), pp. 900–919.

[33] N. A. Stanton & P. Marsden (1996): *From fly-by-wire to drive-by-wire: safety implications of automation in vehicles*. *Safety Science* 24, pp. 35–49.

[34] Neville A. Stanton & Mark S. Young (2000): *A proposed psychological model of driving automation*. *Theoretical Issues in Ergonomics Science* 1(4), pp. 315–331.

[35] Neville A. Stanton & Mark S. Young (2005): *Driver behaviour with adaptive cruise control*. *Ergonomics* 48(10), pp. 1294–1313.

[36] P. Thiffault & J. Bergeron (2003): *Monotony of road environment and driver fatigue: a simulator study*. *Accident Analysis & Prevention* 35(3), pp. 381–391.

[37] F. Viti, S.P. Hoogendoorn, T.P. Alkim & G. Bootsma (2008): *Driving behavior interaction with ACC: results from a Field Operational Test in the Netherlands*. In: *Intelligent Vehicles Symposium, 2008 IEEE*, pp. 745–750.

[38] Mark Vollrath, Susanne Schleicher & Christhard Gelau (2011): *The influence of Cruise Control and Adaptive Cruise Control on driving behaviour A driving simulator study*. *Accident Analysis & Prevention* 43(3), pp. 1134 – 1139.

Formal Model-Driven Engineering: Generating Data and Behavioural Components

Chen-Wei Wang
McMaster Centre for Software Certification,
McMaster University
Hamilton, Canada L8S 4K1
jackie@cse.yorku.ca

Jim Davies
Department of Computer Science,
University of Oxford
Oxford, United Kingdom OX1 3QD
jim.davies@cs.ox.ac.uk

Model-driven engineering is the automatic production of software artefacts from abstract models of structure and functionality. By targeting a specific class of system, it is possible to automate aspects of the development process, using model transformations and code generators that encode domain knowledge and implementation strategies. Using this approach, questions of correctness for a complex, software system may be answered through analysis of abstract models of lower complexity, under the assumption that the transformations and generators employed are themselves correct. This paper shows how formal techniques can be used to establish the correctness of model transformations used in the generation of software components from precise object models. The source language is based upon existing, formal techniques; the target language is the widely-used SQL notation for database programming. Correctness is established by giving comparable, relational semantics to both languages, and checking that the transformations are semantics-preserving.

1 Introduction

Our society is increasingly dependent upon the behaviour of complex software systems. Errors in the design and implementation of these systems can have significant consequences. In August 2012, a 'fairly major bug' in the trading software used by Knight Capital Group lost that firm $461m in 45 minutes [16]. A software glitch in the anti-lock braking system caused Toyota to recall more than 400,000 vehicles in 2010 [25]; the total cost to the company of this and other software-related recalls in the same period is estimated at $3bn. In October 2008, 103 people were injured, 12 of them seriously, when a Qantas airliner [3] dived repeatedly as the fly-by-wire software responded inappropriately to data from inertial reference sensors. A modern car contains the product of over 100 million lines of source code [4], and in the aerospace industry, it has been claimed that "the current development process is reaching the limit of affordability of building safe aircraft" [11].

The solution to the problems of increasing software complexity lies in the automatic generation of correct, lower-level software from higher-level descriptions: precise *models* of structure and functionality. The intention is that the same generation process should apply across a class of systems, or at least multiple versions of the same system. Once this process has been correctly implemented, we can be sure that the behaviour of the generated system will correspond to the descriptions given in the models. These models are strictly more abstract than the generated system, easier to understand and update, and more amenable to automatic analysis. This *model-driven* approach [12] makes it easier to achieve correct designs and correct implementations. Despite the obvious appeal of the approach, and that of related approaches such as domain-specific languages [9] and software product lines [19], much of the code that could be generated automatically is still written by hand; even where precise, abstract specifications exist, their implementation remains a time-consuming, error-prone, manual process.

Submitted to:
FTSCS 2012

The reason for the delay in uptake is simple: in any particular development project, the cost of producing a new language and matching generator, is likely to exceed that of producing the code by hand. As suitable languages and generators become available, this situation is changing, with significant implications for the development of complex, critical, software systems. In the past, developers would work to check the correctness of code written in a general-purpose programming language, such as C or Ada, against natural language descriptions of intended functionality, illuminated with diagrams and perhaps a precise, mathematical account of certain properties. In the future, they will check the correctness of more abstract models of structure and behaviour, written in a range of different, domain-specific languages; and rather than relying upon the correctness of a single, widely-used compiler, they will need to rely upon the correctness of many different code generators. The correctness of these generators, usually implemented as a sequence of model transformations, is thus a major, future concern.

In this paper, we present an approach to model-driven development that is based upon formal, mathematical languages and techniques. The objective is the correct design and implementation of components with complex state, perhaps comprising a large number of inter-related data objects. The approach is particularly applicable to the iterative design and deployment of systems in which data integrity is a primary concern. The modelling language employed has the familiar, structural features of object notations such as UML—classes, attributes, and associations—but uses logical predicates to characterise operations. An initial stage of transformation replaces these predicates with guarded commands that are guaranteed to satisfy the specified constraints: see, for example, [24]. The focus here is upon the subsequent generation of executable code, and the means by which we may prove that this generation process is correct.

The underlying thesis of the approach is that the increasing sophistication of software systems is often reflected more in the complexity of data models than in the algorithmic complexity of the operations themselves. The intended effect of a given event or action is often entirely straightforward. However, the intention may be only part of the story: there may be combinations of inputs and before-states where the operation, as described, would leave the system in an inconsistent after-state; there may be other attributes to be updated; there may be constraints upon the values of other attributes that need to be taken into account. Furthermore, even if the after-state is perfectly consistent, the change in state may have made some other operation, or sequence of operations, inapplicable.

Fortunately, where the intended effect of an operation upon the state of a system is straightforward, it should be possible to express this effect as a predicate relating before and after values *and* generate a candidate implementation. Using formal techniques, we may then calculate the domain of applicability of this operation, given the representational and integrity constraints of the data model. If this is smaller than required, then a further iteration of design is called for; if not, then the generated implementation is guaranteed to work as intended. In either case, code may be generated to throw an exception, or otherwise block execution, should the operation be called outside its domain. Further, by comparing the possible outcomes with the calculated domains of other operations, we can determine whether or not one operation can affect the availability of others.

The application of formal techniques at a modelling level—to predicates, and to candidate implementations described as abstract programs—has clear advantages. The formal semantics of a modern programming language, considered in the context of a particular hardware or virtual machine platform, is rich enough to make retrospective formal analysis impractical. If we are able to establish correctness at the modelling level, and rely upon the correctness of our generation process, then we may achieve the level of formal assurance envisaged in new standards for certification: in particular, DO-178C [21]. We show here how the correctness of the process can be established: in Section 3, we present the underlying semantic framework; in Section 4, the translation of expressions; in Section 5, the implementation of operations; in Section 6, the approach to verification.

2 Preliminaries

The BOOSTER language [6] is an object modelling notation in which model constraints and operations are described as first-order predicates upon attributes and input values. Operations may be composed using the logical combinators: conjunction, disjunction, implication, and both flavours of quantification. They may be composed also using relational composition, as sequential phases of a single operation or transaction. The constraints describing operations are translated automatically into programs written in an extended version of the Generalised Substitution Language (GSL), introduced as part of the B Method [1]. There may be circumstances under which a program would violate the model constraints, representing business rules, critical requirements, or semantic integrity properties. Accordingly, a guard is calculated for each operation, as the weakest precondition for the corresponding, generated program to maintain the model constraints. The result is an abstract program whose correctness is guaranteed, in a language defined by the following grammar:

$$
\begin{array}{lcll}
\mathit{Substitution} & ::= & \mathit{skip} & \mid \ \langle\langle \mathit{PATH} \rangle\rangle := \langle\langle \mathit{Expression} \rangle\rangle \\
& \mid & \langle\langle \mathit{Predicate} \rangle\rangle \longrightarrow \langle\langle \mathit{Substitution} \rangle\rangle & \mid \ \langle\langle \mathit{Substitution} \rangle\rangle \parallel \langle\langle \mathit{Substitution} \rangle\rangle \\
& \mid & \langle\langle \mathit{Substitution} \rangle\rangle \ ; \ \langle\langle \mathit{Substitution} \rangle\rangle & \mid \ \langle\langle \mathit{Substitution} \rangle\rangle \ \Box \ \langle\langle \mathit{Substitution} \rangle\rangle \\
& \mid & !\langle\langle \mathit{Variable} \rangle\rangle : \langle\langle \mathit{Expression} \rangle\rangle \bullet \langle\langle \mathit{Substitution} \rangle\rangle & \\
& \mid & @\langle\langle \mathit{Variable} \rangle\rangle : \langle\langle \mathit{Expression} \rangle\rangle \bullet \langle\langle \mathit{Substitution} \rangle\rangle &
\end{array}
$$

Here, the usual notation of assignable variables is replaced with *paths*, each being a sequence of attribute names, using the familiar object-oriented 'dot' notation as a separator. *Predicate* and *Expression* represent, respectively, first-order predicates and relational and arithmetic expressions. *skip* denotes termination, $:=$ denotes assignment, and $\longrightarrow$ denotes a program guard: to be implemented as an assertion, a blocking condition, or as (the complement of) an exception. $\Box$ denotes alternation, and @ denotes selection: the program should be executed for exactly one of the possible values of the bound variable. Similarly, $\parallel$ denotes parallel composition, with ! as its generalised form: all of the program instances should be performed, in parallel, as a single transaction. ; denotes relational or sequential composition. Inputs and outputs to operations need not be explicitly declared; instead, they are indicated using the decorations ? and ! at the end of the attribute name.

These abstract programs are interpreted as operations at a component applications programming interface (API), with the data model of the component given by a collection of class and association declarations in the usual object-oriented style. The integrity constraints and business rules for the data model can be given as predicates in the same notation, or using the object constraint language (OCL) of the Unified Modelling Language (UML) [12].

As a simple, running example, consider the following description of (a fragment of) the data model for a hotel reservations system

```
class Hotel {
  attributes
    reservations : seq(Reservation.host) [*] }
```

```
class Reservation {
  attributes
    host : Hotel.reservations }
```

A single hotel may be the **host** for any number of reservations. It may also be the **host** of a number of rooms and allocations: see the class association graph [10] of Figure 1. The action of creating a new reservation may be specified using a simple operation predicate in the context of the `Hotel` class:

```
reserve { # allocations < limit & reservations' = reservations ^ <r!> & r!.room = m? }
```

This requires that a new reservation be created and appended to the existing list, modelled as an ordered association from `Hotel` to `Room`, and that the room involved is given by input `m?`. The operation should not be allowed if the number of reservations in the system has already reached a specified `limit`.

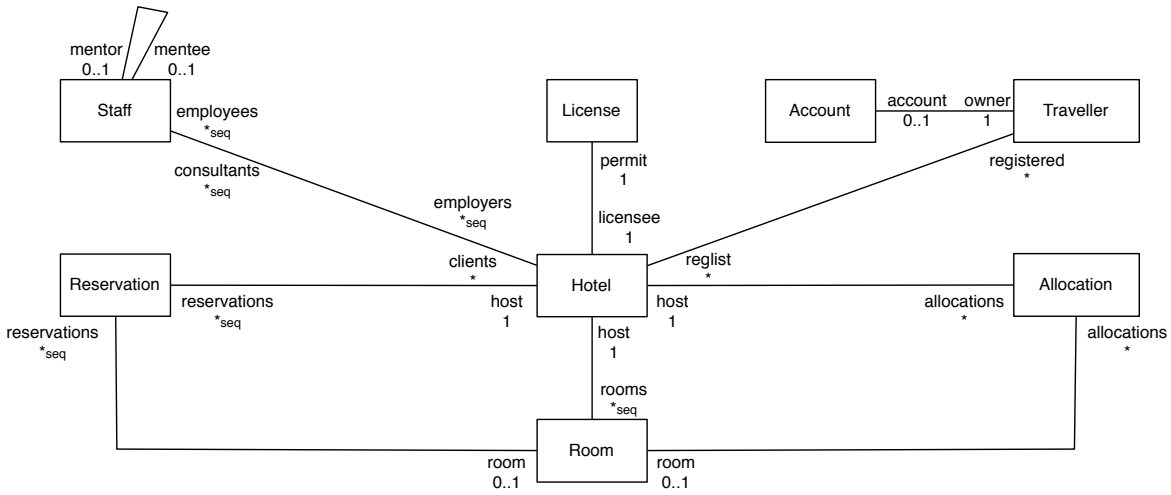


Figure 1: Hotel Reservation System (HRS)—Graph of Class Associations

If the constructor operation predicate on `Reservation` mentions a set of dates `dates?`, then this will be added as a further input parameter. We might expect to find also a constraint insisting that any two different reservations associated with the same room should have disjoint sets of dates, and perhaps constraints upon the number of reservations that can held by a particular traveller for the same date. For the purposes of this paper, however, we will focus simply upon the required, consequential actions and the description of the operation as an abstract program.

```
reserve {
     r! : extent(Reservation) & dates? : set(Date) & m? : extent(Room)
       & card(allocations) < limit
  ==>
     r!.dates := r!.dates \/ dates? || r!.status := "unconfirmed"
  || r!.host := this || reservations := ins(reservations, #reservations + 1, r!)
  || r!.room := m?   || m?.reservations := ins(m?.reservations, #m?.reservations + 1, r!)}
```

In this abstract program, the two reservations attributes, in the hotel and room objects, are updated with a reference to the new reservation, the dates attribute of the new reservation is updated to include the supplied dates, and the status attribute is set to `"unconfirmed"`, presumably as a consequence of the constructor predicate for the `Reservation` class.

3 A Unified Implementation and Semantic Framework

To illustrate our formal, model-driven approach, we will consider the case in which the target is a relational database platform. The above program would then be translated into a SQL query, acting on a relational equivalent of our original object model. The transformations can be described using the Haskell [2] functional programming language: in the diagram of Figure 2, thin-lined, unshaded boxes represent to denote Haskell program data types, and thin arrows the executable transformations between them. These constitute an *implementation* framework. The thick-lined, shaded boxes denote the relational semantics of corresponding data types, thick lines with circles at one end the process of assigning a formal meaning, and arrows with circles at each end the relationship between formalised concepts. These constitute a corresponding *semantic* framework for establishing correctness.

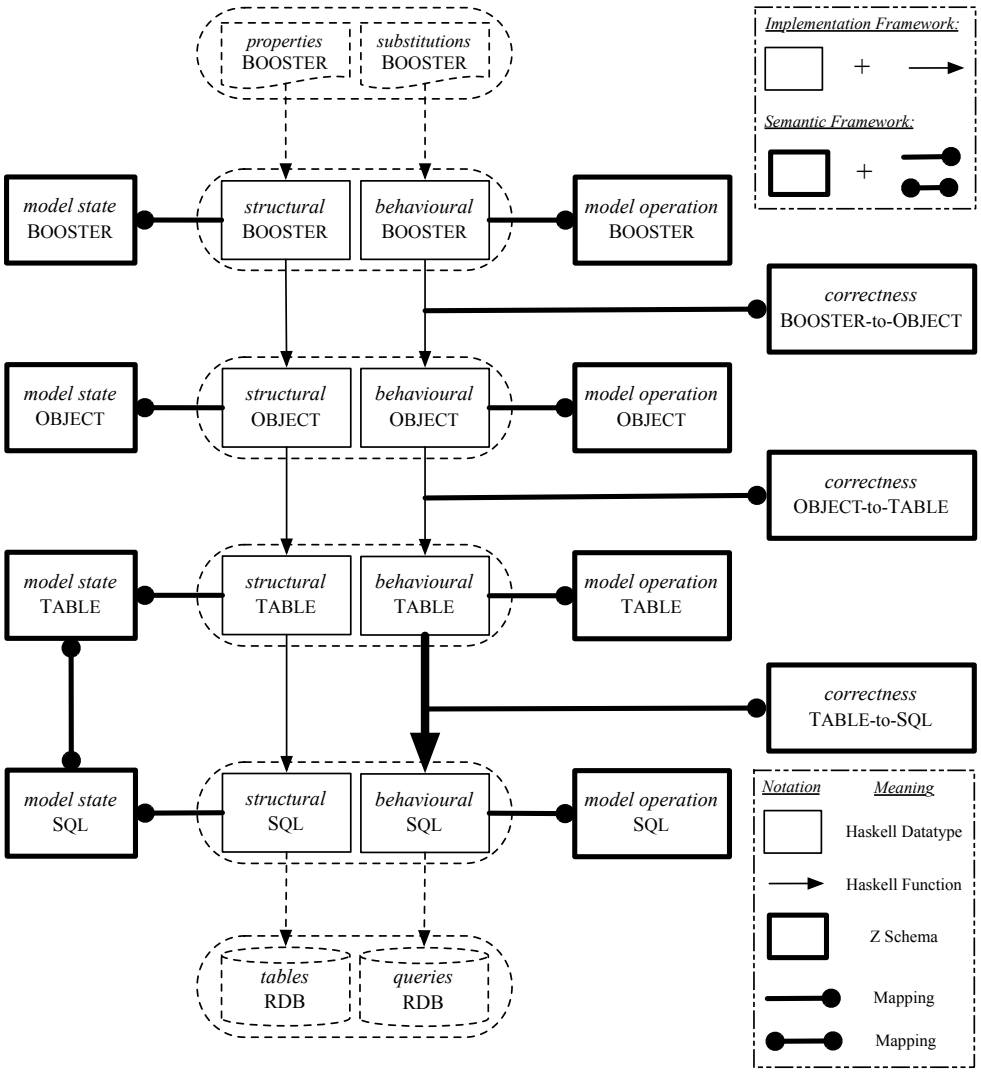


Figure 2: BOOSTER Model to SQL Database: Implementation & Semantic Framework

Four kinds of models are involved in our transformation pipeline: 1) a BOOSTER model, in extended GSL notation, generated from the original predicates; 2) an OBJECT model representing an object-oriented relational semantics for that model; 3) an intermediate TABLE model reflecting our implementation strategy; 4) a SQL model expressed in terms of tables, queries, and key constraints. A final model-to-text transformation will be applied to generate a well-formed SQL database schema.

We use Haskell to define metamodels of model structures and operations as data types. Our transformations are then defined as Haskell functions: from BOOSTER to OBJECT, then to TABLE, and finally to SQL. Our relational semantics is most easily described using the Z notation [26]. Other formal languages with a transformational semantics would suffice for the characterisation of model and operation constraints, but Z has the distinct advantage that each operation, and each relation, may be described as a single predicate: rather than, for example, a combination of separate pre- and post-conditions; this facilitates operation composition, and hence a compositional approach to verification.

4 Path & Expression Transformation

The descriptions of operations in the BOOSTER, OBJECT, and TABLE models are all written in the GSL language; the difference between them lies in the representation of attribute and association references. Instead of creating three versions of a language type `Substitution`, one for each of the reference notations, we employ a type `PATH` as a generic solution: see Figure 3.

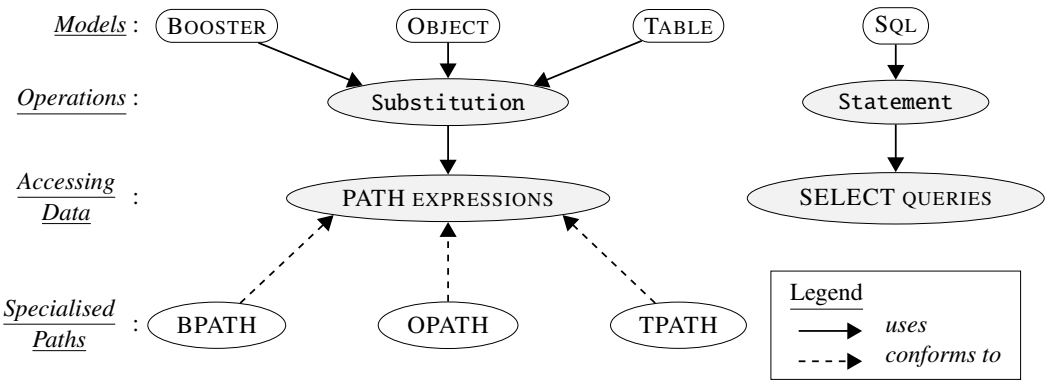


Figure 3: Datatypes of Behavioural Models

We define

```
data PATH = BPath BPATH | OPath OPATH | TPath TPATH
```

where **BPath**, **OPath**, and **TPath** are type constructors. A BOOSTER model path (of type `BPATH`) is represented as a sequence $\langle a_1, \ldots, a_n \rangle$ of name references to attributes/properties. We will refer to this range $1 \ldots n$ of indices for explaining the corresponding OBJECT and TABLE model paths.

