Mariëlle Stoelinga
Ralf Pinger (Eds.)

LNCS 7437

Formal Methods for Industrial Critical Systems

17th International Workshop, FMICS 2012
Paris, France, August 2012
Proceedings

Springer

Mariëlle Stoelinga   Ralf Pinger (Eds.)

Formal Methods for Industrial Critical Systems

17th International Workshop, FMICS 2012
Paris, France, August 27-28, 2012
Proceedings

Springer

Volume Editors

Mariëlle Stoelinga
University of Twente, Department of Computer Science
Formal Methods and Tools
P.O. Box 217, 7500 AE Enschede, The Netherlands
E-mail: marielle@cs.utwente.nl

Ralf Pinger
Siemens AG, Infrastructure and Cities Sector
Mobility and Logistics Division, Rail Automation
Ackerstraße 22, 38126 Braunschweig, Germany
E-mail: ralf.pinger@siemens.com

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-32468-0 e-ISBN 978-3-642-32469-7
DOI 10.1007/978-3-642-32469-7
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2012943602

CR Subject Classification (1998): D.2.4, F.3.1, D.2, C.3, J.1, J.2, F.1.1

LNCS Sublibrary: SL 2 – Programming and Software Engineering

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

This volume contains the papers presented at FMICS 2012, the 17th International Workshop on Formal Methods for Industrial Critical Systems, taking place August 27–28, 2012, in Paris, France. Previous workshops of the ERCIM Working Group on Formal Methods for Industrial Critical Systems were held in Oxford (March 1996), Cesena (July 1997), Amsterdam (May 1998), Trento (July 1999), Berlin (April 2000), Paris (July 2001), Malaga (July 2002), Trondheim (June 2003), Linz (September 2004), Lisbon (September 2005), Bonn (August 2006), Berlin (July 2007), L'Aquila (September 2008), Eindhoven (November 2009), Antwerp (September 2010), and Trento (August 2011). The FMICS 2012 workshop was co-located with the 18th International Symposium on Formal Methods (FM 2012).

The aim of the FMICS workshop series is to provide a forum for researchers who are interested in the development and application of formal methods in industry. In particular, FMICS brings together scientists and engineers that are active in the area of formal methods and interested in exchanging their experiences in the industrial usage of these methods. The FMICS workshop series also strives to promote research and development for the improvement of formal methods and tools for industrial applications.

The topics of interest include, but are not limited to:

- Design, specification, code generation and testing based on formal methods
- Methods, techniques and tools to support automated analysis, certification, debugging, learning, optimization and transformation of complex, distributed, dependable, real-time systems and embedded systems
- Verification and validation methods that address shortcomings of existing methods with respect to their industrial applicability, e.g., scalability and usability issues
- Tools for the development of formal design descriptions
- Case studies and experience reports on industrial applications of formal methods, focusing on lessons learned or identification of new research directions
- Impact of the adoption of formal methods on the development process and associated costs
- Application of formal methods in standardization and industrial forums

This year we received 37 submissions. Papers had to pass a rigorous review process in which each paper received three reports. The international Program Committee of FMICS 2012 decided to select 14 papers for presentation during the workshop and inclusion in these proceedings. The workshop was highly enriched by our two invited talks given by Dimitra Giannakopoulou, NASA Ames, USA, and Hubert Garavel, INRIA Grenoble Rhone-Alpes, France.

We would like to thank the local organizers Kamel Barkaoui, CNAM Paris, and Béatrice Bérard, University Pierre et Marie Curie, for taking care of all the local arrangements in Paris, the ERCIM FMICS working group coordinator Radu Mateescu, INRIA Grenoble, for his fruitful discussions, and especially Alessandro Fantechi, Università degli Studi di Firenze and ISTI-CNR, Italy, for inviting us to co-chair this workshop, EasyChair for supporting the review process, Springer for the publication, all Program Committee members and external reviewers for their substantial reviews and discussions, all authors for submitting 37 papers and all attendees of the workshop. Thanks to all for making FMICS 2012 a success.

August 2012

Mariëlle Stoelinga
Ralf Pinger

Organization

Program Committee

Lubos Brim	Masaryk University, Czech Republic
Alessandro Cimatti	FBK-irst, Italy
Maria Del Mar Gallardo	University of Malaga, Spain
Michael Dierkes	Rockwell Collins, France
Cindy Eisner	IBM Haifa, Israel
Georgios Fainekos	Arizona State University, USA
Alessandro Fantechi	DSI - Università di Firenze, Italy
Holger Hermanns	Saarland University, Germany
Michaela Huhn	Technische Universität Clausthal, Institut für Informatik, Germany
Franjo Ivancic	NEC Laboratories America, Inc., USA
Joost-Pieter Katoen	RWTH Aachen, Germany
Stefan Kowalewski	RWTH Aachen University, Germany
Juliana Küster Filipe Bowles	University of St. Andrews, UK
Frederic Lang	INRIA Rhône-Alpes / VASY, France
Odile Laurent	Airbus, France
Stefan Leue	University of Konstanz, Germany
Tiziana Margaria	University of Potsdam, Germany
Mieke Massink	CNR-ISTI, Italy
David Parker	University of Oxford, UK
Corina Pasareanu	CMU/NASA Ames Research Center, USA
Thomas Peikenkamp	OFFIS e.V., Germany
Jan Peleska	TZI, Universität Bremen, Germany
Ralf Pinger	Siemens AG, Braunschweig, Germany
Jakob Rehof	TU Dortmund, Germany
Judi Romijn	Movares, The Netherlands
John Rushby	SRI International, USA
Gwen Salaün	Grenoble INP - INRIA - LIG, France
Bernhard Schätz	TU München, Germany
Marjan Sirjani	Reykjavik University, Reykjavik, Iceland
Mariëlle Stoelinga	University of Twente, The Netherlands

Additional Reviewers

Acharya, Mithun
Barnat, Jiri
Beer, Adrian
Belinfante, Axel
Biallas, Sebastian
Blech, Jan Olaf

Bracciali, Andrea
Bushnell, David
Ceska, Milan
Düdder, Boris
Edmunds, Andrew
Eggers, Andreas
Gay, Gregory
Gdemann, Matthias
Genov, Blagoy
Gorbachuk, Elena
Hartmanns, Arnd
Hayden, Richard
Hugues, Jerome
Hölzl, Florian
Jafari, Ali
Khamespanah, Ehsan
Khosravi, Ramtin
Kratochvila, Tomas
Lapschies, Florian
Leitner-Fischer, Florian
Martens, Moritz
Merino, Pedro
Nguyen, Viet Yen
Noll, Thomas
Ouederni, Meriem
Shafiei, Nastaran
Sieverding, Sven
Tabaei Befrouei, Mitra
Ter Beek, Maurice H.
Teufl, Sabine
Yue, Haidi

Three Decades of Success Stories in Formal Methods

Hubert Garavel[⋆]
with contributions of Susanne Graf

INRIA/LIG – CONVECS team
655 avenue de l'Europe, 38330 Montbonnot St Martin, France
`hubert.garavel@inria.fr`
`http://convecs.inria.fr/people/Hubert.Garavel`

Abstract. This talk presents a selection of successful applications of formal methods to real-life problems. Similar studies already appeared in the scientific literature but are not, we believe, entirely satisfactory. On the one hand, in the cumulative list of applications considered by these studies, certain formal methods are over-represented while others are not mentioned. On the other hand, the essential role of verification tools is not always acknowledged as strongly as it should be.

To ensure a broader coverage of the diversity of formal methods, we selected a set of thirty case-studies, while prior studies often limited themselves to a dozen. These case-studies are distributed regularly over the past three decades, one per year between 1982 and 2011.

We tried to give a balanced panorama of formal methods by featuring different formal approaches (mathematical notations, theorem proving, model checking, static analysis, etc.), different models of computations (sequential, synchronous, asynchronous, timed, probabilistic, hybrid, etc.), and different application domains (hardware, software, telecommunication, embedded systems, operating systems, compilers, etc.).

In our selection, we focused on practical applications of formal methods rather than theoretical results alone. Contrary to some other studies, we gave priority to repeatable experiments, privileging approaches supported by software tools rather than "heroic" approaches relying on pen-and-paper manipulation of mathematical symbols.

Obviously, exhaustivity is impossible as the number and diversity of applications of formal methods cannot be reduced to a collection of thirty samples. Also, we do not claim that our selection represents the "best" case studies ever published, but simply that they correspond to pioneering and inspiring work that the young generation should keep in mind.

[⋆] This study is part of a formal methods survey that has been funded by the German Federal Agency BSI (*Bundesamt für Sicherheit in der Informationstechnik*) under project 875 initiated and led by Dr. Anastasia-Maria Leventi-Peetz.

To Scale or Not to Scale: Experience with Formal Methods and NASA Systems

Dimitra Giannakopoulou

NASA Ames Research Center, USA
`dimitra.giannakopoulou@nasa.gov`

Abstract. The safety-critical nature of aerospace systems mandates the development of advanced formal verification techniques that provide desired correctness guarantees. In this talk, we will discuss our experience with the development and use of formal method techniques in the context of aerospace systems. We will first provide an overview of approaches that we have developed over the last decade for scaling exhaustive verification through divide-and-conquer principles. In particular, we will present learning-based frameworks for automatically generating component abstractions. Such abstractions can be used for documentation, or more efficient modular reasoning. In the domain of human-automation interaction systems, these abstractions can be used for human operators to understand what to expect from their interactions with the system.

The techniques that will be presented use a variety of approaches, including model checking, predicate abstraction, and symbolic execution. Despite the progress that we have made in developing and applying sophisticated formal methods frameworks, the issue of scalability still remains the Achilles tendon in this domain. We will discuss scalability and the trade-offs that we have made in our work, as well as our perspective for the future application of formal methods in industry.

Table of Contents

Real-Time Specification Patterns and Tools[⋆]

Nouha Abid[1,2], Silvano Dal Zilio[1,2], and Didier Le Botlan[1,2]

[1] CNRS, LAAS, 7 avenue du colonel Roche, F-31400 Toulouse France
[2] Univ de Toulouse, LAAS, F-31400 Toulouse, France

Abstract. An issue limiting the adoption of model checking technologies by the industry is the ability, for non-experts, to express their requirements using the property languages supported by verification tools. This has motivated the definition of dedicated assertion languages for expressing temporal properties at a higher level. However, only a limited number of these formalisms support the definition of timing constraints. In this paper, we propose a set of specification patterns that can be used to express real-time requirements commonly found in the design of reactive systems. We also provide an integrated model checking tool chain for the verification of timed requirements on TTS, an extension of Time Petri Nets with data variables and priorities.

1 Introduction

An issue limiting the adoption of model checking technologies by the industry is the difficulty, for non-experts, to express their requirements using the specification languages supported by the verification tools. Indeed, there is often a significant gap between the boilerplates used in requirements statements and the low-level formalisms used by model checking tools; the latter usually relying on temporal logic. This limitation has motivated the definition of dedicated assertion languages for expressing properties at a higher level (see Section 5). However, only a limited number of assertion languages support the definition of timing constraints and even fewer are associated to an automatic verification tool, such as a model checker.

In this paper, we propose a set of real-time specification patterns aimed at the verification of reactive systems with hard real-time constraints. Our main objective is to propose an alternative to timed extensions of temporal logic during model checking. Our patterns are designed to express general timing constraints commonly found in the analysis of real-time systems (such as compliance to deadlines; event duration; bounds on the worst-case traversal time; etc.). They are also designed to be simple in terms of both clarity and computational complexity. In particular, each pattern should correspond to a decidable model checking problem.

Our patterns can be viewed as a real-time extension of Dwyer's specification patterns [11]. In his seminal work, Dwyer shows through a study of 500 specification examples that 80% of the temporal requirements can be covered by a

[⋆] This work was partially supported by the JU Artemisia project CESAR and the FNRAE project Quarteft.

small number of "pattern formulas". We follow a similar philosophy and define a list of patterns that takes into account timing constraints. At the syntactic level, this is mostly obtained by extending Dwyer's patterns with two kind of *timing modifiers*: (1) P within I, which states that the delay between two events declared in the pattern P must fit in the time interval I; and (2) P lasting D, which states that a given condition in P must hold for at least duration D. For example, we define a pattern Present A after B within $]0, 4]$ to express that the event A must occur within 4 units of time (u.t.) of the first occurrence of event B, if any, and not simultaneously with it. Although seemingly innocuous, the addition of these two modifiers has a great impact on the semantics of patterns and on the verification techniques that are involved.

Our second contribution is an integrated model checking tool chain that can be used to check timed requirements. We provide a compiler for Fiacre [6], a formal modelling language for real-time systems, that we extended to support the declaration of real-time patterns. In our tool chain, Fiacre is used to express the model of the system while verification activities ultimately relies on Tina [3], the TIme Petri Net Analyzer. This tool chain provides a reference implementation for our patterns when the systems can be modeled using an extension of Time Petri Nets with data variables and priorities that we call a TTS (see Sect. 2.1). This is not a toy example; Fiacre is the intermediate language used for model verification in Topcased [13], an Eclipse-based toolkit for system engineering, where it is used as the target of model transformation engines for various high-level modelling languages, such as SDL or AADL [4]. In each of these transformations, we have been able to use our specification patterns as an intermediate format between high-level requirements (expressed on the high-level models) and the low-level input languages supported by the model checkers in Tina.

The rest of the paper is organized as follows. In the next section, we define technical notations necessary to define the semantics of patterns. Section 3 gives our catalog of real-time patterns. For each pattern, we give a simple definition in natural language as well as an unambiguous, formal definition based on two different approaches. Before concluding, we review the results of experiments that have been performed using our verification tool chain in Sect. 4.

2 Technical Background

Since patterns are used to express timing and behavioral constraints on the execution of a system, we base the semantics of patterns on the notion of *timed traces*, which are sequences mixing events and time delays, (see Def. 1 below). We use a dense time model, meaning that we consider rational time delays and work both with strict and non-strict time bounds.

The semantics of a pattern will be expressed as the set of all timed traces where the pattern holds. We use two different approaches to define set of traces: (1) Time Transition Systems (TTS), whose semantics relies on timed traces; and (2) a timed extensions of Linear Temporal Logic, called MTL. In our verification tool chain, both Fiacre and specification patterns are compiled into TTS.

2.1 Time Transition Systems and Timed Traces

Time Transition Systems (TTS) are a generalization of Time Petri Nets [17] with priorities and data variables. We describe the semantics of TTS using a simple example. Figure 1 gives a model for a simple airlock system consisting of two doors (D_1 and D_2) and two buttons. At any time, at most one door can be open. This constraint is modeled by the fact that at most one of the places $\mathsf{D_1isOpen}$ and $\mathsf{D_2isOpen}$ may have a token. Additionally, an open door is automatically closed after exactly 4 units of time (u.t.), followed by a ventilation procedure that lasts 6 u.t. This behavior is modeled by adding timing constraints on the transitions $\mathsf{Open_1}$, $\mathsf{Open_2}$ and Ventil. Moreover, requests to open the door D_2 have higher priority than requests to open D_1. This is modeled using a priority (dashed arrow) from the transition $\mathsf{Open_2}$ to $\mathsf{Open_1}$. A shutdown command can be triggered if no request is pending.

To understand the model, the reader may, at first, ignore side conditions and side effects (the pre and act expressions inside dotted rectangles). In this case, a TTS is a standard Time Petri Net, where circles are places and rectangles are transitions. A transition is enabled if there are enough tokens in its input places. A time interval, such as $I = [d_1; d_2[$, indicates that the corresponding transition must be fired if it has been enabled for d units of time with $d \in I$. As a consequence, a transition associated to the time interval $[0; 0]$ must fire as soon as it is enabled. Our model includes two boolean variables, $\mathsf{req_1}$ and $\mathsf{req_2}$, indicating whether a request to open door D_1 (resp. D_2) is pending. Those variables are read by pre-conditions on transitions Open_i, Button_i, and $\mathsf{Shutdown}$ and are modified by post-actions on transitions Button_i and Close_i. For instance, the pre-condition $\neg\mathsf{req_2}$ on $\mathsf{Button_2}$ is used to disable the transition when the door is already open. This implies that pressing the button while the door is open has no further effect.

We introduce some basic notations used in the remainder of the paper. (A complete, formal description of the TTS semantics can be found in [2].) Like with Petri

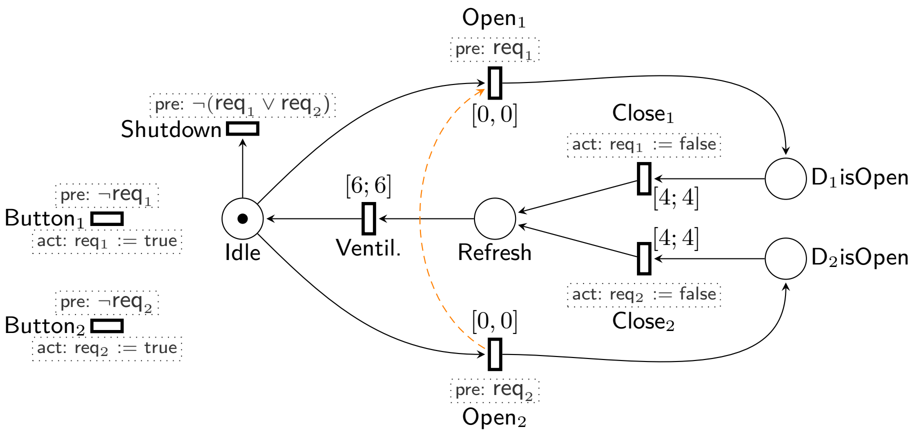


Fig. 1. A TTS model of an airlock system

Nets, the state of a TTS depends on its marking, m, that is the number of tokens in each place. We write $\mathcal{M}$ the set of markings. Since we manipulate values, the state of a TTS also depends on its *store*, that is a mapping from variable names to their respective values. We use the symbol s for a store and write $\mathcal{S}$ for the set of stores. Finally, we use the symbol t for a transition and T for the set of transitions of a TTS. The behavior of a TTS is defined by the set of all its (timed) traces. In this particular case, a trace will contain information about fired transitions (e.g. Open_1), markings, the value of variables, and the passing of time. Formally, we define an event ω as a triple (t, m, s) recording the marking and store immediately after the transition t has been fired. We denote Ω the set $T \times \mathcal{M} \times S$ of events. The set of non-negative rational numbers is written $\mathbb{Q}^+$.

Definition 1 (Timed trace). *A timed trace σ is a possibly infinite sequence of events $\omega \in \Omega$ and durations $d(\delta)$ with $\delta \in \mathbb{Q}^+$. Formally, σ is a partial mapping from $\mathbb{N}$ to $\Omega^* = \Omega \cup \{d(\delta) \mid \delta \in \mathbb{Q}^+\}$ such that $\sigma(i)$ is defined whenever $\sigma(j)$ is defined and $i \leq j$.*

Given a finite trace σ and a—possibly infinite—trace σ', we denote $\sigma\sigma'$ the *concatenation* of σ and σ'. This operation is associative. The semantics of patterns will be defined as a set of timed traces. Given a real-time pattern P, we say that a TTS T satisfies the requirement P if all the traces of T hold for P.

2.2 Metric Temporal Logic and Formulas over Traces

Metric Temporal Logic (MTL) [16] is an extension of LTL where temporal modalities can be constrained by a time interval. For instance, the MTL formula $A\ \mathbf{U}_{[1,3[}\ B$ states that in every execution of the system (in every trace), the event B must occur at a time $t_0 \in [1, 3[$ and that A holds everywhere in the interval $[0, t_0[$. In the following, we will also use a weak version of the "until modality", denoted $A\ \mathbf{W}\ B$, that does not require B to eventually occur. We refer the reader to [18] for a presentation of the logic and a discussion on the decidability of various fragments.

An advantage of using MTL is that it provides a sound and unambiguous framework for defining the meaning of patterns. Nonetheless, this partially defeats one of the original goal of patterns, that is to circumvent the use of temporal logic in the first place. For this reason, we propose an alternative way for defining the semantics of patterns that relies on first-order formulas over traces. For instance, when referring to a timed trace σ and an event A, we can define the "scope" σ after A–that determines the part of σ located after the first occurrence of A–as the trace σ_2 such that $\exists \sigma_1 . \sigma = \sigma_1 A \sigma_2 \wedge A \notin \sigma_1$. We believe that this second approach may ease the work of engineers that are not trained with formal verification techniques. Our experience shows that being able to confront different definitions for the same pattern, using contrasting approaches, is useful for teaching patterns.

2.3 Model Checking, Observers and TTS

We have designed our patterns so that checking whether a system satisfies a requirement is a decidable problem. We assume here that we work on discrete models (with a continuous time semantics), such as timed automata or time Petri Nets, and not on hybrid models. Since the model checking problem for MTL is undecidable [18], it is not enough to simply translate each pattern into a MTL formula to check whether a TTS satisfies a pattern. This situation can be somehow alleviated. For instance, the problem is decidable if we disallow simultaneous events in the system and if we disallow punctual timing constraints, of the form $[d, d]$. Still, while we may rely on timed temporal logics as a way to define the semantics of patterns, it is problematic to have to limit ourselves to a decidable fragment of a particular logic–which may be too restrictive–or to rely on multiple real-time model checking algorithms–that all have a very high complexity in practice.

To solve this problem, we propose to rely on *observers* in order to reduce the verification of timed patterns to the verification of LTL formulas. We provide for each pattern, P, a pair (T_P, ϕ_P) of a TTS model and a LTL formula such that, for any TTS model T, we have that T satisfies P if and only if $T \otimes T_P$ (the composition of the two models T and T_P) satisfies ϕ_P. The idea is not to provide a generic way of obtaining the observer from a formal definition of the pattern. Rather, we seek, for each pattern, to come up with the best possible observer in practice. To this end, using our tool chain, we have compared the complexity of different implementations on a fixed set of representative examples and for a specific set of properties and kept the best candidates.

3 A Catalog of Real-Time Patterns

We describe our patterns using a hierarchical classification borrowed from Dwyer [11] but adding the notion of "timing modifiers". Patterns are built from five categories, listed below, or from the composition of several patterns (see Sect. 3.4):

- **Existence Patterns (Present):** for conditions that must eventually occur;
- **Absence Patterns (Absent):** for conditions that should not occur;
- **Universality Patterns :** for conditions that must occur throughout the whole execution;
- **Response Patterns (Response):** for (trigger) conditions that must always be followed by a given (response) condition;
- **Precedence Patterns :** for (signal) conditions that must always be preceded by a given (trigger) condition.

In each class, generic patterns may be specialized using one of five *scope modifiers* that limit the range of the execution trace over which the pattern must hold:

- **Global :** the default scope modifier, that does not limit the range of the pattern. The pattern must hold over the whole timed trace;
- **Before R :** limit the pattern to the beginning of a time trace, up to the first occurrence of R;
- **After Q :** limit the pattern to the events following the first R;
- **Between Q and R :** limit the pattern to the events occurring between an event Q and the following occurrence of an event R;
- **After Q Until R :** similar to the previous scope modifier, except that we do not require that R must necessarily occur after a Q.

Finally, timed patterns are obtained using one of two possible kind of *timing modifiers* that limit the possible dates of events referred in the pattern:

- **Within** I **:** to constraint the delay between two given events to belong to the time interval I;
- **Lasting** D **:** to constraint the length of time during which a given condition holds (without interruption) to be greater than D.

When defining patterns, the symbols $A, B, \ldots$ stand for predicates on events $\omega \in \Omega$ such as $\mathsf{Open}_2 \vee \mathsf{req}_2$. In the definition of observers, a predicate A is interpreted as the set of transitions of the system that match A. Due to the somewhat large number of possible alternatives, we restrict this catalog to the most important presence, absence and response patterns. Patterns that are not described here can be found in a long version of this paper [1].

For each pattern, we give its denotational interpretation based on First-Order formulas over Timed Traces (denoted FOTT in the following) and a logical definition based on MTL. We provide also the observer and the LTL formula that should be combined with the system in order to check the validity of the pattern. We define some conventions on observers. In the following, Error, Start, ... are transitions that belong to the observer, whereas E_1 (resp. E_2) represents all transitions of the system that match predicate A (resp. B). We also use the symbol I as a shorthand for the time interval $[d_1, d_2]$. The observers for the pattern obtained with other time intervals–such as $]d_1, d_2]$, $]d_1, +\infty[$, or in the case $d_1 = d_2$–are essentially the same, except for some priorities between transitions that may change. By convention, the boolean variables used in the definition of an observers are initially set to false.

3.1 Existence Patterns

An existence pattern is used to express that, in every trace of the system, some events must occur.

Present A **after** B **within** I

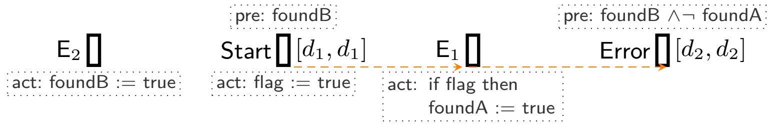
Predicate A must hold between d_1 and d_2 units of time (u.t) after the first occurrence of B. The pattern is also satisfied if B never holds.

Example: **present** Ventil. **after** $\mathsf{Open}_1 \vee \mathsf{Open}_2$ **within** $[0, 10]$

MTL def.: $(\neg B)\ \mathbf{W}\ (B \wedge \mathit{True}\ \mathbf{U}_I\ A)$

FOTT def.: $\forall \sigma_1, \sigma_2\ .\ (\sigma = \sigma_1 B \sigma_2 \wedge B \notin \sigma_1) \Rightarrow \exists \sigma_3, \sigma_4\ .\ \sigma_2 = \sigma_3 A \sigma_4 \wedge \Delta(\sigma_3) \in I$

Observer:



The associated LTL formula is $[]\neg$Error.

Explanation:

In this observer, transition Error is conditioned by the value of the shared boolean variables foundA and foundB. Variable foundB is set to true after transition E_2 and transition Error is enabled only if the predicate foundB $\wedge\neg$ foundA is true. Transition Start is fired d_1 u.t after an occurrence of E_2 (because it is enabled when foundB is true). Then, after the first occurrence of E_1 and if flag is true, foundA is set to true. This captures the first occurrence of E_1 after Start has been fired. After d_2 u.t., in the absence E_1, transition Error is fired. Therefore, the verification of the pattern boils down to checking if the event Error is reachable. The priority (dashed arrows) between Start, Error, and E_1 is here necessary to ensure that occurrences of E_1 at precisely the date d_1 or d_2 are taken in account.

Present first A before B within I

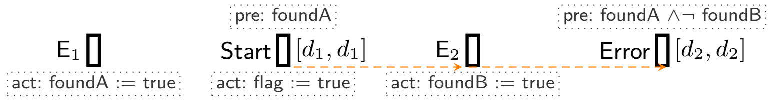
The first occurrence of predicate A holds between d_1 and d_2 u.t. before the first occurrence of B. The pattern is also satisfied if B never holds. (The difference with Present B after A within I is that B should not occur before the first A.)

Example: **present first** $\mathsf{Open}_1 \vee \mathsf{Open}_2$ **before** Ventil. **within** $[0, 10]$

MTL def.: $(\Diamond B) \Rightarrow (\,(\neg A \wedge \neg B)\ \mathbf{U}\ (A \wedge \neg B \wedge (\neg B\ \mathbf{U}_I\ B))\,)$

FOTT def.: $\forall \sigma_1, \sigma_2\ .\ (\sigma = \sigma_1 B \sigma_2 \wedge B \notin \sigma_1) \Rightarrow \exists \sigma_3, \sigma_4\ .\ \sigma_1 = \sigma_3 A \sigma_4 \wedge A \notin \sigma_3 \wedge \Delta(\sigma_4) \in I$

Observer:



The associated LTL formula is $(\Diamond B) \Rightarrow \neg\Diamond$(Error $\vee$ (foundB $\wedge\neg$flag)).

Explanation:

Like in the previous case, variables foundA and foundB are used to record the occurrence of transitions E_1 and E_2. Transition Start is fired, and variable flag is set to true, d_1 u.t. after the first E_1. Then transition Error is fired only if its precondition—the predicate foundA $\wedge\neg$ foundB—is true for d_2 u.t. Therefore transition Error is fired if and only if there is an occurrence of E_2 before E_1 (because then foundB is true) or if the first occurrence of E_2 is not within $[d_1, d_2]$ of the first occurrence of E_1.

Present A lasting D

Starting from the first occurrence when the predicate A holds, it remains true for at least duration D.

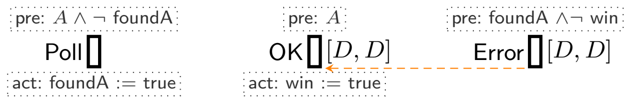
Comment: The pattern makes sense only if A is a predicate on states (that is, on the marking or store); since transitions are instantaneous, they have no duration.