We consider structures of the types `OPATH` and `TPATH` in detail. Paths of type `OPATH` are used to indicate explicitly which properties/classes are accessed, along with its chain of navigation starting from the current class.

```
data OPATH          = BaseOPath REF_START         | RecOPath OPATH TARGET
data REF_START      = ThisRef BASE                | SCRef    IDEN_PROPERTY EXPRESSION BASE
data TARGET         = EntityTarget IDEN_PROPERTY | SCTarget IDEN_PROPERTY EXPRESSION
data BASE           = ClassBase N_CLASS           | SetBase N_SET | IntBase | StrBase
type IDEN_PROPERTY = (N_CLASS, N_ATTRIBUTE)
```

An object path is a left-heavy binary tree, where the left-most child refers to its starting reference and all right children represent target classes/properties that are accessed. The starting reference of an object path—which denotes access to, e.g the current object, an element of a sequence-valued property through indexing, etc.—provides explicit information about the base type of that reference. All intermediate and the ending targets of an object path contextualise the properties with their enclosing classes (i.e. `IDEN_PROPERTY`).

For each context path $\langle a_1, \ldots, a_i \rangle$, where $(1 \leq i \leq n-1)$, an OBJECT model path (of type `OPATH`) identifies a target class C; if the source BOOSTER path is valid, then attribute a_{i+1} must have been declared in C.

Example object path. As an example of how the transformation on paths works in practice, consider the `Account` class (Figure 1 shown on page 4). The path `this.owner.reglist` denotes a list of registered hotels and has its `OPATH` counterpart:

```
RecOPath (RecOPath (BaseOPath (ThisRef (ClassBase Account)))
                   (EntityTarget (Account, owner)))
         (EntityTarget (Traveller reglist))
```

where **RecOPath** and **BaseOPath** are constructors for, respectively, recursive and base OBJECT paths. **EntityTarget** and **ClassBase** construct type information about the three context paths: `(Account)` for `this`, `(Account, owner)` for `this.owner`, and `Traveller, reglist` for `this.owner.reglist`.

On the other hand, we use a path of type `TPATH` to indicate, for each navigation to a property in the OBJECT model, the corresponding access to a table which stores that property.

```
data TPATH    = BaseTPath                  REF_START
              | RecTPath     TPATH         T_ACCESS
data T_ACCESS = ClassTAccess IDEN_PROPERTY
              | AssocTAccess IDEN_PROPERTY
              | SetTAccess   IDEN_PROPERTY
              | SeqTAccess   IDEN_PROPERTY             -- retrieve all indexed components
              | SeqTCAccess  IDEN_PROPERTY EXPRESSION  -- retrieve an  indexed component
```

A table path is left-heavy (as is an `OPATH`), where the left-most child refers to its starting reference and all right children represent target tables that are accessed. The starting reference of a table path provides exactly the same information as its `OPATH` counterpart (i.e. `REF_START`). All intermediate and the ending targets of a table path denote accesses to a variety of tables, predicated upon our implementation strategy. When the target property is sequence-valued, we distinguish between the two cases where one of its indexed components is to be accessed (`SeqTCAccess`) and where all indexed components are to be accessed (`SeqTAccess`).

For each attribute a_i, where $(1 \leq i \leq n)$, a TABLE model path (of type `TPATH`) recursively records which sort of table (e.g. class tables, association tables, or set tables) it is stored, based on the target class of its context path.

Example table path. The above OBJECT path has its `TPATH` counterpart:

```
RecTPath (RecTPath (BaseTPath (ThisRef (ClassBase Account)))
                   (AssocTAccess (Account, owner)))
         (AssocTAccess (Traveller reglist))
```

where **RecTPath** and **BaseTPath** construct, respectively, recursive and base TABLE paths. Properties `owner` and `reglist` are accessed in the two corresponding association tables.

Path transformation. We now specify the above `OPATH`-to-`TPATH` transformation in Haskell:

```
objToTabPath :: OBJECT_MODEL -> PATH -> PATH
objToTabPath om (OPath opath) = TPath (objToTabPath' om opath)
```

where the first line declares a function `objToTabPath`, and the second line gives its definition: matching an input object model as `om` and an input path as `(OPath opath)`, whereas the RHS constructs a new `PATH` via `TPath`. The transformation of object paths is given by

```
objToTabPath' om (RecOPath op tar) =
  case tar of
    EntityTarget (c, p) | (c, p) `elem` biAssoc'    om c -> RecTPath tp (AssocTAccess (c, p))
                        | (c, p) `elem` classTables tm c -> RecTPath tp (ClassTAccess (c, p))
                        | (c, p) `elem` setTables   tm   -> RecTPath tp (SetTAccess   (c, p))
                        | (c, p) `elem` seqTables   tm   -> RecTPath tp (SeqTAccess   (c, p))
    SCTarget (c, p) oe  -> let te = objToTabExpr om oe   in RecTPath tp (SeqTCAccess (c, p) te)
  where tm = objToTab      om
        tp = objToTabPath' om op
```

where each condition specified between `|` and `->` denotes a special case of the matched entity target, consisting of class `c` and property `p`. For example, the condition `(c, p) `elem` classTables tm c` denotes properties that are stored in the table for class `c`.

Each recursive object path is structured as (`RecOPath op tar`), where `op` is its prefix (i.e. context) of type `OPATH`, which we recursively transform into a table path equivalent `tp`; and `tar` is its target property. For each given `tar`, table access is determined by checking membership against various domains: a bidirectional association will be accessed by means of an association table. If the target property is sequence-valued (i.e. the case of `SCTarget`), it cannot be accessed for its entirety, but only one of its members through indexing. The function `objToTabExpr` transforms the index expression `oe` that contains paths of type `OPATH` to `te` that contains paths of type `TPATH`. The function `objToTab` transforms an object model `om` to an equivalent table model `tm`.

SQL database statements express paths via (nested) `SELECT` queries. For example, the above TABLE path has its SQL statement counterpart:

```
SELECT (VAR   'reglist')
       (TABLE 'Hotel_registered_Traveller_reglist')
       (VAR   'oid' = (SELECT (VAR   'owner')
                              (TABLE 'Account_owner_Traveller_account')
                              (VAR    oid = VAR this)))
```

where `oid` is the default column (declared as the primary key) for each table that implements an association. We can show [23] by structural induction that the transformation from *BPATH* to *OPATH*, from *OPATH* to *TPATH*, and from *TPATH* to `SELECT` statements are correct.

Expression transformation. We transform both predicates and expressions on TABLE model into SQL expressions:

```
toSqlExpr  :: TABLE_MODEL -> Predicate  -> SQL_EXPR
toSqlExpr' :: TABLE_MODEL -> Expression -> SQL_EXPR
```

Some transformations are direct

```
toSqlExpr tm (And p q) = toSqlExpr tm p 'AND' toSqlExpr tm q
```

whereas others require an equivalent construction:

```
toSqlExpr' tm (Card e) | isPathExpr e = SELECT [COUNT (VAR oid)] (toSqlExpr' tm e) TRUE
```

5 Assignment Transformation

The most important aspect of the model transformation is the handling of attribute assignments and collection updates. There are 36 distinct cases to consider, given the different combinations of attributes and bidirectional (opposite) associations. We present a single, representative case in Table 1, for an association between an optional attribute (multiplicity 0..1) and a sequence-valued attribute (ordered with multiplicity *) .

Bi-Assoc. Decl.	#	GSL Substitution	SQL Queries

seq-*to*-opt
`class A  class B`
`bs: seq(B.ao)  ao: [A.bs]`
	23	$bs := \mathrm{ins}(bs, i, that)$
		$\Vert$
		$that.ao := this$
			`UPDATE` t `SET` $index = index + 1$
			`WHERE` $ao = this$ `AND` $index \geq i$;
			`INSERT INTO` $t\ (bs, ao, index)$ `VALUE` $(that, this, i)$;

Table 1: Assignment Transformation Pattern for *sequence-to-optional* Bi-Association

From left to right, the columns of the table present declarations of properties, numerical identifiers of patterns, their abstract implementation in the substitution program, and their concrete implementation in database queries. The dummy variables *this* and *that* are used to denote instances of, respectively, the current class and the other end of the association.

For each case (for each row of the completed table), we define a transformation function `toSqlProc` that turns a substitution into a list of SQL query statements.

```
toSqlProc tm _ s@(Assign _ _) = transAssign tm s
transAssign :: TABLE_MODEL -> Substitution -> [STATEMENT]
```

The function `toSqlProc` delegates the task of transforming base cases, i.e. assignments, to another auxiliary function `transAssign` that implements the 36 patterns. The recursive cases of `toSqlProc` are straightforward. For example, to implement a guarded substitution, we transform it into an `IfThenElse` pattern that is directly supported in the SQL domain; and to implement iterators (`ALL`, `ANY`), we instantiate a loop pattern, declared with an explicit variant, that is guaranteed to terminate.

6 Correctness Proofs

The correctness of both BOOSTER-to-OBJECT and OBJECT-to-TABLE transformations can be established by constructing a relational model mapping identifiers and paths to references and primitive values, and then showing that the different reference mechanisms identify the same values in each case. To prove the correctness of the TABLE-to-SQL transformation (shown as the vertical, thick arrow in Figure 2 on page 5), we need also to introduce *linking invariants* between model states. We first formalise states and operations for each model domain. In the Z notation, sets and relations may be described using a schema notation, with separate declaration and constraint components and an optional name:

$$
\begin{array}{|l}
\hline
\mathit{name} \\
\mathit{declaration} \\
\hline
\mathit{constraint} \\
\hline
\end{array}
$$

Either component may include schema references, with the special reference Δ denoting two copies of another schema, typically denoting before- and after-versions, the attributes of the latter being decorated with a prime ($'$). The remainder of the mathematical notation is that of standard, typed, set theory.

We map the state OBJECT model into a relational semantics $\mathscr{S}_{obj}$, characterised by:

$$
\begin{array}{|l}
\hline
\mathscr{S}_{obj} \\
\mathit{OBJECT\_MODEL} \\
\mathit{extent} : \mathit{N\_CLASS} \pfun \mathbb{P}\, \mathit{ObjectId} \\
\mathit{value} : \mathit{ObjectId} \pfun \mathit{N\_PROPERTY} \pfun \mathit{Value} \\
\hline
\mathrm{dom}\, \mathit{extent} = \mathrm{dom}\, \mathit{class} \\
\forall c : \mathit{N\_CLASS};\ o : \mathit{ObjectId} \mid \\
\quad c \in \mathrm{dom}\, \mathit{extent} \land o \in \mathit{extent}(c) \bullet \mathrm{dom}\,(\mathit{value}(o)) = \mathrm{dom}\,((\mathit{class}\, c).\mathit{property}) \\
\hline
\end{array}
$$

The inclusion of *OBJECT_MODEL* (whose details are omitted here) enables us to constrain the two mappings according to the type system of the object model in question. *Value* denotes a structured type that encompasses the possibilities of undefined value (for optional properties), primitive value, and set and sequence of values.

The state of a table model will be composed of: 1) the type system of the object model in context; and 2) functions for querying the state of such a context object model. More precisely,

$$
\begin{array}{|l}
\hline
\mathit{TABLE\_MODEL} \\
\mathit{OBJECT\_MODEL} \\
\mathit{nTableModel} : \mathit{N\_MODEL} \\
\mathit{assocTables}, \mathit{setTables} : \mathbb{P}(\mathit{N\_CLASS} \times \mathit{N\_PROPERTY}) \\
\hline
\end{array}
$$

where *assocTables*, *setTables* and *seqTables* are reflective queries: for example, *assocTables* returns the set of attributes/properties (and their context classes) that are stored in the association tables. We formalise the TABLE model state as:

$$\mathscr{S}_{tab}$$
$$\mathscr{S}_{obj}$$
$$TABLE\_MODEL$$

For each instance of $\mathscr{S}_{obj}$, there should be a corresponding configuration of *TABLE_MODEL*. A SQL database corresponds to a set of named tables, each containing a set of column-value mappings:

$$\mathscr{S}_{sql}$$
$$tuples : N\_TABLE \pfun \mathbb{P}\, Tuple$$

$$Tuple$$
$$values : N\_COLUMN \pfun ScalarValue$$

We use *ScalarValue* to denote the collection of basic types: strings, integers, and Booleans. We require mapping functions to retrieve values from TABLE and SQL:

$$\mathscr{M} : \mathscr{S}_{tab} \times (NClass \times NProperty) \pfun \mathbb{P}(Value \times Value)$$

These return reference–value pairs for each kind of property. For example, set-valued properties are returned by

$$\mathscr{M}_{set} == \lambda\, s : \mathscr{S}_{tab};\ p : NClass \times NProperty \bullet \bigcup \left\{ \begin{array}{l} o : ObjectId;\ v : Value;\ vs : \mathbb{P}\, Primitive \mid \\ \quad o \in s.extent\,(fst\, p) \land v = s.value\,(o)\,(snd\, p) \land v = setValue\,(vs) \bullet \\ \quad\quad \{ v' : vs \bullet [\![o]\!]^{SV} \mapsto [\![v']\!]^{SV} \} \end{array} \right\}$$

The set of mappings for a particular table is given by

$$\lambda\, s : \mathscr{S}_{sql};\ n : NTable;\ c_1, c_2 : NColumn \bullet \{\ row : s.tuples\,(n) \bullet row.values\,(c_1) \mapsto row.values\,(c_2)\ \}$$

and the necessary linking invariant is:

TABLE $\leftrightarrow$ SQL
$$\mathscr{S}_{tab}$$
$$\mathscr{S}_{sql}$$
$$C$$

where C comprises six conjuncts, one for each possible unordered combination of association end multiplicities.

Each operation is implemented as an atomic transaction. $\mathscr{R}_{obj}$ represents the formal context, with the effect upon the state being described as a binary relation ($\leftrightarrow$).

$$\mathscr{R}_{obj}$$
$$input : \mathbb{P}\, N\_VARIABLE$$
$$output : \mathbb{P}\, N\_VARIABLE$$
$$effect : (\mathscr{S}_{obj} \times IO_{obj}) \leftrightarrow (\mathscr{S}_{obj} \times IO_{obj})$$
$$effect \in \mathscr{S}_{obj} \times (input \to Value) \leftrightarrow \mathscr{S}_{obj} \times (output \to Value)$$

Each element of $IO_{obj} == N\_VARIABLE \pfun Value$ represents a collection of inputs and outputs.

Using $\mathscr{S}_{obj}$ and $\mathscr{R}_{obj}$, we may write $System_{obj}$ to denote the set of possible object system configurations, each characterised through its current state (of type $\mathscr{S}_{obj}$) and its set of indexed operations (of type $\mathscr{R}_{obj}$). More precisely,

$$
\begin{array}{|l}
\hline
System_{obj} \\
state : \mathscr{S}_{obj} \\
relation : N\_CLASS \pfun N\_OPERATION \pfun \mathscr{R}_{obj} \\
\hline
\end{array}
$$

We will describe the effect of a primitive assignment (:=), and use this as the basis for a recursive definition of effect, based on the grammar of the GSL notation. If *AssignInput* is the schema $[path? : PATH;\ e? : Expression]$, then we may define

$$
\begin{array}{|l}
\hline
AssignEffect \\
s, s' : \mathscr{S}_{obj} \\
AssignInput \\
\hline
s.nObjModel = s'.nObjModel \\
s.sets = s'.sets \\
s.classes = s'.classes \\
s.extent = s'.extent \\
\textbf{let}\ p == target\,[\![path?]\!]\ ;\ o == context\,[\![path?]\!] \bullet s'.value = s.value \oplus \{o \mapsto \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad s.value\,(o) \oplus \{p \mapsto eval\,(e?)\}\} \\
\hline
\end{array}
$$

The input *path*? can be either `OPATH` and `TPATH`: for the former, the other input expression *e*? involves paths, if any, of type `OPATH`; for the latter, it is `TPATH`. The (**let** *es* • *p*) expression, where *es* consists of a list of expression-to-variable bindings, denotes a predicate *p* on the variables of *es*.

We start by relating domains of the OBJECT model and TABLE model, where assignment paths are specified in, respectively, `OPATH` and `TPATH` (Fig 3). In the OBJECT model domain, an assignment is parameterised by a path of type `BPATH` and an expression that consists of paths, if any, of the consistent type. We formalise each OBJECT model assignment under the formal context of $\mathscr{R}_{obj}$, by defining its *effect* mapping though the constraint of *AssignEffect* and by requiring that the sets of external inputs and outputs are empty.

$$
\begin{array}{|l}
\hline
Assign_{obj} \\
\mathscr{R}_{obj} \\
op? : OPATH \\
oe? : Expression \\
\hline
\forall s, s' : \mathscr{S}_{obj};\ AssignInput \mid path? = OPath\,(op?) \land e? = oe? \bullet AssignEffect \Leftrightarrow (s, \{\}) \mapsto (s', \{\}) \in effect \\
\hline
\end{array}
$$

The characterisation $Assign_{tab}$ of an assignment in the TABLE model domain is similar to that of $Assign_{obj}$, except that the target is now of type `TPATH`, and the source is now of type *Expression*. We may then map our extended GSL substitution into a relation:

$$[\![\_]\!]_{obj} : Substitution \to ((\mathscr{S}_{obj} \times IO_{obj}) \leftrightarrow (\mathscr{S}_{obj} \times IO_{obj}))$$

of the same type as the *effect* component of $\mathscr{R}_{obj}$. Given a TABLE path *tp*? and an expression *te*?, we represent the assignment substitution $tp? := te?$ by the effect relation of $Assign_{tab}$ that exists uniquely with respect to *tp*? and *te*?. More precisely,

$$[\![tp? := te?]\!]_{obj} = (\mu Assign_{tab}).effect$$

where $\mu Assign_{tab}$ denotes the unique instance of $\mathscr{S}_{tab}$ such that the constraint as specified in $Assign_{tab}$ holds, and *.effect* retrieves its relational effect on the model state. The definition of $Assign_{tab}$ is very similar to that of $Assign_{obj}$, except that the input path is constrained as $path? = TPath(\ldots)$.

We interpret a guarded substitution $g \longrightarrow S$ as a relation that has the same effect as $[\![S]\!]_{obj}$ within the domain of satisfying states of guard g (denoted as $[\![g]\!]^{states}_{obj}$); otherwise, it just behaves like *skip* as it will be blocked and cannot achieve anything. More precisely, we have:

$$[\![g \longrightarrow S]\!]_{obj} = \mathrm{id}\,(\mathscr{S}_{obj} \times IO_{obj}) \oplus ([\![g]\!]^{states}_{obj} \lhd [\![S]\!]_{obj})$$

Similar rules may be defined for other combinators.

Each transaction is composed of SQL queries, and similar to $\mathscr{R}_{obj}$, we collect and produce, respectively, its list of inputs and outputs upon its initiation and completion. We use $\mathscr{R}_{sql}$ to denote such formal context, under which the transformational effect on the state of database is defined accordingly as a function, reflecting the fact that the database implementation is deterministic in its effect.

$\mathscr{R}_{sql}$

$$input, output : \mathbb{P}\, N\_VARIABLE$$
$$effect : (\mathscr{S}_{sql} \times IO_{sql}) \to (\mathscr{S}_{sql} \times IO_{sql})$$

$$this \in input$$

The mechanism of referencing the current object (via *this*) is simulated through providing by default the value of *this* for each generated stored procedure or function. We model inputs and outputs in the same way as we do for IO_{obj}, except that the range of values is now of type *ScalarValue*.