Example: **present** Refresh **lasting** 6

MTL def.: $(\neg A)\ \mathbf{U}\ (\Box_{[0,D]} A)$

FOTT def.: $\exists \sigma_1, \sigma_2, \sigma_3\ .\ \sigma = \sigma_1\sigma_2\sigma_3 \wedge A \notin \sigma_1 \wedge \Delta(\sigma_2) \geqslant D \wedge A(\sigma_2)$

Observer:



The associated LTL formula is $\Box\neg$Error.

Explanation:

Variable foundA is set to true when transition *Poll* is fired, that is when A becomes true for the first time. Transition OK is used to set win to true if A is true for duration D without interruption (otherwise its timing constraint is resetted). Otherwise, if variable win is still false after D u.t., then transition *Error* is fired. We use a priority between *Error* and *OK* to disambiguate the behavior D u.t. after Poll is fired.

3.2 Absence Patterns

Absence patterns are used to express that some condition should never occur.

Absent A after B for interval I

Predicate A must never hold between d_1–d_2 u.t. after the first occurrence of B.

Comment: This pattern is dual to **Present** A **after** B **within** I (it is not equivalent to its negation because, in both patterns, B is not required to occur).

Example: **absent** $\text{Open}_1 \vee \text{Open}_2$ **after** $\text{Close}_1 \vee \text{Close}_2$ **for interval** $[0, 10]$

MTL def.: $\neg B\ \mathbf{W}\ (B \wedge \Box_I \neg A)$

FOTT def.: $\forall \sigma_1, \sigma_2, \sigma_3, \omega\ .\ (\sigma = \sigma_1 B \sigma_2 \omega \sigma_3 \wedge B \notin \sigma_1 \wedge \Delta(\sigma_2) \in I) \Rightarrow \neg A(\omega)$

Observer: We use the same observer as for **Present** A **after** B **within** I, but here Error is the expected behavior.
The associated LTL formula is $\Diamond B \Rightarrow \Diamond$Error.

Explanation:
Same as the explanation for **Present** A **after** B **within** I.

Absent A before B for duration D

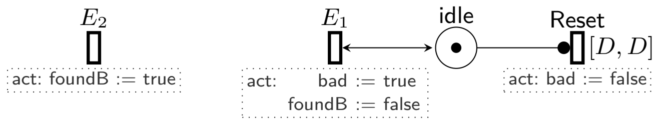
No A can occur less than D u.t. before the first occurrence of B. The pattern holds if there are no occurrence of B.

Example: **absent** Open$_1$ **before** Close$_1$ **for duration** 3

MTL def.: $\Diamond B \Rightarrow (A \Rightarrow (\Box_{[0,D]} \neg B))\ \mathbf{U}\ B$

FOTT def.: $\forall \sigma_1, \sigma_2, \sigma_3, \omega\ .\ (\sigma = \sigma_1 \omega \sigma_2 B \sigma_3 \wedge B \notin \sigma_1 \omega \sigma_2 \wedge \Delta(\sigma_2) \leqslant D) \Rightarrow \neg A(\omega)$

Observer:



The associated LTL formula is $\Box\neg(\mathsf{foundB} \wedge \mathsf{bad})$.

Explanation:
Variable $foundB$ is set to true after each occurrence of E_2. Conversely, we set the variables bad to true and $foundB$ to false at each occurrence of E_1. Therefore $foundB$ is true on every "time interval" between an E_2 and an E_1. We use transition $Reset$ to set bad to false if this interval is longer than D. As a consequence, the pattern holds if we cannot find an occurrence of E_2 ($foundB$ is true) while bad is true.

3.3 Response Patterns

Response patterns are used to express "cause–effect" relationship, such as the fact that an occurrence of a first kind of events must be followed by an occurrence of a second kind of events.

A **leadsto first** B **within** I

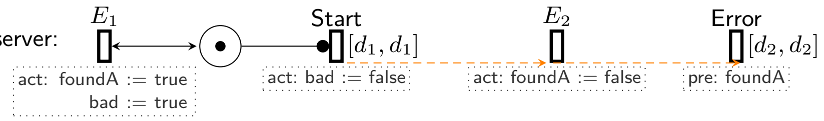
Every occurrence of A must be followed by an occurrence of B within time interval I (considering only the first occurrence of B after A).

Example: Button$_2$ **leadsto first** Open$_2$ **within** $[0, 10]$

MTL def.: $\Box(A \Rightarrow (\neg B)\ \mathbf{U}_I\ B)$

FOTT def.: $\forall \sigma_1, \sigma_2\ .\ (\sigma = \sigma_1 A \sigma_2) \Rightarrow \exists \sigma_3, \sigma_4\ .\ \sigma_2 = \sigma_3 B \sigma_4 \wedge \Delta(\sigma_3) \in I \wedge B \notin \sigma_3$

Observer:



The associated LTL formula is $(\Box\neg\mathsf{Error}) \wedge (\Box\neg(B \wedge \mathsf{bad}))$.

Explanation:
After each occurrence of E_1, variables $foundA$ and bad are set to true and the transition $Start$ is enabled. Variable bad is used to control the beginning of the time interval. After each occurrence of E_2 variable $foundA$ is set to false. Hence $Error$ is fired if there is an occurrence of E_1 not followed by an occurrence of E_2 after d_2 u.t. We use priorities to avoid errors when E_2 occurs precisely at time d_1 or d_2.

A **leadsto first** B **within** I **before** R

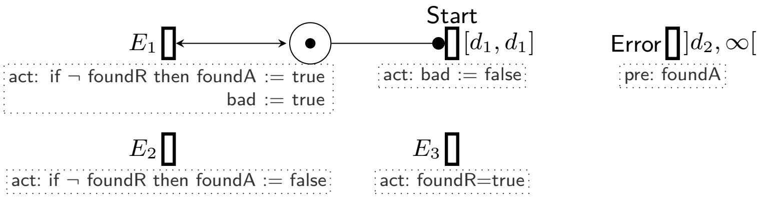
Before the first occurrence of R, each occurrence of A is followed by a B—and these two events occur before R—in the time interval I. The pattern holds if R never occur.

Example: Button$_2$ **leadsto first** Open$_2$ **within** $[0, 10]$ **before** Shutdown

MTL def.: $\lozenge R \Rightarrow (\Box(A \wedge \neg R \Rightarrow (\neg B \wedge \neg R)\ \mathbf{U}_I\ B \wedge \neg R)\ \mathbf{U}\ R$

FOTT def.: $\forall \sigma_1, \sigma_2, \sigma_3\ .\ (\sigma = \sigma_1 A \sigma_2 R \sigma_3 \wedge R \notin \sigma_1 A \sigma_2 \Rightarrow \exists \sigma_4, \sigma_5\ .\ \sigma_2 = \sigma_4 B \sigma_5 \wedge \Delta(\sigma_4) \in I \wedge B \notin \sigma_4$

Observer:



The associated LTL formula is $\lozenge R \Rightarrow (\Box\neg\mathsf{Error} \wedge \Box\neg(B \wedge \mathsf{bad}))$.

Explanation:

Same explanation than for the previous case, but we only take into account transitions E_1 and E_2 occurring before E_3.

A **leadsto first** B **within** I **after** R

Same than with the pattern "A ***leadsto first*** *B* ***within*** *I" but only considering occurrences of A after the first R.*

Example: Button$_2$ **leadsto first** Open$_2$ **within** $[0, 10]$ **after** Shutdown

MTL def.: $\Box(R \Rightarrow (\Box(A \Rightarrow (\neg B)\ \mathbf{U}_I\ B)))$

FOTT def.: $\forall \sigma_1, \sigma_2\ .\ (\sigma = \sigma_1 R \sigma_2 A \sigma_3 \wedge R \notin \sigma_1) \Rightarrow \exists \sigma_4, \sigma_5\ .\ \sigma_3 = \sigma_4 B \sigma_5 \wedge \Delta(\sigma_4) \in I \wedge B \notin \sigma_4$

Observer: It is similar to the observer of the pattern A leadsto first B within I before R . We should just replace ¬foundR in transition E_1 and E_2 by foundR. The associated LTL formula is $\lozenge R \Rightarrow (\Box\neg\mathsf{Error} \wedge \Box\neg(B \wedge \mathsf{bad}))$.

Explanation:

Same explanation than in the previous case, but we only take into account transitions E_1 and E_2 occurring after an E_3.

3.4 Composite Patterns

Patterns can be easily combined together using the usual boolean connectives. For example, the pattern "P_1 **and** P_2" holds for all the traces where P_1 and P_2 both hold. To check a composed pattern, we use a combination of the respective observers, as well as a combination of the respective LTL formulas. For instance, if (T_1, ϕ_1) and (T_2, ϕ_2) are the observers and LTL formulas corresponding to the patterns P_1 and P_2, then the composite pattern P_1 **and** P_2 is checked using the LTL formula $\phi_1 \wedge \phi_2$. Similarly, if we check the LTL formula $\phi_1 \Rightarrow \phi_2$ (implication) then we obtain a composite pattern $P_1 \multimap P_2$ that is satisfied by systems T such that, for all traces of T, the pattern P_2 holds whenever P_1 holds.

4 Use Cases and Experimental Results

In this section, we report on three experiments that have been performed using an extension of a Fiacre compiler that automatically compose a system with the necessary observers. In case the system does not meet its specification, we obtain a counter-example that can be converted into a timed sequence of events exhibiting a problematic scenario. This sequence can be played back using *play* and *nd*, two Time Petri Nets animators provided with Tina.

Avionic Protocol and AADL. Our first example is a network avionic protocol (NPL) which includes several functions allowing the pilot and ground stations to receive and send information relative to the plane: weather, speed, ... AADL has been used to model the dynamic architecture for this demonstrator [5]. The AADL model includes several threads that exchange information through shared memory data and amounts to about 8 diagrams and 800 lines of code (using AADL textual syntax). The AADL code specifies both the hardware and software architecture of the system and defines the real time properties of threads, like for instance their dispatch protocol (periodic or sporadic) or their periods.

We used the AADL2Fiacre plug-in of Topcased to check properties on the NPL specification. The Fiacre model obtained after transformation takes into account the complete behavior described in the AADL model but also the whole language execution model, meaning that our interpretation takes fully into account the scheduling semantics as specified in the AADL standard. The abstract state space for the TTS generated from Fiacre has about 120 000 states and 180 000 transitions and can be generated in less than 12 s on a typical development computer (Intel dual-core processor at 2 GHz with 2 Gb of RAM). On examples of this size, our model checker is able to prove formal properties in a few seconds. We checked a set of 22 requirements that were given together with the description of the system, all expressed using a natural language description and, in one case, a scenario based on a UML sequence diagram. Of these 22 requirements, 18 where instances of "untimed patterns", such as checking the absence of deadlock or that threads are resettable. The four remaining requirements where "response patterns" of the kind A leadsto first B within $[0, d]$. Using patterns, we were able to check the 22 patterns in less than 5 min.

Service Oriented Applications. We consider models obtained from the composition of services expressed using a timed extension of BPEL, the Business Process Execution Language. Our example models a scenario from the health-care domain related to patient handling during a medical examination. The scenario involves three entities, each one managed by a service: a Clinic Service (CS); a Medical Analysis Service (MAS); and a Pharmacy Service (PS). When a patient arrives in clinic, a doctor should check with the MCS whether its social security number is valid. If so, the doctor may order some medical analyses from the MAS and, after analyzing the results, he can order drugs through the PS. Timing constraints can be added to this scenario by associating a duration to each activity of the workflow and a delay to each service invocation.

We use our patterns to express different requirements on this system. An example involving the absence pattern is that we cannot have two medical analyses for a patient in less than 10 days (240 hours): absent MAS.medicalAnalysis after MAS.medicalAnalysis for interval $]0, 240]$. A more complicated example of requirement is to impose that if a doctor does not cancel a drug order within 6 hours, then it should not cancel drugs for another 48 hours. This requirement can be expressed using the composition of two absence patterns (see Sect. 3.4):

(absent MCS.drugsChanging after MCS.drugsAsking for interval $[0; 6]$)
$\multimap$ (absent MCS.drugsChanging after MCS.drugsAsking for interval $[0; 54]$).

Finally, using the notation S.init and S.end to refer to a start (resp. end) event in the service S, we can express that drugs must be delivered within 48 hours of the medical examination start: MCS.init leadsto PS.sendDrugsOrder within $[0; 48]$.

The complete scenario is given in [9], where we describe a transformation tool chain from Timed BPEL processes to Fiacre. For a more complex version of the health care scenario, with seven different services and more concurrent activities, the state graph for the TTS generated from Fiacre is quite small, with only 886 states and 2476 transitions. The generation of the Fiacre specification and its corresponding state space takes less than a second. For examples of this size, the verification time for checking a requirement is negligible (half a second).

Transportation Systems. Our final example is an automated railcar system, taken from [10], that was directly modeled using Fiacre. The system is composed of four terminals connected by rail tracks in a cyclic network. Several railcars, operated from a central control center, are available to transport passengers between terminals. When a car approaches its destination, it sends a request to signal its arrival to the terminal. This system has several real-time constraints: the terminal must be ready to accommodate an incoming car in 5 s; a car arriving in a terminal leaves its door open for exactly 10 s; passengers entering a car have 5 s to choose their destination; etc. There are three key requirements:

(P1) when a passenger arrives in a terminal, a car must be ready to transport him within 15 s. This property can be expressed with a response pattern, where Passenger/sndReq is the state where the passenger requests a car and Car/ackTerm is the state where it is served:

Passenger/sendReq **leadsto** Car/ackTerm **within** $[0, 15]$

(P2) When the car starts moving, the door must be closed:

present CarDoor/closeDoor **after** CarHandler/moving **within** $[0, 10]$

(P3) When a passenger select a destination (in the car), a signal should stay illuminated until the car is arrived:

absent Terminal/buttonOff **before** Control/ackTerm **for duration** 10

We can prove that these three patterns are valid on our Fiacre model. Concerning the performances, we are able to generate the complete state space of the railcar system in 310 ms, using 400 kB of memory. This gives an upper-bound to the complexity of checking simple (untimed) reachability properties on the system, like for instance the absence of deadlocks. The three patterns can all be checked in under 1.5 s. For instance, we observed that checking property (P1) is not more complex than exploring the complete system: the property is checked in 450 ms, using 780 kB of memory. Also, this is roughly the same complexity than checking the corresponding untimed requirement in LTL that is: $\Box(\mathsf{Passenger/sendReq} \Rightarrow \Diamond \mathsf{Control/ackTerm})$.

Conclusion. In other benchmarks, we have often found that the complexity of checking timed patterns is in the same order of magnitude than checking their untimed temporal logic equivalent. An exception to this observation is when the temporal values used in the patterns are far different from those found in the system; for example if checking a periodic system, with a period in the order of the milliseconds, against a requirement using an interval in the order of the minutes. More results on the complexity of our approach can be found in [2]. These experimentation, while still modest in size, gives a good appraisal of the use of formal verification techniques for real industrial software.

These experimental results are very encouraging. In particular, we can realistically envisage that system engineers could evaluate different design choices in a very short time cycle and test the safety of their solutions at each iteration.

5 Related Work and Contributions

We base our approach on the original catalog of specification patterns defined by Dwyer [11]. This work essentially study the expressiveness of their approach and define patterns using different logical framework (LTL, CTL, Quantified Regular Expressions, etc.). As a consequence, they do not need to consider the problem of checking requirements as they can readily rely on existing model checkers. Their patterns language is still supported, with several tools, an online repository of examples [12] and the definition of the Bandera Specification Language [8] that provides a structured-English language front-end. A recent study by Bianculli et al. [7] show the relevance of this pattern-based approach in an industrial context.

Some works consider the extension of patterns with hard real-time constraints. Konrad et al. [15] extend the patterns language with time constraints and give a mapping from timed pattern to TCTL and MTL. Nonetheless, they do not consider the complexity of the verification problem (the implementability of their approach). Another related work is [14], where the authors define observers based on Timed Automata for each pattern. However, they consider a less expressive set of patterns (without the lasting modifier) and they have not integrated their language inside a tool chain or proved the correctness of their observers. By contrast, we have defined a formal framework that has been used to prove the correctness of some of our observers [2]. Work is currently under way to mechanize these proofs using the Coq interactive theorem prover.

Our patterns can be viewed as a subset of the Contracts Specification Language (CSL), defined in the context of the SPEEDS project [19], which is intended as a pragmatic proposal for specifying contract assertions on HRC models. While the semantics for HRC is based on hybrid automata, the only automatic verification tools available for CSL use a discrete time model. Therefore, our verification tool chain provides a partial implementation for CSL (the part concerned by timing constraints) for a dense time model. This is an important result since more conception errors can be captured using a dense rather than a discrete time model.

Compared to these related works, we make several contributions. We extend the specification patterns language of Dwyer et al. with two modifiers for real-time constraints. We also address the problem of checking the validity of a pattern on a real-time system using model-based techniques: our verification approach is based on a set of observers, that are described in Sect. 3. Using this approach, we reduce the problem of checking real-time properties to the problem of checking simpler LTL properties on the composition of the system with an observer. Another contribution is the definition of a formal framework to prove that observers are correct and non-intrusive, meaning that they do not affect the system under observation. This framework is useful for proving the soundness of optimization. Due to space limitations, we concentrate on the definition of the patterns and their semantics in this paper, while most of the theoretical results are presented in a companion research report [2]. Finally, concerning tooling, we offer an EMF-based meta-model for our specification patterns that allow its integration within a model-driven engineering development: our work is integrated in a complete verification tool chain for the Fiacre modelling language and can therefore be used in conjunction with Topcased [13], an Eclipse based toolkit for system engineering.

6 Conclusion and Perspectives

We define a set of high-level specification patterns for expressing requirements on systems with hard real-time constraints. Our approach eliminates the need to rely on model checking algorithms for timed extensions of temporal logics that—when decidable—are very complex and time-consuming. While we have concentrated our attention on model checking—and although we only provide an implementation for TTS models—we believe our notation is interesting in its own right and can be reused in different contexts.

There are several directions for future works. We plan to define a compositional patterns inspired by the "denotational interpretation" used in the definition of patterns. The idea is to define a lower-level pattern language, with more composition operators, that is amenable to an automatic translation into observers (and therefore can dispose with the need to manually prove the correctness of our interpretation). In parallel, we plan to define a new modelling language for observers—adapted from the TTS framework—together with specific optimization techniques and easier soundness proofs. This language, which has nearly reached completion, would be used as a new target for implementing patterns verification.

References

1. Abid, N., Dal Zilio, S., Le Botlan, D.: A Real-Time Specification Patterns Language. Technical Report 11364, LAAS (2011)
2. Abid, N., Dal Zilio, S., Le Botlan, D.: Verification of Real-Time Specification Patterns on Time Transitions Systems. Technical Report 11365, LAAS (2011)
3. Berthomieu, B., Ribet, P.-O., Vernadat, F.: The tool tina – construction of abstract state spaces for Petri nets and time Petri nets. International Journal of Production Research 42, 14 (2004)
4. Berthomieu, B., Bodeveix, J.-P., Chaudet, C., Dal Zilio, S., Filali, M., Vernadat, F.: Formal Verification of AADL Specifications in the Topcased Environment. In: Kordon, F., Kermarrec, Y. (eds.) Ada-Europe 2009. LNCS, vol. 5570, pp. 207–221. Springer, Heidelberg (2009)
5. Berthomieu, B., Bodeveix, J.-P., Chaudet, C., Dal Zilio, S., Dissaux, P., Filali, M., Heim, S., Gaufillet, P., Vernadat, F.: Formal Verification of AADL models with Fiacre and Tina. In: Proc. of ERTSS 2010 5th International Congress and Exhibition on Embedded Real-Time Software and Systems (2010)
6. Berthomieu, B., Bodeveix, J.-P., Farail, P., Filali, M., Garavel, H., Gaufillet, P., Lang, F., Vernadat, F.: Fiacre: an Intermediate Language for Model Verification in the Topcased Environment. In: Proc. of ERTS (2008)
7. Bianculli, D., Ghezzi, C., Pautasso, C., Senti, P.: Specification Patterns from Research to Industry: a Case Study in Service-based Applications. In: The 34th International Conference on Software Engineering. IEEE (2012)
8. Corbett, J.C., Dwyer, M.B., Hatcliff, J., Robby: A Language Framework for Expressing Checkable Properties of Dynamic Software. In: Havelund, K., Penix, J., Visser, W. (eds.) SPIN 2000. LNCS, vol. 1885, pp. 205–223. Springer, Heidelberg (2000)
9. Guermouche, N., DalZilio, S.: Formal Requirement Verification for Timed Choreographies. Technical Report HAL 578436 (2011)
10. Dong, J.S., Hao, P., Qin, S.C., Sun, J., Yi, W.: Timed automata patterns. IEEE Transactions on Software Engineering 52(1) (2008)
11. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: Proc. of ICSE (1999)
12. Dwyer, M.B., Dillon, L.: Online Repository of Specification Patterns, `http://patterns.projects.cis.ksu.edu/`
13. Farail, P., Gaufillet, P., Canals, A., Le Camus, C., Sciamma, D., Michel, P., Crgut, X., Pantel, M.: The TOPCASED project: a Toolkit in Open source for Critical Aeronautic SystEms Design. In: Proc. of ERTS (2006)
14. Gruhn, V., Laue, R.: Patterns for timed property specifications. Electr. Notes Theor. Comput. Sci. 153(2), 117–133 (2006)
15. Konrad, S., Cheng, B.H.C.: Real-time specification patterns. In: Proc. of ICSE. ACM (2005)
16. Koymans, R.: Specifying real-time properties with metric temporal logic. Real-Time Syst. 2, 255–299 (1990)
17. Merlin, P.M.: A study of the recoverability of computing systems. PhD thesis (1974)
18. Ouaknine, J., Worrell, J.: On the decidability and complexity of metric temporal logic over finite words. Logical Methods in Computer Science 3 (2007)
19. Gafni, V.: Contract Specification Language (CSL). In: Speeds D2.5.4–Speculative and Exploratory Design in Systems Engineering (2008)

Automated Extraction of Abstract Behavioural Models from JMS Applications

Elvira Albert[1], Bjarte M. Østvold[2], and José Miguel Rojas[3]

[1] Complutense University of Madrid, Spain
[2] Norwegian Computing Center, Norway
[3] Technical University of Madrid, Spain

Abstract. Distributed systems are hard to program, understand and analyze. Two key sources of complexity are the many possible behaviors of a system, arising from the parallel execution of its distributed nodes, and the handling of asynchronous messages exchanged between nodes. We show how to systematically construct *executable models* of publish/subscribe systems based on the Java Messaging Service (JMS). These models, written in the Abstract Behavioural Specification (ABS) language, capture the essentials of the messaging behavior of the original Java systems, and eliminate details not related to distribution and messages. We report on JMS2ABS, a tool that automatically extracts ABS models from the *bytecode* of JMS systems. Since the extracted models are formal and executable, they allow us to reason about the modeled JMS systems by means of tools built specifically for the modeling language. For example, we have succeeded to apply simulation, termination and resource analysis tools developed for ABS to, respectively, execute, prove termination and infer the resource consumption of the original JMS applications.

1 Introduction

Reverse engineering is a key technique to understand and improve software that is available only in executable form. In this paper we focus on reverse engineering, or decompilation, of distributed Java applications given in *bytecode* form, with the purpose of increasing the understanding of such applications through analysis of reverse-engineered executable specifications. In the context of mobile code, programming languages which are compiled to bytecode and executed on a *virtual machine* are widely used nowadays. This is the approach used by Java bytecode and .NET. The execution model based on virtual machines has two important benefits when compared to classical machine code. First, bytecode is *platform-independent*, i.e., the same compiled code can be run on multiple platforms. Second, since the virtual machine is not directly executed on the hardware, it is possible to apply a *sandbox* model which guarantees that the bytecode does not have access to certain assets of the platform unless the code is explicitly granted access to them. In languages such as Java and C#, handling bytecode has a much wider application area than handling source code since the latter is often not available.

We study a specific class of distributed systems called *publish/subscribe* systems [9]. Furthermore we focus on applications built using the Java Messaging Service (JMS) [13], an industry-standard technology for realizing publish/-subscribe enterprise systems in Java. Our goal is to extract *abstract behavioral specifications* that capture the essentials of the messaging behavior, eliding implementation details, but preserving enough behavior so that analysis can draw conclusions about distribution and resource consumption of the original systems. The modeling language, called abstract behavioral specification language (ABS) allows to abstract from implementation details: Abstract data types and functions specify internal, sequential computations, while concurrency and distribution are handled using *active objects*. Analysis of ABS models is supported by a set of research tools.[1]

We report on JMS2ABS, a tool which automatically extracts an ABS model from a JMS application in bytecode form. The main phases of the extraction process are: (1) Decompile the bytecode into a higher-level intermediate representation with structured control flow. (2) Based on annotations added by the programmer, generate an ABS model from the intermediate representation. (3) During generation, insert calls to a pre-written ABS library of the JMS middleware, in order to model publish/subscribe middleware behavior.

The main contributions of our work can be summarized as follows:

– Section 4 provides a general and system-independent model of a subset of JMS publish/subscribe systems;
– In Section 5, we define a procedure for translating the code of a JMS publish/subscribe system into an executable model, and realize this as a tool;
– Section 6 applies existing tools developed for the ABS language in order to draw conclusions about the systems;
– Finally, Section 7 reports on a prototype implementation of our approach and evaluates it on two JMS examples.

2 Publish/Subscribe Communication in JMS

JMS is an industry standard for message communication in Java enterprise systems [13]. It offers APIs for configuring message passing services and for performing the message passing (i.e., encode, send, receive, and decode messages). One may realize various kinds of messaging systems using JMS; we focus on publish/subscribe systems.

Fig. 1 provides an overview of the publish/subscribe programming model of JMS. *Subscribers* have the ability to express interest in events or messages in order to be later notified of any message generated by a *publisher* that matches their registered interest. The basic model for publish/subscribe interaction relies on a message notification service (middleware) to provide storage and management for subscriptions, to mediate between and decouple publishers and

[1] These tools are currently being developed by the ongoing EU project HATS (FP7-231620), `http://www.hats-project.eu`.

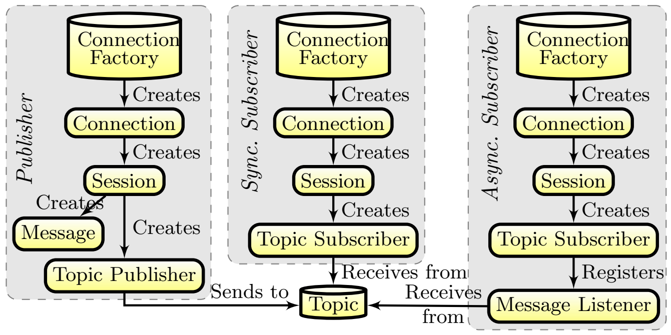


Fig. 1. Overview of the JMS publish/subscribe programming model

subscribers, and to deliver messages efficiently. Such middleware manages addressable message destination objects denoted as *topics*. The steps that *publishers* and *subscribers* perform, as depicted in Fig. 1, are: (1) Discover and join a topic of interest by means of a *connection factory* of topics. (2) Establish a *connection* to the factory and then start a new *session* for such a connection. (3) Create a *topic subscriber* for the session which allows receiving (subscribers) and sending (publishers) messages related to the topic of interest. (4) Create and publish a message (publisher). (5) Receive a message (subscriber).

We consider the subset of JMS components depicted in Fig. 1, capturing the essence of the publish/subscribe communication model. In addition, a model of a JMS system must include the state information and logic that decides how messages are processed and exchanged. Features such as transactions or failure recovery are outside the scope of this paper. Fig. 2 shows an example of a JMS publish/subscribe implementation of a basic fruit supply business model, consisting of a `FruitSupplier` class that acts as a publisher of updates for topic `"PriceLists"`; `SuperMarket` class implements asynchronous updates receipts from the topic, time-decoupled (i.e., non-blocking) from the publisher; and `Example` class provides the `main` method that initializes instances of `FruitSupplier` and `SuperMarket`.

Note that the different components are created and retrieved by invoking API methods. In particular, `ConnectionFactory` and `Topic` objects can be either created dynamically or found using JNDI services[2] . Subscribers can retrieve messages either asynchronously using a `MessageListener` object or synchronously through the (blocking) `receive` method of a `TopicSubscriber` object.