For each SQL statement, we assign to it a relational semantics by mapping it to a relation on states (of type $\mathscr{S}_{sql}$). This is a similar process to that for $[\![\_]\!]_{obj}$. More precisely, we define:

$$[\![\_]\!]_{sql} : Statement \to ((\mathscr{S}_{sql} \times IO_{sql}) \leftrightarrow (\mathscr{S}_{sql} \times IO_{sql}))$$

And since a SQL stored procedure is defined as a sequential composition, we also define

$$[\![\_]\!]_{\mathrm{seq}\,sql} : \mathrm{seq}\, Statement \to ((\mathscr{S}_{sql} \times IO_{sql}) \leftrightarrow (\mathscr{S}_{sql} \times IO_{sql}))$$

to derive its effect through combining those of its component statements via relational composition. For primitive query statements, we refer to their schema definitions. For example, we have:

$$[\![\texttt{UPDATE}\ t\ \texttt{SET}\ sets\ \texttt{WHERE}\ cond]\!]_{sql} = (\mu\,\texttt{UPDATE}).effect$$

where the state effect of query (`UPDATE` *table*? `SET` *sets*? `WHERE` *cond*?) is formally specified in a schema named `UPDATE`. The `UPDATE` query modifies in a table those tuples that satisfy a condition and takes as inputs *table*? a table name, *sets*? a mapping that specifies how relevant columns should be modified, and *cond*? a Boolean condition that chooses the range of tuples to be modified. The schema `UPDATE` is defined similarly as is $Assign_{tab}$, except that it imposes constraints on the model state $\mathscr{S}_{sql}$. We formalise an `IF`...`THEN`...`ELSE`... statement as the union of the semantic interpretations of the two sequences of statements in its body, each suitably restricted on its domain.

$$[\![\texttt{IF}\ b\ \texttt{THEN}\ stmts_1\ \texttt{ELSE}\ stmts_2]\!]_{sql} = ([\![b]\!]^{states}_{sql} \lhd [\![stmts_1]\!]_{\mathrm{seq}\,sql}) \cup ([\![\texttt{NOT}\,b]\!]^{states}_{sql} \lhd [\![stmts_2]\!]_{\mathrm{seq}\,sql})$$

where $[\![b]\!]^{states}_{sql}$ denotes the set of satisfying state of a SQL expression b.

To define the semantics of a `WHILE` loop, we intend for the following equation to hold

$$[\![\texttt{WHILE}\ b\ \texttt{DO}\ stmts\ \texttt{END WHILE}]\!]_{sql} = [\![\texttt{IF}\ b\ \texttt{THEN}\ stmts \frown \langle \texttt{WHILE}\ b\ \texttt{DO}\ stmts\ \texttt{END WHILE}\rangle\ \texttt{ELSE}\ \langle\rangle]\!]_{sql}$$

where $\frown$ is the operator for sequence concatenation. By applying the definition of $[\![\_]\!]_{sql}$ on `IF`...`THEN`...`ELSE`... and $[\![\_]\!]_{\mathrm{seq}\,sql}$ on $\langle\rangle$, we have

$$\boxed{[\![\texttt{WHILE}\ b\ \texttt{DO}\ stmts\ \texttt{END WHILE}]\!]_{sql}} = [\![b]\!]^{states}_{sql} \lhd ([\![stmts]\!]_{\mathrm{seq}\,sql} \fatsemi \boxed{[\![\texttt{WHILE}\ b\ \texttt{DO}\ stmts\ \texttt{END WHILE}]\!]_{sql}}) \cup [\![\texttt{NOT}\,b]\!]^{states}_{sql} \lhd \mathrm{id}\,(\mathscr{S}_{sql} \times IO_{sql})$$

Let us define a function

$$F(X) = (\,[\![b]\!]^{states}_{sql} \lhd ([\![stmts]\!]_{\mathrm{seq}\,sql} \fatsemi X)\,) \cup (\,[\![\texttt{NOT}\ b]\!]^{states}_{sql} \lhd \mathrm{id}\,(\mathscr{S}_{sql} \times IO_{sql})\,)$$

When $X = [\![\texttt{WHILE}\ b\ \texttt{DO}\ stmts\ \texttt{END WHILE}]\!]_{sql}$, we obtain

$$[\![\texttt{WHILE}\ b\ \texttt{DO}\ stmts\ \texttt{END WHILE}]\!]_{sql} = F(\,[\![\texttt{WHILE}\ b\ \texttt{DO}\ stmts\ \texttt{END WHILE}]\!]_{sql}\,)$$

which means that $[\![\texttt{WHILE}\ b\ \texttt{DO}\ stmts\ \texttt{END WHILE}]\!]_{sql}$ should be a fixed-point of function F. The least fixed-point (LFP) of function F—i.e. $\bigcup_{n\in\mathbb{N}} F^n(\emptyset)$—exists by Kleene's fixed-point theorem, since F is easily provable to be continuous. We choose this LFP of F for the value of $[\![\texttt{WHILE}\ b\ \texttt{DO}\ stmts\ \texttt{END WHILE}]\!]_{sql}$.

We are now able to establish the correctness of the transformation with respect to the linking invariant. The commuting diagram of Figure 4 shows how a substitution program *prog* and its context TABLE model (i.e. $\theta TableModel$), are mapped by the transformation $toSqlProc\,(\theta\,TableModel)\,(prog)$ to produce an SQL implementation. The linking invariant holds for the before states TABLE $\leftrightarrow$ SQL and for the after states TABLE $\leftrightarrow$ SQL$'$. We then establish that for each state transformation, characterised by the relational effect of the generated SQL code from *prog*, there is at least a corresponding state transformation, characterised by the relational effect of the TABLE program, $[\![prog]\!]_{obj}$. This is an example of simulation between abstract data types [8].

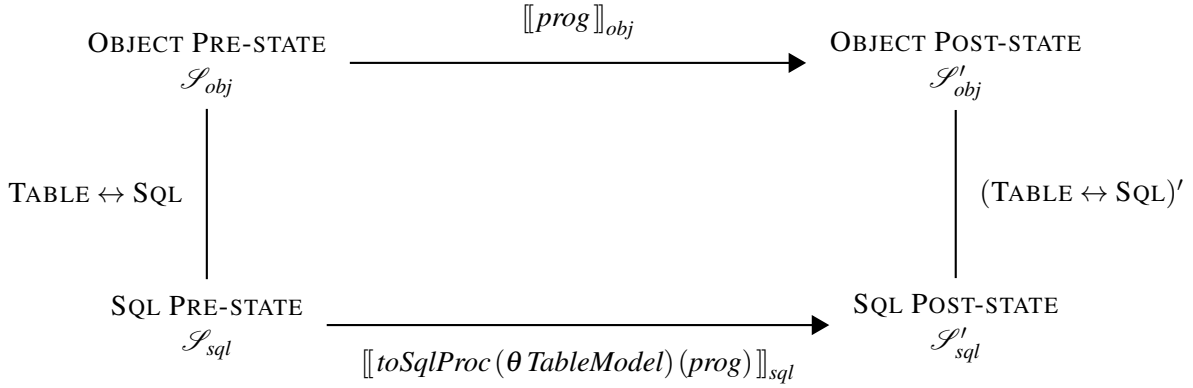


Figure 4: Correctness of Model Transformation

We use a universal quantification $(\forall x \mid R(x) \bullet P(x))$ to state our correctness criterion: the x part declares variables, the $R(x)$ part constrains the range of state values, and the $P(x)$ part states our concern. Schemas defined above (i.e. $\mathscr{S}_{obj}$, $\mathscr{S}_{sql}$, and TABLE $\leftrightarrow$ SQL) are used as both declarations and predicates. If we declare

TransInput
TableModel
prog : *Substitution*

to represent the inputs to the transformation, then

$$\forall\, TransInput;\ \Delta\mathscr{S}_{obj};\ \Delta\mathscr{S}_{sql} \mid$$
$$\text{TABLE} \leftrightarrow \text{SQL} \wedge [\![toSqlProc\,(\theta\,TableModel)\,(prog)]\!]_{\mathrm{seq}\,sql} \bullet (\ \exists \mathscr{S}'_{obj} \bullet (\text{TABLE} \leftrightarrow \text{SQL})' \wedge [\![prog]\!]_{obj}\ )$$

With the relational semantics outlined above, we may establish this result through a combination of case analysis and structural induction.

7 Example Implementation

Consider the implementation, on a relational database platform, of the operation `reserve` introduced in Section 2. Having translated the object model into a collection of database tables, the generation process will produce a

stored procedure for each operation. The guard for `reserve` requires that the current number of allocations—characterised through the cardinality of the set-valued attribute `allocations`—is below a specific bound. We might include such a condition, for example, to ensure that the memory or storage requirements of the system remain within particular bounds; this is may not be an issue for a hotel reservation system, but is a realistic concern in critical systems development. In the implementation, a stored function is generated that will establish whether or not the guard constraint holds for the current state, together with any input values. The remainder of the generated code will achieve the effect specified in the original operation constraint, translated into the representation, or orientation, of the database platform.

Class `Reservation` has `status` as an attribute, and this is stored in the corresponding class table. In the function, `AUTO_INCREMENT` allows the target SQL platform to generate a unique identifier for each inserted row. Set-valued properties, like attribute `dates` in class `Reservation` are stored in separate tables, with an `oid` column to identify the current object in a given method call. Associations such as `host` and `reservations` are stored in separate tables, with an `oid` column to identify the exact association instance. Since attribute `reservations` are also sequence-valued, an `index` column is required.

Schema of Tables Updated by 'reserve'

```
1 CREATE TABLE ‘Reservation‘(‘oid‘ INTEGER AUTO_INCREMENT, PRIMARY KEY (‘oid‘), ‘status‘ CHAR(30));
2 CREATE TABLE ‘Room_reservations_Reservation_room‘(‘oid‘ INTEGER AUTO_INCREMENT,
3         PRIMARY KEY (‘oid‘), ‘reservations‘ INTEGER, ‘room‘ INTEGER, ‘index‘ INTEGER);
```

We generate also integrity constraints for association tables: although the generated procedures are guaranteed to preserve semantic integrity, this affords some protection against errors in the design of additional, manually-written procedures.

The value of the model-driven approach should be apparent following a comparison of the original specification for `reserve` with the fragments of the following SQL implementation. Manual production of queries that need to be take account of a large number of complex model constraints—as well as, for examples, constraints arising from data caching strategies—is time-consuming and error-prone. Furthermore, we may expect the design of a system to evolve during use: the challenge of maintaining correctness in the face of changing specifications (and platforms) adds another dimension of complexity to systems development; some degree of automation, in production and in proof, is essential.

In the following, variable names have been preserved from the BOOSTER domain, e.g. the input and output parameters `dates?` and `r!` at line 2, as well as caching variables ‘`r!.status`‘, ‘`r!.host`‘, and ‘`r!.room`‘ at line 4. Meta-variables are used to implement the `ALL` iterator in method `reserve`: Line 5 declares, respectively, the bound variable ‘`x`‘ and ‘`x_variant`‘ the variant of the loop, and Line 6 declares a cursor over the set-valued input `dates?`.

Queries Implementing 'reserve': Declarations

```
1 CREATE PROCEDURE ‘Hotel_reserve‘  (IN ‘this?‘ INTEGER,
2     IN ‘dates?‘ CHAR(30), IN ‘m?‘ INTEGER, OUT ‘r!‘ INTEGER)
3 BEGIN
4   DECLARE ‘r!.status‘ CHAR(30); DECLARE ‘r!.host‘ INTEGER; DECLARE ‘r!.room‘ INTEGER;
5   DECLARE ‘x‘ Date; DECLARE ‘x_variant‘ INTEGER;
6   DECLARE ‘x_cursor‘ CURSOR FOR (SELECT * FROM ‘dates?‘ WHERE TRUE);
```

Line 7 first creates a new instance of `Reservation` by inserting, for output `r!`, a row formatted as $\langle oid, \ldots \rangle$ into the appropriate class table, where *oid* is a unique value generated by the built-in function `last_insert_id()`, with the guarantee that each subsequent call to this functions returns a new value. It then assigns this unique identifier to `r!` for queries in later fragments to refer to.

Queries Implementing 'reserve': Creating an Empty Output

```
7     INSERT INTO ‘Reservation‘ () VALUE (); SET ‘r!‘ = last_insert_id ();
```

In Lines 8 to 10 the pair of DROP TEMPORARY TABLE and CREATE TEMPORARY TABLE queries update the value of a cache variable ‘m?.reservations‘ that denotes a multi-valued property: this kind of caching is useful in large database implementations. In Line 11 we update the caching variable ‘r!.host‘ of single-valued types of properties through a SELECT INTO query. We cache the value of attribute host possessed by the reservation r!. Any later paths with ‘r!.host‘ or ‘m?.reservations‘ as its prefix will be able to use its value directly without re-evaluation.

Queries Implementing 'reserve': Updating Caching Vars

```
8   DROP TEMPORARY TABLE IF EXISTS ‘m?.reservations‘;
9   CREATE TEMPORARY TABLE ‘m?.reservations‘ AS
10    SELECT ‘reservations‘ FROM ‘Room_reservations_Reservation_room‘ WHERE ‘room‘ = ‘m?‘;
11  SELECT ‘status‘ INTO ‘r!.status‘ FROM ‘Reservation‘ WHERE ‘oid‘ = ‘r!‘;
```

Lines 12 to 20 instantiate a finite loop pattern. In Line 12 we activate the declared cursor and and fetch its first available value. In Line 13 we also calculate the size of the data set that the cursor will iterate over and use it as the variant of the loop defined in Lines 14 to 20. The exit condition (Line 14) is characterised through decreasing—via the 2nd statement in Line 19—the value of x_cursor; the bound variable x is updated to the next data item at the end of each iteration (via the 1st statement in Line 19). In each iteration of the loop, from Lines 15 to 17 we re-cache the value of the set-valued path r!.dates, in case there are other paths which contain it as a prefix and are used later in the loop. In Line 18 we perform the first substitution in the specification of method reserve: we implement the substitution r!.dates := r!.dates \/ dates? via iterating through the input dates? with a bound variable ‘x‘.

Queries Implementing 'reserve': Terminating Loop

```
12  OPEN ‘x_cursor‘; FETCH ‘x_cursor‘ INTO ‘x‘;
13  SELECT COUNT(*) INTO ‘x_variant‘ FROM ‘dates?‘ WHERE TRUE;
14  WHILE (‘x_variant‘) > (0) DO
15    DROP TEMPORARY TABLE IF EXISTS ‘r!.dates‘;
16    CREATE TEMPORARY TABLE ‘r!.dates‘ AS
17      SELECT ‘dates‘ FROM ‘Reservation_dates‘ WHERE ‘oid‘ = ‘r!‘;
18    INSERT INTO ‘Reservation_dates‘ (‘oid‘, ‘dates‘) VALUE (‘r!‘, ‘x‘);
19    FETCH ‘x_cursor‘ INTO ‘x‘; SET ‘x_variant‘ = ‘x_variant‘ - 1;
20  END WHILE; CLOSE ‘x_cursor‘;
```

Line 21 implements the update r!.status := unconfirmed. The two generated query statements—that are located in Lines 22 to 27 and Lines 28 to 31—implement the last two parallel assignments in reserve that update the *optional-to-sequence* association. They correspond exactly to the rules specified for pattern 23 in Section 5. The queries for the middle two parallel assignments in reserve, updating the *one-to-sequence* association, are entirely similar.

Queries Implementing 'reserve': Performing Updates

```
21  UPDATE ‘Reservation‘ SET ‘status‘ = ’unconfirmed’ WHERE (‘oid‘) = (‘r!‘);
22  UPDATE ‘Room_reservations_Reservation_room‘
23  SET ‘index‘ = (‘index‘) + (1)
24  WHERE ‘room‘ = ‘m?‘ AND
25        ‘index‘>=(SELECT COUNT(‘oid‘)
26                  FROM (SELECT ‘reservations‘ FROM ‘m?.reservations‘ WHERE TRUE) AS reservations
27                  WHERE TRUE) + 1;
28  INSERT INTO ‘Room_reservations_Reservation_room‘ (‘reservations‘, ‘room‘, ‘index‘) VALUE
29    (‘r!‘,‘m?, (SELECT COUNT(‘oid‘)
30                FROM (SELECT ‘reservations‘ FROM ‘m?.reservations‘ WHERE TRUE) AS reservations
31                WHERE TRUE) + 1);
```

8 Discussion

The principal contribution of this paper is the presentation of a practical, formal, model-driven approach to the development of critical systems. Both the modelling notation and the target programming language are given a formal, relational semantics: the latter only for a specific subset of the language, sufficient for the patterns of implementation produced by the code generation process. The generation process is formalised as a functional program, easily related to the corresponding transformation on the relational semantics. It is perfectly possible to prove the generator correct; indeed, a degree of automatic proof could be applied here. The task of system verification is then reduced to the strictly simpler task of model verification or validation.

The implementation platform chosen to demonstrate the approach is a standard means of storing data, whether that data was originally described in a hierarchical, a relational, or an object-oriented schema. In particular, there are many products that offer a means of mapping [20] from object models (as used here) to a relational database implementation: Hibernate [5] is perhaps the best-known example. However, translating the data model to a data schema is relatively straightforward; the focus here is the generation of correct implementations for operations.

At the same time, much of the work on program transformation is focussed, unsurprisingly, upon code rewriting rather than the generation of complete software components with persistent data. The work on Vu-X [17], where modifications to a web interface are reflected back into the data model is an interesting exception, but has yet to be extended to a formal treatment of data integrity. The work on UnQL [13] supports the systematic development of model transformation through the composition of graph-based transformations: this is a powerful approach, but again no similar framework has been proposed.

Some work has been done in precise data modelling in UML, for example [7], but no formal account has been given for the proposed translation of operations. The Query/View/Transformation approach [18] focuses on design models, but the transformations [14] are described in an imperative, stateful, style, making proofs of correctness rather more difficult. Recent work on generating provably correct code, for example [22], is restricted to producing primitive getter and setter methods, as opposed to complex procedures. Mammar [15] adopts a formal approach to generating relational databases from UML models. However, this requires the manual definition of appropriate guards for predefined update methods: the automatic derivation of guards, and the automatic generation of methods from arbitrary constraint specifications, as demonstrated here, is not supported.

The unified implementation and semantic framework for transformation (Figure 2) presented here can be applied to any modelling and programming notation that admits such a relational semantics for the behaviour of components. It is important to note that the style of this semantics effectively limits the approach to the development of sequential data components: that is, components in which interactions with important data are managed as exclusive transactions; our semantic treatment does not allow us to consider the effects of two or more complex update operations executing concurrently.

In practice, this is not a significant limitation. Where data is encapsulated within a component, and is subject to complex business rules and integrity constraints, we may expect to find locking or caching protocols to enforce data consistency in the face of concurrent requests, by means of an appropriate sequentialisation. Where concurrency properties are important, they can be addressed using process semantics and model-checking techniques; a degree of automatic generation may even be possible, although this is likely to be at the level of workflows, rather than data-intensive programs.

Work is continuing on the development of the transformation and generation tools discussed here, with a particular emphasis upon the incremental development of operation specifications and models. It is most often the case that a precise model will prove too restrictive: when a property is written linking two or more attributes, it constrains their interpretation; if one of these attributes is used also elsewhere in the model, or within an operation, then that usage may not always be consistent with the now formalised interpretation. In our approach, such a problem manifests itself in the unavailability of one or more operations, in particularly circumstances.

As a guard is generated for each operation, sufficient to protect any data already acquired, each incremental version of the system can be deployed without risk of data loss. It can then be used in practice and in earnest, allowing users to determine whether or not the availability—or the overall design—of each operation and data view matches their requirements and expectations. Where an operation has a non-trivial guard, additional analysis may be required to demonstrate that the resulting availability matches requirements: in many cases, the necessary check or test can be automated. The work described here provides a sound foundation for this development process.

References

[1] J.-R. Abrial (1996): *The B-book: assigning programs to meanings*. Cambridge University Press.

[2] R. Bird (1998): *Introduction to Functional Programming using Haskell*. Prentice Hall.

[3] Australian Transport Safety Bureau (2011): *In-flight upset 154km West of Learmouth, WA, VH-QPA, Airbus A330-303* AO-2008-070.

[4] R. N. Charette (2009): *This car runs on code*. In: *IEEE Spectrum*.

[5] JBoss Community: *Hibernate — Relational Persistence for Java and .NET*. `http://www.hibernate.org/`.

[6] J. Davies, C. Crichton, E. Crichton, D. Neilson & I. H. Sørensen (2005): *Formality, evolution, and model-driven software engineering*. *Electronic Notes in Theoretical Computer Science (ENTCS)* 130, pp. 39–55.

[7] B. Demuth & H. Hussmann (1999): *Using UML/OCL Constraints for Relational Database Design*. In: *UML*, *LNCS* 1723, pp. 598–613.

[8] W. DeRoever & K. Engelhardt (1999): *Data Refinement: Model-Oriented Proof Methods and Their Comparison*. Cambridge University Press.

[9] A. van Deursen, P. Klint & J. Visser (2000): *Domain-Specific Languages: An Annotated Bibliography*. *SIGPLAN Notices* 35(6), pp. 26–36.