3 ABS: A Distributed Modeling Language

Within the OO paradigm, there are two main approaches to concurrency: (1) thread-based concurrency models (like those of Java and C#) are based on threads which share memory and are scheduled preemptively, i.e., threads can

[2] http://www.oracle.com/technetwork/java/jndi/

```
class FruitSupplier extends Thread {
  void run() {
    fac = new TopicConnectionFactory();
    con = fac.createTopicConnection();
    ses = con.createTopicSession(...);
    topic = ses.createTopic("PriceLists");
    publisher = ses.createPublisher(topic);
    message = ses.createObjectMessage(priceList);
    publisher.publish(message);//execution continues
    con.close();  } }
class SuperMarket extends Thread implements MessageListener {
  PriceList priceList;
  void onMessage(ObjectMessage m) {
      newPriceList = m.getObject();
      updatePrices(newPriceList); }
  void updatePrices(PriceList l) {
      Product p;
      for (int i = 1; i <= l.length(); i++) {
        p = l.get(i);
        if (priceList.contains(p)) priceList.update(p);
        else priceList.insert(p); } }
  void run() {
    fac = new TopicConnectionFactory();
    con = fac.createTopicConnection();
    ses = con.createTopicSession(...);
    topic = ses.createTopic("PriceLists");
    subsc = topicSession.createSubscriber(topic);
    subsc.setMessageListener(this);
    con.start();//execution continues
    con.close(); } }
class Example {
  void main(...){ new SuperMarket().start();
                  new FruitSupplier().start(); } }
```

Fig. 2. Excerpt of implementation of publish/subscribe in JMS

be suspended or activated at any time. To prevent threads from undesired interleavings, low-level synchronization mechanisms such as locks have to be used. Experience has shown that software written in the thread-based model is error-prone, difficult to debug, verify and maintain [20]. (2) In order to overcome these problems, the *active objects* model [20,15,8] aims at providing programmers with simple language extensions which allow programming concurrent applications with relatively little effort. The common idea is to take advantage of the inherent concurrency implicit in the notion of object in the following way: a concurrent object, conceptually, has a dedicated processor and it encapsulates a local heap which is not accessible from outside the object. Active (also called concurrent) objects operate similar to actors [12] and Erlang processes [5].

ABS [14] is the *abstract behavioral specification* language for distributed concurrent objects that we use to define the models. ABS has a *functional sublanguage* with abstract data types and functions to specify internal, sequential computations. The functional language is a standard strict functional language (the details are elsewhere [14]). As regards the concurrent imperative part, the central concept is the notion of component object group (COG), which generalizes the notion of concurrent or active object [12]. Intuitively, each COG has a dedicated *processor* and the COG is a concurrently running, isolated component. A COG can be considered as a container for objects. Its state is a heap

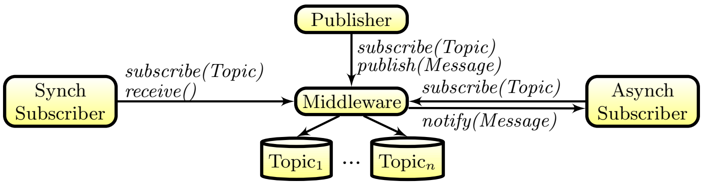


Fig. 3. Concurrent entities in ABS models (counterpart of Figure 1 for JMS)

of objects which are owned by the COG for their entire lifetime. The behavior of a COG consists of a set of cooperative tasks, which, again, are owned by the COG for their entire lifetime.

All communication is via asynchronous method calls between named objects, typed by interfaces. Method calls may be seen as triggers of concurrent activity, spawning new activities (so-called *processes*) in the called object without transferring control from the caller. The method caller may decide at runtime when to synchronize with the reply from a call. In general, an object may have many method activations competing to be executed. Among these, at most one process (or *task*) is active and the other processes are suspended in a process pool. Process scheduling is non-deterministic and occurs only at processor release points. This means that switching between tasks of the same object happens only at specific scheduling points during program execution, which are explicit in the source code and can be syntactically identified. This particular feature of process scheduling makes machine analysis notably simpler (when compared to the thread-based concurrency model).

In ABS syntax, asynchronous method calls are denoted $o!m(\overline{e})$. After asynchronously calling $x := o!m(\overline{e})$, the caller may proceed with its execution without blocking on the call. Here x is a future variable, o is an object (typed by an interface), and $\overline{e}$ are expressions. A future variable x refers to a return value which has yet to be computed. There are two operations on future variables, which control synchronization in ABS. First, a return test $x?$ evaluates to false unless the reply to the call can be retrieved. Second, the return value is retrieved by the expression $x.\texttt{get}$, which blocks all execution in the object until the return value is available. The statement sequence $x = o!m(\overline{e});\ v = x.\texttt{get}$ encodes a blocking, *synchronous call*, abbreviated $v = o.m(\overline{e})$, whereas the statement sequence $x = o!m(\overline{e});\ \texttt{await}\ x?;\ v = x.\texttt{get}$ encodes a non-blocking, *preemptable call*.

4 Modeling Publish/Subscribe Systems in ABS

This section shows how to model the behavior of a publish/subscribe system implemented using JMS by means of the ABS language. The model abstracts away implementation-related details of a distributed Java application while still capturing the essence of cooperation among the components of the system. Our goal is to preserve all essential application properties concerning distribution and performance but improve on clarity and tractability for automatic analysis purposes. Our starting point is the JMS system of Fig. 2, whose model in ABS is shown in Fig. 4 (in the model, *updatePrices* is a function that will be defined later). In particular, we focus on the components that participate in the system (Sec. 4.1) and the operations that can be executed (Sec. 4.2). Sec. 5 will then describe how to automate the model extraction process.

4.1 Distributed Entities

Our ABS model creates only one concurrent object per participant in the distributed communication, namely publishers, subscribers and the middleware; see Fig. 3. Thus, each concurrent entity in ABS encapsulates the behavior of several JMS components that will communicate with the remaining entities by means of asynchronous calls and future variables.

Clients: Publishers and Subscribers. A JMS system relies on a number of objects in order to perform distributed operations. This design makes JMS portable and interoperable across multiple messaging products. However, it often makes the resulting programs harder to understand and thus analyze. In the ABS models we simplify this into a smaller set of objects. Namely, a publish/subscribe client in ABS will just need to create a session object and a publisher/subscriber object to interact with the middleware.

Table 1. Example mapping from Java/JMS to Distributed ABS

JMS instructions	**Equivalent ABS models**
Create a distributed object	
`obj = new C();` `//class C implements Runnable` `//interface or extends Thread`	`obj = new cog C();`
Establish new session	
`f = new TopicConnectionFactory();` `c = f.createTopicConnection();` `s = c.createTopicSession(...);` `t = new Topic("TopicName");`	`s = middleware.createSession();` `t = middleware.createTopic("TopicName");`
Send a message	
`pub = s.createPublisher(t);` `connection.start();` `m = s.createTextMessage();` `m.setText("message");` `pub.publish(m);`	`pub = s.createPublisher(t);` `m = "message";` `pub!publish(m);`
Receive a message synchronously	
`sub = s.createSubscriber(t);` `connection.start();` `m = topicSubscriber.receive();`	`sub = s.createSubscriber(t);` `Fut<Message> f = s!receive();` `message = f.get;`
Receive a message asynchronously	
`sub = s.createSubscriber(t);` `l = new TextListener();` `sub.setMessageListener(l);` `connection.start();`	`sub = s.createSubscriber(t);` `l = new MessageListener();` `sub.setMessageListener(l);`

Middleware. In a real publish/subscribe system the middleware (the message-oriented middleware, or MOM) is a highly distributed entity. In our model of JMS we simplify the middleware to one central entity. While not a desirable choice in an actual implementation, it still allows to analyze many properties of applications. The middleware entity relies on the concurrency model of ABS to provide publish/subscribe services. The `main` block of the ABS model creates the initial configuration of the publish/subscribe system, see Line 20 of Fig. 4 (the counterpart of Lines 32–33 of Fig. 2). Observe that the model uses COGs to represent each of the distributed/concurrent entities.

4.2 Operations

Here we consider the operations of a publish/subscribe system.

Message Sending. A publisher sends a message to the topic using a session, see method `run` of class `FruitSupplier` (Lines 3–10 of Fig. 2). The asynchronous semantics of the operation can be simulated by an ABS asynchronous method call, see Lines 3–8 of Fig. 4. In JMS, the sending operation implies some decisions regarding delivery mode, priority and time-to-live for the message. These configuration parameters can be global to a message publisher or specific for each message. For flexibility, we use the latter option and include configuration parameters as properties of messages.

Asynchronous Message Receipt. Method `run` of class `SuperMarket` in Fig. 2 shows that asynchronous message receipt in JMS is achieved by instantiating the `MessageListener` class. The new object is bound to the subscriber object and is able to receive and process incoming messages in its `onMessage` method (named `notify` in the publish/subscribe literature [9]). This method is triggered from the JMS provider upon arrival of a new message to the topic (Lines 23–30 of Fig. 2).

```
 1 class FruitSupplier(Middleware mw) {
     Unit run() {
 3     Topic topic = "PriceLists";
       TopicSession session = mw.createSession();
 5     TopicPublisher publisher = session.createPublisher(topic);
       ObjectMessage message = new ObjectMessage(priceList);
 7     session!publish(message);
   } }
 9 class SuperMarket(Middleware mw) implements MessageListener {
     Unit onMessage(ObjectMessage m) {
11     newPriceList = m.getObject();
       priceList = updatePrices(newPriceList);
13   }
     Unit run() {
15     Topic topic = "PriceLists";
       TopicSession session = mw.createSession();
17     TopicSubscriber subscriber = session.createSubscriber(topic);
       subscriber!setMessageListener(this);
19   }
   { Middleware mw = new cog Middleware();
21   new cog SuperMarket(mw); new cog FruitSupplier(mw);
   } //main block
```

Fig. 4. Extracted ABS model for the running example

In ABS, an equivalent asynchronous message receipt is implemented in Lines 15–19 of Fig. 4. The concurrent behavior, i.e., the interaction with different topics simultaneously, is achieved by sharing the single-threaded session object among clients within the same COG. A serial order of outgoing and incoming messages is implicitly modelled when using a shared session object. Table 1 summarizes the mappings that we have described along this section for our particular example.

5 Automatic Extraction of ABS Models from JMS

Figure 5 provides an overview of the main steps performed by JMS2ABS for automatically extracting ABS models from JMS publish/subscribe systems. The tool receives as input the *bytecode* associated to the JMS publish/subscribe system and, optionally, a set of *annotations* that indicate which methods of the code should be transformed into functions and which ones into imperative methods. The absence of annotations brings about a purely imperative translation. Intuitively, the following stages are carried out by the extraction process. First, the bytecode is decompiled into a higher-level intermediate representation (IR) which, among other things, features structured control flow. Then, a driver module reads the IR and the set of annotations and directs the model extraction process either towards a functional implementation or towards an imperative one. The extraction of functional code requires a static single assignment (SSA) transformation [4] and automatically generates abstract data types and functions. The ABS library functions include standard data types for lists, trees, etc. and some common functions on these types. They are used by the translation when possible. The imperative object-oriented extraction is based on the modeling of JMS using ABS defined in Table 1. As an external component to this process, we have available the ABS implementation of the specific JMS middleware in use. As a result of the process, an ABS model is obtained which includes abstract data types, functions and classes. The following sections describe the main components of JMS2ABS.

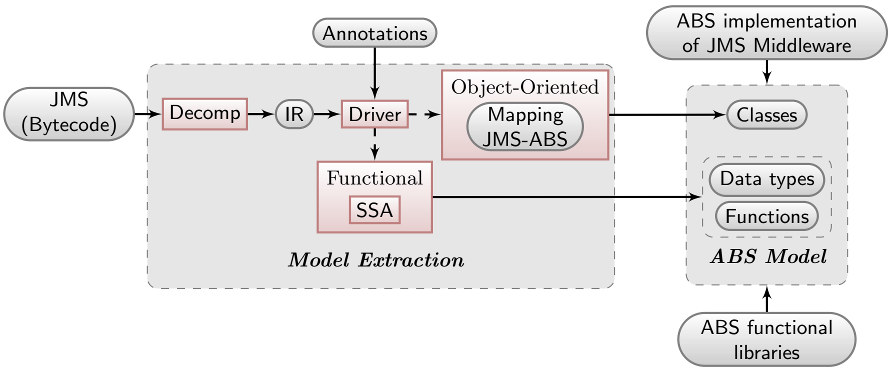


Fig. 5. Overview of main components of JMS2ABS

5.1 From Bytecode to Intermediate Representation

A method m in a Java (bytecode) program is represented by a set of *procedures* in the IR such that there is an entry procedure named m and the remaining ones are intermediate procedures invoked only from m. The translation of a program into the IR works by first building the control flow graph (CFG) from the program, and then representing each block of the CFG in the IR as a rule. The process is identical to that of Albert *et al.* [2], hence, we will not go into the details of the transformation but just show the syntax of the transformed program. A *program* in the IR consists of a set of *procedures* which are defined as a set of (recursive) rules. A procedure p is defined by a set of *guarded rules* which adhere to the following grammar:

$$\begin{array}{ll}
\mathit{rule} ::= p(\bar{x}, \bar{y}) \leftarrow g, b_1, \ldots, b_n & g ::= \mathit{true} \mid \mathit{exp}_1 \ \mathit{op} \ \mathit{exp}_2 \mid \mathsf{type}(x, C) \\
\mathit{exp} ::= x \mid \mathsf{null} \mid n \mid x{-}y \mid x{+}y \mid x{*}y & \mathit{op} ::= \ > \mid < \mid \leq \mid \geq \mid = \mid \neq \\
b ::= x{:=}\mathit{exp} \mid x{:=}\mathsf{new}\ c \mid x{:=}y.f \mid x.f{:=}y \mid q(\bar{x}, \bar{y}) &
\end{array}$$

where $p(\bar{x}, \bar{y})$ is the *head* of the rule; $\bar{x}$ (resp. $\bar{y}$) are the input (resp. output) parameters; g its guard, which specifies conditions for the rule to be applicable;

```
0: iconst_1
1: istore_3
2: iload_3
3: aload_1
4: invokevirtual length:()I
7: if_icmpgt 43
10: aload_1
11: iload_3
12: invokevirtual get:(I)LProduct;
15: astore_2
16: aload_0
17: aload_2
18: invokevirtual exists:(LProduct;)Z
21: ifeq 32
24: aload_0
25: aload_2
26: invokevirtual updatePrice:(LProduct;)V
29: goto 37
32: aload_0
33: aload_2
34: invokevirtual add:(LProduct;)V
37: iinc 3, 1
40: goto 2
43: return
```

```
updatePrices([this,l],[]) ← i := 1,
    rule_2([this,l,i],[]).
rule_2([this,l,i],[]) ←
    length([l],[$s_1$]),
    rule_7([this,l,i,$s_1$],[]).
rule_7$_1$([this,l,i,$s_1$],[]) ←
    i ≤ $s_1$,
    get([l,i],[p]),
    exists([this,p],[$s_2$]),
    rule_21([this,l,p,i,$s_2$],[$i_p$]).
rule_7$_2$([this,l,i,$s_1$],[]) ←
    i > $s_1$.

rule_21$_1$([this,l,p,i,$s_1$],[]) ←
    $s_1$ = 0,
    add([this,p],[]),
    rule_37([this,l,i],[]).
rule_21$_2$([this,l,p,i,$s_1$],[]) ←
    $s_1$ ≠ 0,
    updatePrice([this,p],[]),
    rule_37([this,l,i],[]).
rule_37([this,l,i],[]) ←
    $i_p$ := i +1,
    rule_2([this,l,$i_p$],[]).
```

Fig. 6. Pretty-printed IR for method `updatePrices` of class `PriceList`

$b_1, \ldots, b_n$ the body of the rule; n an integer; x and y variables; f a field name, and $q(\bar{x}, \bar{y})$ a call-by-value procedure call. The IR supports class definition and includes instructions for object creation, field manipulation, and type comparison through the instruction `type`(x, C), which succeeds if the runtime class of x is exactly C. A class C is a finite set of fields with either numeric (integer) or reference (class name) type.. The key features of this representation, which will simplify the transformation later, are: (1) input and output parameters are explicit variables of rules, (2) *recursion* is the only iteration mechanism, (3) *guards* are the only form of conditional, and (4) objects can be regarded as records, and the behavior induced by dynamic dispatch is compiled into *dispatch* rules guarded by a `type` check.

As an example, let us consider method `updatePrices` in Fig. 2. The left-hand column of Fig. 6 shows the bytecode of this method (which is the input to JMS2ABS) and the right-hand column contains the IR that JMS2ABS uses which features the three first points above. We can observe that instructions in the IR have an almost one-to-one correspondence with bytecode instructions (*rule_7* in the IR corresponds to the CFG block starting at bytecode instruction 7, for example), but they contain as explicit parameters the variables on which they operate (the operand stack is represented by means of variables). Another important aspect of the IR is that unstructured control flow of bytecode (i.e., the use of `goto` statements) is transformed into recursion and loop conditions become `guards`, as in rules *rule_2* and *rule_37* for instance.

5.2 From IR to Functional and Distributed ABS

To generate functions from a set of procedures in the IR, JMS2ABS performs three main steps: (1) An SSA transformation on the IR guarantees that variables are only written once [4]. (2) Then, for each recursive rules in the IR, it generates an associated function with the same name, where each instruction is transformed into an equivalent one in the functional sub-language of ABS. The process is similar to decompilation from bytecode to a simply typed functional language [16], to TRS [17] or to CLP programs [11]. Hence, we do not go into the details of the process but rather show an example. (3) Finally, JMS2ABS generates definitions of the data types involved in the functions. This is done by recursively inspecting the types of the class fields until reaching a primitive type, and using data constructors to group the fields that form an object.

The following function corresponds to the bytecode in Fig. 6. It is extracted from the above IR in a fully automatic way. ABS's *let* and *case* expressions, resp., are used to represent variable bindings and conditional statements in the original program. Moreover, observe that several data types declarations have been generated from class `PriceList`. The new algebraic data type `PriceList` has two data constructors: one for the empty list (`EmptyPriceList`) and one for the combination of a product and another list (`ConsPriceList(Product,PriceList)`).

```
//Data type declarations
type ProductID = Int;  type Price = Int;
data Product = EmptyProduct | ConsProduct(ProductID, Price);
data PriceList = EmptyPriceList | ConsPriceList(Product, PriceList);
//Function definitions
PriceList updatePrices(PriceList l) {
    let Int i = 1 in loop(priceList, newPriceList, i);
}
PriceList loop(PriceList l1, PriceList l2, Int i) {
    let n = length(l2) in
      case i <= n {
        True  => let p = get(l2, i) in
                      case (contains(l2, p)) {
                        True  => return loop(update(l1, p), l2, i+1);
                        False => return loop(add(l1, p), l2, i+1);}
        False => return l1;}
}
```

All procedures which have not been transformed into functions will become methods of the ABS models. Each ABS class will have as attributes the same ones as in the original Java program. Then, the translation of each method is performed by mapping each instruction in the IR into an equivalent one in ABS. The instructions which involve the distribution aspects of the application are translated by relying on the mapping of Table 1.

Fig. 7 shows the IR for the Java method `SuperMarket.run`. Observe how instructions in lines 2–5 match with the pattern for session establishment shown in Table 1. Instructions in lines 7–9 correspond to the asynchronous receiving of a message. From this IR it is straightforward to extract the model for method `run` showed in Fig. 4. Because of the correspondence between the involved operations in JMS and ABS, the main properties of the JMS systems (e.g., those regarding *reliability* and *safety* [6]) are preserved.

```
0 run([this],[]) ← tConFac := null, tCon := null, tSes := null, topic := null,
1                  tSubscriber := null, tListener := null,
2                  tConFac := new TopicConnectionFactory,
3                  createTopicConnection([tConFac],[tCon]),
4                  createTopicSession([tCon],[tSes]),
5                  createTopic([tSes,this.topicName],[topic])
6                  createSubscriber([tSes,topic],[tSubscriber]),
7                  tListener := new PriceListener,
8                  setMessageListener([tSubscriber,tListener],[]),
9                  start([tCon],[]), close([tCon],[]).
```

Fig. 7. Pretty-printed IR for method `run` of class `SuperMarket`

6 Using the ABS Toolset on the Extracted Models

The final goal of the extraction of ABS models from bytecode systems is to be able to perform machine analysis of JMS systems via their equivalent ABS models. This section outlines the application of two ABS tools: the simulator [14] and the COSTABS termination and resource usage analyzer [1].

6.1 Simulation

Once compiled, ABS models can be run in a simulator. The ABS toolset has two main simulators, with corresponding back-ends in the ABS compiler: One simulator is defined using rewriting logic and the Maude system [7], and the other is written in Java. The Maude simulator allows modellers to explore the model's state-space declaratively and model check it. The Java simulator does source-level simulation, meaning that modellers can follow the model's control flow at the statement level and observe object or method state. Both simulators allow modellers to control scheduling of methods, for example, control when a JMS message is sent and when it is received.

6.2 Resource and Termination Analysis

Resource analysis (a.k.a. cost analysis) aims at automatically inferring bounds on the resource consumption of programs statically, i.e., without having to execute the program. The inferred bounds are symbolic expressions given as functions of its *input data sizes*. For instance, given a method `void traverse(List l)`, an upper bound (e.g., on the number of execution steps) can be an expression on the form *l*200+10*, where *l* refers to the size of the list `l`. The analysis guarantees that the number of steps of executing `traverse` will never exceed the amount inferred by analysis. COSTABS [1], a COSt and Termination analyzer for ABS, is a system able to prove termination and obtain *resource usage bounds* for both the imperative and functional fragments of ABS programs. The resources that COSTABS can infer include termination, number of execution steps, memory consumption, number of asynchronous calls. Knowledge of the number of asynchronous calls is useful to understand and optimize the distributed behavior of the application (e.g., to detect bottlenecks when one object is receiving a large amount of asynchronous calls). COSTABS allows using *asymptotic* (i.e., big O complexity) notation for the results of the analysis and obtain simpler cost expressions.

Table 2. Resource analysis results

Method	#Instructions	Memory	#Async Calls
`run`	`max`(allSubscribers)	`max`(allSubscribers)	1
`onMessage`	m*(m+`max`(priceList))+m^2	`max`(priceList)	1

Let us analyze the resource consumption of the methods of class `SuperMarket` from the extracted ABS model. Table 2 shows the asymptotic results that COSTABS computes. The upper bound on the number of instructions inferred for method `run` depends on the number of clients that are subscribed to the topic (field `allSubscribers` of class `TopicSession`). `max`(f) denotes the maximum value that field f can take. This is because in our current implementation the size of the list of subscribers is not statically known, as it is updated when a new subscriber arrives (the analysis uses `max`(allSubscribers) to bound its size). As regards the analysis of `onMessage`, it requires analyzing `updatePrices` which traverses the new list of prices `priceList` and, for each of its elements, it checks whether it already exists or must be added to the local list of prices. The latter requires inspecting the object message `m` which is an input parameter of the method. Hence, we obtain a quadratic complexity on the sizes of `m` and `priceList`. The memory allocation accounts for the creation of the functional data structures. Namely, in method `run` (resp. `onMessage`), we create the data structure `allSubscribers` (resp. `PriceList`). Finally, it can be observed that both methods perform a constant number of asynchronous method calls, hence the rightmost column shows a constant complexity (denoted by 1). A main novelty of COSTABS, which is not available in other systems, is the notion of *cost centers*. This is motivated by the fact that distribution does not match well with the traditional monolithic notion of cost which aggregates the cost of all distributed components together. Albert *et al.* [1] propose the use of cost centers to keep the resource consumption of the different distributed components separate.

The cost bounds that are shown in Table 2 are computed as a monolithic expression which accumulates the resources consumed by all objects together. More interestingly, COSTABS can show the results separated by cost centers. In particular, we consider that all objects of the same class belong to the same cost center (i.e., the share the processor). Now, the execution of method `SuperMarket.run` performs steps in three cost centers, namely in `SuperMarket`, `Middleware` and in `TopicSubscriber`. By enabling the cost centers option, COSTABS shows that `max`(allSubscribers) is the upper bound on both number of instructions and memory in the cost center `Middleware`. In cost center `TopicSubscriber`, the upper bounds on number of instructions and on memory consumption are constant. Also, in cost center `SuperMarket`, the upper bound for both cost models is constant. Method `onMessage` is integrally executed in the `SuperMarket` cost center (hence the same results of Table 2 are obtained). Performing cost analysis of a distributed system, using cost centers, allows detecting bottlenecks if one distributed component (cost center) has a large resource consumption while siblings are idle most of the time.

7 Prototype Implementation

JMS2ABS can be used on 32-bit Linux systems through a command-line interface, is open-source and can be downloaded from `http://tools.hats-project.eu/`. Also, available from the same place, is our ABS model of JMS middleware, examples of how to write publish/subscribe ABS models using the middleware model, and Java/JMS example applications from which models may be extracted. These examples correspond to the running example of this paper, and a Chat example borrowed from Richards *et al.* [19] and slightly simplified. The Java code is accompanied by the necessary Java/JMS libraries and a makefile which may be used to run the tool on the Java examples. Although still a research prototype, JMS2ABS is reasonably efficient. For instance, on an Intel(R) Core(TM) i5 CPU at 1.7GHz with 4GB of RAM running Ubuntu Linux 11.10, the overall time to extract the model for the running example is 910 msec. This time is divided into the time for building the CFG (240 msec.), generating and optimizing the intermediate representation (40 msec.) and building and refining the ABS model (630 msec.). The Chat example is smaller and its overall model extraction time is 790 msec. In this case, the most costly phase is also the model generation and refinements, which takes 490 msec. of the overall time.

8 Related Work

Reverse engineering higher-level specifications from complex third party or legacy code has applications in analyzing, documenting and improving the code. Broadly speaking, we can classify reverse engineering tools into two categories: (1) When the higher-level specification is some sort of software visualization formalism which abstracts away most of the program semantics (e.g., UML class diagrams, control flow graphs or variable data flow graphs), reverse engineering is usually applied in order to understand the structure of the source code faster and more accurately. This in turn can detect problems related to the design of the application, to task interactions, etc. (2) When the higher-level specification provides an abstraction of the program semantics, but still the properties of interest are observable on it, reverse engineering can be used to develop analysis tools that reason about the original code by means of analyzing the reverse engineered specification. This has the advantage that, instead of analyzing the complex original code, we develop the tools on a simpler formalism which allows inferring the properties of interest more easily.

Our work falls into the second category. The overall motivation behind our work is to be able to analyze (complex) distributed Java JMS applications by means of tools developed for (simpler) ABS models. In particular, we have been able to apply simulation and *cost analysis* techniques developed for ABS programs [3,18] to reason on JMS applications. It is widely recognized that publish/-subscribe systems are difficult to reason about, and there are several previous approaches to modeling their behavior using different formalisms. Baldoni et al. [6] provide one of the first formal computational frameworks for modeling publish/subscribe systems. The focus in this work is different from ours; their main concern is the notion of time in the communication, which allows them to evaluate the overall performance, while we do not consider this aspect. Another formalism for publish/subscribe system is provided by Garlan et al. [10]. Instead of building executable programs as we do, they rely on a finite state machine that can be checked using existing model checking tools.

9 Conclusions and Future Work

Our goal is to show that it is possible to build a tool that automatically extracts useful models for complex distributed systems such as the JMS publish/subscribe using the concurrency and distribution mechanisms provided by the ABS modeling language. These mechanisms are not very different from those used by other distributed object-based modeling languages [12], and so we expect our study to provide useful conclusions beyond the mere case study performed.

Publish/subscribe systems come in a large range of flavors, depending on applications and requirements [9]. The common idea is to asynchronously decouple publishers from subscribers. In a purely centralized model such as the one used in this paper, providing the expected service is not hard, as the server has full knowledge to ensure that messages are sent only to active subscribers, in the same order in which they come in. In general, however, a reusable and general model must allow for decentralized implementations in which full consistency (in the sense that messages are received by only and all subscribers at any given time) and order preservation (same order of messages for all subscribers) cannot be achieved with good performance. We are currently examining ways in which the ABS framework can be extended to allow richer families of implementations.

Acknowledgments. The authors would like to thank Mads Dam for his valuable comments and insights. This work was funded in part by the Information & Communication Technologies program of the European Commission, Future and Emerging Technologies (FET), under the ICT-231620 *HATS* project, by the Spanish Ministry of Science and Innovation (MICINN) under the TIN-2008-05624 and PRI-AIBDE-2011-0900 projects, by UCM-BSCH-GR35/10-A-910502 grant and by the Madrid Regional Government under the S2009TIC-1465 *PROMETIDOS-CM* project.

References

1. Albert, E., Arenas, P., Genaim, S., Gómez-Zamalloa, M., Puebla, G.: Cost Analysis of Concurrent OO Programs. In: Yang, H. (ed.) APLAS 2011. LNCS, vol. 7078, pp. 238–254. Springer, Heidelberg (2011)
2. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost Analysis of Object-Oriented Bytecode Programs. Theor. Comput. Sci. 413 (January 2012)
3. Albert, E., Genaim, S., Gómez-Zamalloa, M., Johnsen, E.B., Schlatte, R., Lizeth Tapia Tarifa, S.: Simulating Concurrent Behaviors with Worst-Case Cost Bounds. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 353–368. Springer, Heidelberg (2011)
4. Appel, A.W.: SSA is functional programming. SIGPLAN Not. 33, 17–20 (1998)
5. Armstrong, J., Virding, R., Wistrom, C., Williams, M.: Concurrent Programming in Erlang, 2nd edn. Prentice Hall (1996)
6. Baldoni, R., Beraldi, R., Tucci Piergiovanni, S., Virgillito, A.: On the modelling of publish/subscribe communication systems. Concurr. Comput.: Pract. Exper. 17, 1471–1495 (2005)
7. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007)
8. de Boer, F.S., Clarke, D., Johnsen, E.B.: A Complete Guide to the Future. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 316–330. Springer, Heidelberg (2007)

9. Eugster, P., Felber, P., Guerraoui, R., Kermarrec, A.: The many faces of publish/-subscribe. ACM Comput. Surv. 35, 114–131 (2003)
10. Garlan, D., Khersonsky, S., Kim, J.S.: Model Checking Publish-Subscribe Systems. In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 166–180. Springer, Heidelberg (2003)
11. Gómez-Zamalloa, M., Albert, E., Puebla, G.: Decompilation of Java Bytecode to Prolog by Partial Evaluation. Information and Software Technology 51, 1409–1427 (2009)
12. Haller, P., Sommers, F.: Actors in Scala: Concurrent programming for the multi-core era. Artima, PrePrint edition (March 2011)
13. Hapner, M., Burridge, R., Sharma, R., Fialli, J., Stout, K.: Java Message Service Specification. Sun Microsystems, Inc., Version 1.1 (April 2002)
14. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: A Core Language for Abstract Behavioral Specification. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2010. LNCS, vol. 6957, pp. 142–164. Springer, Heidelberg (2011)
15. Johnsen, E.B., Owe, O.: An asynchronous communication model for distributed concurrent objects. Software and Systems Modeling 6(1), 35–58 (2007)
16. Katsumata, S.-Y., Ohori, A.: Proof-Directed De-compilation of Low-Level Code. In: Sands, D. (ed.) ESOP 2001. LNCS, vol. 2028, pp. 352–366. Springer, Heidelberg (2001)
17. Otto, C., Brockschmidt, M., von Essen, C., Giesl, J.: Automated Termination Analysis of Java Bytecode by Term Rewriting. In: RTA 2010. LIPIcs, vol. 6 (2010)
18. HATS Project. Report on Resource Guarantees, Deliv. 4.2 of project FP7-231620 (HATS) (March 2011), http://www.cse.chalmers.se/research/hats/sites/default/files/Deliverable42.pdf
19. Richards, M., Monson-Haefel, R., Chappell, D.A.: Java Message Service - Creating Distributed Enterprise Applications, 2nd edn. O'Reilly (2009)
20. Schäfer, J., Poetzsch-Heffter, A.: JCoBox: Generalizing Active Objects to Concurrent Components. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 275–299. Springer, Heidelberg (2010)

Certifying and Reasoning on Cost Annotations in C Programs

Nicolas Ayache[1,2], Roberto M. Amadio[1], and Yann Régis-Gianas[1,2]

[1] Université Paris Diderot (UMR-CNRS 7126)
[2] INRIA (Team πr^2)

Abstract. We present a so-called labelling method to enrich a compiler in order to turn it into a "cost annotating compiler", that is, a compiler which can *lift* pieces of information on the execution cost of the object code as cost annotations on the source code. These cost annotations characterize the execution costs of code fragments of constant complexity. The first contribution of this paper is a proof methodology that extends standard simulation proofs of compiler correctness to ensure that the cost annotations on the source code are sound and precise with respect to an execution cost model of the object code.

As a second contribution, we demonstrate that our label-based instrumentation is scalable because it consists in a modular extension of the compilation chain. To that end, we report our successful experience in implementing and testing the labelling approach on top of a prototype compiler written in ocaml for (a large fragment of) the C language.

As a third and last contribution, we provide evidence for the usability of the generated cost annotations as a mean to reason on the concrete complexity of programs written in C. For this purpose, we present a Frama-C plugin that uses our cost annotating compiler to automatically infer trustworthy logic assertions about the concrete worst case execution cost of programs written in a fragment of the C language. These logic assertions are synthetic in the sense that they characterize the cost of executing the entire program, not only constant-time fragments. (These bounds may depend on the size of the input data.) We report our experimentations on some C programs, especially programs generated by a compiler for the synchronous programming language Lustre used in critical embedded software.

1 Introduction

The formal description and certification of software components is reaching a certain level of maturity with impressing case studies ranging from compilers to kernels of operating systems. A well-documented example is the proof of functional correctness of a moderately optimizing compiler from a large subset of the C language to a typical assembly language of the kind used in embedded systems [11].

In the framework of the *Certified Complexity* (CerCo) project[1] [4], we aim to refine this line of work by focusing on the issue of the *execution cost* of

[1] CerCo project `http://cerco.cs.unibo.it`

the compiled code. Specifically, we aim to build a formally verified C compiler that given a source program produces automatically a functionally equivalent object code plus an annotation of the source code which is a sound and precise description of the execution cost of the object code.

We target in particular the kind of C programs produced for embedded applications; these programs are eventually compiled to binaries executable on specific processors. The current state of the art in commercial products such as Scade[2] [8] is that the *reaction time* of the program is estimated by means of abstract interpretation methods (such as those developed by AbsInt[3] [7]) that operate on the binaries. These methods rely on a specific knowledge of the architecture of the processor and may require explicit (and uncertified) annotations of the binaries to determine the number of times a loop is iterated (see, *e.g.*, [14] for a survey of the state of the art).

In this context, our aim is to produce a mechanically verified compiler which can *lift* in a provably correct way the pieces of information on the execution cost of the binary code to cost annotations on the source C code. Then the produced cost annotations are manipulated with the Frama − C[4] [5] automatic tool to infer synthetic cost annotations. We stress that the practical relevance of the proposed approach depends on the possibility of obtaining accurate information on the execution cost of relatively short sequences of binary instructions. This seems beyond the scope of current Worst-Case Execution Time (WCET) tools such as AbsInt or Chronos[5] which do not support a *compositional* analysis of WCET. For this reason, we focus on processors with a simple architecture for which manufacturers can provide accurate information on the execution cost of the binary instructions. In particular, our experiments are based on the 8051 [10][6]. This is a widely popular 8-bits processor developed by Intel for use in embedded systems with no cache and no pipeline. An important characteristic of the processor is that its cost model is 'additive': the cost of a sequence of instructions is exactly the sum of the costs of each instruction.

The rest of the paper is organized as follows. Section 2 describes the labelling approach and its formal application to a toy compiler. The report [2] gives standard definitions for the toy compiler and sketches the proofs. A formal and browsable Coq development composed of 1 *Kloc* of specifications and 3.5 *Kloc* of proofs is available at `http://www.pps.univ-paris-diderot.fr/cerco`. Section 3 reports our experience in implementing and testing the labelling approach for a compiler from C to 8051 binaries. The CerCo compiler is composed of 30 *Kloc* of ocaml code; it can be both downloaded and tested as a web application at the URL above. More details are available in report [2] Section 4 introduces the automatic Cost tool that starting from the cost annotations produces certified synthetic cost bounds. This is a Frama − C plug-in composed of 5 *Kloc* of ocaml code also available at the URL above.

[2] Esterel Technologies. `http://www.esterel-technologies.com`
[3] AbsInt Angewandte Informatik. `http://www.absint.com/`
[4] Frama − C software analyzers. `http://frama-c.com/`
[5] Chronos tool. `www.comp.nus.edu.sg/~rpembed/chronos`
[6] The recently proposed ARM Cortex M series would be another obvious candidate.

2 A "Labelling" Method for Cost Annotating Compilation

In this section, we explain in general terms the so-called "labelling" method to turn a compiler into a cost annotating compiler while minimizing the impact of this extension on the proof of the semantic preservation. Then to make our purpose technically precise, we apply the method to a toy compiler.

2.1 Overview

As a first step, we need a clear and flexible picture of: (i) the meaning of cost annotations, (ii) the method to provide them being sound and precise, and (iii) the way such proofs can be composed. The execution cost of the source programs we are interested in depends on their control structure. Typically, the source programs are composed of mutually recursive procedures and loops and their execution cost depends, up to some multiplicative constant, on the number of times procedure calls and loop iterations are performed. Producing a *cost annotation* of a source program amounts to:

– enrich the program with a collection of *global cost variables* to measure resource consumption (time, stack size, heap size,...)
– inject suitable code at some critical points (procedures, loops,...) to keep track of the execution cost.

Thus, producing a cost-annotation of a source program P amounts to build an *annotated program* $An(P)$ which behaves as P while self-monitoring its execution cost. In particular, if we do *not* observe the cost variables then we expect the annotated program $An(P)$ to be functionally equivalent to P. Notice that in the proposed approach an annotated program is a program in the source language. Therefore, the meaning of the cost annotations is automatically defined by the semantics of the source language and tools developed to reason on the source programs can be directly applied to the annotated programs too. Finally, notice that the annotated program $An(P)$ is *only* meant to *reason* on the execution cost of the unannotated program P and it will never be compiled or executed.

Soundness and precision of cost annotations. Suppose we have a functionally correct compiler $\mathcal{C}$ that associates with a program P in the source language a program $\mathcal{C}(P)$ in the object language. Further suppose we have some obvious way of defining the execution cost of an object code. For instance, we have a good estimate of the number of cycles needed for the execution of each instruction of the object code. Now, the annotation of the source program $An(P)$ is *sound* if its prediction of the execution cost is an upper bound for the 'real' execution cost. Moreover, we say that the annotation is *precise* with respect to the cost model if the *difference* between the predicted and real execution costs is bounded by a constant which only depends on the program.

Compositionality. In order to master the complexity of the compilation process (and its verification), the compilation function $\mathcal{C}$ must be regarded as the result of the composition of a certain number of program transformations $\mathcal{C} = \mathcal{C}_k \circ \cdots \circ \mathcal{C}_1$. When building a system of cost annotations on top of an existing compiler, a certain number of problems arise. First, the estimated cost of executing a piece of source code is determined only at the *end* of the compilation process. Thus, while we are used to define the compilation functions $\mathcal{C}_i$ in increasing order, the annotation function An is the result of a progressive abstraction from the object to the source code. Second, we must be able to foresee in the source language the looping and branching points of the object code. Missing a loop may lead to unsound cost annotations while missing a branching point may lead to rough cost predictions. This means that we must have a rather good idea of the way the source code will eventually be compiled to object code. Third, the definition of the annotation of the source code depends heavily on *contextual information.* For instance, the cost of the compiled code associated with a simple expression such as $x+1$ will depend on the place in the memory hierarchy where the variable x is allocated. A previous experience described in [1] suggests that the process of pushing 'hidden parameters' in the definitions of cost annotations and of manipulating directly numerical cost is error prone and produces complex proofs. For this reason, we advocate next a 'labelling approach' where costs are handled at an abstract level and numerical values are produced at the very end of the construction.

2.2 The Labelling Approach, Formally

The 'labelling' approach to the problem of building cost annotations is summarized in the following diagram.

$$\begin{tikzcd}
L_1 & L_{1,\ell} \arrow[l, "\mathcal{I}"'] \arrow[r, "\mathcal{C}_1"] \arrow[d, bend left, "er_1"] & L_{2,\ell} \arrow[d, "er_2"] & \ldots \arrow[r, "\mathcal{C}_k"] & L_{k+1,\ell} \arrow[d, "er_{k+1}"] \\
 & L_1 \arrow[u, bend left, "\mathcal{L}"] \arrow[r, "\mathcal{C}_1"] & L_2 & \ldots \arrow[r, "\mathcal{C}_k"] & L_{k+1}
\end{tikzcd}$$

$$\begin{array}{rcl}
er_{i+1} \circ \mathcal{C}_i & = & \mathcal{C}_i \circ er_i \\
er_1 \circ \mathcal{L} & = & id_{L_1} \\
An & = & \mathcal{I} \circ \mathcal{L}
\end{array}$$

For each language L_i considered in the compilation process, we define an extended *labelled* language $L_{i,\ell}$ and an extended operational semantics. The labels are used to mark certain points of the control. The semantics makes sure that whenever we cross a labelled control point a labelled and observable transition is produced.

For each labelled language there is an obvious function er_i erasing all labels and producing a program in the corresponding unlabelled language. The compilation functions $\mathcal{C}_i$ are extended from the unlabelled to the labelled language so that they enjoy commutation with the erasure functions. Moreover, we lift the soundness properties of the compilation functions from the unlabelled to the labelled languages and transition systems.

A *labelling* $\mathcal{L}$ of the source language L_1 is a function such that $er_{L_1} \circ \mathcal{L}$ is the identity function. An *instrumentation* $\mathcal{I}$ of the source labelled language $L_{1,\ell}$ is a function replacing the labels with suitable increments of, say, a fresh global *cost* variable. Then, an *annotation* An of the source program can be derived simply as the composition of the labelling and the instrumentation functions: $An = \mathcal{I} \circ \mathcal{L}$.

Suppose s is some adequate representation of the state of a program. Let P be a source program. The judgement $(P, s) \Downarrow s'$ is the big-step evaluation of P transforming state s into a state s'. Let us write $s[v/x]$ to denote a state s in which the variable x is assigned a value v. Suppose now that its annotation satisfies the following property:

$$(An(P), s[c/cost]) \Downarrow s'[c + \delta/cost] \tag{1}$$

where c and δ are some non-negative numbers. Then, the definition of the instrumentation and the fact that the soundness proofs of the compilation functions have been lifted to the labelled languages allows to conclude that

$$(\mathcal{C}(\mathcal{L}(P)), s[c/cost]) \Downarrow (s'[c/cost], \lambda) \tag{2}$$

where $\mathcal{C} = \mathcal{C}_k \circ \cdots \circ \mathcal{C}_1$ and λ is a sequence (or a multi-set) of labels whose 'cost' corresponds to the number δ produced by the annotated program. Then, the commutation properties of erasure and compilation functions allows to conclude that the *erasure* of the compiled labelled code $er_{k+1}(\mathcal{C}(\mathcal{L}(P)))$ is actually equal to the compiled code $\mathcal{C}(P)$ we are interested in. Given this, the following question arises: under which conditions the sequence λ, *i.e.*, the increment δ, is a sound and possibly precise description of the execution cost of the object code?

To answer this question, we observe that the object code we are interested in is some kind of assembly code and its control flow can be easily represented as a control flow graph. The idea is then to perform two simple checks on the control flow graph. The first check is to verify that all loops go through a labelled node. If this is the case then we can associate a finite cost with every label and prove that the cost annotations are sound. The second check amounts to verify that all paths starting from a label have the same cost. If this check is successful then we can conclude that the cost annotations are precise.

2.3 A Toy Compiler

As a first case study, we apply the labelling approach to a *toy compiler*.

The syntax of the source, intermediate and target languages is given in Figure 1. The three languages considered can be shortly described as follows: Imp is a very simple imperative language with pure expressions, branching and looping commands, Vm is an assembly-like language enriched with a stack, and Mips is a Mips-like assembly language [9] with registers and main memory.

The semantics of Imp is defined over configurations (S, K, s) where S is a statement, K is a continuation and s is a state. A *continuation* K is a list of

Syntax for Imp

$$\begin{array}{rl}
id & ::= x \parallel y \parallel \ldots \\
n & ::= 0 \parallel -1 \parallel +1 \parallel \ldots \\
v & ::= n \parallel \mathsf{true} \parallel \mathsf{false} \\
e & ::= id \parallel n \parallel e + e \\
b & ::= e < e \\
S & ::= \mathsf{skip} \parallel id := e \parallel S;S \\
 & \parallel \ \mathsf{if}\ b\ \mathsf{then}\ S\ \mathsf{else}\ S \\
 & \parallel \ \mathsf{while}\ b\ \mathsf{do}\ S \\
P & ::= \mathsf{prog}\ S
\end{array}$$

Syntax for Vm

$$\begin{array}{rl}
instr_{\mathsf{Vm}} & ::= \mathtt{cnst}(n) \parallel \mathtt{var}(n) \\
 & \parallel \ \mathtt{setvar}(n) \parallel \mathtt{add} \\
 & \parallel \ \mathtt{branch}(k) \parallel \mathtt{bge}(k) \parallel \mathtt{halt}
\end{array}$$

Syntax for Mips

$$\begin{array}{rl}
instr_{\mathsf{Mips}} & ::= \mathtt{loadi}\ R, n \parallel \mathtt{load}\ R, l \\
 & \parallel \ \mathtt{store}\ R, l \parallel \mathtt{add}\ R, R, R \\
 & \parallel \ \mathtt{branch}\ k \parallel \mathtt{bge}\ R, R, k \parallel \mathtt{halt}
\end{array}$$

Fig. 1. Syntax definitions

commands which terminates with a special symbol halt. The semantics of Vm is defined over stack-based machine configurations $C \vdash (i, \sigma, s)$ where C is a program, i is a program counter, σ is a stack and s is a state. The semantics of Mips is defined over register-based machine configurations $C \vdash (i, m)$ where C is a program, i is a program counter and m is a machine memory (with registers and main memory).

The first compilation function $\mathcal{C}$ relies on the stack of the Vm language to implement expression evaluation while the second compilation function $\mathcal{C}'$ allocates (statically) the base of the stack in the registers and the rest in main memory. This is of course a naive strategy but it suffices to expose some of the problems that arise in defining a compositional approach. The formal definitions of these compilation functions $\mathcal{C}$ from Imp to Vm and $\mathcal{C}'$ from Vm to Mips are standard and thus eluded. (See report [2] for formal details about semantics and the compilation chain.)

Applying the labelling approach to this toy compiler results in the following diagram. The next sections aim at describing this diagram in details.

$$\begin{tikzcd}
\mathsf{Imp} & \mathsf{Imp}_\ell \arrow[l, "\mathcal{I}"'] \arrow[r, "\mathcal{C}"] \arrow[d, bend left, "er_{\mathsf{Imp}}"] & \mathsf{Vm}_\ell \arrow[r, "\mathcal{C}'"] \arrow[d, "er_{\mathsf{Vm}}"] & \mathsf{Mips}_\ell \arrow[d, "er_{\mathsf{Mips}}"] \\
 & \mathsf{Imp} \arrow[u, bend left, "\mathcal{L}"] \arrow[r, "\mathcal{C}"] & \mathsf{Vm} \arrow[r, "\mathcal{C}'"] & \mathsf{Mips}
\end{tikzcd}$$

$$\begin{array}{rcl}
er_{\mathsf{Vm}} \circ \mathcal{C} & = & \mathcal{C} \circ er_{\mathsf{Imp}} \\
er_{\mathsf{Mips}} \circ \mathcal{C}' & = & \mathcal{C}' \circ er_{\mathsf{Vm}} \\
er_{\mathsf{Imp}} \circ \mathcal{L} & = & id_{\mathsf{Imp}} \\
An_{\mathsf{Imp}} & = & \mathcal{I} \circ \mathcal{L}
\end{array}$$

2.4 Labelled languages: Syntax and Semantics

Syntax The syntax of Imp is extended so that statements can be labelled: $S ::= \ldots \parallel \ell : S$. A new instruction $\mathsf{emit}(\ell)$ (resp. $(\mathsf{emit}\ \ell)$) is introduced in the syntax of Vm (resp. Mips).

Semantics. The small step semantics of Imp statements is extended as described by the following rule.

$$(\ell : S, K, s) \xrightarrow{\ell} (S, K, s)$$

We denote with $\lambda, \lambda', \ldots$ finite sequences of labels. In particular, the empty sequence is written ϵ. We also identify an unlabelled transition with a transition labelled with ϵ. Then, the small step reduction relation we have defined on statements becomes a *labelled transition system*. We derive a *labelled* big-step semantics as follows: $(S, s) \Downarrow (s', \lambda)$ if $(S, \mathsf{halt}, s) \xrightarrow{\lambda_1} \cdots \xrightarrow{\lambda_n} (\mathsf{skip}, \mathsf{halt}, s')$ and $\lambda = \lambda_1 \cdots \lambda_n$.

Following the same pattern, the small step semantics of Vm and Mips are turned into a labelled transition system as follows:

$$\begin{array}{ll} C \vdash (i, \sigma, s) \xrightarrow{\ell} (i+1, \sigma, s) & \text{if } C[i] = \mathsf{emit}(\ell)\ . \\ M \vdash (i, m) \xrightarrow{\ell} (i+1, m) & \text{if } M[i] = (\mathsf{emit}\ \ell)\ . \end{array}$$

The evaluation predicate for labelled Vm is defined as $(C, s) \Downarrow (s', \lambda)$ if $C \vdash (0, \epsilon, s) \xrightarrow{\lambda_1} \cdots \xrightarrow{\lambda_n} (i, \epsilon, s')$, $\lambda = \lambda_1 \cdots \lambda_n$ and $C[i] = \mathsf{halt}$. The evaluation predicate for labelled Mips is defined as $(M, m) \Downarrow (m', \lambda)$ if $M \vdash (0, m) \xrightarrow{\lambda_1} \cdots \xrightarrow{\lambda_n} (j, m')$, $\lambda = \lambda_1 \cdots \lambda_n$ and $M[j] = \mathsf{halt}$.

2.5 Erasure Functions

There is an obvious *erasure* function er_{Imp} from the labelled language to the unlabelled one which is the identity on expressions and boolean conditions, and traverses commands removing all labels.

The erasure function er_{Vm} amounts to remove from a Vm code C all the $\mathsf{emit}(\ell)$ instructions and recompute jumps accordingly. Specifically, let $n(C, i, j)$ be the number of emit instructions in the interval $[i, j]$. Then, assuming $C[i] = \mathsf{branch}(k)$ we replace the offset k with an offset k' determined as follows:

$$k' = \begin{cases} k - n(C, i, i+k) & \text{if } k \geq 0 \\ k + n(C, i+1+k, i) & \text{if } k < 0 \end{cases}$$

The *erasure function* er_{Mips} is also similar to the one of Vm as it amounts to remove from a Mips code all the $(\mathsf{emit}\ \ell)$ instructions and recompute jumps accordingly. The compilation function $\mathcal{C}'$ is extended to Vm_ℓ by simply translating $\mathsf{emit}(\ell)$ as $(\mathsf{emit}\ \ell)$:

$$\mathcal{C}'(i, C) = (\mathsf{emit}\ \ell) \text{ if } C[i] = \mathsf{emit}(\ell)$$

2.6 Compilation of Labelled Languages

The compilation function $\mathcal{C}$ is extended to Imp_ℓ by defining:

$$\mathcal{C}(\ell : b, k) = (\mathsf{emit}(\ell)) \cdot \mathcal{C}(b, k) \qquad \mathcal{C}(\ell : S) = (\mathsf{emit}(\ell)) \cdot \mathcal{C}(S)\ .$$

Proposition 1. *For all commands S in Imp_ℓ, we have that:*
(1) $er_{\mathsf{Vm}}(\mathcal{C}(S)) = \mathcal{C}(er_{\mathsf{Imp}}(S))$.
(2) *If* $(S, s) \Downarrow (s', \lambda)$ *then* $(\mathcal{C}(S), s) \Downarrow (s', \lambda)$.

The following proposition relates Vm_ℓ code and its compilation and it is similar to proposition 1. Here $m \Vdash \sigma, s$ means "the low-level Mips memory m realizes the Vm stack σ and state s".

Proposition 2. *Let C be a Vm_ℓ code. Then:*
(1) $er_{\mathsf{Mips}}(\mathcal{C}'(C)) = \mathcal{C}'(er_{\mathsf{Vm}}(C))$.
(2) *If* $(C, s) \Downarrow (s', \lambda)$ *and* $m \Vdash \epsilon, s$ *then* $(\mathcal{C}'(C), m) \Downarrow (m', \lambda)$ *and* $m' \Vdash \epsilon, s'$.

2.7 Labellings and Instrumentations

Assuming a function κ which associates an integer number with labels and a distinct variable *cost* which does not occur in the program P under consideration, we abbreviate with $inc(\ell)$ the assignment $cost := cost + \kappa(\ell)$. Then we define the instrumentation $\mathcal{I}$ (relative to κ and *cost*) as follows:

$$\mathcal{I}(\ell : S) = inc(\ell); \mathcal{I}(S) \ .$$

The function $\mathcal{I}$ just distributes over the other operators of the language. We extend the function κ on labels to sequences of labels by defining $\kappa(\ell_1, \ldots, \ell_n) = \kappa(\ell_1) + \cdots + \kappa(\ell_n)$. The instrumented Imp program relates to the labelled one as follows.

Proposition 3. *Let S be an Imp_ℓ command. If $(\mathcal{I}(S), s[c/cost]) \Downarrow s'[c+\delta/cost]$ then $\exists\, \lambda \;\; \kappa(\lambda) = \delta$ and $(S, s[c/cost]) \Downarrow (s'[c/cost], \lambda)$.*

Definition 1. *A* labelling *is a function $\mathcal{L}$ from an unlabelled language to the corresponding labelled one such that $er_{\mathsf{Imp}} \circ \mathcal{L}$ is the identity function on the* Imp *language.*

Proposition 4. *For any labelling function $\mathcal{L}$, and* Imp *program P, the following holds:*

$$er_{\mathsf{Mips}}(\mathcal{C}'(\mathcal{C}(\mathcal{L}(P))) = \mathcal{C}'(\mathcal{C}(P)) \ . \quad (3)$$

Proposition 5. *Given a function κ for the labels and a labelling function $\mathcal{L}$, for all programs P of the source language if $(\mathcal{I}(\mathcal{L}(P)), s[c/cost]) \Downarrow s'[c+\delta/cost]$ and $m \Vdash \epsilon, s[c/cost]$ then $(\mathcal{C}'(\mathcal{C}(\mathcal{L}(P))), m) \Downarrow (m', \lambda)$, $m' \Vdash \epsilon, s'[c/cost]$ and $\kappa(\lambda) = \delta$.*

2.8 Sound and Precise Labellings

With any Mips_ℓ code M, we can associate a directed and rooted (control flow) graph whose nodes are the instruction positions $\{0, \ldots, |M| - 1\}$, whose root is the node 0, and whose directed edges correspond to the possible transitions between instructions. We say that a node is labelled if it corresponds to an instruction $\mathsf{emit}\ \ell$.

Definition 2. *A* simple path *in a* Mips_ℓ *code* M *is a directed finite path in the graph associated with* M *where the first node is labelled, the last node is the predecessor of either a labelled node or a leaf, and all the other nodes are unlabelled.*

Definition 3. *A* Mips_ℓ *code* M *is* soundly labelled *if in the associated graph the root node* 0 *is labelled and there are no loops that do not go through a labelled node. Besides, we say that a soundly labelled code is* precise *if for every label* ℓ *in the code, the simple paths starting from a node labelled with* ℓ *have the same cost.*

In a soundly labelled graph there are finitely many simple paths. Thus, given a soundly labelled Mips code M, we can associate with every label ℓ a number $\kappa(\ell)$ which is the maximum (estimated) cost of executing a simple path whose first node is labelled with ℓ. Thus for a soundly labelled Mips code the sequence of labels associated with a computation is a significant information on the execution cost.

For an example of command which is not soundly labelled, consider ℓ : $\mathsf{while}\ 0 < x\ \mathsf{do}\ x := x + 1$, which when compiled, produces a loop that does not go through any label. On the other hand, for an example of a program which is not precisely labelled consider $\ell : (\mathsf{if}\ 0 < x\ \mathsf{then}\ x := x + 1\ \mathsf{else}\ \mathsf{skip})$. In the compiled code, we find two simple paths associated with the label ℓ whose cost will be quite different in general.

Proposition 6. *If* M *is soundly (resp. precisely) labelled and* $(M, m) \Downarrow (m', \lambda)$ *then the cost of the computation is bounded by* $\kappa(\lambda)$ *(resp. is exactly* $\kappa(\lambda)$*).*

The next point we have to check is that there are labelling functions (of the source code) such that the compilation function does produce sound and possibly precise labelled Mips code. To discuss this point, we introduce in table 1 a labelling function $\mathcal{L}_p$ for the Imp language. This function relies on a function "*new*" which is meant to return fresh labels and on an auxiliary function $\mathcal{L'}_p$ which returns a labelled command and a binary directive $d \in \{0, 1\}$. If $d = 1$ then the command that follows (if any) must be labelled.