[10] H.-E. Eriksson, M. Penker & D. Fado (2003): *UML 2 Toolkit*. Wiley.

[11] P.H. Feiler (2010): *Model-based validation of safety-critical embedded systems*. In: *Aerospace Conference*, IEEE, pp. 1 – 10.

[12] D. S. Frankel (2003): *Model Driven Architecture: Applying MDA to Enterprise Computing*. Wiley.

[13] S. Hidaka, Z. Hu, H. Kato & K. Nakano (2009): *Towards a compositional approach to model transformation for software development*. In: *SAC*, ACM, pp. 468–475, doi:10.1145/1529282.1529383.

[14] F. Jouault, F. Allilaire, J. Bézivin & I. Kurtev (2008): *ATL: A model transformation tool*. *Science of Computer Programming* 72(1–2), pp. 31–39. Available at `http://dx.doi.org/10.1016/j.scico.2007.08.002`.

[15] A. Mammar (2009): *A systematic approach to generate B preconditions: application to the database domain*. *Software Systems Modeling* 8(3), pp. 385–401.

[16] A. Massoudi (2012): *Knight Capital glitch loss hits $461m*. *Financial Times*.

[17] K. Nakano, Z. Hu & M. Takeichi (2009): *Consistent Web site updating based on bidirectional transformation*. *Int. J. Softw. Tools Technol. Transf.* 11(6), pp. 453–468, doi:10.1007/s10009-009-0124-3.

[18] OMG (2009): *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification*. OMG Document ptc/09-12-05, Object Management Group. `http://www.omg.org/spec/QVT/1.1/Beta2/PDF/`.

[19] K. Pohl, G. Böckle & F. J. van der Linden (2005): *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer.

[20] C. Russell (2008): *Bridging the Object-Relational Divide*. *ACM Queue* 6(3), pp. 18–28, doi:10.1145/1394127.1394139.

[21] RTCA SC-205 (2011): *DO-178C, Software Considerations in Airborne Systems and Equipment Certification*. Approved by Special Committee 205 of Radio Technical Commission for Aeronautic.

[22] K. Stenzel, N. Moebius & W. Reif (2011): *Formal Verification of QVT Transformations for Code Generation*. In: *MoDELS*, pp. 533–547. Available at `http://dx.doi.org/10.1007/978-3-642-24485-8_39`.

[23] C.-W. Wang (2012): *Model-Driven Development of Information Systems*. Ph.D. thesis, University of Oxford, Oxford University Research Archive. Available at `http://ora.ox.ac.uk/objects/uuid:9d70647c-e1b6-4cbb-b88c-707f09431db6`.

[24] J. Welch, D. Faitelson & J. Davies (2008): *Automatic Maintenance of Association Invariants*. *Software and Systems Modeling* 7(3), pp. 287–301, doi:10.1007/s10270-008-0085-0.

[25] M. Williams (2010): *Toyota to recall Prius hybrids over ABS software*. *Computerworld*.

[26] J. Woodcock & J. Davies (1996): *Using Z*. Prentice Hall.

MDM: A Mode Diagram Modeling Framework for Periodic Control Systems

Zheng Wang, Geguang Pu, Jianwen Li, Jifeng He
Shanghai Key Laboratory of Trustworthy Computing
East China Normal University
{wangzheng,ggpu,jifeng}@sei.ecnu.edu.cn

Shenchao Qin
University of Teesside
s.qin@tees.ac.uk

Kim G. Larsen
Aalborg University of Denmark
kgl@cs.aau.dk

Jan Madsen
Technical University of Denmark
jan@imm.dtu.dk

Bin Gu
Beijing Institute of Control Engineering
gubin88@yahoo.com.cn

Periodic control systems used in spacecrafts and automotives are usually period-driven and can be decomposed into different modes with each mode representing a system state observed from outside. Such systems may also involve intensive computing in their modes. Despite the fact that such control systems are widely used in the above-mentioned safety-critical embedded domains, there is lack of domain-specific formal modelling languages for such systems in the relevant industry. To address this problem, we propose a formal visual modeling framework called MDM as a concise and precise way to specify and analyze such systems. To capture the temporal properties of periodic control systems, we provide, along with MDM, a property specification language based on interval logic for the description of concrete temporal requirements the engineers are concerned with. The statistical model checking technique can then be used to verify the MDM models against desired properties. To demonstrate the viability of our approach, we have applied our modelling framework to some real life case studies from industry and helped detect two design defects for some spacecraft control system.

1 Introduction

Control systems that are widely used in safety-critical embedded domains, such as spacecraft control and automotive control, usually reveal periodic behaviours. Such *periodic* control systems share some interesting features and characteristics:

- They are *mode-based*. A periodic control system is usually composed of a set of modes, with each mode representing an important state of the system. Each mode either contains a set of sub-modes or performs controlled computation periodically.
- They are *computation-oriented*. In each mode, a periodic control system may perform control algorithms involving complex computations. For instance, in certain mode, a spacecraft control system may need to process intensive data in order to determine its space location.
- They behave *periodically*. A periodic control system is reactive and may run for a long time. The behaviour of each mode is regulated by its own period. That is, most computations are performed within a period and may be repeated in the next period if mode switch does not take place. Mode switch may only take place at the end of a period under certain conditions.

Despite the fact that periodic control systems have been widely used in areas such as spacecraft control, there is lack of a concise and precise domain specific formal modelling language for such systems.

Submitted to:
FTSCS 2012

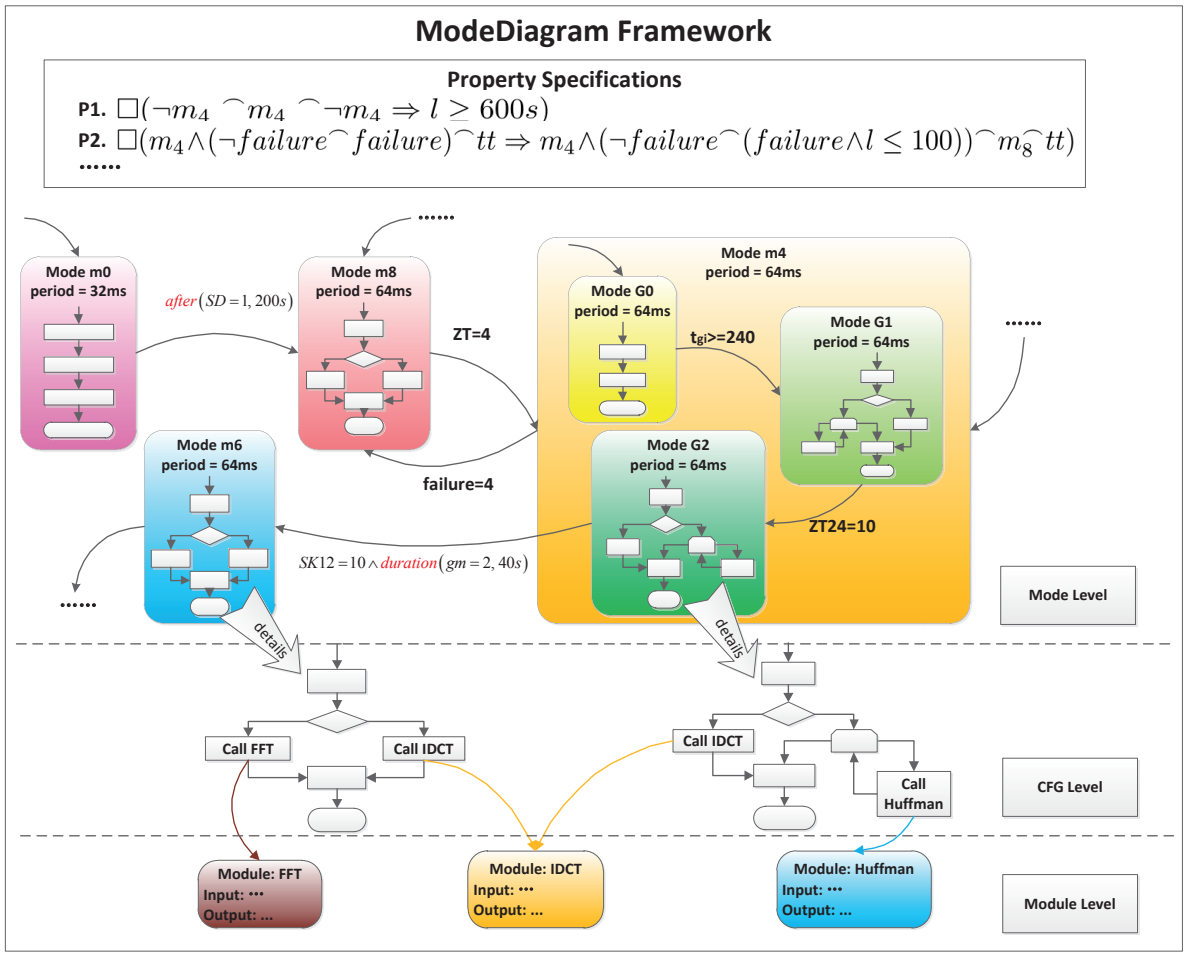


Figure 1: MDM: An (Incomplete) Example

In our joint project with China Academy of Space Technology (CAST), we have started with several existing modelling languages but they are either too complicated therefore require too big a learning curve for domain engineers, or are too specific/general, therefore require non-trivial restrictions or extensions. This motivates us to propose a new formal but lightweight modelling language that matches exactly the need of the domain engineers, the so called Mode Diagram Modelling framework (MDM).

Although the proposed modelling notation MDM can be regarded as a variant of Statecharts [11], it has been specifically designed to cater for the domain-specific need in modelling periodic control systems. We shall now use an example to illustrate informally the MDM framework, and leave the formal syntax and semantics to the next section. As shown in Fig 1, the key part of a MDM model is the collection of modes given in the mode level. Each mode has a period, and the periods for different modes can be different. A mode can be nested and the transitions between modes or sub-modes may take place. A transition is enabled if the associated guard is satisfied. In MDM, the transition guards may involve complex temporal expressions. For example, in the transition from mode G2 to mode m6, in addition to the condition SK12=10, it also requires that the condition gm=2 has held for 40s, as captured by the duration predicate.

An MDM model is presented hierarchically. A mode that does not contain any sub-modes (termed a *leaf* mode) contains a control flow graph (CFG) encapsulating specific control algorithms or computation

tasks. The details of CFGs are given in the CFG level. The CFGs may refer to modules (similar to procedures in conventional languages) details of which are given in the module level.

To support formal reasoning about MDM models, we also provide a property specification language inspired by an interval-like calculus [7], which facilitates the capture of temporal properties system engineers may be interested in. Two example properties are listed in Fig 1. The property P1 says that "whenever the system enters the m4 mode, it should stay there for at least 600s". The formal details of the specification language is left to a later section.

To reason about whether an MDM model satisfies desired properties specified by system engineers using the property specification language, we employ statistical model checking techniques [23, 24]. Since MDM may involve complex non-linear computation in its control flow graph, complete verification is undecidable. Apart from incompleteness, statistical model checking can verify hybrid systems efficiently [4]. Our experimental results on real life cases have demonstrated that the statistical model checking can help uncover potential defects of MDM models.

In summary, we have made the following contributions in this paper:

- We propose a novel visual formal modelling notation MDM as a concise yet precise modelling language for periodic control systems. Such a notation is inspired from the industrial experiments of software engineers.
- We present a formal semantics for MDM and a property specification language to facilitate the verification process.
- We develop a new statistical model checking algorithm to verify MDM models against various temporal properties. Some real life case studies have been carried out to demonstrate the effectiveness of the proposed framework. Furthermore, some design defects of a real spacecraft control system are discovered.

The rest of this paper is organized as follows. Section 2 presents the formal syntax and semantics of MDM. Section 3 introduces our interval-based property specification language and its semantics. The statistical model checking algorithm for MDM is developed in Section 4, followed by related work and concluding remarks.

2 The MDM Notations

Before developing the formal model of MDM, we will begin by giving its informal description. An MDM model is composed by several modes, variables used in the mode, and mode transitions specifying the mode switch relations. A mode essentially refers to the state of the system which can be observed from outside. The mode body can be either a Control Flow Graph (CFG), which prescribes the computational tasks the system can perform in every period, or several other modes as sub-modes. If the mode has sub-modes, when the system is in this mode, it should be in one of its sub-modes. We say that the mode is a leaf mode if its mode body is a control flow graph. A leaf mode usually encapsulates the control algorithms involving complicated computation. The CFG in a leaf mode follows the standard notation, which contains assignment, conditional and loop. It also supports function unit similarly to programming languages.

2.1 The Syntax of MDM

We briefly list its syntactical elements in Fig. 2(a). An MDM is composed of a list of modes ($Mode^+$) and modules ($Module^+$), as well as a list of variables (Var^+) used in those modes and modules.

$$
\begin{array}{rcl}
md & ::= & (Var^+, Mode^+, Module^+) \\
Mode & ::= & (name, period, initial, \\
 & & Body, Transition^+) \\
Body & ::= & Mode^+ \mid CFG \\
Transition & ::= & (source, guard, priority, target) \\
Module & ::= & (name, V_I, V_O, CFG)
\end{array}
$$

(a)MDM

$$
\begin{array}{rcl}
SExpr & ::= & Const \mid Var \mid f^{(n)}(SExpr\ldots) \\
BTerm & ::= & \mathsf{true} \mid \mathsf{false} \mid p^{(n)}(SExpr\ldots) \\
IExpr & ::= & (\mathsf{after} \mid \mathsf{duration})(BTerm, SExpr) \\
GTerm & ::= & IExpr \mid BTerm \\
BExpr & ::= & BTerm \mid \neg BExpr \\
 & & \mid BExpr \vee BExpr \mid BExpr \wedge BExpr \\
guard & ::= & GTerm \mid \neg guard \\
 & & \mid guard \vee guard \mid guard \wedge guard
\end{array}
$$

(b) Expressions and Guards

$$
\begin{array}{rcl}
CFG & ::= & stmts \\
stmts & ::= & pStmt \mid cStmt \\
pStmt & ::= & aStmt \mid \mathsf{call}\ name \mid \mathsf{skip} \mid \\
aStmt & ::= & x := SExpr \\
cStmt & ::= & stmts;\ stmts \mid \mathsf{while}\ BExpr\ \mathsf{do}\ stmts \mid \\
 & & \mathsf{if}\ BExpr\ \mathsf{then}\ stmts\ \mathsf{else}\ stmts
\end{array}
$$

(c) CFG

Figure 2: The Syntax of MDM

Intuitively, a mode refers to a certain state of the system which can be observed from outside. A mode has a name, a period, a body and a list of transitions. For simplicity, we assume all mode names are distinct in an MDM model. The mode period (an integer number) is used to trigger the periodic behavior of the mode. The *initial* denotes a mode is an initial mode or not. The mode body can be composed of either a control flow graph (CFG), prescribing the computational tasks the system can perform in the mode in every period, or a list of other modes as the immediate sub-modes of the current mode. If a mode has sub-modes, when the control lies in this mode, the control should also be in one of the sub-modes. A leaf mode does not have sub-modes, so its body contains a CFG. A mode is either a leaf mode, or it directly or indirectly has leaf modes as its sub-modes. A mode is called top mode if it is not a sub-mode of any other mode. The CFG in a leaf mode is the standard control flow graph, which contains nodes and structures like assignment, module call, conditional and loop. It also supports function units like the ones in conventional programming languages. The syntax of CFG is presented in Figure 2(c).

A module encapsulates computational tasks as its CFG. V_I specifies the set of variables used in the CFG, while V_O is the set of variables modified in the CFG. A module can be invoked by some modes or other modules. As a specification for embedded systems, recursive module calls are forbidden.

A transition (from $Transition^+$) specifying a mode switch from one mode to another is represented as a quadruple, where the first element is the name of the source mode, the second specifies the transition condition, the third is the priority of the transition and the last element is the name of the target mode. The MDM supports mode switches at different levels in the mode-hierarchy. The transition condition (i.e. *guard*) is defined in Fig. 2(b). A state expression can be either a constant, a variable, or a real-value function on state expressions. A boolean term is either a boolean constant, or a predicate on state expressions. There are two kinds of interval expressions, after and duration. These interval expression are very convenient to model system behaviors related with past states. A guard term can be either an interval expression, or a boolean term. A guard is the boolean combination of guard terms. To ensure the mode switches deterministic, we require that the priority of a transition has to be different from the

others in the same mode chain:

$$\forall m \in Mode \cdot \forall t_1, t_2 \in \mathsf{outs}(\mathsf{super\_modes}(md, m)) \cdot t_1 \neq t_2 \Rightarrow \mathsf{prio}(\mathsf{t}_1) \neq \mathsf{prio}(\mathsf{t}_2)$$

The functions $\mathsf{super\_modes}(md, m)$ and $\mathsf{outs}(mlist)$ will be defined later.

2.1.1 Auxiliary Definitions

Given an MDM $md ::= (Var^+, Mode^+, Module^+)$, we introduce two auxiliary relations:
$Contains(md) \subseteq Modes \times Modes$ for mode-subsume relation and
$Trans(md) \subseteq Modes \times Int \times guard \times Modes$ for mode-switch relation.

Given a mode $m = (n, per, ini, b, tran)$ and a transition $t = (m, g, pri, m')$, we define these operations/predicates:

$$\begin{array}{llll} \mathsf{period}(\mathsf{m}) = per & \mathsf{is\_initial}(\mathsf{m}) = ini & \mathsf{CFG}(\mathsf{m}) = b & \\ \mathsf{prio}(\mathsf{t}) = pri & \mathsf{guard}(\mathsf{t}) = g & \mathsf{source}(\mathsf{t}) = m & \mathsf{target}(\mathsf{t}) = m' \end{array}$$

We also define the following auxiliary functions:

$$\begin{array}{l} \mathsf{super\_modes}(md, m) \triangleq \langle m_1, m_2, \ldots, m_k \rangle, \text{ where} \\ \quad m_k = m \wedge m_1 \in TopModes(md) \wedge \forall 1 < i \leq k \cdot (m_{i-1}, m_i) \in Contains(md) \\ \quad \text{and } m \in TopModes(md) \triangleq m \in Modes(md) \wedge \neg \exists m' \cdot (m', m) \in Contains(md) \end{array}$$

$$\mathsf{up\_modes}(md, m, k) \triangleq \{m_i \mid m_i \in \mathsf{super\_modes}(md, m) \wedge \mod(k, \tfrac{\mathsf{period}(m_i)}{\mathsf{period}(m)}) = 0\}$$

$$\mathsf{sub\_mode}(md, m) \triangleq m', \text{ where } (m, m') \in Contains(md) \wedge \mathsf{is\_initial}(m')$$

$$\mathsf{outs}(md, mlist) \triangleq \bigcup_{m \in mlist} \{t \mid t \in Trans(md) \wedge \mathsf{source}(t) = m\}$$

The function $\mathsf{super\_modes}(md, m)$ retrieves a sequence of modes from a top mode to m using the *Contains* relation. The set $TopModes(md)$ consists all the modes which are not sub-modes of any other mode. The function $\mathsf{up\_modes}(md, m, k)$ returns those modes in $\mathsf{super\_modes}(md, m)$ whose periods are consistent with the period count k. MDM requires that the period of a mode should be equal to or multiple to the period of its sub-modes. The function $\mathsf{sub\_mode}(md, m)$ returns the initial sub-mode for a non-leaf node m, and the predicate $\mathsf{is\_initial}(m')$ means that the sub-mode m' is the initial sub-mode in its hierarchy. The function $\mathsf{outs}(mlist)$ returns all outgoing transitions from modes in *mlist*.

2.2 The Semantics

In order to precisely analyze the behaviors of MDM, for instance, model checking of MDM , we need its formal semantics. In this section, we present the operational semantics for MDM.