Table 1. A labelling for the Imp language

$$\begin{aligned}
&\mathcal{L}_p(\mathsf{prog}\ S) &&= \mathsf{prog}\ \mathcal{L}_p(S)\\
&\mathcal{L}_p(S) &&= \mathit{let}\ \ell = \mathit{new},\ (S', d) = \mathcal{L'}_p(S)\ \mathit{in}\ \ell : S'\\
&\mathcal{L'}_p(S) &&= (S, 0) \quad \text{if } S = \mathsf{skip} \text{ or } S = (x := e)\\
&\mathcal{L'}_p(\mathsf{if}\ b\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2) &&= (\mathsf{if}\ b\ \mathsf{then}\ \mathcal{L}_p(S_1)\ \mathsf{else}\ \mathcal{L}_p(S_2), 1)\\
&\mathcal{L'}_p(\mathsf{while}\ b\ \mathsf{do}\ S) &&= (\mathsf{while}\ b\ \mathsf{do}\ \mathcal{L}_p(S), 1)\\
&\mathcal{L'}_p(S_1; S_2) &&= \mathit{let}\ (S'_1, d_1) = \mathcal{L'}_p(S_1),\ (S'_2, d_2) = \mathcal{L'}_p(S_2)\ \mathit{in}\\
& &&\quad\ \mathit{case}\ d_1\\
& &&\quad\ 0 : (S'_1; S'_2, d_2)\\
& &&\quad\ 1 : \mathit{let}\ \ell = \mathit{new}\ \mathit{in}\ (S'_1; \ell : S'_2, d_2)
\end{aligned}$$

Proposition 7. *For all* Imp *programs* P, $\mathcal{C}'(\mathcal{C}(\mathcal{L}_p(P))$ *is a soundly and precisely labelled* Mips *code.*

Once a sound and possibly precise labelling $\mathcal{L}$ has been designed, we can determine the cost of each label and define an instrumentation $\mathcal{I}$ whose composition with $\mathcal{L}$ will produce the desired cost annotation.

Definition 4. *Given a labelling function* $\mathcal{L}$ *for the source language* Imp *and a program* P *in the* Imp *language, we define an annotation for the source program as follows:*

$$An_{\mathsf{Imp}}(P) = \mathcal{I}(\mathcal{L}(P)) \ .$$

Proposition 8. *If* P *is a program and* $\mathcal{C}'(\mathcal{C}(\mathcal{L}(P)))$ *is a sound (sound and precise) labelling then* $(An_{\mathsf{Imp}}(P), s[c/cost]) \Downarrow s'[c+\delta/cost]$ *and* $m \Vdash \epsilon, s[c/cost]$ *entails that* $(\mathcal{C}'(\mathcal{C}(P)), m) \Downarrow m'$, $m' \Vdash \epsilon, s'[c/cost]$ *and the cost of the execution is bounded by (is exactly)* δ.

3 A C Compiler Producing Cost Annotations

We now consider an untrusted C compiler prototype in ocaml in order to experiment with the scalability of our approach. Its architecture is described below:

$$\begin{tikzcd}
& \mathsf{C} \arrow[r] & \mathsf{Clight} \arrow[r] & \mathsf{Cminor} \arrow[r] & \mathsf{RTLAbs} \arrow[d] & \text{(front end)} \\
\mathsf{Mips}\text{ or }\mathsf{8051} & \mathsf{LIN} \arrow[l] & \mathsf{LTL} \arrow[l] & \mathsf{ERTL} \arrow[l] & \mathsf{RTL} \arrow[l] & \text{(back-end)}
\end{tikzcd}$$

The most notable difference with CompCert [11] is that we target the Intel 8051 [10] and Mips assembly languages (rather than PowerPc). The compilation from C to Clight relies on the CIL front-end [13]. The one from Clight to RTL has been programmed from scratch and it is partly based on the Coq definitions available in the CompCert compiler. Finally, the back-end from RTL to Mips is based on a compiler developed in ocaml for pedagogical purposes[7]; we extended this back-end to target the Intel 8051. The main optimizations the back-end performs are liveness analysis and register allocation, and graph compression. We ran some benchmarks to ensure that our prototype implementation is realistic. The results are given in report [2].

This section informally describes the labelled extensions of the languages in the compilation chain (see report [2] for details), the way the labels are propagated by the compilation functions, and the (sound and precise) labelling of the source code. A related experiment concerning a higher-order functional language of the ML family is described in [3].

3.1 Labelled Languages

Both the Clight and Cminor languages are extended in the same way by labelling both statements and expressions (by comparison, in the toy language Imp we

[7] `http://www.enseignement.polytechnique.fr/informatique/INF564/`

just used labelled statements). The labelling of expressions aims to capture precisely their execution cost. Indeed, Clight and Cminor include expressions such as $a_1?a_2;a_3$ whose evaluation cost depends on the boolean value a_1. As both languages are extended in the same way, the extended compilation does nothing more than sending Clight labelled statements and expressions to those of Cminor.

The labelled versions of RTLAbs and the languages in the back-end simply consist in adding a new instruction whose semantics is to emit a label without modifying the state. For the CFG based languages (RTLAbs to LTL), this new instruction is emit *label* $\to$ *node*. For LIN, Mips and 8051, it is emit *label*. The translation of these label instructions is immediate.

3.2 Labelling of the Source Language

As for the toy compiler, the goals of a labelling are soundness, precision, and possibly economy. Our labelling for Clight resembles that of Imp for their common instructions (e.g. loops). We only consider the instructions of Clight that are not present in Imp[8].

Ternary expressions. They may introduce a branching in the control flow. We achieve precision by associating a label with each branch.

Program Labels and Gotos. Program labels and gotos are intraprocedural. Their only effect on the control flow is to potentially introduce an unguarded loop. This loop must contain at least one cost label in order to satisfy the soundness condition, which we ensure by adding a cost label right after each program label.

Function calls. In the general case, the address of the callee cannot be inferred statically. But in the compiled assembly code, we know for a fact that the callee ends with a return statement that transfers the control back to the instruction following the function call in the caller. As a result, we treat function calls according to the following invariants: (1) the instructions of a function are covered by the labels inside this function, (2) we assume a function call always returns and runs the instruction following the call. Invariant (1) entails in particular that each function must contain at least one label. Invariant (2) is of course an over-approximation of the program behavior as a function might fail to return because of an error or an infinite loop. In this case, the proposed labelling remains correct: it just assumes that the instructions following the function call will be executed, and takes their cost into consideration. The final computed cost is still an over-approximation of the actual cost.

4 A Tool for Reasoning on Cost Annotations

Frama – C is a set of analysers for C programs with a specification language called ACSL. New analyses can be dynamically added through a plug-in system.

[8] We do not consider expressions with side-effects because they are eliminated by CIL.

For instance, the Jessie plug-in allows deductive verification of C programs with respect to their specification in ACSL, with various provers as back-end tools.

We developed the Cost plug-in for the Frama – C platform as a proof of concept of an automatic environment exploiting the cost annotations produced by the CerCo compiler. It consists of an ocaml program of 5 *Kloc* which in first approximation takes the following actions: (1) it receives as input a C program, (2) it applies the CerCo compiler to produce a related C program with cost annotations, (3) it applies some heuristics to produce a tentative bound on the cost of executing the C functions of the program as a function of the value of their parameters, (4) the user can then call the Jessie tool to discharge the related proof obligations. In the following we elaborate on the soundness of the framework, the algorithms underlying the plug-in, and the experiments we performed with the Cost tool.

4.1 Soundness

The soundness of the whole framework depends on the cost annotations added by the CerCo compiler, the synthetic costs produced by the Cost plug-in, the verification conditions (VCs) generated by Jessie, and the external provers discharging the VCs. The synthetic costs being in ACSL format, Jessie can be used to verify them. Thus, even if the added synthetic costs are incorrect (relatively to the cost annotations), the process in its globality is still correct: indeed, Jessie will not validate incorrect costs and no conclusion can be made about the WCET of the program in this case. In other terms, the soundness does not really depend on the action of the Cost plug-in, which can in principle produce *any* synthetic cost. However, in order to be able to actually prove a WCET of a C function, we need to add correct annotations in a way that Jessie and subsequent automatic provers have enough information to deduce their validity. In practice this is not straightforward even for very simple programs composed of branching and assignments (no loops and no recursion) because a fine analysis of the VCs associated with branching may lead to a complexity blow up.

4.2 Inner Workings

The cost annotations added by the CerCo compiler take the form of C instructions that update by a constant a fresh global variable called the *cost variable*. Synthesizing a WCET of a C function thus consists in statically resolving an upper bound of the difference between the value of the cost variable before and after the execution of the function, i.e. find in the function the instructions that update the cost variable and establish the number of times they are passed through during the flow of execution. The plug-in proceeds as follows.

– Each function is independently processed and is associated a WCET that may depend on the cost of the other functions. This is done with a mix between abstract interpretation [6] and syntactic recognition of specific loops.

- As result of the previous step, a system of inequations is built and its solution is attempted by an iterative process. At each iteration, one replaces in all the inequations the references to the cost of a function by its associated cost if it is independent of the other functions. This step is repeated till a fixpoint is reached.
- ACSL annotations are added to the program according to the result of the above fixpoint. The two previous steps may fail in finding a concrete WCET for some functions, because of imprecision inherent in abstract interpretation, and because of recursive definitions in the source program not solved by the fixpoint. At each program point that requires an annotation (function definitions and loops), annotations are added if a solution was found for the program point.
- The most difficult instructions to handle are loops. We consider loops for which we can syntactically find a counter (its initial, increment and last values are domain dependent). Other loops are associated an undefined cost ($\top$). When encountering a loop, the analysis first sets the cost of its entry point to 0. The cost inside the loop is thus relative to the loop. Then, for each exit point, we fetch the value of the cost at that point and multiply it by an upper bound of the number of iterations (obtained through arithmetic over the initial, increment and last values of the counter); this results in an upper bound of the cost of the whole loop, which is sent to the successors of the considered exit point.

Figure 2 shows the action of the Cost plug-in on a C program. The most notable differences are the added so-called *cost variable*, some associated update (increment) instructions inside the code, and an `ensures` clause that specifies the WCET of the `is_sorted` function with respect to the cost variable. One can notice that this WCET depends on the inputs of the function. Running Jessie on the annotated and specified program generates VCs that are all proved by the automatic prover AltErgo[9].

4.3 Experiments

The Cost plug-in has been developed in order to validate CerCo's framework for modular WCET analysis. The plug-in allows (semi-)automatic generation and certification of WCET for C programs. Also, we designed a wrapper for supporting Lustre files. Indeed, Lustre is a data-flow language to program synchronous systems and the language comes with a compiler to C. The C function produced by the compiler implements the *step function* of the synchronous system and computing the WCET of the function amounts to obtain a bound on the reaction time of the system.

We tested the Cost plug-in and the Lustre wrapper on the C programs generated by the Lustre compiler. We also tested it on some basic algorithms and cryptographic functions; these examples, unlike those generated by the Lustre

[9] AltErgo prover. `http://ergo.lri.fr/`

```
int is_sorted (int *tab, int size) {
  int i, res = 1;
  for (i = 0 ; i < size-1 ; i++) if (tab[i] > tab[i+1]) res = 0;
  return res; }
```

(a) The initial C source code.

```
int _cost = 0;

/*@ ensures (_cost ≤ \old(_cost)+(101+(0<size-1?(size-1)*195:0))); */
int is_sorted (int *tab, int size) {
  int i, res = 1, _cost_tmp0;
  _cost += 97; _cost_tmp0 = _cost;
  /*@ loop invariant (0 < size-1) ⇒ (i ≤ size-1);
    @ loop invariant (0 ≥ size-1) ⇒ (i ≡ 0);
    @ loop invariant (_cost ≤ _cost_tmp0+i*195);
    @ loop variant (size-1)-i; */
  for (i = 0; i < size-1; i++) {
    _cost += 91;
    if (tab[i] > tab[i+1]) { _cost += 104; res = 0; }
    else _cost += 84; }
  _cost += 4; return res; }
```

(b) The annotated source code generated by Cost.

Fig. 2. An example of the Cost plug-in action

File	Type	Description	LOC	VCs
`3-way.c`	C	Three way block cipher	144	34
`a5.c`	C	A5 stream cipher, used in GSM cellular	226	18
`array_sum.c`	S	Sums the elements of an integer array	15	9
`fact.c`	S	Factorial function, imperative implementation	12	9
`is_sorted.c`	S	Sorting verification of an array	8	8
`LFSR.c`	C	32-bit linear-feedback shift register	47	3
`minus.c`	L	Two modes button	193	8
`mmb.c`	C	Modular multiplication-based block cipher	124	6
`parity.lus`	L	Parity bit of a boolean array	359	12
`random.c`	C	Random number generator	146	3
S: standard algorithm C: cryptographic function L: C generated from a Lustre file				

Fig. 3. Experiments on CerCo and the Cost plug-in

compiler include arrays and for-loops. Table 3 provides a list of concrete programs and describes their type, functionality, the number of lines of the source code, and the number of VCs generated. In each case, the Cost plug-in computes a WCET and AltErgo is able to discharge all VCs. Obviously the generation of synthetic costs is an undecidable and open-ended problem. Our experience just shows that there are classes of C programs which are relevant for embedded applications and for which the synthesis and verification tasks can be completely automatized.

Acknowledgement. The master students Kayvan Memarian and Ronan Saillard contributed both to the Coq proofs and the CerCo compiler in the early stages of their development.

References

1. Amadio, R.M., Ayache, N., Memarian, K., Saillard, R., Régis-Gianas, Y.: Compiler Design and Intermediate Languages. Deliverable 2.1 of [4]
2. Ayache, N., Amadio, R.M., Régis-Gianas, Y.: Certifying and reasoning on cost annotations of C programs. Research Report 00702665 (June 2012)
3. Amadio, R.M., Régis-Gianas, Y.: Certifying and Reasoning on Cost Annotations of Functional Programs. In: Peña, R., van Eekelen, M., Shkaravska, O. (eds.) FOPARA 2011. LNCS, vol. 7177, pp. 72–89. Springer, Heidelberg (2012)
4. Certified complexity (Project description). ICT-2007.8.0 FET Open, Grant 243881
5. Correnson, L., Cuoq, P., Kirchner, F., Prevosto, V., Puccetti, A., Signoles, J., Yakobowski, B.: Frama-C user manual. CEA-LIST, Software Safety Laboratory, Saclay, F-91191, http://frama-c.com/
6. Cousot, P., Cousot, R.: Abstract Interpretation Frameworks. Jou. of Logic and Computation 2(4), 511–547 (1992)
7. Ferdinand, C., Heckmann, R., Le Sergent, T., Lopes, D., Martin, B., Fornari, X., Martin, F.: Combining a high-level design tool for safety-critical systems with a tool for WCET analysis of executables. In: Embedded Real Time Software (2008)
8. Fornari, X.: Understanding how SCADE suite KCG generates safe C code. White paper, Esterel Technologies (2010)
9. Larus, J.: Assemblers, linkers, and the SPIM simulator. Appendix of Computer Organization and Design: the hw/sw interface. Hennessy and Patterson (2005)
10. MCS 51 Microcontroller Family User's Manual. Publication number 121517. Intel Corporation (1994)
11. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM 52(7), 107–115 (2009)
12. Leroy, X.: Mechanized semantics, with applications to program proof and compiler verification. In: Marktoberdorf Summer School (2009)
13. Necula, G., McPeak, S., Rahul, S.P., Weimer, W.: CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In: Horspool, R.N. (ed.) CC 2002. LNCS, vol. 2304, pp. 213–228. Springer, Heidelberg (2002)
14. Wilhelm, R., et al.: The worst-case execution-time problem - overview of methods and survey of tools. ACM Trans. Embedded Comput. Syst. 7(3) (2008)

Waiting for Locks: How Long Does It Usually Take?*

Christel Baier[1], Marcus Daum[1], Benjamin Engel[2], Hermann Härtig[2], Joachim Klein[1], Sascha Klüppelholz[1], Steffen Märcker[1], Hendrik Tews[2], and Marcus Völp[2]

[1] Institute for Theoretical Computer Science
[2] Operating-Systems Group
Technische Universität Dresden, Germany

Abstract. Reliability of low-level operating-system (OS) code is an indispensable requirement. This includes functional properties from the safety-liveness spectrum, but also quantitative properties stating, e.g., that the average waiting time on locks is sufficiently small or that the energy requirement of a certain system call is below a given threshold with a high probability. This paper reports on our experiences made in a running project where the goal is to apply probabilistic model checking techniques and to align the results of the model checker with measurements to predict quantitative properties of low-level OS code.

1 Introduction

For safety-critical systems such as space, flight, and automotive control systems one wants correctness guarantees for the software not only for the functional behaviour of components but also, e.g., for their timing behaviour. Worst-case execution-time analyses (see e.g. [3,11,22]) are able to provide these guarantees, but only in the form of upper bounds on the execution times of all involved components, which hold even in the most extreme situations. Many computer systems are however either not safety critical or they include fail-safe mechanisms that prevent damage in highly exceptional situations. Quantitative analyses can provide detailed information on the probabilities of certain events or on the average behaviour. First, the requirement that certain requirements hold for all possible execution sequences is a very strong condition. E.g., for uncontended locks the property that a process will find a lock free without waiting might not hold universally, but with some high probability. Second, probabilistic features are crucial for the evaluation of complex architectures such as x86 that are optimised according to their average-performance, for systems that rely on imprecise real-time computing techniques to deal with transient overload [18,6], or for systems that may fail in extreme cases. Third, the probabilistic analysis may guide OS level optimisation justifying, for instance, the use of a simple test-and-test-and-set (TTS) lock implementation over more complex ticket [1] or queue-based locks [15].

* This work was in part funded by the German Research Council (DFG) through the QuaOS project and the collaborative research center 912 Highly-Adaptive Energy-Efficient Computing (HAEC).

In this paper, we report on a running project of the operating-system and the formal-methods group at TU Dresden whose aim it is to establish quantitative properties of low-level operating-system (OS) code using probabilistic model checking techniques. By low-level OS code, we mean drivers, the kernel of monolithic operating systems, microkernels or microhypervisors and similar code that directly interacts with hardware devices and that is therefore often optimised to fully exploit the intrinsic behaviour of modern processor architectures. Although the applicability of probabilistic model checking techniques is expected to work in principle, several non-trivial problems have to be addressed.

Modelling. The first challenge is to find a reasonable abstraction level for the formal model based on the probabilistic analysis that will be carried out. E.g., there are several details on the realisation of hardware primitives (such as caches, busses and controllers of the memory subsystem) that have impact on the timing behaviour of low-level OS code. The model must cover all features that dominate the quantitative behaviour, while still being compact enough for the algorithmic analysis. The latter requires to abstract away from details that have negligible impact on the quantitative behaviour or that would render the model unmanageable. The abstraction of many of these details is indispensable because only little information on the hardware realisation is available, and even if these details are known, too fine-grained hardware models make the state-explosion problem unscalable and lead to quantitative results that are too hardware specific. Instead, we incorporate hardware timing effects in the distributions for the execution times and use measurement-based simulation techniques to obtain empirical evidence for the models and the model checking results.

Measurement-based simulation. Generating this evidence from measurement data is the second major challenge because the quantities of interest for many relevant OS-level properties are in a range where measurements significantly disturb the normal system behaviour and where instrumentation-induced noise blurs the results. For instance, the update rate and resolution of CPU-internal energy sensors necessitate a statistical analysis over a multitude of measurements to extract an energy profile for a single system call [5]. To counteract these effects and to obtain empirical evidence for the models and model checking results, we construct microbenchmarks that place the to-be-measured code into a manageable environment and that mimic the formalisation as close as possible.

Quantitative properties. A third major step is the identification of the types of quantitative properties that are relevant for low-level OS code. At a first glance, it seems that constrained reachability conditions such as "what is the probability for threads to find the requested resource locked for longer than 1 microsecond?" can be expressed as probabilistic queries of the form $\mathbb{P}_{=?}(\varphi)$ using comparably simple patterns of path formulas φ in standard temporal logics, such as PCTL. (The notation $\mathbb{P}_{=?}(\varphi)$ refers to the probability for the event specified by φ.) However, the main interest is in deducing probabilities of this kind for the *long run* rather than for fixed initial distributions. Typically, the long-run behaviour of programs (e.g., the time programs hold a certain resource) shows fundamentally different characteristics when compared to the initialisation-phase behaviour (when resources are requested for the first time). These differences are caused by the fact that the system had time to learn and adjust to the program characteristics, e.g., by warming up the disk or processor caches or by adjusting the scheduling parameters of the program to meet its responsiveness and interactivity demands. Queries for questions of the above type must therefore be able to ask for long-run probabilities of path formulas *under the condition* that the system is in a certain set of states. The above question translates into a condition about the states in the long run (e.g., the probability of finding a resource held in the long run) and a temporal formula on the paths starting in these states (e.g., "what is the likelihood that such a held resource will be released and granted to the requesting thread within 1 microsecond?"). We refer to these type of queries as *conditional steady-state queries*. A second class of important queries for low-level OS code asks for the value of a quantity that is not exceeded in the majority of all cases: the *quantile*. Two examples of important quantile-based queries are "how long does a thread wait for a resource in 99.9% of all cases?" and "what is the energy that must remain in the battery to guarantee the complete playback of a certain video with a probability greater than 95%?". To our surprise, neither conditional steady-state queries nor quantile-based queries are supported by state-of-the-art probabilistic model checkers.

Outline. In this paper, we report on our experiences with a simple TTS spinlock as a starting point and initial experiment for a more elaborate investigation of the general feasibility of probabilistic model checking techniques and the limitations of existing tool support. Sec. 2 presents the TTS spinlock and a set of relevant quantitative properties. Sec. 3 explains our discrete-time Markov chain model. Sec. 4 presents the results of the quantitative analysis that we have carried out with the model checker PRISM [13]. We explain how we dealt with conditional long-run probabilities and quantile-based properties and report on the results of the measurement-based simulation and the lessons learned. Sec. 5 concludes this paper. Due to space limitations, we present the detailed model checking statistics in an extended version[1] and provide here only a brief summary.

2 A Test-And-Test-And-Set Lock

Fig. 1 shows the C/C++ code of a simple TTS lock. To acquire the lock, the requesting process executes the atomic swap operation in Line 3 to atomically read the value of the shared variable `occupied` and to then set it to `true`. The loop exits if the process was first to perform this swap after another process has released the lock by setting `occupied` to `false`. For as long as in Line 4 `occupied` is `true`, the process only reads this variable to avoid unnecessary contention on the core-to-core interconnect.

```
1  volatile bool occupied = false;
2  void lock(){
3     while(atomic_swap(occupied,true)){
4        while(occupied){}
5     } }
6  void unlock(){
7     occupied = false
8  }
```

Fig. 1. Simple TTS spinlock

[1] http://wwwtcs.inf.tu-dresden.de/ALGI/spinlock-FMICS2012.pdf

Properties. For our case study, we investigate four questions as representatives for complex conditional long-run and quantile-based properties:

(A1) probability that a process finds a free lock when it seeks to acquire this lock;

(A2) probability of re-acquiring a previously held lock (without spinning) without other processes having acquired the lock in the meantime;

(A3) expected amount of time a process waits for a lock;

(A4) the 95% quantile of the time processes wait for a lock.

Properties such as (A1)–(A4), which characterise the quantitative behaviour of locks, are highly relevant to guide design decisions and optimisation of low-level OS code. For instance, high probabilities in (A1) and (A2) justify the use of less complex lock implementations, respectively of simpler execution-time analyses. An analysis which assumes low fixed costs for acquiring and releasing a lock is justified by a high probability of (A2) because cache eviction of the `occupied` variable is unlikely. The expected waiting time (A3) is important to judge whether a lock implementation is suitable for the common cases of a given scenario. And the quantile in (A4) replaces the worst-case lock acquisition time in imprecise real-time systems [18,6] and in systems with a fail safe override in case of late results. It returns an upper bound t for the lock-acquisition time that will be met with the specified probability (here 95%).

We investigate these measures in a scenario where a fixed number of processes repeatedly acquire a single shared lock, execute a critical section while holding the lock, and then wait for some time after they have released the lock before they attempt to re-acquire it. We call the time between release and the attempt to re-acquire the *interim time* and refer to the code that is executed during this time as the *interim section*. We assume here, that the length of the critical section is more or less constant, while the interim time varies. Further, critical sections are typically very short in comparison with the times when no lock is held. For the distribution of the length of the interim section we draw inspiration from a video decoding example. For video decoding, the different frame types (I and P-frames) lead to clusters of the interim time in certain small intervals. Our approach used for both model checking and the measure-based techniques relies on discretisation of these distributions using finitely many sampling points.

The modelled lock-acquisition pattern allows for the derivation of results about the common-case behaviour of applications when they use a certain operating-system functionality, but gives also rise to extreme-case analyses where one assumes malicious or erroneous applications to attack the operating system. For example, by setting the interim time to the execution time of a system call minus its critical sections it is possible to deduce the contention of locks that protect these sections under the assumption that malicious applications invoke this system call as fast as they can.

3 Markov Chain Model for the Spinlock Protocol

To model the spinlock protocol, we have chosen a *discrete-time Markov chain* (DTMC) model for the following reasons. The clear demand for probabilistic guarantees about the *long-run* behaviour requires a model where steady-state probabilities are mathematically well defined and supported by model checking tools. This, for instance, rules out probabilistic timed automata and other stochastic models with nondeterminism (e.g., Markov decision processes). Continuous-time Markov chains are not adequate given that the distribution specifying the duration of the critical and interim sections are not exponential. Approximations with phase-type distributions lead to large and unmanageable state spaces. From a mathematical point of view, we could use semi-Markovian models with continuous-time uniform distributions, but we are not aware of tools that provide engines for all queries (A1)–(A4).

Preliminaries: Discrete-Time Markov Chains. We briefly explain our notations used for DTMCs and refer to standard textbooks for further details, see e.g. [12,7]. A (probabilistic) *distribution* on a countable set X is a function $\mu : X \to [0,1]$ such that $\sum_{x\in X} \mu(x) = 1$. The support $\mathit{supp}(\mu)$ of μ consists of all elements $x \in X$ with $\mu(x) > 0$. μ is called a Dirac distribution if its support is a singleton. If $C \subseteq X$ then $\mu(C) = \sum_{x\in C} \mu(x)$.

In our approach, a DTMC is a tuple $\mathcal{M} = (S, \mathbf{P}, s_{init}, \mathit{rew})$ where S is a finite state space, $s_{init} \in S$ the initial state, $\mathbf{P} : S\times S \to [0,1]$ the transition probability matrix and $\mathit{rew} : S \to \mathbb{N}$ the reward function. We require $\sum_{u\in S} \mathbf{P}(s,u) = 1$ for all $s \in S$ and refer to $\mathbf{P}(s,u)$ as the probability to move from s to u within one (time) step. A path in $\mathcal{M}$ is a finite or infinite sequence $\pi = s_0\, s_1\, s_2 \ldots$ of states with $\mathbf{P}(s_i, s_{i+1}) > 0$ for all i. Let $\pi(k) = s_k$ be the (k+1)-st state and $\pi \downarrow k = s_0\, s_1 \ldots s_k$ the prefix consisting of the first k+1 states. $\mathit{Paths}(s)$ denotes the set of infinite paths starting in state s. The accumulated reward for a finite path $\pi = s_0\, s_1 \ldots s_k$ is $\mathit{Rew}(\pi) = \mathit{rew}(s_0) + \ldots + \mathit{rew}(s_{k-1})$. If π is a finite path then the cylinder set $\mathit{Cyl}(\pi)$ spanned by π consists of all infinite paths π' where π is a prefix of π'. Using well-known concepts of measure theory, given some probabilistic distribution $\mu : S \to [0,1]$, there exists a unique *probability measure* $\Pr^{\mathcal{M}}_{\mu}$ on the σ-algebra generated by the cylinder sets of finite paths such that $\Pr^{\mathcal{M}}_{\mu}\big(\mathit{Cyl}(s_0\, s_1 \ldots s_k)\big) = \mu(s_0) \cdot \prod_{0\leqslant i<k} \mathbf{P}(s_i, s_{i+1})$. If μ is a Dirac distribution with $\mathit{supp}(\mu) = \{s\}$ we simply write $\Pr_s$ or $\Pr^{\mathcal{M}}_s$ for $\Pr^{\mathcal{M}}_{\mu}$.

For specifying measurable path events (i.e., sets of paths that belong to the σ-algebra generated by the cylinder sets of finite paths), we use the standard LTL-like notations with the symbols $\bigcirc$ (next), $\mathcal{U}$ (until) and $\Diamond$ (eventually) and time-bounded variants thereof. For $X, Y \subseteq S$, $\bigcirc X$ stands for the set of infinite paths π with $\pi(1) \in X$. $X\,\mathcal{U}^{=k}\,Y$ denotes the set of infinite paths π such that $\pi(n) \in X \setminus Y$ for $0 \leqslant n < k$ and $\pi(k) \in Y$. We write $X\,\mathcal{U}^{\leqslant K}\,Y$ for the union of the sets $X\,\mathcal{U}^{=k}\,Y$ where k ranges over the elements in $\{0, 1, \ldots, K\}$ and $X\,\mathcal{U}\,Y$ for the union of the sets $X\,\mathcal{U}^{=k}\,Y$ where $k \in \mathbb{N}$. $\Diamond\, Y$, $\Diamond^{=k}\, Y$ and $\Diamond^{\leqslant K}\, Y$ are short forms of $S\,\mathcal{U}\,Y$, $S\,\mathcal{U}^{=k}\,Y$ and $S\,\mathcal{U}^{\leqslant K}\,Y$, respectively. To deal with query (A2), we will also use LTL-like formulas with cascades of until-operators and suppose here the standard LTL-semantics for paths. For further details see [20,4,21].

For $m \in \mathbb{N}$, $\theta_m : S \to [0,1]$ denotes the state distribution for $\mathcal{M}$ after m steps. Formally, θ_0 is the Dirac distribution with $\mathit{supp}(\theta_0) = \big\{s_{init}\big\}$ and $\theta_{m+1} = \mathbf{P} \cdot \theta_m$ for $m \geqslant 0$. The function $\theta : S \to [0,1]$,

$$\theta(s) \;=\; \lim_{k\to\infty} \tfrac{1}{k+1} \cdot \sum_{m=0}^{k} \theta_m(s),$$

is called the *steady-state distribution* for $\mathcal{M}$. Then, $\theta(s) > 0$ iff s belongs to a bottom strongly connected component (BSCC) that is accessible from s_{init}. If C is a set of states with $\theta(C) > 0$ and $\mathit{\Pi}$ a measurable set of paths then the *conditional long-run probability* for $\mathit{\Pi}$ (under condition C) is defined by:

$$\mathbb{P}^{\mathcal{M}}\big(\mathit{\Pi} \,\big|\, C\big) \;\overset{\text{def}}{=}\; \sum_{s\in C} \theta(s)/\theta(C) \cdot \mathrm{Pr}^{\mathcal{M}}_s(\mathit{\Pi})$$

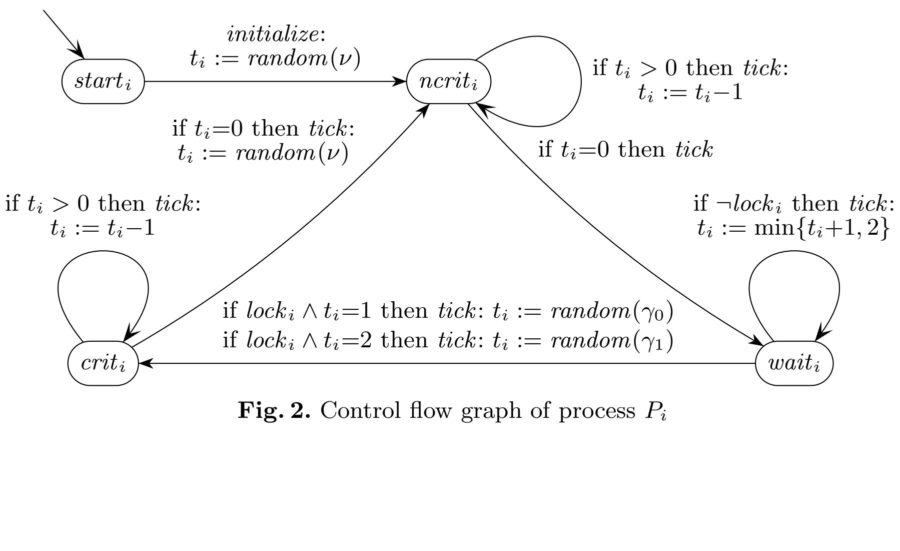
Here, $\theta(s)/\theta(C)$ is the conditional steady-state probability for state s, again under condition C. The intuitive meaning of $\theta(s)/\theta(C)$ is the portion of time spent in state s on long runs relative to the total time spent in states of C. With the factor $\mathrm{Pr}^{\mathcal{M}}_s(\mathit{\Pi})$, the above weighted sum represents the long-run probability for the event specified by $\mathit{\Pi}$ under condition C. Analogously, conditional steady-state average values of random variables can be defined as weighted sums. (A3) will be formalised as an instance of *conditional long-run accumulated reward* for reaching a goal set Y defined by:

$$\mathbb{R}^{\mathcal{M}}\big(\Diamond Y \,\big|\, C\big) \;\overset{\text{def}}{=}\; \sum_{s\in C} \theta(s)/\theta(C) \cdot \mathrm{ExpAccRew}^{\mathcal{M}}_s(\Diamond Y)$$

where $\mathrm{ExpAccRew}^{\mathcal{M}}_s(\Diamond Y)$ denotes the expected accumulated reward for reaching Y from state s. It is defined by:

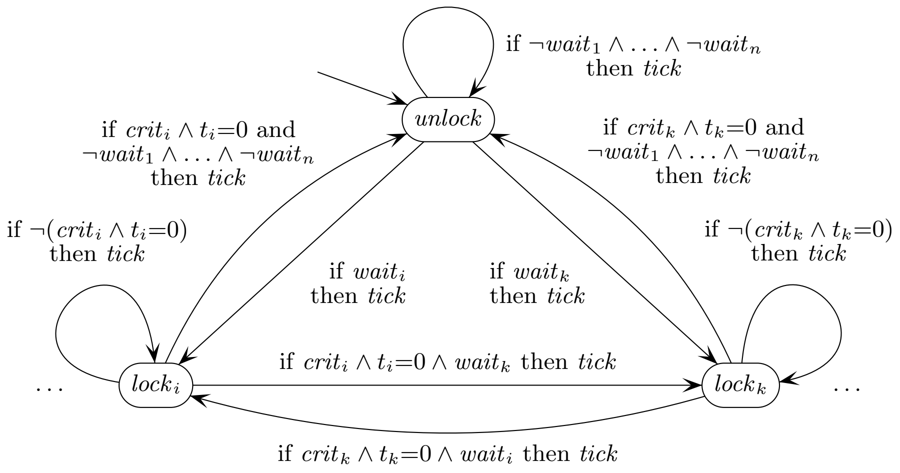
$$\sum_{r=0}^{\infty} r \cdot \mathrm{Pr}^{\mathcal{M}}_s\big\{\pi \in \mathit{Paths} : \exists k \in \mathbb{N} \text{ s.t. } \pi \in \Diamond^{=k}\, Y \wedge \mathit{Rew}(\pi{\downarrow}k) = r\big\}$$

Markov Chain Model for the Spinlock Protocol. To analyse the quantitative behaviour of the spinlock protocol for n processes $P_1, \ldots, P_n$, we use a DTMC that results as the synchronous parallel composition of one module representing the spinlock in Fig. 1 (see Fig. 3) and one module for each of the processes (see Fig. 2). The t_i's are integer variables that serve as timers for the critical and noncritical section. Distribution ν models the *interim time*, i.e., the time required for the (noncritical) activities of the processes between two critical sections, including the request to acquire the lock, but without the spinning



initialize: $t_i := \mathit{random}(\nu)$
start_i
ncrit_i
if $t_i > 0$ then *tick*: $t_i := t_i{-}1$
if $t_i{=}0$ then *tick*: $t_i := \mathit{random}(\nu)$
if $t_i{=}0$ then *tick*
if $t_i > 0$ then *tick*: $t_i := t_i{-}1$
if $\neg\mathit{lock}_i$ then *tick*: $t_i := \min\{t_i{+}1, 2\}$
if $\mathit{lock}_i \wedge t_i{=}1$ then *tick*: $t_i := \mathit{random}(\gamma_0)$
if $\mathit{lock}_i \wedge t_i{=}2$ then *tick*: $t_i := \mathit{random}(\gamma_1)$
crit_i
wait_i

Fig. 2. Control flow graph of process P_i



if $\neg wait_1 \wedge \ldots \wedge \neg wait_n$ then *tick*
unlock
if $crit_i \wedge t_i{=}0$ and $\neg wait_1 \wedge \ldots \wedge \neg wait_n$ then *tick*
if $crit_k \wedge t_k{=}0$ and $\neg wait_1 \wedge \ldots \wedge \neg wait_n$ then *tick*
if $\neg(crit_i \wedge t_i{=}0)$ then *tick*
if $\neg(crit_k \wedge t_k{=}0)$ then *tick*
if $wait_i$ then *tick*
if $wait_k$ then *tick*
if $crit_i \wedge t_i{=}0 \wedge wait_k$ then *tick*
$lock_i$
$lock_k$
...
...
if $crit_k \wedge t_k{=}0 \wedge wait_i$ then *tick*

Fig. 3. Control flow graph of the spinlock

time. Distributions γ_0 and γ_1 serve to model the total length of the critical section (including lock acquisition and release). In order to account for cache effects that lead to different time behaviour for the lock acquisition, depending on whether the lock is taken with and without spinning, we use two distributions for the critical section length. Distribution γ_0 is used when the lock was obtained without spinning, while distribution γ_1 is used if some spinning was necessary. To obtain a DTMC model, ν and γ_0, γ_1 are discrete distributions with finitely many sampling points in the relevant time intervals. E.g., the distribution $\nu(8) = \nu(10) = \nu(12) = \frac{1}{3}$ indicates that with equal probability the duration of the interim time is 8, 10 or 12 time units. The assignment $t_i := random(\nu)$ means that a sample is drawn according to distribution ν and assigned to t_i.

For each process P_i, we distinguish four locations. Location $start_i$ serves to set the timers for the first noncritical phase. The other three locations have the obvious meaning. Location $wait_i$ signals that P_i is trying to acquire the lock. If the lock is granted to P_i, the lock process switches to location $lock_i$, which in turn enables the transition from $wait_i$ to $crit_i$. In location $wait_i$, process P_i waits until the lock has been made accessible for it. In location $wait_i$, variable t_i does not serve as a timer. Instead, t_i indicates whether process P_i has just entered the waiting location ($t_i \in \{0, 1\}$) or P_i is spinning as some other process is holding the lock ($t_i{=}2$). The control flow graph of the lock process contains for each P_i one location $lock_i$ (indicating that P_i may take or holds the lock) and one location *unlock* (the lock is free). For the synchronisation, we followed the approach of PRISM's input language with synchronisation over shared actions. Action *initialize* has to be synchronised by all processes, while *tick* indicates one time step and must be executed synchronously by all processes and the lock.

The states of the DTMC $\mathcal{M}$ for the composite model have the form $s = \langle \ell_1, \ldots, \ell_n, m, t_1{=}b_1, \ldots, t_n{=}b_n \rangle$ where ℓ_i is the current location of process P_i, m the current location of the lock and b_i the current value of variable t_i. Then,

P_i is spinning in state s iff $s.m \neq lock_i$, $s.\ell_i = wait_i$ and $s.b_i = 2$, where we use the dot notation to refer to parts of a state s. If process P_i performs its last critical action and moves from location $crit_i$ to $ncrit_i$ then either the lock returns to its initial location *unlock* (if no process P_k is spinning) or there is a uniform probabilistic choice for the lock to move to one of the locations $lock_k$ where process P_k is spinning. To compute the long-run average spinning time (query (A3)), we deal with the reward function $rew\_spin_i(s) = 1$ for each state s where process P_i is spinning. For all other states s, we have $rew\_spin_i(s) = 0$.

Formalisation of Queries (A1)–(A4). In the sequel, we use propositional formulas over the locations and conditions on the values of $t_1, \ldots, t_n$ to characterise sets of states. For instance, $crit_i$ is identified with the set $\{s \in S : s.\ell_i = crit_i\}$, where S denotes the state space of the DTMC $\mathcal{M}$ for the composite system. Condition $request_i = wait_i \wedge t_i{=}0$ characterises the set of states s in $\mathcal{M}$ where process P_i has just requested the lock. The set of states where process P_i spins is specified by $spin_i = wait_i \wedge t_i{=}2 \wedge \neg lock_i$. Finally, $release_i = crit_i \wedge t_i{=}0$ characterises the states s where process P_i is just performing its last critical actions and the lock is to be released next. The relevant quantitative measures (A1)–(A4) of Section 2 now correspond to the following values.

(A1) $\mathbb{P}^{\mathcal{M}}\big(\varphi_1 \,\big|\, request_i\big)$ where $\varphi_1 = \bigcirc\, lock_i$

(A2) $\mathbb{P}^{\mathcal{M}}\big(\varphi_2 \,\big|\, release_i\big)$ where $\varphi_2 = \bigcirc\,(\, unlock\, \mathcal{U}\, lock_i\,)$

(A3) $\mathbb{R}^{\mathcal{M}}\big(\Diamond\, lock_i \,\big|\, request_i\big)$

(A4) $\min\big\{\, t \in \mathbb{N} \,:\, \mathbb{P}^{\mathcal{M}}\big(\Diamond^{\leqslant t+1} lock_i \,\big|\, request_i\big) \,\geqslant\, 0.95 \,\big\}$

(A1), (A3) and (A4) refer to the conditional steady-state distribution for the condition that process P_i has just performed its first request operation. (A1) corresponds to the long-run probability under the condition $request_i$ for the path event $\varphi_1 = \bigcirc\, lock_i$ stating that in the next time step process P_i will win the race between the waiting processes, i.e., it will enter its critical section without spinning. (A3) stands for the long-run average spinning time from states where P_i has just requested its critical section. Replacing the condition $request_i$ in (A3) with $wait_i \wedge t_1{=}1 \wedge \neg lock_i$, we obtain the long-run average spinning time, provided that the first attempt to acquire the lock was not successful. The quantile in (A4) corresponds to the minimal number t of time steps minus one such that the long-run probability from the $request_i$-states for the path event $\Diamond^{\leqslant t+1} lock_i$ stating that the lock will be granted for process P_i in $t{+}1$ or fewer steps is at least 0.95. For the long-run probability of acquiring the lock, again without interference by other processes (see (A2)), there are several reasonable formalisations. For the constraint that no process other than P_i requested the lock in the mean time, we can deal with the path event $\varphi_2 = \bigcirc(\, unlock\, \mathcal{U}\, lock_i)$ under the condition that process P_i will release the lock in the next step (condition $release_i$). The variant of (A2) where other processes may have acquired and released the lock in between the critical sections of P_i and where P_i has not to spin and hence experiences a low overhead lock can be formalised as $\mathbb{P}^{\mathcal{M}}\big(\varphi_2' \,\big|\, request_i\big)$ for the event specified by the LTL-formula $\varphi_2' = \bigcirc\big(lock_i\, \mathcal{U}\, (ncrit_i \wedge (\neg spin_i\, \mathcal{U}\, crit_i))\big)$. The treatment of (A2) using φ_2 or φ'_2 is rather complex since the standard treatment of LTL-queries relies on a probabilistic reachability analysis of a product construction of a deterministic ω-automaton and the DTMC (see e.g. [2]). To avoid this automaton-based approach, one can replace φ_2 and φ'_2 with a simpler reachability condition when refining the control flow graph of P_i by duplicating the control loop $\mathit{ncrit}_i\ \mathit{wait}_i\ \mathit{crit}_i$. This can be realised by introducing a Boolean variable b that flips its value after leaving the critical section, see the extended version of this paper, and justified using an appropriate notion of bisimulation. The control flow graphs for the lock and the other processes remain unchanged. Then, instead of φ_2 and φ'_2 we can then deal with $\psi_2 = (\mathit{lock}_i \vee \mathit{unlock})\,\mathcal{U}\,(\mathit{crit}_i \wedge b)$ and $\psi'_2 = \neg \mathit{spin}_i\,\mathcal{U}\,(\mathit{crit}_i \wedge b)$ under the condition $\mathit{release}_i \wedge \neg b$ and $\mathit{request}_i \wedge \neg b$, respecively. Indeed, our experiments show that the analysis of $\mathcal{M}'$ with the modified queries is more efficient than the analysis of $\mathcal{M}$.

4 Quantitative Analysis of the TTS Spinlock

Our general approach to the quantitative analysis of low-level operating-system code proceeds in four steps: (1) The targeted operating-system code is formalised at a suitable level of abstraction in the input language of a probabilistic model checker. For our TTS lock case study, we used the prominent model checker PRISM [13]. (2) The parameters of the model are determined with the help of measurements in the targeted setting. If this is not possible due to unduly high interference with the instrumentation code, we extract the relevant code and measure it in the form of a microbenchmark in a controlled environment. (3) The queries of interest are evaluated both by the model checker and with a microbenchmark that again executes the code in a manageable environment. This step is necessary to determine how well the model corresponds to the targeted setting. (4) The queries are evaluated on the parameters obtained for the real workload and if possible cross-checked against measurements in this setting. There are two situations in which such a comparison is infeasible, namely when the interference between the instrumentation code and the targeted operating-system code fundamentally changes the behaviour of the latter or when the analysis is performed with parameters of a system that does not yet exist.

The measurements done in the microbenchmarks do as well interfere with the to-be-analysed code. However, these measurements only extract the required model parameters (possibly only one at a time). The challenge is to construct a setting where this interference is limited only to those parameters that are not currently extracted. We now report on our experiences in adjusting the model with the help of a microbenchmark and measurement-based simulation.

Measurement-Based Simulation. For the extraction of the two distributions γ_0 and γ_1 for the critical section and for the distribution ν for the interim section, we can resort to random sampling and similar techniques [14] to deduce the characteristics of the workload we seek to investigate. Very short critical sections and the acquire and release costs of the lock must however be measured in an environment where it is possible to control side effects on the quantity of interest. For the TTS lock, we construct such an environment by mimicking the behaviour of the formal model in the critical and interim section. When a process enters a critical section, it selects pseudorandomly a sampling point of the distribution γ_0 if the lock was free and of γ_1 if it had to spin. A counter in the spinning loop (in Line 4 in Fig. 1) reveals whether or not the process was able to obtain the lock without spinning. We pass this counter in a processor register not to disturb the timing of the TTS code. The actual instrumentation consists of three reads of the per core time stamp counter, before and after acquiring the invocation of the `lock()` function in Line 2 and after the `unlock()` function returns. Both the critical and interim section consist of a loop of an integer instruction: `rep; dec %eax`. We have confirmed that the execution time of this loop is very regular. The loop executes for a time that is proportional to the sampling point selected for the respective critical or interim section.

Quantitative Analysis Using PRISM. For the quantitative analysis of the DTMC of the TTS spinlock we used the probabilistic model checker PRISM [13]. We mainly concentrated on (A1)–(A4), but also considered functional and a few more quantitative queries. To obtain empirical evidence in the model and in the model checking results we compare the model checking results with measurements of the model mimicking a microbenchmark. Unfortunately, PRISM has no direct support for computing conditional long-run probabilities or quantile-based queries. We extended the PRISM code by operators that compute conditional long-run probability $\mathbb{P}^{\mathcal{M}}\big(\varphi \,\big|\, C\big)$ and conditional long-run accumulated rewards $\mathbb{R}^{\mathcal{M}}\big(\Diamond Y \,\big|\, C\big)$ where φ is a PCTL path formula and Y, C sets of states. Although there is also no direct support for (A4) in PRISM, quantiles that refer to the amount of time until some event occurs can be calculated with the same iterative bottom-up computation scheme as for bounded reachability properties. In (A4), we are interested in the minimal $t \in \mathbb{N}$ such that the conditional long-run probability is greater than some fixed probability value. For finding the minimal value $t \in \mathbb{N}$ with the above property efficiently, we modified the implementation of the bounded until operator in PRISM to store the intermediate probabilities for all $0 \leqslant j \leqslant t$. Instead of a more direct evaluation at the MTBDD level of PRISM, this storing of intermediate results allows for an external script to check the gradually increasing values of the bounded until formula without restarting the model checking for each such check.

Lessons Learned. During the analysis of the queries (A1)–(A4) and in the course of performing and evaluating the measurements for the simulation, we encountered several difficulties that we would like to share.
Cascade effects. We first evaluated (A1)–(A4) using a model where the length of the critical section and the interim time are deterministic, i.e., where $\gamma_0 = \gamma_1$ and ν are Dirac distributions with values T_{crit} and T_{int} and where $T_{crit} \ll T_{int}$. A simulation run of the model revealed a probabilistic choice of the order in which processes acquire the lock for the first time. For all subsequent turns, the processes received the lock in this same order and without having to spin. Small variations in the lock acquisition times and in the points in time when

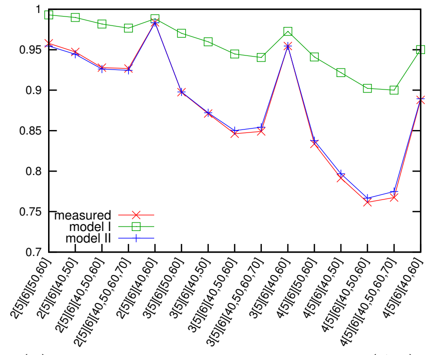
the processes start prevented the measurement-based simulation from entering a similar cascade. These cascade effects, however, do not appear for more realistic models where at least ν is a distribution with $|supp(\nu)| \geqslant 2$. Assuming that the values for ν are much larger than those for γ_0 and γ_1, such DTMC models only have one bottom strongly component, which justifies to apply the measurements for just a few simulation runs.

Short critical sections. One of the first workloads we evaluated, was a system call of the Nova microhypervisor [19] in which very short TTS-lock protected critical sections alternate with relatively long sequences of interim activities. The measure-based approach encountered situations where the lock-acquisition times exceeded the critical section length. To avoid a too fine granular (discrete) time domain for the DTMC model that would rule out the feasibility of model checking techniques, we used time domains of different granularity for the measurements and the DTMC model and relate them via a scaling factor sf. E.g., for the scaling factor $sf = 1000$, one time unit in the DTMC model corresponds to 1000 cycles (approx. 362 ns) on the target system.

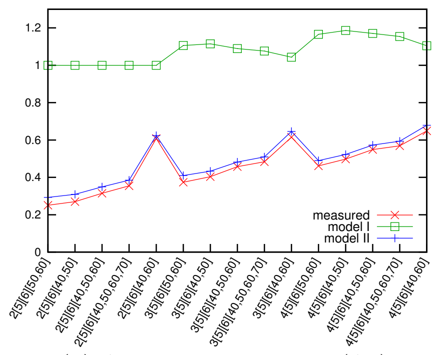
Varying acquisition times. In an early version of the microbenchmark, we adjusted the `rep; dec %eax` loops, which together with the pseudo random choice of a sampling point mimic the distributions of the critical and interim sections, to the same constant value for all processes. More precisely, for $\gamma(x) = 1$, we adjusted the loop in critical to consume as close as possible to 1000 cycles minus the time required to execute the instrumentation code. However, the average costs for acquiring and releasing a lock increase with the number of processes that require this lock. One explanation for this behaviour could be that the costs for invalidating a cache line when acquiring or releasing the lock vary with varying number of cores. This is because the on-chip networks in modern multicore processors connect cores in a point-to-point fashion while maintaining information about the locations of copies of cache lines.

Controlled measurement environment. We realised early variants of our microbenchmark as a Linux user-level application, disabled all obvious sources of interference and raised the priority of this application into the otherwise empty real-time priority band. From the results, we classified spikes as interference and ignored these points in our comparison with the formal model. Still, we experienced high fluctuations of the measurement results, which could not easily be explained. In a repeated measurement on top of a small kernel binary, which we just used to bootstrap our microbenchmark and to communicate the results after the measurement part completed, these variations did not reappear. We therefore take this effect as an indicator and as a warning that the interference of large operating-system kernels on short executing microbenchmarks should not be underestimated and best be avoided whenever this is possible.

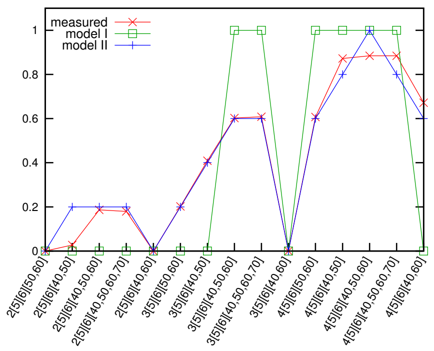
Need for two critical distributions. After having performed the above adjustments, we observed a discrepancy between the model checking results and the measurements of approx. 20% (see the model I results in Fig. 4(a)–4(c)). Following an in-depth search for possible causes both in the model and in the



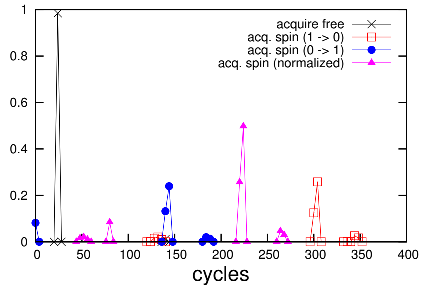
(a) Probability to grab the lock (A1)



(b) Average waiting time (A3)



(c) 95% quantile (A4)



(d) Frequency of acquire free and spin

Fig. 4. (a)-(c) results for model I and II and (d) the histogram for acquire time

microbenchmark, we identified small variations between the costs of acquiring a lock with and without spinning. In the course of this search, we encountered several other possible causes such as the quantisation due to scaling, which showed similar small variations in the measurements. To confirm these factors, we adjusted a copy of the measurement data to mimic the effect we suspected to determine the magnitude of the impact this effect could have. If this impact was in a range where it could have explained a significant part of this discrepancy, we adjusted the formal model accordingly. For the small variations between lock acquisition times, we changed the model from a single Dirac distribution for the critical section length to the two Dirac distributions γ_0 and γ_1 in Fig. 2.

Unfortunately, the value of the singleton sampling point of γ_1 could not be directly measured because it would require the inclusion of `rdtsc` in the spinning loop (Line 4 in Fig. 1) to read the core cycle counter, which would significantly change the timing behaviour of the lock. We therefore calculate the average costs for acquiring a lock under the condition that a process had to spin by comparing the time stamps of the releasing cores with the time stamps of the lock acquiring cores. However, although the time stamp counter for cores on the same die are derived from the same clock, there is an offset caused by the barrier on which these cores synchronise to start the measurement at approx. the same point in time. The two dashed lines in Fig. 4(d) show the acquire spin costs per core. The figure also shows the acquire spin costs after normalising the offset between the two clocks. We choose the spike as the sampling point for γ_1. The scaling factor $sf = 1000$ that we used to deal with the single-distribution model (where $\gamma(1) = 1$) was no longer adequate for integrating the resulting distributions ($\gamma_0(5){=}\gamma_1(6){=}1$) into the DTMC model. We therefore decreased the scaling factor to $sf{=}200$.

Evaluation of the Results. Fig. 4(a)–4(c) show the results of our quantitative analysis of the queries (A1)–(A4). We omit the plot for (A2) because, for selected distributions, the chance to re-acquire the lock without interference by other processes is close to zero. The plots compare our measurements with the results from two models. Model I is our earlier, simpler model which uses only one distribution for the critical section ($\gamma_0{=}\gamma_1$). Model II is more precise because it uses different distributions γ_0 and γ_1 as explained before. The x-axis displays the number of processes and the distributions used for model II and, with scaling factor $sf{=}200$, for the measurement. The label $3[5][6][40, 50, 60]$ stands for $n{=}3$ processes and the distributions $\gamma_0(5){=}\gamma_1(6){=}1$ and $\nu(40){=}\nu(50){=}\nu(60){=}\frac{1}{3}$. The distribution for model I, which uses scaling factor $sf{=}1000$, is obtained by dividing all values by 5 and using the value of γ_0 for γ_1 too. Thus, for model I, the same label stands for the model with 3 processes, $\gamma_0(1){=}\gamma_1(1){=}1$ and $\nu(8){=}\nu(10){=}\nu(12){=}\frac{1}{3}$. We performed the measurements on an Intel i7 920 quadcore machine at 2.67 GHz. The benchmark and the sampling area for storing the measurement results fitted completely in the on-die caches. Fig. 4(a) shows that the introduction of γ_1 reduced the error between the measured and analysed results from 20% for model I to below 1% for model II. The discrepancy between model II and the measurements in Fig. 4(c) is due to the quantisation of the model. That is, the model considers changes of the waiting time only in steps of $sf{=}200$ cycles, which corresponds to one fifth (0.2) of the critical section length. The differences between the results obtained for the models with parameters $n[5][6][40, 50, 60]$ and $n[5][6][40, 60]$ (where $n \in \{2, 3, 4\}$) illustrate that not only the mean value, but also the variance of distribution ν for the interim time has non-negligible impact on (A1)-(A4).

For model I ($sf{=}1000$ and $\gamma_0{=}\gamma_1$), we used PRISM to compute (A1)–(A4) for the DTMC with up to six processes. For model II ($sf{=}200$, $\gamma_0 \neq \gamma_1$) the analysis has been carried out with up to four processes. Because of the reduced scaling factor, model II is far more complex. E.g., for $n{=}4$ processes and the distribution $\nu(8) = \nu(14) = \frac{1}{2}$ the DTMC for model I has about 10^4 states, while for model II with the corresponding distribution $\nu(40) = \nu(70) = \frac{1}{2}$ the DTMC has about $2.5 \cdot 10^6$ states. In a nutshell, for the DTMC with $n{=}6$ and $sf{=}1000$ PRISM needs a few minutes for all queries, while for $n{=}4$ and $sf{=}200$ the computation can take a few hours. In most cases, the computation of the steady-state probabilities is the most time consuming part. For more information on the PRISM statistics (MTBDD sizes, time for the model construction and the quantitative analysis) we refer to the extended version.