2.2.1 Configuration

The configuration in our operational semantics is represented as $(md, m, l, pc, k, \Sigma)$, where

- md is the MDM, and m is the mode the system control currently lies in.
- $l \in \{Begin, Execute, End\}$ specifies the system is in the beginning, middle, or end of a period.

$$
\begin{array}{lcl}
\sigma_1 \cdot \ldots \cdot \sigma_n \models b & \Leftrightarrow & \sigma_n \models b \\
\sigma_1 \cdot \ldots \cdot \sigma_n \models \neg g & \Leftrightarrow & \neg(\sigma_1 \cdot \ldots \cdot \sigma_n \models g) \\
\sigma_1 \cdot \ldots \cdot \sigma_n \models g_1 \vee g_2 & \Leftrightarrow & \sigma_1 \cdot \ldots \cdot \sigma_n \models g_1 \text{ or } \sigma_1 \cdot \ldots \cdot \sigma_n \models g_2 \\
\sigma_1 \cdot \ldots \cdot \sigma_n \models g_1 \wedge g_2 & \Leftrightarrow & \sigma_1 \cdot \ldots \cdot \sigma_n \models g_1 \text{ and } \sigma_1 \cdot \ldots \cdot \sigma_n \models g_2 \\
\sigma_1 \cdot \ldots \cdot \sigma_n \models \mathsf{duration}(b,l) & \Leftrightarrow & \sigma_n(l) = v \wedge \exists i{<}n \cdot (\sigma_i(ts){+}v \leq \sigma_n(ts) \wedge \\
 & & \sigma_{i+1}(ts){+}v \geq \sigma_n(ts) \wedge \forall i{\leq}j{\leq}n \cdot \sigma_j(b) = \mathsf{true}) \\
\sigma_1 \cdot \ldots \cdot \sigma_n \models \mathsf{after}(b,l) & \Leftrightarrow & \sigma_n(l) = v \wedge \exists i{<}n \cdot (\sigma_i(ts){+}v \leq \sigma_n(ts) \wedge \\
 & & \sigma_{i+1}(ts){+}v \geq \sigma_n(ts)) \wedge \sigma_i(b) = \mathsf{true})
\end{array}
$$

Table 1: The Interpretation of Guards

- $pc \in \mathscr{L}$, where $\mathscr{L} = \mathscr{N} \cup \{Start, Exit, \perp\}$ is the program counter to execute the control flow graph. $\mathscr{N}$ is used to represent the nodes in control flow graphs and *Start*, *Exit* denote the start and exit locations of a control flow graph respectively. If the current mode is not equipped with any flow graph, we use the symbol $\perp$ as a placeholder.
- The fourth component k records the count of periods for the current mode. If the system switches to another mode, it will be reset to 1. The period count is used to distinguish whether a super-mode of the current mode is allowed to check its mode switch guard.
- Σ is a list of states of the form $\Sigma' \cdot \sigma$, where σ denotes the current state ($\sigma \in State \triangleq Vars \to \mathbb{R}$) and Σ' represents a history of states.

Guards The evaluation of a transition guard may depend on the current state as well as some historical states. Table 1 shows how to interpret a guard in a given sequence of states. The symbol ts is the abbreviation of the variable *timestamp*. The guard $\mathsf{duration}(b,l)$ evaluates to true if the boolean expression b has been true during the time interval l up to the current moment. The guard $\mathsf{after}(b,l)$ evaluates to true if the boolean expression b was true the time interval l ago. In this table, b is a pure boolean expression without interval expressions and l is a state expression.

2.2.2 Operational Rules

The operational rules for MDM are given in Table 2. Here we adopt a big-step operational semantics for MDM, which means that we only observe the start and end points of a period in the current mode, while the state changes within a period are not recorded. This is reasonable since in practice engineers usually monitor the states at the two ends of a period to decide if it works well. In the rules, we make use of an auxiliary function *execute* to represent the execution results for the mode in one period.

$$execute : \mathscr{CFG}(V) \times \mathscr{L} \times State \times \mathbb{R}^+ \to \mathscr{L} \times State$$

It receives a flow graph, a program counter, an initial state and the time permitted to execute and returns the state and program counter after the given time is expired. Its detailed definition is left in the report [22]. We now explain the operational rules:

1. (ENTER). When the system is at the beginning of a period, if the current mode m has sub-modes, the system enters the initial sub-mode of m.
2. (DETECT). When the system is at the beginning of a period, if the current mode m is a leaf mode, the system updates its state by sampling from sensors. The function $\mathsf{sampling}$ represents the side-effect on variables during sensor detection. The period label l is changed to be *Execute*, indicating that the system will then perform computational tasks specified by the control flow graph of m.

$$\text{(ENTER)}\quad \frac{\mathsf{CFG}(\mathsf{m}) = \bot}{(md, m, Begin, \bot, k, \Sigma) \longrightarrow (md, m', Begin, pc', k, \Sigma)}$$

$$\text{where } m' = \mathsf{sub\_mode}(md, m) \text{ and } pc' = \begin{cases} \bot, & \text{if } \mathsf{CFG}(\mathsf{m}') = \bot \\ Start, & \text{if } \mathsf{CFG}(\mathsf{m}') \neq \bot \end{cases}$$

$$\text{(DETECT)}\quad \frac{\mathsf{CFG}(\mathsf{m}) \neq \bot}{(md, m, Begin, pc, k, \Sigma \cdot \sigma) \longrightarrow (md, m, Execute, pc, k, \Sigma \cdot \mathsf{sampling}(\sigma))}$$

$$\text{(EXECUTE)}\quad \frac{execute(\mathsf{CFG}(\mathsf{m}), pc, \sigma, \mathsf{period}(\mathsf{m})) = (pc', \sigma')}{(md, m, Execute, pc, k, \Sigma \cdot \sigma) \longrightarrow (md, m, End, pc', k, \Sigma')}$$

$$\text{where} \quad \Sigma' = \Sigma \cdot \sigma'[ts \mapsto \sigma(ts) + \mathsf{period}(\mathsf{m})]$$

$$\text{(CONTINUE)}\quad \frac{pc \neq Exit}{(md, m, End, pc, k, \Sigma) \longrightarrow (md, m, Execute, pc, k, \Sigma)}$$

$$\text{(REPEAT)}\quad \frac{\forall t \in \mathsf{outs}(\mathsf{up\_modes}(md, m, k)) \cdot \Sigma \not\models \mathsf{guard}(\mathsf{t})}{(md, m, End, Exit, k, \Sigma) \longrightarrow (md, m, Begin, Start, k+1, \Sigma)}$$

$$\text{(SWITCH)}\quad \frac{\begin{array}{l}\exists t \in \mathsf{outs}(md, \mathsf{up\_modes}(md, m, k)) \cdot \Sigma \models \mathsf{guard}(\mathsf{t}) \wedge \\ \forall t' \in \mathsf{outs}(\mathsf{up\_modes}(md, m, k)) - \{t\} \cdot (\Sigma \not\models \mathsf{guard}(\mathsf{t}') \vee \mathsf{prio}(\mathsf{t}') < \mathsf{prio}(\mathsf{t}))\end{array}}{(md, m, End, Exit, k, \Sigma) \longrightarrow (md, m', Begin, pc', 1, \Sigma)}$$

$$\text{where } m' = \mathsf{target}(\mathsf{t}) \text{ and } pc' = \begin{cases} \bot, & \text{if } \mathsf{CFG}(\mathsf{m}') = \bot \\ Start, & \text{if } \mathsf{CFG}(\mathsf{m}') \neq \bot \end{cases}$$

Table 2: Operational Semantic Rules for MDM

3. (EXECUTE). This rule describes the behaviors of executing CFG of the leaf mode m. The function *execute* is used to compute the new state σ' from σ. The computation task may be finished in the current period and $pc' = Exit$ holds or the task is not finished and the program counter points to some location in the control flow graph. The value of the timestamp variable ts in σ' is equal to its value in state σ plus the period of the mode m.

4. (CONTINUE). This rule tells that when the computation task in leaf mode is not finished in a period, it will continue its task in the next period. In this case, the system is implicitly not allowed to switch to other modes from the current mode. When moving to the next period, sensor detection is skipped.

5. (REPEAT). This rule specifies the behavior of restarting the flow graph when the computation task is finished in a period. When it is at the end of a period and the system finishes executing the flow graph ($pc = Exit$), if there is no transition guard enabled, the system stays in the same mode and restarts the computation specified by the flow graph.

6. (SWITCH). This rule specifies the behavior of the mode transition. There exists a transition t, whose guard holds on the sequence of states Σ. And the priority of t is higher than that of any other enabled transitions.

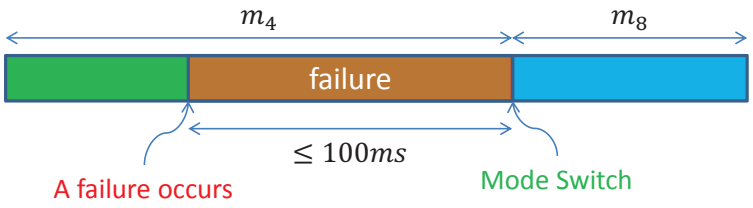


Figure 3: A Property about Failure

$$\begin{array}{lll} \textit{Terms} & \theta & \triangleq \quad r \mid v \mid l \mid f(\theta_1, \ldots, \theta_n) \\ \textit{Formulas} & \phi, \psi & \triangleq \quad tt \mid ff \mid p(\theta_1, \ldots, \theta_n) \mid \neg\phi \mid \phi \wedge \psi \mid \phi \frown \psi \end{array}$$

Figure 4: The Syntax of ITL

3 The Property Specification Language

We adopt the Interval Temporal Logic (ITL) [17] as the property specification language. The reason why we adopt the interval-based logic instead of state-based logics like LTL or CTL is that most of the properties the domain engineers care about are related to some duration of time. For instance, the engineers would like to check if the system specified by MDM can stay in a specific state for a continuous period of time instead of just reaching this state. Another typical scenario illustrated in Fig. 3 is that, "*when the system control is in mode m_4, if a failure occurs, it should switch to mode m_8 in 100 ms*". The standard LTL formula $\Box(failure \wedge m_4 \Rightarrow \Diamond m_8)$ can be used to specify that "*when the system is in mode m_4, and a failure occurs, it should switch to mode m_8*". But the real-time feature "*in 100 ms*" is lost. Though the extensions of LTL or CTL may also describe the interval properties to some extent, it is more natural for the domain engineers to use interval-based logic since the intuitive chop operator ($\frown$) is available in ITL.

An interval logic formula can be interpreted over a time interval [7] or over a "state interval" (a sequence of states) [17] . As explained later in this section, our proposed specification language will be interpreted in the latter way [17] except for a small modification on the interpretation of the chop operator ($\frown$).

3.1 Syntax

The syntax of the specification language is defined in Fig 4, where

- The set of terms θ contains real-value constants r, temporal variables v, a special variable l, and functions $f(\theta_1, \ldots, \theta_n)$ (with f being an n-arity function symbol and $\theta_1, \ldots, \theta_n$ being terms).
- Formulae can be boolean constants (tt, ff), predicates ($p(\theta_1, \ldots, \theta_n)$ with p, an n-arity predicate symbol), classical logic formulae (constructed using $\neg$, $\wedge$, etc), or interval logic formulae (constructed using $\frown$). If the formula $\phi \frown \psi$ holds for an interval ℓ, it means that the interval ℓ can be "chopped" into two sub-intervals, where ϕ holds for the first sub-interval and ψ holds for the second one.

As a kind of temporal logic, ITL also provides the $\Box$ and $\Diamond$ operators. They are defined as the abbreviations of $\frown$.

$$\Diamond\phi \triangleq tt \frown (\phi \frown tt), \text{ for some sub-interval }, \quad \Box\phi \triangleq \neg\Diamond(\neg\phi), \text{ for all sub-intervals}$$

$$\mathscr{I}_{\mathscr{T}}(r,\Sigma) = r$$
$$\mathscr{I}_{\mathscr{T}}(l,\Sigma) = \begin{cases} \sigma_{n-1}(ts) - \sigma_0(ts) & \text{if } \Sigma = \sigma_0 \ldots . \sigma_{n-1} \\ \infty & \text{if } \mid \Sigma \mid = \infty \end{cases}$$
$$\mathscr{I}_{\mathscr{T}}(v,\sigma_0.\Sigma) = \sigma_0(v)$$
$$\mathscr{I}_{\mathscr{T}}(f(\theta_1,\ldots,\theta_n),\Sigma) = f(\mathscr{I}_{\mathscr{T}}(\theta_1,\Sigma),\ldots,\mathscr{I}_{\mathscr{T}}(\theta_n,\Sigma))$$

$$\begin{array}{rll} \mathscr{I}_{\mathscr{F}}(p(\theta_1,\ldots,\theta_n),\Sigma) = \mathsf{true} & \text{iff} & p(\mathscr{I}_{\mathscr{T}}(\theta_1,\Sigma),\ldots,\mathscr{I}_{\mathscr{T}}(\theta_n,\Sigma)) \\ \mathscr{I}_{\mathscr{F}}(tt,\Sigma) = \mathsf{true} & \text{iff} & \textit{always} \\ \mathscr{I}_{\mathscr{F}}(ff,\Sigma) = \mathsf{false} & \text{iff} & \textit{always} \\ \mathscr{I}_{\mathscr{F}}(\neg\phi,\Sigma) = \mathsf{true} & \text{iff} & \mathscr{I}_{\mathscr{F}}(\phi,\Sigma) = \mathsf{false} \\ \mathscr{I}_{\mathscr{F}}(\phi \wedge \psi,\Sigma) = \mathsf{true} & \text{iff} & \mathscr{I}_{\mathscr{F}}(\phi,\Sigma) = \mathsf{true} \text{ and } \mathscr{I}_{\mathscr{F}}(\psi,\Sigma) = \mathsf{true} \\ \mathscr{I}_{\mathscr{F}}(\phi^\frown\psi,\Sigma) = \mathsf{true} & \text{iff} & \exists k < \infty \cdot\ \Sigma = (\sigma_0 \ldots \sigma_k \cdot \Sigma') \wedge \\ & & \quad \mathscr{I}_{\mathscr{F}}(\phi,\sigma_0 \ldots \sigma_k) = \mathsf{true} \wedge \mathscr{I}_{\mathscr{F}}(\psi,\Sigma') = \mathsf{true} \end{array}$$

Table 3: Interpretation of the Specification Language

By the specification language proposed here, we can describe the properties the domain engineers may desire. For instance, the following property describes the scenario shown in Fig. 3.

$$\Box(m_4 \wedge (\neg \mathit{failure}^\frown \mathit{failure})^\frown tt \Rightarrow m_4 \wedge (\neg \mathit{failure}^\frown (\mathit{failure} \wedge l \leq 100))^\frown m_8{}^\frown tt)$$

3.2 Interpretation

Terms/formulae in our property specification language are interpreted in the same way as in Maszkowski [17], where an interval is represented by a finite or infinite sequence of states ($\Sigma = \sigma_0\sigma_1 \ldots \sigma_{n-1} \ldots$), where $\sigma_i \in \mathit{State}$. The interpretation is given by two functions (1) term interpretation :$\mathscr{I}_{\mathscr{T}} \in \mathit{Terms} \times \mathit{Intv} \mapsto \mathbb{R}$, and (2) formula interpretation function: $\mathscr{I}_{\mathscr{F}} \in \mathit{Formulas} \times \mathit{Intv} \mapsto \{\mathsf{true}, \mathsf{false}\}$. Table 3 defines these two functions, where ts denotes the variable *timestamp*. The value of the variable *timestamp* increases with the elapse of the time. i.e., for any two states in the same interval σ_i, σ_j, if $i<j$, then $\sigma_i(ts)<\sigma_j(ts)$. Thus, we can compute the length of time interval based on the difference of the two time stamps located in the first and last states respectively. The interpretation of a variable v on Σ is the evaluation of v on the first state of Σ. Note that our chop operator requires that the first sub-interval of Σ is restricted to be finite no matter whether the interval Σ itself is finite or not.

4 MDM Verification by Statistical Model Checking

As a modelling & verification framework for periodic control systems, MDM supports the modelling of periodic behaviors, mode transition, and complex computations involving linear or non-linear mathematical formulae. Moreover, it also provides a property specification language to help the engineers capture requirements. In this section, we will show how to verify that an MDM model satisfies properties formalized in the specification language. There are two main obstacles to apply classic model checking techniques on MDM: (1) MDM models involve complex computations like non-linear mathematic formulae; (2) MDM models are open systems which need intensive interactions with outside.

Our proposed approach relies on Statistical Model Checking(SMC) [20, 23, 16, 6]. SMC is a simulation-based technique that runs the system to generate traces, and then uses statistical theory to

analyze the traces to obtain the verification estimation of the entire system. SMC usually deals with the following quantitative aspect of the system under verification [23]:

What is the probability that a random run of a system will satisfy the given property ϕ?

Since the SMC technique verifies the target system with the probability estimation instead of the accurate analysis, it is very effective when being applied to open and non-linear systems. Because SMC depends on the generated traces of the system under verification, we shall briefly describe how to simulate an MDM and then present an SMC algorithm for MDM.

4.1 MDM Simulation

The MDM model captures a reactive system [10]. The MDM model executes and interacts with its external environment in a *control loop* in one period as follows: (1) Accept inputs via sensors from the environment. (2) Perform computational tasks. (3) Generate outputs to drive other components. The MDM simulation engine simulates the process of the control loop above.

Generally speaking, the simulation is implemented according to the inference rules defined in Table 2. However, the behaviors of an MDM model depends not only on the MDM itself, but also on the initial state and the external environment. When we simulate the MDM model, the initial values are randomly selected from a range specified by the control engineers from CAST. As a specification language, the type of variables defined in MDM can be real number. To implement the simulation, we use float variables instead, which may introduce some problems on precision. There are lots of techniques can be adopted to check if any loss of precision may cause problems[14]. Because the simulation doesn't take care of the platform to deploy the system specified by the MDM, the time during simulation is not the real time, but the logic time. For each iteration in the *control loop*, the time is increased by the length of period of the current mode.

To make the simulation executable, we have to simulate the behaviors of the environment to make the MDM model to be closed with its environment. The environment simulator involving kinematical computations designed by the control engineers is combined with the MDM to simulate the physical environment the MDM model interacts with. In the beginning of each period, the simulator checks whether there are sub-modes in the current mode. If so, the simulator takes the initial sub-mode as the new current mode. When the current mode is a leaf mode, the simulator invokes the library simulating the physical environments and updates the internal state by getting the value detected from sensors. Then the simulator executes the control flow graph in the leaf mode. We assume that there is enough time to execute the CFG. The situation that tasks are allowed not to be finished in one period is not considered during simulation. In the end of each period, the guard of transitions is checked. The satisfaction of duration and after guards does not only depend on the current state, but also the past states. The simulator sets a counter for each duration/after guard instead of recording the past states. As an MDM model is usually a non-terminating periodic system, the bound of periods is set during the process of simulation.

4.2 SMC Algorithm

We apply the methodology in [23] to estimate the probability that a random run of an MDM will satisfy the given property ϕ with a certain precision and certain level of confidence. The statistical model checking algorithm for MDM is illustrated in Fig. 5. Since the run of the MDM usually is infinite, the users can set the length of the sequence by the number of periods based on the concrete application. This algorithm firstly computes the number N of runs based on the formula $N := 4 * \log(1/\delta)/\varepsilon^2$ which involves

input *md*: the MDM, ϕ: property, B: bound of periods
δ: confidence, ε: approximation
output p: the probability that ϕ holds on an arbitrary run of *md*
begin
10 $N := 4 * \frac{\log \frac{1}{\delta}}{\varepsilon^2}$, $a := 0$
20 **for** $i := 1$ **to** N **do**
30 generate an initial state s_0 randomly
40 simulate the MDM from s_0 in B periods to get the state trace Σ
50 **if** $(\mathscr{I}_{\mathscr{F}}(\phi, \Sigma) = \mathsf{true})$ **then** $a := a + 1$
60 **end for**
70 **return** $\frac{a}{N}$
end

Figure 5: Probability Estimation for MDM

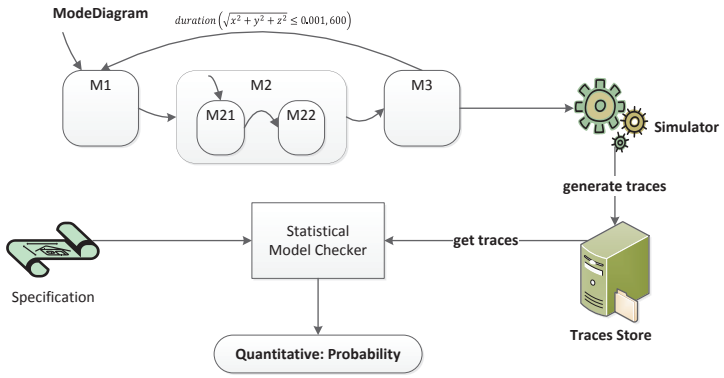


Figure 6: The Framework of Implementation

the confidence interval $[p-\delta, p+\delta]$ with the confidence level $1-\varepsilon$. Then the algorithm generates the initial state (line 30) and gets a state trace Σ by the inference rules defined in Table 2 (line 40). The algorithm in line 50 decides whether ϕ holds on the constructed interval based on the interpretation for the specification language mentioned in Section 3. If the interpretation is true, the algorithm increases the number of traces on which property ϕ holds. Line 70 returns the probability for the satisfaction of ϕ on the MDM.

4.3 Experiments

We have implemented the MDM modeling and verification framework and applied it onto several real periodic control systems. The implementation framework of SMC is illustrated in Fig. 6, where the simulator is used to simulate the MDM by the proposed operational semantics and the generated traces are for the statistical model checker. One of the real periodic control systems (termed as A) is for spacecraft control developed by CAST. Fig. 1(shown in Section 1) is a small portion of the MDM model for system A.

We communicate with the engineers in CAST, summarize several properties the two models of spacecrafts should obey, and present these properties in our specification language. A total of 12 properties are

developed by the engineers and these properties are verified on the systems A. We only highlight three properties because the verification results on these three properties reveal two defects.

- *After 3000 seconds, the system will eventually reach the stable state forever*

$$\ell \geq 3000 \Rightarrow tt^\frown\Box(\sqrt{\omega_x^2+\omega_y^2+\omega_z^2} \leq 0.1 \wedge \sqrt{\dot{\omega}_x{}^2+\dot{\omega}_y{}^2+\dot{\omega}_z{}^2} \leq 0.01))$$

  where ω_x, ω_x and ω_z are angles. $\dot{\omega}_x$, $\dot{\omega}_x$ and $\dot{\omega}_z$ are angle rates.

- *The system starts from mode m*0*, and then it will finally switch to mode m*5 *or m*6 *or m*8*, and stay in one of these three modes forever*

$$(\mathsf{mode}=0)^\frown tt^\frown\Box(\mathsf{mode}=5 \vee \mathsf{mode}=6 \vee \mathsf{mode}=8)$$

- *Whenever the system switches to mode m*4 *and then leaves m*4*, during its stay in m*4*, it firstly stays in sub-mode G*0*, and then it switches to sub-mode G*1*, and then G*2*.*

$$\Box(\mathsf{mode}\neq 4^\frown\mathsf{mode}=4^\frown\mathsf{mode}\neq 4 \Rightarrow \mathsf{mode}\neq 4^\frown \\ \mathsf{mode}=4\wedge(\mathsf{gm}=0^\frown\mathsf{gm}=1^\frown\mathsf{gm}=2)^\frown tt)$$

For the parameters of the statistical model checking algorithm, we set the half length of confident interval to be 1% ($\delta = 1\%$) and the error rate to be 5% ($\varepsilon = 5\%$). Based on this algorithm, the total 7369 traces for each control system are required to be generated to compute the probabilities during the verification process.

During the verification phase by the statistical model checking on MDM, two design defects in system A are uncovered by analyzing the verification results: (1) A variable is not initialized properly. (2) A value from sensors is detected from the wrong hardware address. In the traditional developing process in CAST, these two defects may be revealed only after a prototype of the software is developed and then tested. Our approach can find such bugs in design phase and reduce the cost to fix defects.

5 Related Work

Our MDM can be broadly considered as a variant of Statecharts [11], where a mode in MDM is similar to a state in the Statecharts. However, we note the following distinctions: (1) In Statecharts, when a transition guard holds, the system immediately switches to the target state. But in MDM, mode switches are only allowed to be triggered at the end of a period. (2) In Statecharts, a transition guard is usually a boolean expression on the current(source) state; while in MDM, transition guards may involve past states via predicates like during and after. (3) In Statecharts, all observations on the system are the states; while MDM also concerns about the computation aspect of the system by means of the flow graphs provided in the leaf modes.

Timed Automata are a modeling tool for the description and verification of real-time systems [3, 5]. It provides the *clock* variable to support the time explicitly. Timed Automata only focus on the linear computation for time since it has nice time zone semantics supporting the timed verification. Hybrid Automata [2] extend the traditional automata to deal with complex computation like the difference and differentiation while it is not a systematic modeling tool which supports the rich modeling mechanisms like the hierarchy, types etc.

Giese et al. [8] have proposed a semantics of real-time variant of Statecharts by introducing the Hierarchical Timed Automata. In another work [9] they have presented a compositional verification

approach to the real time UML designs. A. K. Mok et al. have developed a kind of herarchical real-time chart named "Modechart" [15]. Compared with Giese et al. [8], parallel modes are supported in Modechart.

Stateflow is the Statechart-like language used in the commercial software Matlab/Simulink [1]. The Stateflow language enriches Statecharts to allow it to support flow-based and state-based computations together for specifying discrete event systems. Our MDM focuses more on periodic control systems, which can be regarded as a specific type of discrete event systems, and it provides the first class element *period* to facilitate the precise modeling of periodic-driven systems. The transitions in Stateflow can be attached with a flowchart to describe complicated computation, the MDM specifies the flow graph for the computation in its leaf modes. While Stateflow focuses only on the modelling aspect of the systems, the MDM integrates modelling and reasoning by providing a property specification language and a verification algorithm.

Some researchers introduce *operational modes* [18, 19] during the modeling in hardware/software co-synthesis. The operational mode is essentially a state in the automaton, but it can be attached a flowchart for the description of the computation. It does not support the nested mode and period explicitly. However, it is actually an informal modeling notation because it allows to specify the system behaviors in natural language. Our MDM is a lightweight formal notation for the modeling with its precise operational semantics.

Giotto is also a periodic-driven modeling language proposed by Henzinger et al. [13]. The main difference between Giotto and MDM is the computation mechanism provided. The tasks in a mode can be performed in parallel in Giotto while the details of the tasks are omitted and are moved to the implementation stage. The MDM supports the detailed description of the computation in their leaf modes since the design of it is targeted for control systems which may involve rich algorithms. The MDM does not support the parallel computation explicitly at present since it could bring the nondeterminism at the design level. The emphasis of the Giotto is more for the modeling and synthesis of parallel tasks while the MDM is for the modeling and verification based on the proposed specification language.

Runtime Verification is a verification approach based on extracting information by executing the system and using the information to detect whether the observed behaviors violating the expected properties [12, 21]. The verification approach we apply in this paper is also a kind of runtime verification. But our methodology is the off-line analysis, while [21] applies an on-line monitoring approach using Aspect-J. The reason to propose off-line analysis is that the cost to decide if an ITL formula is satisfiable on a given trace is expensive, so information extraction and analysis are separated to two phases in our approach.

6 Conclusion

In this paper, we propose the Mode Diagram Modelling framework (MDM), a domain-specific formal visual modelling language for periodic control systems. To support formal reasoning, MDM is equipped with a property specification language based on interval temporal logic and a statistical model checking algorithm. The property specification language allows engineers to precisely capture various properties they desire, while the verification algorithm allows them to reason about MDM models with respect to those properties. The viability and effectiveness of the proposed MDM framework have been demonstrated by a number of real life case studies, where defects of spacecraft control systems have been detected and uncovered in the early design stage.

References

[1] *The Mathworks: Stateflow and Stateflow Coder, User's Guide*. www.mathworks.com/help/releases/R13sp2/pdf_doc/stateflow/sf_ug.pdf.

[2] Rajeev Alur, Costas Courcoubetis, Thomas A. Henzinger & Pei-Hsin Ho (1992): *Hybrid Automata: An Algorithmic Approach to the Specification and Verification of Hybrid Systems*. In Robert L. Grossman, Anil Nerode, Anders P. Ravn & Hans Rischel, editors: *Hybrid Systems*, *Lecture Notes in Computer Science* 736, Springer, pp. 209–229. Available at http://dx.doi.org/10.1007/3-540-57318-6_30.

[3] Rajeev Alur & David L. Dill (1994): *A theory of timed automata*. *Theor. Comput. Sci.* 126, pp. 183–235, doi:10.1016/0304-3975(94)90010-8. Available at http://dl.acm.org/citation.cfm?id=180782.180519.

[4] Ananda Basu, Saddek Bensalem, Marius Bozga, Benoît Delahaye & Axel Legay (2012): *Statistical abstraction and model-checking of large heterogeneous systems*. *STTT* 14(1), pp. 53–72. Available at http://dx.doi.org/10.1007/s10009-011-0201-2.

[5] Gerd Behrmann, Alexandre David, Kim Guldstrand Larsen, Paul Pettersson & Wang Yi (2011): *Developing UPPAAL over 15 years*. *Softw., Pract. Exper.* 41(2), pp. 133–142. Available at http://dx.doi.org/10.1002/spe.1006.

[6] Alexandre David, Kim G. Larsen, Axel Legay, Marius Mikucionis & Zheng Wang (2011): *Time for Statistical Model Checking of Real-Time Systems*. In Ganesh Gopalakrishnan & Shaz Qadeer, editors: *CAV*, *Lecture Notes in Computer Science* 6806, Springer, pp. 349–355. Available at http://dx.doi.org/10.1007/978-3-642-22110-1_27.

[7] Bruno Dutertre (1995): *Complete Proof Systems for First Order Interval Temporal Logic*. In: *LICS*, IEEE Computer Society, pp. 36–43.

[8] Holger Giese & Sven Burmester (2003): *Real-Time Statechart Semantics*. Technical Report TR-RI-03-239, Software Engineering Group, University of Paderborn, Warburger Str. 100, D-33098 Paderborn, Germany.

[9] Holger Giese, Matthias Tichy, Sven Burmester, Wilhelm Schäfer & Stephan Flake (2003): *Towards the compositional verification of real-time UML designs*. In: *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-11, ACM, New York, NY, USA, pp. 38–47, doi:http://doi.acm.org/10.1145/940071.940078.

[10] D. Harel & A. Pnueli (1985): *On the development of reactive systems*, pp. 477–498. Springer-Verlag New York, Inc., New York, NY, USA. Available at http://dl.acm.org/citation.cfm?id=101969.101990.

[11] David Harel (1987): *Statecharts: A visual formalism for complex systems*. *Science of Computer Programming* 8(3), pp. 231–274, doi:http://dx.doi.org/10.1016/0167-6423(87)90035-9.

[12] Klaus Havelund (2008): *Runtime Verification of C Programs*. In: *Proceedings of the 20th IFIP TC 6/WG 6.1 international conference on Testing of Software and Communicating Systems: 8th International Workshop*, TestCom '08 / FATES '08, Springer-Verlag, Berlin, Heidelberg, pp. 7–22, doi:http://dx.doi.org/10.1007/978-3-540-68524-1_3.

[13] Thomas A. Henzinger, Benjamin Horowitz & Christoph M. Kirsch (2001): *Giotto: a Time-triggered Language for Embedded Programming*. Technical Report, Department of Electronic Engineering and Computer Science, University of California, Berkeley, Berkeley, CA, USA.

[14] Nicholas J. Higham (2002): *Accuracy and stability of numerical algorithms (2. ed.)*. SIAM.

[15] Farnam Jahanian & Aloysius K. Mok (1994): *Modechart: A Specification Language for Real-Time Systems*. *IEEE Trans. Softw. Eng.* 20, pp. 933–947, doi:http://dx.doi.org/10.1109/32.368134. Available at http://dx.doi.org/10.1109/32.368134.

[16] Kim G. Larsen, Axel Legay, Louis-Marie Traonouez & Andrzej Wasowski (2011): *Robust Specification of Real Time Components*. In Uli Fahrenberg & Stavros Tripakis, editors: *FORMATS*, *Lecture*

Notes in Computer Science 6919, Springer, pp. 129–144. Available at http://dx.doi.org/10.1007/978-3-642-24310-3_10.

[17] Ben C. Moszkowski & Zohar Manna (1983): *Reasoning in Interval Temporal Logic*. In Edmund M. Clarke & Dexter Kozen, editors: *Logic of Programs*, *Lecture Notes in Computer Science* 164, Springer, pp. 371–382. Available at http://dx.doi.org/10.1007/3-540-12896-4_374.

[18] Hyunok Oh & Soonhoi Ha (2002): *Hardware-software cosynthesis of multi-mode multi-task embedded systems with real-time constraints*. In: *Proceedings of the tenth international symposium on Hardware/software codesign*, CODES '02, ACM, New York, NY, USA, pp. 133–138, doi:http://doi.acm.org/10.1145/774789.774817. Available at http://doi.acm.org/10.1145/774789.774817.

[19] Marcus T. Schmitz, Bashir M. Al-Hashimi & Petru Eles (2005): *Cosynthesis of energy-efficient multimode embedded systems with consideration of mode-execution probabilities*. *IEEE Trans. on CAD of Integrated Circuits and Systems* 24(2), pp. 153–169. Available at http://dx.doi.org/10.1109/TCAD.2004.837729.

[20] Koushik Sen, Mahesh Viswanathan & Gul Agha (2004): *Statistical Model Checking of Black-Box Probabilistic Systems*. In Rajeev Alur & Doron Peled, editors: *CAV*, *Lecture Notes in Computer Science* 3114, Springer, pp. 202–215. Available at http://dx.doi.org/10.1007/978-3-540-27813-9_16.

[21] Volker Stolz & Eric Bodden (2006): *Temporal Assertions using AspectJ*. *Electron. Notes Theor. Comput. Sci.* 144, pp. 109–124, doi:http://dx.doi.org/10.1016/j.entcs.2006.02.007.

[22] Zheng Wang, Geguang Pu, Shengchao Qin, Jianwen Li, Kim G. Larsen, Jan Madsen, Bin Gu & Jifeng He (2011): *ModeDiagram: A Modeling Notation for Requirement Analysis in Aerospace*. Technical Report LAB-205-TR-HT-11-0812, Software Engineering Insititute, East China Normal University, North Zhongshan Road. 3663, Shanghai, China. Http://www.lab205.org/MDM/reports.html.

[23] Håkan L. S. Younes (2005): *Probabilistic Verification for "Black-Box" Systems*. In Kousha Etessami & Sriram K. Rajamani, editors: *CAV*, *Lecture Notes in Computer Science* 3576, Springer, pp. 253–265. Available at http://dx.doi.org/10.1007/11513988_25.

[24] Håkan L. S. Younes & Reid G. Simmons (2002): *Probabilistic Verification of Discrete Event Systems Using Acceptance Sampling*. In Ed Brinksma & Kim Guldstrand Larsen, editors: *CAV*, *Lecture Notes in Computer Science* 2404, Springer, pp. 223–235. Available at http://dx.doi.org/10.1007/3-540-45657-0_17.

Towards the Formal Verification of Quantum Optical Systems

M. Y. Mahmoud V. Aravantinos S. Tahar

Electrical and Computer Engineering Dept.
Concordia University
Montreal, Canada

mosolim@ece.concordia.ca vincent@ece.concordia.ca tahar@ece.concordia.ca

Nowadays, optics has several applications in many industries, e.g, Internet and mobile communications, and laser surgery. Quantum optics in particular plays an important role in many disciplines, e.g., information technology. The systems developed using quantum optics have several applications which can be critical (with respect to either safety or financial aspects). Their verification is thus an extremely important problem. This is done usually with paper-and-pencil analysis, simulation or computer algebra systems. However these techniques have some flaws that we propose to address using formal verification, and, more specifically, theorem proving. In this position paper, we sketch a formalization of quantum optics using a theorem prover and describe potential applications of these techniques. We focus in particular on the implementation of quantum bits (i.e., the first step towards a quantum computer) using coherent laser light.

1 Introduction

Classical physics (more precisely, electromagnetic theory) studies light as an electromagnetic wave. On the contrary, quantum optics studies light as a stream of particles, called *photons* [5]. Based on this concept, quantum optics investigates new properties and phenomena about the light, especially with a low number of photons [13]. This investigation allows a better use of existing optical devices, e.g., beam splitters [10], and the invention of totally new quantum devices, e.g., single photon devices [12]. These devices help in different fields; sometimes they enhance the performance, e.g., detection of gravitational waves, and in other cases they define totaly new solutions, e.g., quantum communications [20]. In addition, quantum optics is one of the most practical implementations of quantum computers [17].

System verification represents a critical issue in every design process. For quantum mechanics and especially quantum optics, the available verification methods are simulation, paper-and-pencil, numerical methods, and computer algebra systems ("CAS"). In the first case, the systems are simulated on a computer or physically in an optical laboratory. For large systems, laboratories are more efficient and effective than computer simulation since it was proved in 1982 by Feyman that quantum systems cannot be simulated on ordinary computers [4]. Although laboratories can be sufficient, they raise cost and safety issues. In the paper-and-pencil approach, all the verification process is done by modeling the system and proving, using existing physic knowledge, that the system satisfies its specifications. However all this process is done by a human and is thus much error-prone, particularly when the system is very large. Thus, computer methods can be used to help the human – and thus decrease the risk of errors – which yields the two last methods: numerical methods (typically Matlab [19]) and CAS (typically Mathematica [3]). Both kinds of tools are used to help the simplification and generation of intermediate mathematical steps. However, these tools are not sufficient: they cannot be substituted fully to the paper-and-pencil approach since they cannot mathematically express the whole model of the system.

Formal verification is an alternative to the techniques mentioned above. It involves the development of a formal (i.e., mathematical) proof that the system satisfies its specifications by using a *theorem prover*,

Submitted to:
FTSCS 2012

i.e., a software allowing to express and reason about mathematical theories. We can then prove properties about the system *inside* the theorem prover (typically, we prove that the model of the system satisfies its specifications). The whole interest in this method is that the theorem prover is able to ensure that the provided proof is mathematically flawless. The language allowing to express the mathematical properties in the theorem prover is usually *first-order logic (FOL)* or *higher-order logic (HOL)*. Several theorem provers exist such as HOL4 [18], HOL Light [7], PVS [16], Isabelle [15] or Coq [14].

In this work, we investigate the use of theorem proving for the verification of quantum optical systems. Theorem proving has proved, so far, to be able to express everything that mathematics can express. Therefore, it gives us strong confidence in the fact that they can handle quantum mechanics, contrary to CAS. We target the formalization of quantum optics in HOL Light and the use of this formalization in quantum system verification. To this end, we propose a general methodology and give a possible application of this methodology: the verification of a quantum bit implementation as a coherent laser light [17].

2 Formal Verification Methodology

Figure 1 shows our proposed methodology. The right part follows the classical paradigm: We take as input a system and some specifications about its expected behavior. From the informal description of the system, we make a formal model, i.e., the model is expressed mathematically using higher-order logic. The specifications are similarly expressed. Once this is done, we use HOL Light to prove formally that the model satisfies its specifications. If this is the case, we can conclude that the system is correct, otherwise it is faulty.

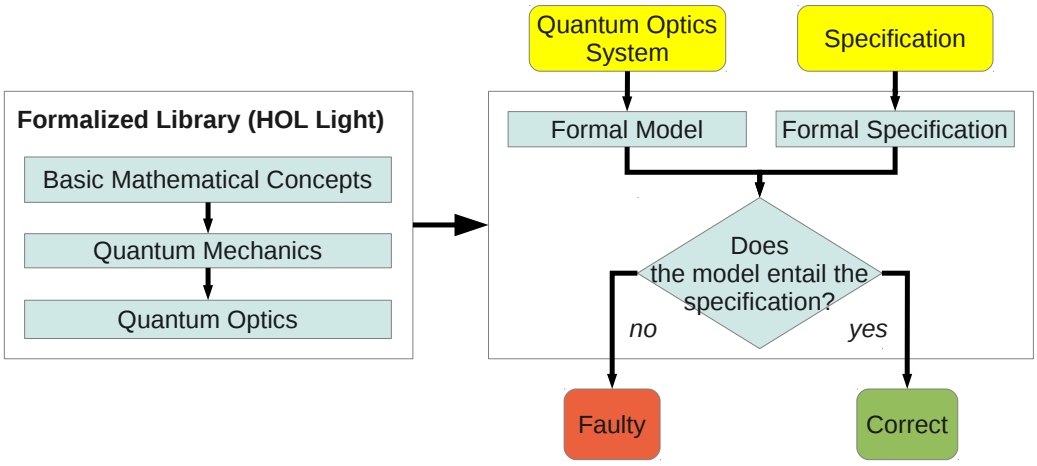


Figure 1: Overview of the Verification Methodology

To formalize the system and its specifications, we need to express quantum optics in HOL Light. This requires that we formalize quantum mechanics, which themselves require some basic mathematical concepts: an algebra of complex-valued functions, an inner product over this algebra (yielding notions of orthogonality and norm), an algebra of operators over complex-valued functions (linear and hermitian operators, closure properties of such operators), definitions of eigenvalues and eigenvectors for these operators, and a statistical interpretation of these objects (expectation, variance, etc.). We can then develop quantum mechanics notions: quantum state space (formally represented by the above complex-valued

functions), quantum system (made of a quantum state space, a set of coordinates, and a Hamiltonian formalized by operators), observables and dynamics (represented by hermitian operators as above).