5 Conclusions

The paper presents a first step towards the application of probabilistic model checking techniques for the quantitative analysis of low-level operating-system code. We reported on the difficulties we encountered when analysing a simple test-and-test-and-set (TTS) spinlock with the model checker PRISM and on our solutions to address them. A major challenge was to find an appropriate level of abstraction that allows to capture all relevant behaviour and allows to abstract from the precise timing behaviour of the cache and other CPU parts. We performed extensive measurement-based simulations of real spinlocks to demonstrate that our abstract model does indeed reproduce important aspects of the timing behaviour. We considered a representative list of properties that are of high interest to the system designer when he has to choose the right lock implementation. These properties involve conditional steady-state probabilities and quantiles. To overcome the lack of direct tool support for both types of queries, we added the relevant features in the PRISM code.

Related work. Many researchers performed case studies with probabilistic model checkers for mutual exclusion protocols and other coordination algorithms for distributed systems (see e.g. [8,16,17] or the PRISM web pages [13]). While some of these case studies address the analysis of randomised protocols, we deal with non-randomised operating-system primitives (the TTS spinlock). Our models rely on stochastic assumptions on the execution times of the critical section and interim sections of competing processes). Unlike the wide range of case studies with continuous-time models, where rates of exponential distributions specify, e.g., the frequency of the arrival of requests or the average duration of events (see e.g. [7]), we deal with a DTMC model and discretisations of non-exponential distributions. Of course, there have been plenty of case studies with the DTMC engine of PRISM, but we are not aware of experiments that have been carried out where the modelling and evaluation process was accompanied by measure-based techniques. To the best of our knowledge, none of these case studies considers conditional steady-state probabilities or quantile-based queries. The majority of work on model checking low-level operating-system code concentrates on properties in the safety-liveness domain. E.g., [23] used model checking to find serious file system bugs. Formal quantitative analyses often consider only worst-case execution times as measure. For the special case of probabilistic worst-case execution times (i.e. queries similar to Query (A3)), [3] presents a timing-schema based on independent or only pairwise dependent (i.e., joint) execution profiles.

Future work. It would be very interesting to scale the analysis of basic and more advanced spinlocks for CPUs with more than 100 cores. Such results could justify the use of more simple locks with less overhead for certain tasks. Given the exponential growth of the system model with the number of cores, this is an extremely challenging task and requires clever encoding, abstraction and reduction techniques such as symmetry reduction. Other promising candidates are bisimulation quotienting techniques as supported by the model checker MRMC [9,10] and the use of sophisticated (MT)BDD-based techniques to increase the efficiency of PRISM's symbolic engine.

References

1. Anderson, T.E.: The performance of spin lock alternatives for shared-memory multiprocessors. IEEE Trans. Parallel Distrib. Syst. 1(1), 6–16 (1990)
2. Baier, C., Katoen, J.-P.: Principles of Model Checking. MIT Press (2008)
3. Bernat, G., Colin, A., Petters, S.: WCET analysis of probabilistic hard real-time systems. In: RTSS 2002, pp. 279–288. IEEE (2002)
4. Courcoubetis, C., Yannakakis, M.: The complexity of probabilistic verification. Journal of the ACM 42(4), 857–907 (1995)
5. Hähnel, M.: Energy-utility functions. Diploma thesis, TU Dresden, Germany (2012)
6. Hamann, C.-J., Löser, J., Reuther, L., Schönberg, S., Wolter, J., Härtig, H.: Quality-assuring scheduling - using stochastic behavior to improve resource utilization. In: RTSS 2001, pp. 119–128. IEEE (2001)
7. Haverkort, B.: Performance of Computer Communication Systems: A Model-Based Approach. Wiley (1998)
8. Irani, S., Singh, G., Shukla, S.K., Gupta, R.: An overview of the competitive and adversarial approaches to designing dynamic power management strategies. IEEE Trans. VLSI Syst. 13(12), 1349–1361 (2005)
9. Katoen, J.-P., Kemna, T., Zapreev, I., Jansen, D.: Bisimulation Minimisation Mostly Speeds Up Probabilistic Model Checking. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 87–101. Springer, Heidelberg (2007)
10. Katoen, J.-P., Zapreev, I., Hahn, E., Hermanns, H., Jansen, D.: The ins and outs of the probabilistic model checker MRMC. Perform. Eval. 68(2), 90–104 (2011)
11. Knapp, S., Paul, W.: Realistic Worst-Case Execution Time Analysis in the Context of Pervasive System Verification. In: Reps, T., Sagiv, M., Bauer, J. (eds.) Wilhelm Festschrift. LNCS, vol. 4444, pp. 53–81. Springer, Heidelberg (2007)
12. Kulkarni, V.: Modeling and Analysis of Stochastic Systems. Chapman & Hall (1995)
13. Kwiatkowska, M., Norman, G., Parker, D.: Probabilistic symbolic model checking with PRISM: A hybrid approach. STTT 6(2), 128–142 (2004)
14. Liedtke, J., Islam, N., Jaeger, T., Panteleenko, V., Park, Y.: Irreproducible benchmarks might be sometimes helpful. In: ACM SIGOPS European Workshop, pp. 242–246. ACM (1998)
15. Mellor-Crummey, J., Scott, M.: Scalable reader-writer synchronization for shared-memory multiprocessors. In: PPOPP 1991, pp. 106–113. ACM (April 1991)
16. Norman, G.: Analysing Randomized Distributed Algorithms. In: Baier, C., Haverkort, B.R., Hermanns, H., Katoen, J.-P., Siegle, M. (eds.) Validation of Stochastic Systems. LNCS, vol. 2925, pp. 384–418. Springer, Heidelberg (2004)
17. Norman, G., Parker, D., Kwiatkowska, M., Shukla, S., Gupta, R.: Using probabilistic model checking for dynamic power management. Formal Aspects of Computing 17(2), 160–176 (2005)
18. Shih, W.K., Liu, J.W.-S., Chung, J.-Y.: Algorithms for scheduling imprecise computations with timing constraints. SIAM J. Comput. 20(3), 537–552 (1991)

19. Steinberg, U., Kauer, B.: NOVA: a microhypervisor-based secure virtualization architecture. In: EuroSys 2010, pp. 209–222. ACM (2010)
20. Vardi, M.: Automatic verification of probabilistic concurrent finite-state programs. In: FOCS 1985, pp. 327–338. IEEE (1985)
21. Vardi, M.: Probabilistic Linear-Time Model Checking: An Overview of the Automata-Theoretic Approach. In: Katoen, J.-P. (ed.) ARTS 1999. LNCS, vol. 1601, pp. 265–276. Springer, Heidelberg (1999)
22. Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., Stenström, P.: The worst-case execution-time problem - overview of methods and survey of tools. Trans. Embedded Comput. Syst. 7(3), 1–53 (2008)
23. Yang, J., Twohey, P., Engler, D., Musuvathi, M.: Using model checking to find serious file system errors. ACM Trans. Comput. Syst. 24(4), 393–423 (2006)

Microcontroller Assembly Synthesis from Timed Automaton Task Specifications*

Victor Bandur[1], Wolfram Kahl[2], and Alan Wassyng[2]

[1] The University of York, York, UK
[2] McMaster University, Hamilton, Canada

Abstract. A method for the automatic refinement of single-task timed automaton specifications into microcontroller assembly code is proposed. The outputs of the refinement are an assembly implementation and a timed automaton describing its exact behaviour. Implementation is only possible when all specified timing behaviours can be met by the target microcontroller. Crucially, the implementation does not make the simplifying synchrony assumption, yet correctness with respect to timing is guaranteed. Currently this method copes with parallel inputs and outputs, but is restricted to timed automaton specifications with only one clock variable that is reset after each transition. Further generalization is possible. A tool illustrates the method on a simple example.

1 Introduction

We propose a method for automating the implementation of single-task timed automaton specifications targeted strictly at microcontroller architectures without caching or pipelining. These microcontrollers have deterministic instruction execution times. This method does not assume the use of an embedded operating system and therefore does not address the issue of task scheduling.

Existing work addresses these issues, but at various costs. The TIMES tool [2] translates a timed automaton model of task arrival patterns, together with task WCET and deadline, into a complete, multi-tasking executable package that satisfies the required scheduling behaviour by construction. The tool does not implement tasks from their specifications but rather allows the user to input the task code.

PLC automata [6,8] are a complete automaton formalism that is tailored to implementation on PLC hardware. It comes equipped with an algorithm for the direct translation of specifications into PLC hardware implementations. This algorithm uses the hardware's timing facilities to implement the specified timing behaviour, which makes any hardware platform that provides similar timing facilities a valid target for the algorithm. The value ϵ of the specification encodes the minimum required cycle time of the implementation hardware and is derived from the shortest duration for which the implementation must guarantee to

* This research is supported by the National Science and Engineering Research Council of Canada (NSERC).

detect an input (see discussion in Sec. 2.3). Moreover, Dierks [7] also presents an algorithm for the translation of Duration Calculus [21] refinements known as *implementables* into PLC automata.

More recent work by Jee *et al.* [12] tackles reactive control by capturing the behaviour of each system component as a timed automaton model and using the cyclic executive structure of the output of the TIMES tool to implement the combined behaviour without the need for externally provided task code. The structure of the (instrumented) output code is such that it suffers from variations in execution period due to guard condition checking and the way in which transitions are selected for execution. Because of this, the synthesized code must be checked for correctness of timing behaviour by direct execution or simulation. The timing requirements in the specification automata are then modified in response to the *observed* timing behaviour of the synthesized code (which is not guaranteed to meet the behaviour specified in the first place) and the process is repeated. Jee *et al.* also present a multithreaded implementation which confirms our preliminary ideas for how to generalize our proposed method to multiple clock variables. This is discussed further in Sec. 7.

These and other approaches (i.e., [4,13]) make the synchrony assumption, that the underlying system acts in negligible time when compared with the implemented task. Since this time is never in fact zero, it must be accounted for either in the specification or in the development of the task code, making for an iterative and error-prone development approach that circumvents the advantages of automatic code generation. The main contribution of this paper is a refinement method that does not make this assumption, thus being able to guarantee that the specified timing behaviour is preserved by the implementation, at the cost of a constrained specification language. The implementation is thus treated as a purely derived artifact and does not impose any time-consuming iteration upon the development process. The timing guarantee is only contingent on the behaviour of the clock signal driving the target microcontroller, but, as this is outside the software domain, its quality is not considered here. Our second contribution is an intermediate assembly language that can facilitate specification refinement for non-cached, non-pipelined microcontroller architectures in general.

2 The Specification Language Used

Our method is aimed at specifications written as Timed Automata. A full development of the theory is given by Alur and Dill [1]. This section only presents a summary of the restrictions and changes required for our approach.

2.1 Notational Modifications

For improved readability of timed automaton specifications we make the following stylistic changes. In the first place, interval membership notation is preferred over the original logical notation (i.e. we write $x \in [3.2, 9.1]$ for $3.2 \leq x \wedge x \leq 9.1$).

This makes it easier to visualize how the time spent by an automaton in a state is divided into sub-intervals during which individual outgoing transitions are enabled and disabled with the passage of time. In the second place, we impose a convention on message names in the spirit of Tuttle and Lynch [19]: input events are suffixed with '?', output events with '!'. These are collected in the two disjoint sets $\Sigma^?$ and $\Sigma^!$ such that $\Sigma^? \cup \Sigma^! \cup \{\epsilon\} = \Sigma$, where Σ is the complete alphabet and ϵ represents the empty event. Whenever any of the events is ϵ it will be elided from the written specification. This second modification is captured below.

Definition 1. *A modified timed automaton is a tuple $\langle \Sigma^?, \Sigma^!, S, s_0, C, E\rangle$ where,*

- *$\Sigma^?$ is the input alphabet*
- *$\Sigma^!$ is the output alphabet, disjoint from $\Sigma^?$*
- *S is the set of states*
- *$s_0 \in S$ is the start state*
- *C is a finite set of non-negative real-valued clock variables*
- *$E \subseteq S \times S \times \mathcal{P}(\Sigma^? \cup \{\epsilon\}) \times \mathcal{P}(\Sigma^! \cup \{\epsilon\}) \times \mathcal{P}(C) \times \Phi(C)$ is the transition relation and each element of E is a tuple $\langle s, s', \sigma^?, \sigma^!, \lambda, \kappa\rangle$ where,*
    - *$\lambda \subseteq C$ is a set of clock variables to be reset to zero on the given transition*
    - *$\kappa \in \Phi(C)$ is a clock constraint formula*

A timed word in this model is a tuple $\langle \sigma^?, \sigma^!, \tau\rangle$ where $\sigma^?$ and $\sigma^!$ are infinite words over the input and output alphabets respectively, and τ is an infinite sequence of real time values where,

- $\tau_0 > 0$ (definite start time)
- for all $i \geq 1$, $\tau_i > \tau_{i-1}$ (monotonicity of time)
- for all $t \in \mathbb{R}$ with $t > 0$, there is an i such that $\tau_i > t$ (infinite time progress)

2.2 Implementable Specifications

We restrict this method to an implementable subset of timed automata which satisfy the following conditions.

1. $|C| = 1$ (i.e., only one clock variable) and $\lambda = C$ for each transition.
2. For every state, the time intervals specified on any concurrently enabled outgoing transitions are identical.

It is possible to normalize specifications with overlapping time intervals that do not satisfy condition 2 above by splitting them into multiple transitions and distributing I/O actions accordingly among them.

Two different types of acceptance conditions are defined for timed automata, Büchi and Muller conditions [1,17]. For our purposes, specifications need not specify acceptance conditions. They may be useful in a theory of testing for our method, but this is not investigated here.

2.3 Response Allowance and Timing Resolution

A primary factor determining the feasibility of an implementation is whether it can satisfy the *timing resolution* specified for each monitored variable and the *response allowance* for each controlled/monitored variable pair [20]. In general, the timing resolution for a monitored variable is a determination made based on the nature of the monitored signal, which specifies the shortest duration of a condition (or state) of the signal that must be guaranteed to be detected by the implementation. As this relates directly to the rate at which this signal is sampled in the software domain, it governs the feasibility of any given implementation. The response allowance is likewise a determination made of an output generated by the implementation in response to the status of an input (the initiating event), but specifies instead the amount of time that the implementation can take to react. Response allowance likewise governs the feasibility of implementations.

Inputs and outputs are visible in a timed automaton specification on transitions. In general it is possible to determine the controlled/monitored variable pairs themselves if it is known how the given timed automaton specification is derived from its requirements. However it is not possible in general to determine this information from only a timed automaton specification, which makes it impossible for a tool to determine the feasibility of an implementation based on response allowance and timing resolution without explicit annotations to this end. Our tool, therefore, does not implement this feasibility check, though it may be carried out *post hoc* using the definition of the control problem.

3 Overview of Synthesis Method

For brevity we give a condensed overview of the proposed approach. Bandur [3] gives a detailed explanation and rationale.

3.1 Underlying Approach

Instead of calculating the execution speed required of a target microcontroller to implement all timing requirements in a given specification, we propose starting with the capabilities of a given microcontroller and restricting the set of specifications whose timing requirements can be implemented on it. We observe that there is a small set of hardware features common to a large number of microcontrollers on the market that can be targeted as an abstract microcontroller platform. We capture these capabilities in an intermediate assembly language whose instructions can be implemented in a trivial way on a large number of common microcontrollers. This approach is the opposite of the treatment of De Wulf *et al.* [5], where the necessary hardware capabilities are calculated from a given specification.

To start, we look at what each type of outgoing transition of a specification automaton dictates must happen. A simple example transition is shown in Fig. 1, here enabled in isolation on $[0, a]$, with regard to condition 2 of Sec. 2.2. When viewed as a requirement, this transition states the following:

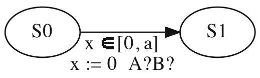


Fig. 1. Example transition

1. If message A or B arrives within a time units since entering state S0, the transition to state S1 is made and the clock variable x is reset to 0.
2. If neither message arrives within this time, the transition becomes disabled and inputs are no longer accepted. If other transitions outgoing from S0 exist, the clock variable is not reset to 0.

This interpretation is consistent with the definition of timed automata. The requirement can be implemented as follows:

1. After entering state S0 accept messages A and B for *at most* a time units.
2. If either message arrives in this time, proceed with subsequent actions.
3. If no message arrives, and S0 has no other outgoing transitions, halt.
4. If S0 has other outgoing transitions, proceed with their implementation.

In a *polling* approach to inputs, the software must enter a loop in which the channels via which messages A and B arrive are polled for a predetermined period of time, until one of the messages arrives. The order in which the channels are polled is not specified and so can be treated as underspecification. Therefore if both messages arrive, the choice of which is accepted can be considered to be nondeterministic. If no message arrives, the software must not proceed further, perhaps entering an infinite loop of inactivity. This is done for every transition of the specification.

The polling loop described above is specific to the time interval specified, $[0, a]$, and consists of instructions for reading the value of each channel, determining whether a message has arrived and keeping track of the passage of time. These instructions take a deterministic amount of time to execute. Depending on these execution times, the number of repetitions of the loop can be calculated and fixed *a priori*, as a function of the instructions in the loop, to span only this time interval. Therefore, assuming that the specification automaton is consistent, the behaviour specified for any time interval is unimplementable only if the hardware is too slow to execute all the specified behaviour in that interval. Fortunately, under this approach even a microcontroller driven by a 4 MHz clock signal can satisfy microsecond-level timing constraints while retaining the ability to discriminate among parallel inputs, and most significantly here, without making any assumption of ideal synchronous hardware behaviour. The decision to take a polling rather than an interrupt-driven approach is fundamental to our method. Interrupts are much more energy-efficient, but the mechanism is more difficult to model over a wide range of microcontrollers. One such treatment can be found in McEwan and Woodcock [14]. Moreover, the simple microcontrollers to which this approach is tailored have very low power consumption relative to more sophisticated high-speed, multi-core architectures, making polling an arguably viable approach.

3.2 Time Intervals and Counter Registers

The polling loops described above are executed a calculated number of times such that the bounds of the time intervals can be met as closely as possible. Controlling these executions are pre-calculated countdown values stored in counter registers. These values are decremented in a systematic fashion with each iteration. The number of registers required for an interval specified between any two states S_i and S_j is $N_{\mathrm{S}_i \to \mathrm{S}_j}$, and each one of these $N_{\mathrm{S}_i \to \mathrm{S}_j}$ registers receives a value which contributes to the total countdown for that interval. A larger number of registers will accommodate a wider time interval. Obtaining these values poses an integer optimization problem in $N_{\mathrm{S}_i \to \mathrm{S}_j}$ variables. These integer optimizations are not solved by the embedded implementation, but rather when the implementation is generated. Thus they do not introduce any time variations in the behaviour of the implementation.

3.3 Input and Output

In a sequential polling scheme such as ours, communication with the environment is generally a deterministic task, whether through digital I/O ports or off-chip circuitry. Our proposed implementation structure assumes that the state of inputs and outputs can be captured as a bit pattern that can be tested (for input) or modified (for output), and that communication carries no data payload. For the remainder we assume the I/O port mechanism, though the intermediate assembly language proposed below can cope with other mechanisms. In the I/O port setting, to each message appearing in the specification there will correspond a port of the microcontroller and a mask value. The mask is used to isolate the bits of the port whose asserted state indicate the receipt or dispatch of that message. The examples following deal only with active-high messages, though our tool can handle both types.

3.4 Intermediate Assembly Language

In order to implement the polling loops described in Sec. 3.1 a target microcontroller needs to provide facilities to address the following:

– Determining whether messages have arrived by obtaining port values and loading them to working registers for analysis.
– Emitting messages by writing arbitrary values out through ports.
– Counting down time by decrementing specific, pre-determined values stored in registers or other memory.

Fortunately these are not special-purpose requirements and all simple microcontrollers on the market (i.e., [22,11,9,16]) can satisfy them. Furthermore, from these requirements a necessarily simple intermediate assembly language can be extracted (Tab. 1) that is sufficient for their implementation *in general.* Each intermediate instruction can be implemented in turn on any microcontroller with either a single instruction or a small, positionally independent sequence of instructions, i.e., the structure of each sequence is independent of its location in the full body of code. This is a virtue of architectures without pipelines or caches that makes it possible to achieve guaranteed microsecond-level behaviour, without making the synchrony assumption, on an entire class of relatively slow, cheap and energy-efficient microcontrollers. An example translation to the PIC 18F452 instruction set [16] can be found in Sec. 6.

Table 1. Intermediate assembly language

Instruction	**Effect**
LI **register value**	Load the literal **value** into **register**.
DEC **register**	Decrement the value in **register** by 1.
J **label**	Unconditional jump to **label**.
JNZ **register label**	Jump to **label** if the value in **register** is not 0.
JZ **register label**	Jump to **label** if the value in **register** is 0.
LPR **port register**	Load the value at **port** to **register**.
LRP **register port**	Load the value in **register** to **port**.
AND **register mask**	Perform the bitwise AND of the value in **register** and the literal **mask**, store result in **register**.
OR **register mask**	Perform the bitwise OR of the value in **register** and the literal **mask**, store result in **register**.

A Note on Optimization Many microcontrollers relevant to this method provide individual instructions that have the same effect as a sequence of two or more instructions in the intermediate language proposed. For example, the intermediate sequence {LPR **p wreg**; AND **wreg m**; LRP **wreg p**} can be implemented on the PIC18F452 by the single instruction {ANDWF **p**, **1**, **0**}, but our proposal is that each of the three instructions be implemented in turn, displaying some obvious redundancy. Carrying out peephole optimization [15] on the proposed implementation would detect this redundancy and make this substitution, but the error in performing this optimization is two-fold. First, the formula for the time to execute the implementation of the intermediate sequence can not be described in a general way. Second, it would obliterate the correct timing behaviour of the implementation. Therefore, omission of any algorithmic optimization is central to the generality of this method.

4 Synthesis of Intermediate Implementations

For brevity we illustrate the synthesis method on the most general outgoing transition configuration allowed for any single state in the restricted formalism, that of multiple transitions specifying concurrent communication within a time window. Less generic scenarios have a similar structure and will not be elaborated here [3].

The outgoing transition configuration of any state of a specification automaton satisfying the restrictions in Sec. 2.2 can only be one of the following:

– Exactly one outgoing transition with no timing constraint (always enabled).
– One or more outgoing transitions with non-overlapping timing constraints.
– More than one outgoing transitions, where any overlapping time intervals are identical.

Therefore, for each state there exists a set of time intervals which never overlap (the identical overlapping timing constraints in the third case above yield one such interval). There exists a natural ordering in time for this set. As a result, the implementation can systematically step through the polling loops corresponding to each interval with the passage of time.

Due to the granularity of each polling loop, sub-interval endpoints can not be met exactly in general. For any interval $[l, u]$, a correct implementation must only attempt to operate within an interval $[l^\star, u^\star]$ such that $l \leq l^\star$ and $u^\star \leq u$. Taking the specified action outside the interval $[l, u]$ would not be correct with respect to the specification. Assume two intervals $[l_i, u_i]$ and $[l_j, u_j]$. Even if $u_i = l_j$ in the specification, the values $u_i^\star$ and $l_j^\star$ that can be satisfied by the implementation will not be equal, thus introducing a gap of inactivity between any two transitions. This is summarized in Fig. 2. The values $l^\star$ and $u^\star$ for the implementation can be calculated exactly as parameters for each polling loop. In the general case of N counter registers and s concurrent transitions on time interval $[a, b]$ (Fig. 3), we propose the intermediate implementation summarized in Fig. 4, where WR is the working register, R_i are the counter registers, $P_{A_{ij}}$ is

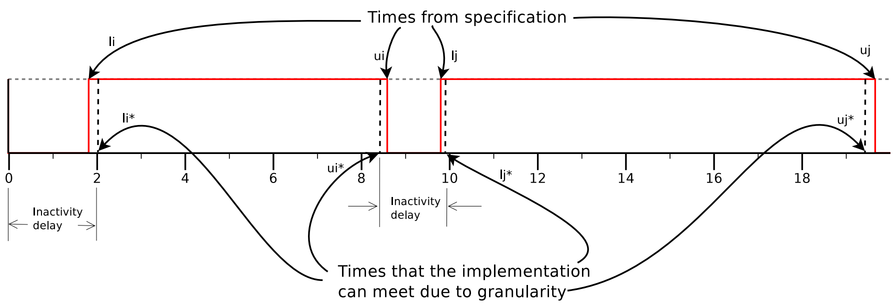


Fig. 2. Approximating time intervals at implementation time

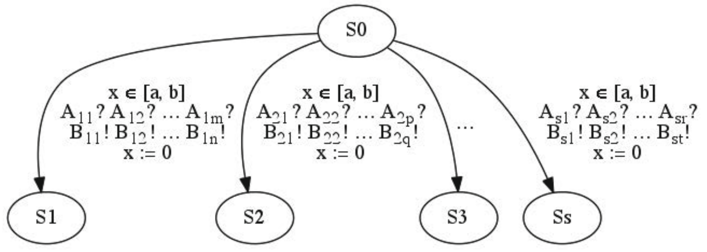


Fig. 3. Concurrent I/O requirements in time interval $[a, b]$

```
S0_0:         LI $R_1$ $\rho_1$
                 ⋮
              LI $R_N$ $\rho_N$
S0_1:         DEC $R_1$
              JZ $R_1$ S0_2
              J S0_(N + 1)
   ⋮
S0_N:         DEC $R_N$
              JZ $R_N$ S0_NoRcv
              LI $R_{N-1}$ $\rho_{N-1}$
                 ⋮
              LI $R_1$ $\rho_1$
              J S0_(N + 1)
S0_(N + 1):   LPR $P_{A_{11}}$ $WR$
              AND $WR$ $M_{A_{11}}$
              JNZ $WR$ S0_(N + 2)
                 ⋮
              LPR $P_{A_{21}}$ $WR$
              AND $WR$ $M_{A_{21}}$
              JNZ $WR$ S0_(N + 3)
                 ⋮
              J S0_1
S0_(N + 2):   LPR $P_{B_{11}}$ $WR$
              OR $WR$ $M_{B_{11}}$
              LRP $WR$ $P_{B_{11}}$
                 ⋮
              J S1_0
S0_(N + 3):   LPR $P_{B_{21}}$ $WR$
              OR $WR$ $M_{B_{21}}$
              LRP $WR$ $P_{B_{21}}$
                 ⋮
              J S2_0
                 ⋮
```

Fig. 4. Intermediate implementation of specification of concurrent inputs and outputs in time window $[a, b]$

the port on which message A_{ij} is received, $P_{B_{ij}}$ the port on which message B_{ij} is sent. Mask values $M_{A_{ij}}$ and $M_{B_{ij}}$ are used to isolate the port bits corresponding to messages A_{ij} and B_{ij}, respectively. The pre-calculated counter values ρ_i are loaded to their respective counter registers between labels S0_0 and S0_1. The countdown over the specified time interval is performed between labels S0_1 and S0_N, with the aid of the block at label S0_(N+1), where polling for the input messages takes place after each iteration of the loop from S0_1 and S0_N. Note that in this block of code the input is discriminated, so that the corresponding set of outputs B is emitted. The blocks starting at S0_(N + 2) output these corresponding messages and perform a jump to the correct subsequent state. If no specified input arrives in the prescribed time interval, the implementation jumps to the block of code at S0_NoRcv, usually code implementing the idle delay of duration $l_j^\star - b^\star$ (recall that currently $l_i = a$ and $u_i = b$) between the current cluster of transitions and the next transition to become enabled.

The values ρ_i to be stored in the countdown registers are calculated as follows.