Developing quantum optics from quantum mechanics then requires that we formalize the so-called "canonical quantization" of classical optics: In quantum mechanics, we make a distinction between *originally quantum systems* and *originally classical systems* [6]. Originally classical systems have to be transformed into quantum systems by a *quantization* process. In 1930, Dirac [2] defined such a transformation called the *canonical quantization*. Applying it to classical optics (more precisely to electromagnetism) results in the quantum optics theory. The quantization of an electromagnetic field, especially a single-mode field (i.e., an electromagnetic field with single resonance frequency), forms the basis of quantum optics.

Among the above mentioned theories, we have currently formalized the basic mathematical concepts required for quantum mechanics and the basics of quantum mechanics. We have used these to quantize single-mode electromagnetic fields and proved some theorems about it: e.g., the fact that the total energy in the field is discrete, or the relation between the number of photons and the total energy in the field. We are currently working on the implementation of the coherent light definitions and related theorems. As a future work, we plan to formalize multi-mode electromagnetic field quantization, light/atom interaction, light states, and other theories that are required in order to target tangible applications. In the next section we give an example of such an application.

3 Application: Verification of a Quantum Bit Implementation

One of the most promising applications of quantum optics is quantum computers. In this section, we propose to apply our framework to this field and, more specifically, to the implementation of quantum bits. We hope that using formal verification in this field can help the development of the quantum computers in the industry, since it is a cheap, but accurate, verification tool compared to other techniques.

The first model of quantum computer was proposed in 1985 by Deutsch [1]. The essential advantage of quantum computers is that they can run exponentially faster than ordinary ones [9]. In a way similar to ordinary computers which are based on bits, quantum computers are based on quantum bits, called *qubits*. Then, similarly, operations between qubits are achieved by *quantum gates*. There are different implementations of quantum computers, e.g., [11], [8] and [21]. Their main differences lie in the implementation of qubits which can be either photons, electrons or ions. Among these implementations, the ones based on photons and quantum optics seem to be the most promising for a practical use [17].

In quantum mechanics, any system has a collection of quantum states $|\varphi_i\rangle$ called *pure states*. At any time, the system state $|\varphi\rangle$ is formally defined as $|\varphi\rangle = \sum_i c_i|\varphi_i\rangle$ where c_i is a complex number and $\sum_i |c_i|^2 = 1$. For a qubit, the system has only two pure states: $|0\rangle$ and $|1\rangle$ (being the quantum counterparts of the states 0 and 1 for a classical bit). Thus, the qubit state is defined as $\delta|0\rangle + \beta|1\rangle$, where δ and β are complex numbers. More details about the definition of a qubit and its benefits with respect to a classical bit can be found in [9]. Now, we briefly sketch the implementation of qubits using coherent light.

Light is called *coherent* when the number of photons, at any time, is randomly distributed with Poisson probability distribution function (p.d.f). The state of a coherent light is related to the parameter of the p.d.f.: if this parameter is $|\alpha|^2$ for some real α, then the coherent light is at quantum state $|\alpha\rangle$. In [17], a qubit is implemented as a prepared coherent light with two pure states: $|\alpha\rangle$ and $|-\alpha\rangle$ which represent $|0\rangle$ and $|1\rangle$, respectively. In addition, the implementation of basic quantum gates is also introduced in [17]. For example, the quantum flip gate, which converts $\delta|0\rangle + \beta|1\rangle$ into $\beta|0\rangle + \delta|1\rangle$, is implemented as an optical phase shifter. We plan, after finishing the formalization of coherent light, to formally verify

that:

- $|\alpha\rangle$ and $|-\alpha\rangle$ are orthonormal (remember that quantum states are vectors in a Hilbert space, so there is a notion of orthonormality between them);
- for all δ and β, an optical phase shifter converts $\delta|\alpha\rangle+\beta|-\alpha\rangle$ into $\beta|\alpha\rangle+\delta|-\alpha\rangle$.

Notice that the latter property cannot be verified by simulation since it would require to simulate all the possible values of δ and β, which is impossible since there are infinitely many of them. In addition, the ability of our proposed framework to mathematically express all the system specifications makes it advantageous over computer algebra systems.

4 Conclusion

We introduced a new alternative for the verification of quantum optics systems which covers the flaws of laboratories simulation and computer algebra systems, since it is computer-based and able to express mathematically all the bases of quantum mechanics. Libraries for quantum mechanics and single-mode field quantization were implemented in HOL Light. We are currently working on the formalization of coherent light to formally verify the implementation of qubits. In the long run, we plan to develop a full, generic, library that could be easily applied to the verification of more complex optical systems.

References

[1] D. Deutsch (1985): *Quantum Theory, the Church-Turing Principle and the Universal Quantum Computer*. *Proceedings of the Royal Society* 400(1818), pp. 97–117, doi:10.1098/rspa.1985.0070.

[2] P. A. M. Dirac (1925): *The Fundamental Equations of Quantum Mechanics*. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences* 109(752), pp. 642–653, doi:10.1098/rspa.1925.0150.

[3] J.M. Feagin (2002): *Quantum Methods with Mathematica*. Springer.

[4] R. Feynman (1982): *Simulating Physics with Computers*. *International Journal of Theoretical Physics* 21, pp. 467–488, doi:10.1007/BF02650179 10.1.1.45.9310.

[5] M. Fox (2006): *Quantum Optics: An Introduction*. Oxford Master Series in Physics, Oxford University Press.

[6] D.J. Griffiths (2005): *Introduction to Quantum Mechanics*. Pearson Prentice Hall.

[7] J. Harrison (1996): *HOL Light: A Tutorial Introduction*. In: *Formal Methods in Computer-Aided Design*, 1166 of LNCS, Springer, pp. 265–269, doi:10.1007/BFb0031814.

[8] H. Hffner, C.F. Roos & R. Blatt (2008): *Quantum Computing with Trapped Ions*. *Physics Reports* 469(4), pp. 155 – 203, doi:10.1016/j.physrep.2008.09.003.

[9] M. Hirvensalo (2004): *Quantum Computing*. Natural Computing Series, Springer.

[10] U. Leonhardt (2003): *Quantum Physics of Simple Optical Instruments*. *Reports on Progress in Physics* 66(7), p. 1207, doi:10.1088/0034-4885/66/7/203.

[11] Y. Li, D. E. Browne, L. C. Kwek, R. Raussendorf & T.-C. Wei (2011): *Thermal States as Universal Resources for Quantum Computation with Always-On Interactions*. *Physical Review Letters* 107, p. 060501, doi:10.1103/PhysRevLett.107.060501.

[12] B. Lounis & M. Orrit (2005): *Single-Photon Sources*. *Reports on Progress in Physics* 68(5), p. 1129, doi:10.1088/0034-4885/68/5/R04.

[13] L. Mandel & E. Wolf (1995): *Optical Coherence and Quantum Optics*. Cambridge University Press.

[14] The Coq development team (2004): *The Coq Proof Assistant Reference Manual*. LogiCal Project. Version 8.0.

[15] T. Nipkow, L.C. Paulson & M. Wenzel (2002): *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer.

[16] S. Owre, J. M. Rushby & N. Shankar (1992): *PVS: A Prototype Verification System*. In: *Automated Deduction*, 607 of LNCS, Springer, pp. 748–752, doi:10.1007/3-540-55602-8_217.

[17] T. C. Ralph, A. Gilchrist, G. J. Milburn, W. J. Munro & S. Glancy (2003): *Quantum Computation with Optical Coherent States*. *Physical Review A* 68, p. 042319, doi:10.1103/PhysRevA.68.042319.

[18] K. Slind & M. Norrish (2008): *A Brief Overview of HOL4*. In: *Theorem Proving in Higher Order Logics*, 5170 of LNCS, Springer, pp. 28–32, doi:10.1007/978-3-540-71067-7_6.

[19] S. M Tan (1999): *A Computational Toolbox for Quantum and Atomic Optics*. *Journal of Optics B: Quantum and Semiclassical Optics* 1(4), p. 424, doi:10.1088/1464-4266/1/4/312.

[20] H. Vahlbruch, M. Mehmet, S. Chelkowski, B. Hage, A. Franzen, N. Lastzka, S. Goßler, K. Danzmann & R. Schnabel (2008): *Observation of Squeezed Light with 10-dB Quantum-Noise Reduction*. *Physical Review Letters* 100, p. 033602, doi:10.1103/PhysRevLett.100.033602.

[21] J. H. Wesenberg, A. Ardavan, G. A. D. Briggs, J. J. L. Morton, R. J. Schoelkopf, D. I. Schuster & K. Mølmer (2009): *Quantum Computing with an Electron Spin Ensemble*. *Physical Review Letters* 103, p. 070502, doi:10.1103/PhysRevLett.103.070502.

Timed Automata Modeling of CCSL Constraints

Jagadish Suryadevara
Mälardalen Real-Time Centre (MRTC)
Mälardalen University, Sweden
jagadish.suryadevara@mdh.se

Ling Yin
Institute of Software Engineering
East China Normal University, Shanghai, China
yinling86@gmail.com

The UML profile MARTE includes CCSL (Clock Constraint Specification Language) for specifying logical (synchronous/asynchronous) and chronometric timing constraints. A reference semantics for CCSL has been defined and transformation techniques proposed e.g. CCSL to Promela. In this paper, we present a transformation of CCSL into timed automata, to enable verification with the UPPAAL model checker. Also, we compare the transformation with related work in Promela.

1 Introduction

Formal semantics and robust transformation techniques are prerequisites for the successful application of formal methods in industrial safety-critical applications. For real-time and embedded systems (RTE), UML (Unified Modeling Language) provides the MARTE (Modeling and Analysis of Real-Time and Embedded systems) Profile[1]. MARTE includes the Clock Constraint Specification Language (CCSL), for specification of logical and physical clock constraints; synchronous, asynchronous, and chronometric time. Also, a reference semantics for a kernel subset of CCSL has been defined [1]. Further, Mallet & Yin have proposed a 'constructive approach' for transforming CCSL into Promela and also proved the correctness by the $checkpoint - bisimulation$ approach[3].

In this paper, we describe a transformation technique for CCSL specifications into timed automata, the input language for the UPPAAL model checker [2]. The approach is similar to the above mentioned transformations for Promela. However, there are differences due to the flexibility and time modeling features in UPPAAL. For example, in addition to intuitive and visual specifications, it is simpler to model coincident instants using *urgent*(u) and *committed*(c) locations in UPPAAL. Also, UPPAAL supports modeling chronometric time using *clock* variables and clock *invariants*.

2 CCSL: Clock Constraint Specification Language

CCSL supports the specification of logical and physical timing constraints; while logical constraints address the functionality and associated causality, physical constraints concern the chronometric time. CCSL constraints are of three kinds: *synchronous*, *asynchronous* and *mixed*. Synchronous constraints rely on *coincidence* relation, denoted by $\equiv$, between the clock instants. For example, `subclocking` specifies that each instant of the subclock must coincide with one instant of the superclock (preserving the order), c `subClock` c' iff $\forall i,j : c_i \prec c_j \ \exists k,l : (c_i \equiv c'_k) \wedge (c_j \equiv c'_l) \wedge (c'_k \prec c'_l)$, c_i being the i^{th} tick of clock c. If the `subClock` is tight, then c ticks iff $c^{'}$ ticks. Asynchronous constraints are based on *precedence* relation, denoted by $\prec$, between instants. For example, c `strictPre` $c^{'}$ specifies that c is strictly faster than $c^{'}$, i.e., c `strictPre` c' iff $\forall i, c_i \prec c'_i$; `a alternatesWith b` is: $\forall k \in \mathbb{N}^*, a[k] \prec b[k] \wedge b[k] \prec a[k+1]$.

[1] Specialization of a subset of UML elements for a given domain.

Submitted to:
FTSCS 2012

3 Checkpoint Transition Systems & CCSL

Formally, a CTS (Checkpoint Transition System) is a tuple $T = \{S, A, \rightarrow, I, clp\}$, where S is a finite set of states, $clp : S \rightarrow \{0,1\}$ such that $s \in S$ with $clp(s) = 1$ is called a *checkpoint*. A is a finite set of actions containing the invisible action τ. For each initial state $i \in I \subseteq S$, $clp(i) = 1$. The set of transitions is specified by $\rightarrow \subseteq S \times L \times S$, where $L = \mathcal{P}(A)$. A label $l \in L$ is a set of actions, representing the simultaneously performed actions during that transition.

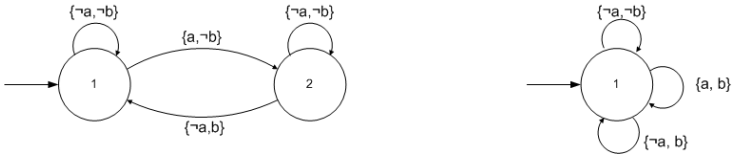


Figure 1: CCSL constraints: (a) `a alternatesWith b` (b) `a subClock b`

A CCSL constraint can be modeled as a CTS where a visible action represents a ticking condition of a clock. An action is denoted as a (clock a ticks) or $\neg a$ (clock a does not tick). Also, the transition relation in a CTS is extended to "checkpoint transitions", denoted by $\Rightarrow$, as defined below where $preclp(s)$ and $poclp(s)$ denote the set of immediate pre- and post-checkpoints of the state s respectively.

$$\begin{aligned}(s, \mu, s^{'}) \in \Rightarrow iff\ \exists\ s, ..., s^{'}, s = q_0, s^{'} = q_n,\ q_i \xrightarrow{l_i} q_{i+1} (0 \leq i < n),\\ where \quad s \in preclp(s^{'}), s^{'} \in poclp(s), and\ \mu = \bigcup_0^{n-1} l_i.\end{aligned} \tag{1}$$

The CTSs of CCSL constraints '`a alternatesWith b`' and '`a subClock b`' are shown in Fig. 1(a) and Fig. 1.(b) respectively. A transition in a CTS represents a valid ticking configuration of clocks in the constraint. Also in every state, the transition with invisible action τ (not shown) corresponds to configuration where none of the clocks of the constraint ticks. If no clock ticks, the constraint is however *not* violated. Also, every state is marked as a checkpoint. The CTS formalism gives a state-based semantics to CCSL and is the basis for transformations from CCSL to timed automata (in this paper) and Promela [3]. Correctness of transformations are proved through checkpoint bisimulation: Let $T_i = \{S_i, \mathcal{A}, \Rightarrow_i, I_i, clp_i\}, (i \in \{1,2\})$, a checkpoint bisimulation is a binary relation $\mathcal{R} \subseteq S_1 \times S_2$ such that

1. $\forall s_1 \in I_1, \exists s_2 \in I_2, (s_1, s_2) \in \mathcal{R}$ and $\forall s_2 \in I_2, \exists s_1 \in I_1, (s_2, s_1) \in \mathcal{R}$
2. for any $(s_1, s_2) \in \mathcal{R}$ it holds that
   (a) for all $s_1 \stackrel{\mu_1}{\Longrightarrow} s_1^{'}$, there exists $s_2 \stackrel{\mu_2}{\Longrightarrow} s_2^{'}$, where $\mu_1 - \{\tau\} = \mu_2 - \{\tau\}$.
   (b) $(s_1^{'}, s_2^{'}) \in \mathcal{R}$

   and vice versa.

T_1 and T_2 are checkpoint bisimulation equivalent if there exists a checkpoint bisimulation R between S_1 and S_2. Further details of CTS & CCSL can be found in [3].

4 Timed Automata Specification of CCSL

A timed automaton (TA) is a tuple $< L, l_0, C, A, E, I >$, where L is a set of *locations*, $l_0 \in L$ is the initial location, C is the set of clocks, A is the set of actions, co-actions and the internal τ-action, $E \subseteq L \times A \times B(C) \times 2^C \times L$ is a set of edges between locations with an action, a guard, a set of clocks to be

reset, and $I : L \to B(C)$ assigns clock *invariants* to locations. Graphically, a timed automaton is a directed graph, e.g. Fig. 2.(a), where nodes represent locations. The initial location is denoted by a special node with two concentric circles. Further details of timed automata can be found in [2].

Logical clocks, instants, clock ordering and non-determinism. In Fig. 2 (a), we present the TA-modeling of a logical clock. It uses attributes `must_tick` (`m`) and `cannot_tick` (`c`), initialized to *false*, to denote if the clock 'ticks' or 'not tick' in the current instance. The clock inconsistency i.e. both the attribute values above are *true* leads to the *location* 'ClkErr'. Also, if both are `false`, a non-deterministic choice of 'tick' or 'not-tick' is made globally. The clock tick itself is denoted by the boolean variable '*t*'. Further, the clock *instants* occur in a global non-deterministic *firable* phase, also a timed automaton shown in Fig. 2 (b). It uses the synchronization channels `fire` and `step` and additional channels e.g. `fire_a` and `fire_b` for clock ordering, as explained later.

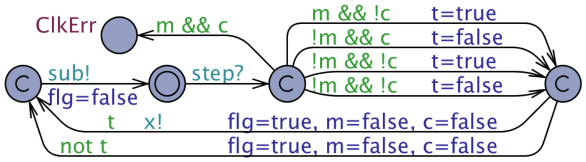
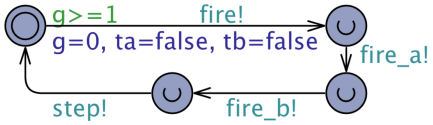


Figure 2: (a) A logical clock TA.



(b) Instants and clock ordering TA.

For each CCSL constraint, based on its CTS representation, we model a timed automaton. It specifies the update actions for attributes `must_tick` and `cannot_tick` for the logical clocks of the constraint. Further, asynchronous constraints are state-based, hence the state is also encoded and updated locally after each global *fireable* phase. However, synchronous constraints are state-less but depend on clock ordering as explained below.

Modeling asynchronous constraints. In Fig. 3(a), we present a TA-model of the asynchronous constraint '`a alternatesWith b`' (based on the CTS in Fig.1 (a)). Locations **Start** to **End** represent an *enable* phase when the attributes '`cannot_tick`' (`c`) are updated i.e. '*ca*' and '*cb*' for logical clocks 'a' and 'b'. The above update action is forced due to *committed* nature of **Start**. Following this, the global fireable phase starts (note the delay-location 'Fire' allowing time to progress at the initial location in the global TA (Fig. 2)). The locations **End** to **Start** represent an *update* phase that updates the state variable and is executed after a *firable* phase completes (note the synchronization signal *step*?).

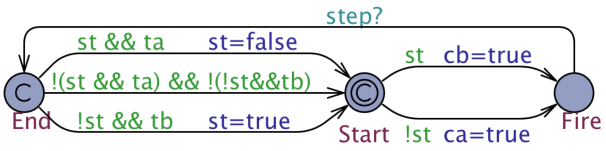
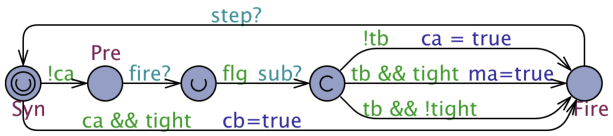


Figure 3: (a) `a alternatesWith b`



(b) `a subClock b`

Modeling synchronous constraints. In Fig. 3(b), we present a TA-model of the synchronous constraint '`a subClock b`' (based on the CTS in Fig.1 (b)). It is state-less and depends on the clock order-

ing; $b < a$ (ticks of 'a' depend on the corresponding ticks of 'b). The locations **Pre** to **Fire** represent a *sub-enable* phase that executes 'within' a global *fireable* phase and coincidently as the super clock 'b' $fires$. The model also specifies a stricter version of causality between 'a' and 'b' using the boolean value 'tight'; if 'a' cannot tick 'b' also cannot tick (ca $\Rightarrow$ cb) and if 'b' ticks 'a' must tick as well (tb $\Rightarrow$ ma).

5 Comparison to Promela transformation

5.1 Overview of Promla transformation

A transformation from CCSL to Promela based on CTS semantics is proposed in [3]. The key idea behind the transformation is implementing the simultaneous clock ticks in one instant in three phases: $start \rightarrow firing \rightarrow end$, in *start* ticking conditions of clocks are computed, in $firing$ each clock fires or not according to its ticking condition, in *end* states are updated to prepare next instant. The Promela transformation of `alternateWith` and `subClock` are shown in Fig. 4, and their CTSs are shown in Figure 5(actions that are not relevant are abstracted as τ). Only *start* and *end* are marked as checkpoints. Comparing the CTSs of `alternateWith` and `subClock` in Fig. 1, the Promela transformation satisfies checkpoint bisimulation.

```
proctype alternateWith (int cL,cR){
    bool state=true;
    do
      :: atomic{start?true;
        if
          ::state->clks[cR].cannot_tick=true
          :: !state->clks[cL].cannot_tick=true
        fi;}
        atomic{end?true;
        if
        ::state->
            if
            :: clks[cL].act_tick->state=false
            :: else->skip
            fi
        ::!state->
            if
            :: clks[cR].act_tick->state=true
            :: else->skip
            fi
        fi}
    od}
```

```
proctype subClock (int cL,cR; bool istight){
byte index;
do
:: start?true;
        atomic{subenable?true;
        if
        :: istight->
            if
            :: clks[cL].cannot_tick->clks[cR].cannot_tick=true
            :: else->skip
            fi
        ::else->skip
        fi;}
loopsub:atomic{subfireable?index;
          if
          :: index==cR->sub++;
            if
            :: clks[cR].act_tick->
              if
              ::istight->clks[cL].must_tick=true
              ::else->skip
              fi
            :: else->clks[cL].cannot_tick=ture
            fi
          ::gotosf?true->goto loopsub
          fi;} fireable?true;
od}
```

Figure 4: Promela encodingsof `alternateWith` and `subClock`

5.2 Comparison

The transformations in Uppaal and Promela, as presented in previous sections, use similar check-point bisimulation approach in implementing CTS representations of CCSL constraints. However, some differences exist, as discussed below:

- **Clock instants and clock ordering**. In Uppaal transformation, a global automata (Fig. 2) defines clock instants through global *firable* phases using a *clock* variable 'g'. The actual duration how-

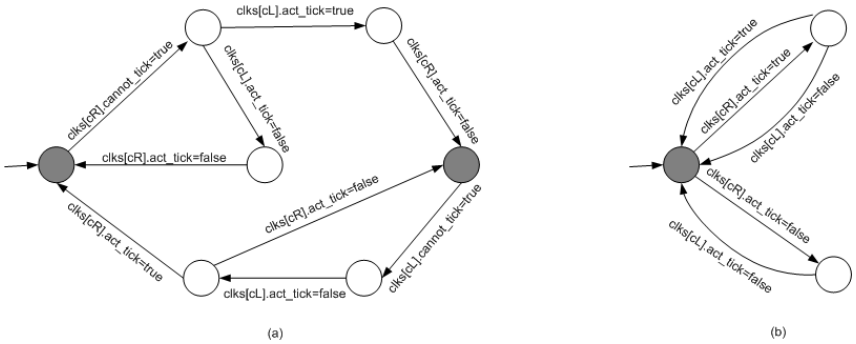


Figure 5: CTSs of Promela encodings: (a) `alternatesWith` (b) `subClock`

ever is indeterminate (the invariant $g >= 1$). The clock ordering is specified using synchronization channels that trigger the logical clocks only in the specified order. While in Promela transformation, there is no global clock if it is not explicitly specified in the CCSL specification. The clock ordering is ensured by indexing the clock declarations and firing them in the specific order.

- **Atomicity and synchrony**. In Uppaal transformations, the atomicity of a sequence of transitions is implemented using *committed* and *urgent* locations. Combined with synchronization channels, this provides a mechanism to implement *coincident* relations between logical clocks. While in Promela, the simultaneous tickings inside one instant are implemented as a sequence of firings between two rendezvous communications (the start and end checkpoints respectively).
- **Chronometric time**. Clock variables in Uppaal can be used to model chronometric time which lacks in Promela implementation. For example, the clock variable 'g' in Fig. 2 can specify a chronometric time duration. Different clock variables may be used to associate different chronometric durations for logical clocks.

6 Conclusions

In this paper, we have presented timed automata transformations for CCSL constraints in UML/MARTE profile. The approach paves the way for formal verification with Uppaal model checker. We have also compared the approach with existing Promela implementation of CCSL. However some significant differences exist, as described in the paper, leading to difference in statespace size and verification aspects. As future work, we plan to investigate both transformations and compare the performance of verification. Also, we will extend the timed automata transformations for CCSL chronometric constraints.

References

[1] Charles André (2009): *Syntax and Semantics of the Clock Constraint Specification Language (CCSL)*. Rapport de recherche RR-6925, INRIA. Available at `http://hal.inria.fr/inria-00384077`.

[2] Gerd Behrmann, Alexandre David & Kim G. Larsen (2004): *A Tutorial on UPPAAL*. In M. Bernardo & F. Corradini, editors: *International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004. Revised Lectures*, *Lecture Notes in Computer Science* 3185, Springer Verlag, pp. 200–237. Available at `http://doc.utwente.nl/51010/`.

[3] Frédéric Mallet & Ling Yin (2012): *Correct Transformation from CCSL to Promela for verification*. Rapport de recherche RR-7491, INRIA. Available at `http://hal.inria.fr/hal-00667849`.

Towards system development methodologies: From software to cyber-physical domain

Maria Spichkova Alarico Campetelli
Institut für Informatik
Technische Universität München
spichkov@in.tum.de campetel@in.tum.de

In many cases, it is more profitable to apply existing methodologies than to develop new ones. This holds, especially, for system development within the cyber-physical domain: up to a certain abstraction level we can (re)use the methodologies for the software system development to benefit from the advantages of these techniques.

1 Introduction

A lot of new ideas are just well-forgotten old ones, and a lot of newly developed methodologies are, in fact, the reinvention of the wheel. In many cases new methods or languages are introduced to deal with cyber-physical systems, where the already existing techniques could be more suitable to represent them, especially after an extension or adaptation to cover some special domain features. Moreover, the existing techniques could provide a connection to the representation of systems from other domains.

Cyber-physical systems are widespread in safety-critical domains, as vehicles, machines, aircrafts or medical instruments. A failure in these systems may lead to considerable loss of money or even endanger human lives. Therefore, it follows the importance of a correct behaviour of the systems, which can be guaranteed through analysis techniques, mostly in form of formal verification (cf. [4]). The support for a formal analysis approach is facilitated by a formal model representation, and a suitable modelling theory for these systems helps in their development, maintenance, simulation, and verification.

The main challenge here is to combine two worlds, the physical and the virtual one: software components operate in discrete program steps, meanwhile the physical components evolves over time intervals following physical constraints. Nevertheless, many physical properties can be represented in a similar manner as the software properties, e.g., in many cases it is possible to switch from the continuous time representation to the digital one without loosing the essential properties of the represented system [12].

From our experience within a number of industrial projects, speaking about the system architecture and properties on a certain abstraction level, ones do not need to distinguish physical signals and component from the software ones. In fact this difference may not have any advantages and, moreover, could make the system description too complicated and hardly readable. Thus, until we are speaking about logical level, we can benefit from using the software system development processes. For this reason, we present a system development methodology, which is a generalization of two methodologies successfully evaluated on three case studies from the automotive domain, with a suggestion to apply its general ideas for the development of cyber-physical systems.

Submitted to:
FTSCS 2012

2 System Development Methodology

One of the typical examples of cyber-physical system from the automotive field is a Cruise Control System (CCS). We modeled two different variants of the system using two development methodologies with similar strategies but different focal points. The first case study [7], developing an Adaptive Cruise Control (ACC) system with Pre-Crash Safety functionality, was motivated and supported by DENSO Corporation; while the second case study [19, 18], developing a CCS with focus on system architecture and verification, was supported by Robert Bosch GmbH. Another case study [8] motivated and supported by DENSO Corporation was the development of the Keyless Entry-System, which was not only a comfort system with the distributed deployment, but also a system having a huge state space. We did not distinguish between physical and virtual parts of the system focusing on the logical representation here. Thus, a sample-property of a Cruise Control System can be represented as follows: *If the driver pushes the ACC-button while the system is On and none of the switch-off constraints occurs, the system must accelerate the vehicle during the next time unit respectively to the predefined acceleration schema.* This means that the system must analyze the information from sensors to check whether any switch-off constraint occurs, e.g., if the battery voltage is too low or if the gas pedal sensor fails.

Fig. 1 illustrates the structure of the generalized development methodology in a top-down manner. However, by developing a large system a number of iterations is needed to cope with gain of knowledge about the system, as well as modifications in the requirements. The boxes represent here the artifacts that have been developed and the arrows show from which other artifacts they were derived. The process starts by structuring of initial requirements in the way they follow specific syntactic patterns: this first step raises the level of precision by transforming the free text requirements into a structured form using specific pre-defined syntactic patterns, as presented in [9]. An informal specification consists of a set of words, which can be distinguished into two categories: content words and keywords. Content words are system-specific words or phrases, e.g., *"Off-button is pressed"*. The set of all content words forms the *logical interface* of the system, which can be understood as some kind of domain specific glossary that must be defined in addition. Keywords are domain-independent and form relationships between the content words (e.g., "if", "then"). Thus, a semiformal specification consists of a number of requirements described via textual pattern, which is easily understood also by engineers unfamiliar with the formal methods. Using this description to structure the informal specification, we can find out missing information quite fast. Furthermore, we identify possible synonyms that must be unified before proceeding to a formal specification. This specification can be schematically rewritten to a *Message Sequence Charts* (MSCs) [11] representation, as an optional step relevant for highly interacting systems.

The methodology proceeds by the translation of semiformal specification to FOCUS [3], a framework for formal specifications and development of distributed interactive systems, preferred here over other specification frameworks. Because it has an integrated notion of time and provides a number of specification techniques for distributed systems and concepts of refinement, as well as graphical notation, which is extremely important when we are dealing with systems of industrial size. We represent in FOCUS two kinds of specifications: system requirements and architecture. This prepares the basis to verify the system architecture against the requirements by translating both to the theorem prover Isabelle/HOL [16] via the framework "FOCUS on Isabelle" [17]. As a next step, we translate the architecture specification to a representation in the tool AutoFOCUS [1] to simulate the system. The requirements specification can be schematically translated to temporal logic or specification patterns, which gives basis to model-check the model (cf. [5]). The integration of model checking in AutoFOCUS approaches usability with

[1] The AUTOFOCUS homepage: `http://af3.in.tum.de`

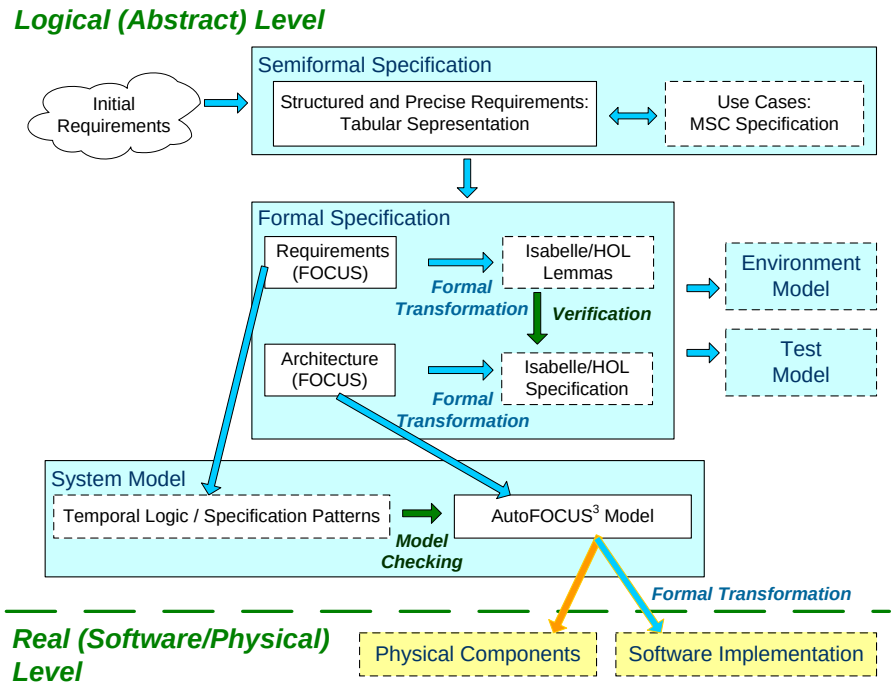


Figure 1: Generalized Development Methodology

the following points: tight coupling of verification properties with model elements, visualization and simulation of counterexamples, and different specification languages for the formulation of properties. Dealing with these issues leads to one of the first model-based development environments incorporating property specification, model checking and debugging.

In the ACC case study [7] we apply model checking technique for temporal logic formulas, ranging between SAT solving and interactive verification with regard to its notational power and complexity. The verification was supplied by the SMV tool [15] and the properties are specified in LTL (Linear Temporal Logic, e.g. [14]), a widespread specification notation suitable for automatic model checking. In this formalism are supported boolean logical operators and temporal operators, further it allows for formalising properties of system states and their changes during system execution. We used SMV to check requirements in form of temporal logic formulas also in the Keyless Entry-System case study [8]. When necessary we abstracted variables/data types and/or restrict value ranges, especially for integer variables.

AutoFocus models can also be exported to Isabelle/HOL to prove its properties – it is in general a *refinement* of a Focus specification, thus its properties can be slightly different, i.e., more strict, from the ones specified on the Focus layer, but the proof schema, which has been developed for the Focus specifications, can be (partially) reused. Optionally, we can also represent an environment or a test model if this benefit the analysis of the concrete system. In the Cruise Control System case study [19, 18], we applied a semiautomatic verification with a theorem prover and fully automatic by a model checker. Formal requirements can be checked using a theorem prover, through a translator, which generates Isabelle/HOL theories from AutoFocus models. To generate the Isabelle/HOL code from the models, at first the user initiates code export for the data dictionary, which generates a theory containing data type declarations and function definitions used in the model. The internal behaviour

of components in the models may be defined by an automaton or a function specification. So, from each component in the models may be generated a theory containing the input and output interface definition and the transition function originating from the automaton or function specification. If the component is composite recursively the theories for all subcomponents will be generated and, ultimately, the theory for the considered component. Generating code for the root component of an AutoFOCUS model produces a set of theories encoding this model in Isabelle/HOL, as well as proof of theorems that support subsequent verification of properties in Isabelle/HOL. This twofold verification guaranteed, firstly that are verified actual properties of the design model and the implementation code; secondly the results of both techniques indicate an implementation fault in one of the generators.

Finally, we can switch from the logical to the real level, where we have to split our model into software and physical components. Transformation to a corresponding C code can be done using the corresponding code generator: we have shown that the C program produced by our code generator is an admissible simulation of the model. Altogether, the methodology guides us from an informal specification via stepwise refinement to a verified formal specification, a corresponding executable verified model, and also a corresponding verified C code implementation.

3 Related Work

There are many approaches on mechatronic/cyber-physical systems, however, most of them do not focus on the logical level of the system representation and loose the advantages of the abstract respresentation: a better overview, possibility to validate the system in the earlier phases, etc. There is a number of interesting attempts to define a modelling solution for the cyber-physical systems, for instance, the work presented in [20] defines an extensive support to the components communication and time requirements, while the model discussed in [10] proposes a complete model of the processes with communication. Nevertheless, in our opinion one limitation of such approaches is that the system is represented with a flat view, that is, there is only a single abstraction level to represent it. That could be a disadvantage in the project of a cyber-physical system, where experts of different domains should be able to cooperate and work in different views and abstraction levels of the system. Modeling theories for distributed hybrid systems such as SHIFT [6] and R-Charon [13], guarantee a complete simulation and compilation of the models, but they have no comprehensive verification support. The same limitation is for UPPAAL [2] and PHAVer [1], which provide the simulation, but a limited verification with restricted dynamics and only for small fragments.

4 Conclusion

In this paper we have suggested to reuse the generalization of two existing methodologies for the development of software systems to apply them within the cyber-physical domain, according to the results of three case studies motivated and supported by DENSO Corporation and Robert Bosch GmbH. Up to a certain abstraction level we can use the existing methodologies for the development of software systems also within the cyber-physical domain to benefit from the advantages these techniques have shown, as well as extend the modelling artefacts for the domain features. As for instance, introduce at some lower abstraction level continuous dynamic modelling capabilities and so its verification, in our existing tool and theory. The simulation of a full continuous dynamic has still a not optimal performance, so we are oriented for a discretization of the simulation, through the implementation of dynamical adapting sampling.

References

[1] D. A. Van Beek, K. L. Man, M. A. Reniers, J. E. Rooda & R. R. H. Schiffelers (2006): *Syntax and consistent equation semantics of hybrid Chi*. In: *Journal of Logic and Algebraic Programming*, pp. 129–210.

[2] G. Behrmann, A. David & K. G. Larsen (2004): *A Tutorial on Uppaal*. In: *SFM*, pp. 200–236.

[3] M. Broy & K. Stølen (2001): *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement*. Springer.

[4] A. Campetelli (2010): *Analysis Techniques: State of the Art in Industry and Research*. Tech. Rep., TU München.

[5] A. Campetelli, F. Hölzl & P. Neubeck (2011): *User-friendly Model Checking Integration in Model-based Development*. In: *The 24th International Conference on Computer Applications in Industry and Engineering*.

[6] Akash Deshpande, Aleks Gll & Pravin Varaiya (1996): *SHIFT: A Formalism and a Programming Language for Dynamic Networks of Hybrid Automata*. In Panos J. Antsaklis, Wolf Kohn, Anil Nerode & Shankar Sastry, editors: *Hybrid Systems IV*, *Lecture Notes in Computer Science* 1273, Springer, pp. 113–133.

[7] M. Feilkas, A. Fleischmann, F. Hölzl, C. Pfaller, S. Rittmann, K. Scheidemann, M. Spichkova & D. Trachtenherz (2009): *A Top-Down Methodology for the Development of Automotive Software*. Technical Report TUM-I0902, TU München.

[8] M. Feilkas, F. Hölzl, C. Pfaller, S. Rittmann, B. Schätz, W. Schwitzer, W. Sitou, M. Spichkova & D. Trachtenherz (2011): *A Refined Top-Down Methodology for the Development of Automotive Software Systems - The KeylessEntry System Case Study*. Technical Report TUM-I1103, TU München.

[9] A. Fleischmann (2008): *Model-based formalization of requirements of embedded automotive systens*. Ph.D. thesis, TU München.

[10] T. Hadlich, C. Diedrich, K. Eckert, T. Frank, A. Fay & B. Vogel-Heuser (2011): *Common communication model for distributed automation systems*. In: *9th IEEE International Conference on Industrial Informatics*, IEEE INDIN.

[11] D. Harel & P. S. Thiagarajan (2003): *Message Sequence Charts*. In L. Lavagno, G. Martin & B. Selic, editors: *UML for Real: Design of Embedded Real-Time Systems*, Kluwer Academic Publishers, pp. 77–105.

[12] T.A. Henzinger, Z. Manna & A. Pnueli (1992): *What Good Are Digital Clocks?* In: *Proceedings of the 19th International Colloquium on Automata, Languages and Programming*, Springer, pp. 545–558.

[13] F. Kratz, O. Sokolsky, G. J. Pappas & I. Lee (2006): *R-Charon, a Modeling Language for Reconfigurable Hybrid Systems*. In: *HSCC*, pp. 392–406.

[14] Zohar Manna & Amir Pnueli (1992): *The temporal logic of reactive and concurrent systems*. Springer-Verlag New York, Inc., New York, NY, USA.

[15] K. L. McMillan (1999): *Getting started with SMV*. Technical Report, Cadence Berkeley Labs.

[16] T. Nipkow, L. C. Paulson & M. Wenzel (2002): *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*. *LNCS* 2283, Springer.

[17] M. Spichkova (2007): *Specification and Seamless Verification of Embedded Real-Time Systems: FOCUS on Isabelle*. Ph.D. thesis, TU München.

[18] M. Spichkova (2011): *Architecture: Requirements+Decomposition+Refinement*. *Softwaretechnik-Trends 31:4*.

[19] M. Spichkova, F. Hölzl & D. Trachtenherz (2012): *Verified System Development with the AutoFocus Tool Chain*. In: *2nd Workshop on Formal Methods in the Development of Software*, WS-FMDS.

[20] B. Vogel-Heuser, Feldmann S., T. Werner & C. Diedrich (2011): *Modeling network architecture and time behavior of Distributed Control Systems in industrial plant*. In: *37th Annual Conference of the IEEE Industrial Electronics Society*, IECON.