Definition 2. *$T_{spec}(N)$ is the amount of time required by the target microcontroller to execute its implementation of the intermediate code fragment "spec" using N counter registers.*

Definition 3. *$\tau(i)$ is the amount of time required by the target microcontroller to execute its implementation of intermediate instruction i.*

To begin, the widest time interval specification that the code block above can implement for N registers is,

$$T_{max} = T_{setup}(N) + T_{delay}(N) + T_{exit}(N) \ .$$

The constituents of this formula and the base cases are as follows, where T_i is the total number of inputs, *MaxRegVal* is $2^8 - 1$ for 8-bit microcontrollers.

$$\begin{aligned}
T_{setup}(N) &= N\tau(\mathrm{LI}) \ . \\
T_{delay}(N) &= (\mathit{MaxRegVal} - 1)(T_{delay}(N-1) + T_{exit}(N-1) + \\
&\quad \tau(\mathrm{DEC}) + \tau(\text{false JZ}) + (N-1)\tau(\mathrm{LI}) + \tau(\mathrm{J}) + \\
&\quad T_i(\tau(\mathrm{LPR}) + \tau(\mathrm{AND}) + \tau(\text{false JNZ})) + \tau(\mathrm{J})) \ . \\
T_{exit}(N) &= T_{delay}(N-1) + T_{exit}(N-1) + \tau(\mathrm{DEC}) + \tau(\text{true JZ}) \ . \\
T_{setup}(0) &= T_{delay}(0) = T_{exit}(0) = 0 \ .
\end{aligned}$$

Next, the number of registers needed for the interval $[a, b]$ specified is calculated. This is the smallest integer R that satisfies $(b - a^\star) \leq T_{max}(R)$ (in general, $a^\star$ is known from the implementation of the previous interval or inactivity delay). Once an adequate number R of counter registers is chosen, the next step is to calculate the latest time $b^\star = T_{latest}(R)$ at which the implementation can still detect an input.

$$\begin{aligned}
T_{latest}(R) &= T_{lsetup}(R) + T_{ldelay}(R) + T_{lexit}(R) \ . \\
T_{lsetup}(R) &= R\tau(\mathrm{LI}) \ . \\
T_{ldelay}(R) &= (\rho_R - 1)(T_{ldelay}(R-1) + T_{lexit}(R-1) + \tau(\mathrm{DEC}) + \\
&\quad \tau(\text{false JZ}) + (R-1)\tau(\mathrm{LI}) + \tau(\mathrm{J}) + \\
&\quad T_i(\tau(\mathrm{LPR}) + \tau(\mathrm{AND}) + \tau(\text{false JNZ})) + \tau(\mathrm{J})) \ . \\
T_{lexit}(R) &= T_{ldelay}(R-1) + T_{lexit}(R-1) + \tau(\mathrm{DEC}) + \tau(\text{true JZ}) \ . \\
T_{ldelay}(0) &= T_{lexit}(0) = T_{lsetup}(0) = 0 \ .
\end{aligned}$$

In both cases termination of the mutual recursion is ensured by the monotonically decreasing values N and R, respectively. If an input arrives at the latest possible moment for which it must still be detected, then the implementation needs time to process the input, and also to generate the corresponding outputs before transitioning to the next state. All this must be done within the upper limit of the time interval. Therefore the values ρ_i are obtained by solving the following integer optimization problem, where T_{mo} is the largest number of outputs B,

$$\min \ \{(b - a^\star) - (T_{latest}(R) + T_{mo}(\tau(\mathrm{LPR}) + \tau(\mathrm{OR}) + \tau(\mathrm{LRP})) + \tau(\mathrm{J}))\} \ .$$

The optimum vector of values ρ_i is subject to,

$$T_{latest}(R) + T_{mo}(\tau(\mathrm{LPR}) + \tau(\mathrm{OR}) + \tau(\mathrm{LRP})) + \tau(\mathrm{J}) \leq (b - a^\star) \ .$$

The recursion ensures that the index i spans the interval $[1, R]$, yielding the register values $\rho_1, \rho_2, \ldots, \rho_R$ sought. The value $b^\star$ is used to implement the behaviour specified for the next time interval for state S0.

In summary, the code in Fig. 4 implements the behaviour specified by the group of transitions shown for the time interval $[a, b]$. The idle wait between any two adjacent intervals i and j corresponding to state S0 (not shown in Fig. 3) is implemented by a simpler loop whose ρ_i values are calculated similarly for the duration $l_j - u_i^\star$, yielding the value $l_j^\star$ required subsequently. This glue code binds individual transition implementations together into a complete implementation for any given state, and the process is repeated for each state of the specification. Most crucially, this resulting implementation is guaranteed to satisfy the timing requirements in the specification without making any synchrony assumption about the underlying hardware, by accounting for execution times for sending and receiving messages, and for switching between transition implementations, in the code implementing the individual timing requirements. By pushing this extra execution time inside the specified time intervals the implementations remain consistent with the specified timing behaviour.

5 Tool Support

We have developed an algorithm that implements this method with performance $\mathcal{O}(N + E)$ in the number of nodes N and edges E of the specification. The algorithm is partially implemented in a prototype tool[1] written in Haskell. The optimal countdown register values ρ_i are determined through exhaustive search. For narrow time interval requirements (less than a second) this implementation yields acceptable performance on a modern personal computer at the time of writing. All information is passed to the tool via XML files, including the specification automaton, and all XML processing is done using the Haskell HaXml package. The tool outputs the assembly listing, together with graph files corresponding to the original specification, the cleaned specification (with any transient states removed) and the implementation. These are in Graphviz Dot [10] format.

6 Example

In this section we apply our method to an example specification to illustrate the connection between intermediate and target assembly code.

6.1 Microcontroller Characteristics

The target microcontroller is the Microchip PIC 18F452 [16]. This microcontroller is relatively average in terms of core complexity in the class of microcontrollers

[1] Latest version may be obtained online from `http://www.cs.york.ac.uk/~bandurvp.`

with no pipelining and no caching. It is driven at 4 MHz and instructions take either four or eight cycles to execute, as described below, yielding execution times of either 1.0 μs or 2.0 μs per instruction. Of the 1536 bytes of RAM in the PIC 18F452, the lowest 128 bytes form the most conveniently addressable set of 8-bit registers. The accumulator register is accessed by its mnemonic, WREG, and is chosen as the working register. The instruction set is of RISC design and the timing characteristics of each instruction can be found in the unit's data sheet [16].

6.2 Instruction Mappings and Execution Times

Table 3 shows the PIC 18F452 implementation of the intermediate assembly language proposed in Sec. 3.4. Similar tables can be compiled for any microprocessor with deterministic instruction execution times.

Table 3. PIC 18F452 implementation of intermediate instructions

Instruction	**PIC 18F452 Code**	τ **(in** μ**s)**
LI **register value**	clrf WREG, 0 addlw **value** movwf **register**, 0	3.0
DEC **register**	decf **register**, 1, 0	1.0
J **label**	bra **label**	2.0
JZ **register label**	movf **register**, 0, 0 incf WREG, 0, 0 decf WREG, 0, 0 bz **label**	4.0 if false, 5.0 if true
JNZ **register label**	movf **register**, 0, 0 incf WREG, 0, 0 decf WREG, 0, 0 bnz **label**	4.0 if false, 5.0 if true
LPR **port WReg**	movf **port**, 0, 0	1.0
LRP **WReg port**	movwf **port**, 1	1.0
AND **WReg mask**	andlw **mask**	1.0
OR **WReg location**	iorlw **mask**	1.0

6.3 Example Specification

The timed automaton in Fig. 5 specifies a metronome that keeps a tempo amenable to measurement by oscilloscope (values are in microseconds). The indicator is a light toggle. The metronome can be started and stopped via the signals START and STOP. We assume that these signals come from a push button whose high and low levels correspond to START and STOP, respectively. The timed automaton model of the implementation (one of the outputs of the tool) is shown in Fig. 7. Note that the necessary platform-specific preamble is a section of initialization code (Fig. 6) that must precede the implementation proper. Similarly, cleanup code may be required, which in this case is the single command `end`, signifying the end of the listing. This code takes no part in the implementation of the specified behaviour.

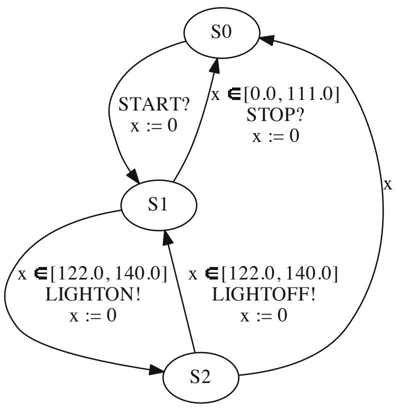


Fig. 5. Metronome specification

```
processor p18f452
#include "p18f452"
movlw 0x00  ;set pins to output
movwf TRISB
movwf PORTB ;turn light off
```

Fig. 6. Preamble code

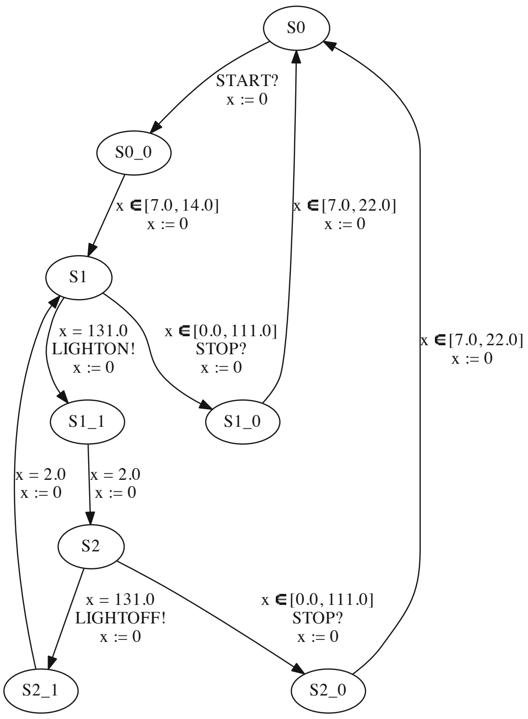


Fig. 7. Implementation behaviour

6.4 Results

A Tektronix TDS 1002 oscilloscope [18] was used to measure the timing characteristics of the implementation. The values measured at run-time of this implementation agree with those predicted on the behaviour automaton within 0.1 μs (Fig. 8). The key feature is the loop between states S1 and S2. The first image notes that the rising edge from the microcontroller's pin is offset by 0.700 μs from the oscilloscope's trigger point (arrow at the top left). The second figure shows the falling edge to be at 133.7 μs. Subtracting the offset we obtain a pulse width of 133 μs, correct with respect to both the specification and the model of the implementation.

7 Conclusions and Future Work

We propose a method for automatically refining timed automaton specifications of single tasks to microcontroller assembly code based on the timing characteristics of the target hardware rather than the behaviour specified. We only allow specifications written in a restricted subset of the timed automaton formalism.

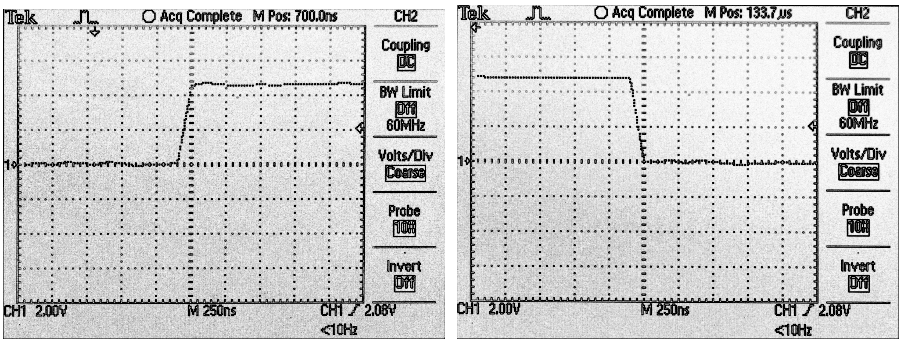


Fig. 8. Oscilloscope measurements of the metronome implementation

By targeting only simple microcontrollers without instruction pipelining, memory caching and other architectural improvements that trade determinism for speed, we can generate code that targets features common to all such microcontrollers and which is correct with respect to its timed specification. We have partially implemented our method in Haskell and have validated it on numerous example specifications by making measurements on the target hardware that agree with the specification. The ability to satisfy microsecond-level timing requirements on such simple microcontroller hardware in a generic way is very promising in terms of cost and guaranteed timing behaviour. Most importantly, we believe that the ability to automatically generate implementations that guarantee timing behaviour without making the synchrony assumption is a strong theoretical result.

As an immediate next step, we believe that it is possible to generalize this method to specifications with multiple clock variables while retaining acceptable performance across this class of microcontroller hardware. Jee *et al.* check and modify timers each time a guard condition is evaluated [12]. These actions would be necessary in such an extension to our method as well, but owing to the strictly sequential and deterministic nature of the implementation code, the time required to execute these actions can be taken into account in such a way that values for multiple clocks can be maintained. Of course this requires a corresponding restriction to the clock predicates allowed. It may also be possible to augment the method to accommodate global variables and actions on those variables, all without making the synchrony assumption. We have not yet investigated extending this approach to specifications with arbitrary tasks, as is done by some of the approaches reviewed in the introduction.

Acknowledgments. We wish to thank the anonymous reviewers for their helpful comments and suggestions.

References

1. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science 126, 183–235 (1994)
2. Amnell, T., Fersman, E., Mokrushin, L., Pettersson, P., Yi, W.: TIMES: A Tool for Schedulability Analysis and Code Generation of Real-time Systems. In: Larsen, K.G., Niebert, P. (eds.) FORMATS 2003. LNCS, vol. 2791, pp. 60–72. Springer, Heidelberg (2004)
3. Bandur, V.: Hard Real-Time Microcontroller Code Generation from Timed Automaton Specifications. Master's thesis, McMaster University (2008), `http://www.cs.york.ac.uk/~bandurvp`
4. Behrmann, G., Cougnard, A., David, A., Fleury, E., Larsen, K.G., Lime, D.: UPPAAL-Tiga: Time for Playing Games! In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 121–125. Springer, Heidelberg (2007)
5. De Wulf, M., Doyen, L., Raskin, J.F.: Almost ASAP semantics: From timed models to timed implementations. Formal Aspects of Computing 17(3), 319–341 (2005)
6. Dierks, H.: PLC-automata: A New Class of Implementable Real-time Automata. In: Bertrán, M., Rus, T. (eds.) ARTS 1997. LNCS, vol. 1231, pp. 111–125. Springer, Heidelberg (1997)
7. Dierks, H.: Synthesizing controllers from real-time specifications. IEEE Trans. on CAD of Integrated Circuits and Systems 18(1), 33–43 (1999)
8. Dierks, H.: PLC-automata: A new class of implementable real-time automata. Theoretical Computer Science 253 (2001)
9. Freescale Semiconductor: MC68HC08AB16A/D Data Sheet (July 2005), document MCH68HC08AB16A
10. Graphviz: Graphviz – graph visualization software (2011), `http://www.graphviz.org`
11. Intel Corporation: Intel MCS 51 Microcontroller Family User's Manual (February 1994), order 272383-002
12. Jee, E., Wang, S., Kim, J.-K., Lee, J., Sokolsky, O., Lee, I.: A safety-assured development approach for real-time software. In: RTCSA 2010, pp. 133–142. IEEE Computer Society (2010)
13. Jessen, J.J., Rasmussen, J.I., Larsen, K.G., David, A.: Guided Controller Synthesis for Climate Controller Using Uppaal Tiga. In: Raskin, J.-F., Thiagarajan, P.S. (eds.) FORMATS 2007. LNCS, vol. 4763, pp. 227–240. Springer, Heidelberg (2007)
14. McEwan, A.A., Woodcock, J.: Unifying Theories of Interrupts. In: Butterfield, A. (ed.) UTP 2008. LNCS, vol. 5713, pp. 122–141. Springer, Heidelberg (2010)
15. McKeeman, W.M.: Peephole optimization. Commun. ACM 8(7), 443–444 (1965)
16. Microchip Technology Inc.: PIC 18FXX2 Data Sheet (2002), document DS39564B
17. Mukund, M.: Finite-state automata on infinite inputs. Tech. Rep. TCS-96-2, SPIC Mathematical Institute (1996)
18. Tektronix, Inc.: TDS1000- and TDS2000-Series Digital Storage Oscilloscope (May 2011), document 071-1064-00
19. Tuttle, M.R., Lynch, N.A.: An introduction to input/output automata. CWI Quarterly 2(3), 219–246 (1989)
20. Wassyng, A., Lawford, M., Hu, X.: Timing Tolerances in Safety-Critical Software. In: Fitzgerald, J.S., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 157–172. Springer, Heidelberg (2005)
21. ChaoChen, Z., Hoare, C.A.R., Ravn, A.P.: A calculus of durations. Information Processing Letters 40(5), 269–276 (1991)
22. Zilog: Zilog Z80 Family CPU User Manual (2004), document UM008005-0205

Tool Chain to Support Automated Formal Verification of Avionics Simulink Designs*

Jiri Barnat[1], Jan Beran[2], Lubos Brim[1], Tomas Kratochvíla[2], and Petr Ročkai[1,**]

[1] Faculty of Informatics, Masaryk University,
Brno, Czech Republic
[2] Honeywell International,
Aerospace Advanced Technology Europe,
Brno, Czech Republic

Abstract. Embedded systems have become an inevitable part of control systems in many industrial domains including avionics. The nature of this domain traditionally requires the highest possible degree of system availability and integrity. While embedded systems have become extremely complex and they have been continuously replacing legacy mechanical components, the amount of defects of hardware and software has to be kept to absolute minimum to avoid casualties and material damages. Despite the above-mentioned facts, significant improvements are still required in the validation and verification processes accompanying embedded systems development. In this paper we report on integration of a parallel, explicit-state LTL model checker (DIVINE) and a tool for requirements-based verification of aerospace system components (HiLiTE, a tool implemented and used by Honeywell). HiLiTE and the proposed partial toolchain use MATLAB Simulink/Stateflow as the primary design language. The work has been conducted within the Artemis project industrial Framework for Embedded Systems Tools (iFEST).

1 Introduction

The complexity of embedded systems and of their architecture has been growing rapidly, primarily due to market demand for more functionality. Moreover, more advanced technologies are becoming available and are being combined in new and innovative ways. This complexity growth demands improved tool support. It is becoming impossible to engineer high quality systems in a cost efficient manner without extensive tool support, which is often coming from heterogeneous, disparate set of tools. It is of utmost importance that these tools work well together.

The safety-critical nature of systems developed in avionics industry increases otherwise high development cost even more. This is because thorough verification, validation, and certification processes must be involved in the development cycle. Automated formal verification methods, such as model checking [8], can significantly help lower

* This work has been partially supported by the Czech Science Foundation grants No. GAP202/11/0312 and GD102/09/H042 and by ARTEMIS-IA iFEST project grant No. 100203.

** Partially supported by Red Hat, Inc. and is a holder of Brno PhD Talent financial aid provided by Brno City Municipality.

the cost of verification and validation process. Incorporation of model checking into the development cycle may result into a development process capable of delivering systems at a reasonable cost and the required level of safety [9].

An important lesson learned from the effort spent on verification of avionics systems [4] shows that the major bottleneck preventing successful application of model checking to verification of avionics systems is the scalability of verification tools. While symbolic model checking tools have been successfully applied in model-based development based on Mathworks Simulink [14], their scalability is unfortunately quite limited. One of the factors limiting scalability is the absence of distributed tools: none of the symbolic model checkers can distribute the work among multiple computation nodes [7]. On the other hand, explicit-state model checking has been repeatedly reported to scale well in a distributed-memory environment [21,5].

Also it is not entirely clear that a symbolic approach is the best possible option for verification of hardware-software co-designed systems. While symbolic methods are more easily applied since they can naturally cope with data-related state space explosion, the explicit-state methods can still be a more efficient [12] alternative. However, some abstraction, ideally automated, is required to deal with data manipulated by the investigated model.

In this paper we aim at chaining tools used for development of embedded systems with parallel, explicit-state model checker DiVinE [2]. This allows for verification of systems against properties specified in Linear Temporal Logic (LTL), a standard formalism for expressing important behavioural properties, cf. Process Specification Language [19].

2 Tools in the Chain

2.1 Simulink

Simulink is a model-based design tool widely used in the aerospace industry. It allows design, simulation and code generation for dynamic and embedded systems [14].

A Simulink model is a design of the system built from a set of interconnected blocks. We will leave out continuous blocks (those with real-valued inputs and/or outputs) in this paper and will focus on discrete blocks: without special precautions, model checking in general (and DiVinE in particular) can only be used with discrete systems. Discrete blocks produce an output at specific points in time, governed by a global discrete clock. Simulink provides libraries of blocks and also allows user to define their own custom blocks. Connected blocks comprise a sub-system and multiple sub-systems are combined to create a hierarchical system model.

2.2 HiLiTE

Honeywell Integrated Lifecycle Tools and Environment (HiLiTE) is a tool used within Honeywell for the requirements-based verification of aerospace systems. HiLiTE has been qualified for use in RTCA DO-178B processes, and has been applied to the verification of thousands of practical Simulink models that are derived from a wide variety of domains, including flight control algorithms, discrete mode logic, built-in tests, hybrid perimeter controls and many others [3]. HiLiTE provides in-depth semantic analysis of models for design consistency and robustness, as well as automatic generation of certifiable code and of test vectors.

2.3 ForReq

In order to automate the validation and verification of requirements, they need to come in a machine-readable form. Whenever we refer to "formal requirements", it is implied that these are available in a suitable machine-readable format. Since the vast majority of legacy requirements are written in informal technical language it is necessary to formalize them. For this purpose, we have implemented a prototype of an HiLiTE extension (ForReq: Formalizing Requirements), a special requirements management tool that aims to simplify the task of formalizing requirements. It guides the user in creation of machine readable requirements from scratch or from legacy requirement documents.

ForReq provides the user with requirement patterns to ease the formalization process. Requirement patterns are similar to design patterns which are typically used in design tools. A requirement pattern consists of a requirement name and classification, a structured English specification without scope, a pattern intent, temporal logic mappings or other mappings, examples, and relationships.

Our requirement patterns are based on a Specification and Pattern System created by Dwyer [10] extended by real-time patterns by Konrad and Cheng [13]. Currently, ForReq is using an improved pattern system adapted to be aligned with aerospace requirements.

Once the requirement pattern and the scope is selected it contains so-called propositions (P, Q, R, S, T, and U) which need to be further refined. At this point our approach provides the engineer with the unique feature to assign a design (Simulink model) to the set of requirements. Our method obtains a list of variables from the design, allowing the engineer to select and use the variables in propositions (P, Q, R, ...). This also enables verification of the coverage of inputs and outputs with requirements and allows the tool to report any inputs or outputs that are not covered by at least one of the requirements.

Once the requirement is formalized, i.e. the requirement pattern is selected and all propositions are specified with logical formulae containing some model design variables, ForReq executes the DiVinE model checker and provides it with the assigned design and the temporal logic formula corresponding to the selected requirement pattern. We have developed a transformation from the design model (Simulink) to a form suitable as an input for the DiVinE model checker. Since every proposition within the temporal logic formulae are logical formulae consisting solely of design variables, the model checker can directly return a confirmation that the requirement is satisfied, or alternatively, a counterexample which demonstrates the behaviour that falsifies the requirement.

2.4 DiVinE

DiVinE is a tool for explicit-state LTL model checking and reachability analysis of discrete distributed systems [2]. The distinguishing quality of the tool is its ability to efficiently exploit the aggregate computing power of multiple network-interconnected multi-core workstations in order to deal with very large verification tasks, beyond the size handled by comparable sequential tools.

3 Model Checking with DiVinE

3.1 Linear Temporal Logic

Model checking is an automated formal verification procedure that takes a behavioral model of a system and decides whether the model satisfies a given property. Properties to be verified by the model-checking procedure vary significantly. Due to the exhaustive statespace search that is behind the model checking procedure, model checkers can formally prove properties such as an absence of a deadlock in a concurrent system, or (un)reachability of a particular system state. However, the real advantage of the technique lies in the ability to check for temporal properties of systems. An example of temporal property that is most often checked with a model checker is a response property, i.e., *System will always react by performing action R to every stimulus action S.*

Linear Temporal Logic (LTL) is often used in combination with explicit-state model checkers to formalize the properties. Formulae of LTL are built from the so-called *atomic propositions*, boolean-typed expressions evaluated on individual system states, and boolean and temporal operators. The key temporal operators used in LTL are the unary operators F (Future), G (Globally), X (Next) and the binary operator U (Until). Formulae of LTL describe properties of individual system runs. A run is a (possibly infinite) sequence of system states. A run satisfies the LTL formula $F\varphi$ if there is a state in the run (suffix of the run, to be more precise) where φ holds. In other words, $F\varphi$ says that φ holds true somewhere along the run (in the *future* of the initial state). Operator $G(\varphi)$ holds for a state on the run if all the following states of the run satisfy φ (i.e. φ is *globally* true). Operator $X(\varphi)$ requires φ to be valid in the next state. Finally, $\varphi U \psi$ requires that φ holds true, *until* ψ is satisfied at some state.

The LTL formalism allows the user to express many important properties of reactive systems. Examples of some LTL-expressible properties are: Infinite repetition of φ, expressed in LTL as $GF\varphi$, inevitable stability of φ, expressed as $FG\varphi$, or a sequence of (alternating) occurrences of φ and ψ, expressed as $((\neg\psi)U(\varphi \wedge (\neg\psi)) \wedge F(\psi))$.

To answer an LTL model checking question, an LTL model checking tool typically employs the automata-theoretic approach, which allows reducing the LTL model-checking problem to the problem of non-emptiness of a Büchi automaton. In particular, the model of a system S is viewed as a finite automaton A_S describing all possible behaviours of the system. The property to be checked (LTL formula φ) is negated and translated into a Büchi automaton $A_{\neg\varphi}$ describing all the behaviours violating φ. In order to check whether the system violates φ, a synchronous product $A_S \times A_{\neg\varphi}$ of A_S and $A_{\neg\varphi}$ is constructed describing all those behaviours of the system that violate φ, i.e. $L(A_S \times A_{\neg\varphi})=L(A_S) \cap L(A_{\neg\varphi})$. The automata A_S, $A_{\neg\varphi}$, and $A_S \times A_{\neg\varphi}$ are referred to as the system, property, and product automaton, respectively. System S satisfies formula φ if and only if the language of the product automaton is empty, which is if and only if there is no reachable accepting cycle in the underlying graph of the product automaton.

The LTL model checking problem is thus reduced to the problem of the detection of an accepting cycle in the product automaton graph.

3.2 Common Explicit-State Model Interface

An important part of the workload of the model-checking tool lies in the interpretation of the model under verification. The interpretation requires the model checker to compute successors of individual states of the model. To reduce this load, DiVinE offers an option to process the input models specified in DVE – DiVinE native modelling language – either as a text file, or as a binary file that results from compilation of the text file into a native binary code (in a form of dynamically loaded library) of the target hardware platform. It should be understood that the binary interface allows DiVinE to load and use binary models other than those resulting from compilation of native DVE modelling language, as long as they provide the mandatory portions of the binary interface. Henceforward, we refer to the binary interface as to "Common Explicit-State Model Interface" (CESMI for short).

CESMI can be used to provide DiVinE with the model to be verified at two different stages of the automata-based approach to LTL model checking. First, CESMI can be used to access the graph of the product Büchi automaton $A_S \times A_{\neg\varphi}$. Note that the graph of an automaton can be given either explictly by enumerating all its states, transitions, etc., or implicitly by providing functions to compute the explicit representation. CESMI assumes the implicit way of definition of the graph, therefore, it requires the following functions to be defined in the binary file:

- *initial state*, a nullary function (i.e. a constant)
- *successors*, a function from states to sets of states
- *is accepting* and *is goal*: unary boolean functions, classifying states as accepting and goal, respectively

The second option is to provide DiVinE with an implicit definition of the model graph only, i.e. graph of automaton A_S. DiVinE can compute the property automaton $A_S \times A_{\neg\varphi}$. internally from a given LTL formula φ and system automaton A_S given though the CESMI-compliant input file. However, to do so, DiVinE requires functions to evaluate atomic propositions used in the input LTL formula at individual states of the system automaton. Therefore, the basic CESMI interface has to be extended with functions to evaluate all atomic propositions used in the formula.

- *AP_a_* unary boolean function, classifying states based on validity of atomic proposition *a*

Should the latter case be used for LTL verification, the *combine* command of DiVinE must be issued on CESMI-specified model file and LTL formula, in order to produce a binary file with the implicit (CESMI) definition of the target product automaton.

An important fact to be understood is that parallel distributed-memory model checking capabilities as offered by DiVinE are independent of the actual model described through the CESMI interface. Once the model is given via CESMI, parameters of parallel and distributed-memory processing are simply part of particular DiVinE invocation.

4 The HiLiTE/DiVinE Interface

The main contribution of the proposed toolchain is a module that allows application of parallel explicit-state LTL model checking to data-flow oriented designs. Even though there is a certain amount of variance between various data-flow programming languages, they share many common traits. Data-flow systems are typically build up from blocks, which are connected using data connections. All the blocks typically execute in parallel, but unlike the standard concurrent software that is typically subject to model checking, this execution is synchronous across the whole system: all computation is done strictly in a lockstep manner.

Like with protocol design, the dataflow program often has no clear single line of execution: the systems are often reactive in nature. This naturally gives rise to the need for temporal properties.

4.1 Explicit-State Interpretation of Data-Flow Programs

In order to verify a data-flow program by means of model checking, we have to define an executable model of it. In our case, the primary language of interest is Matlab Simulink, which is widely used in design of avionic systems. For getting an executable model from the Matlab Simulink model, we opted for a two-level approach. First, we translate the Simulink model into a specific intermediate language of ours, and then we translate the intermediate representation into the form that is accepted by the model checker. The intermediate langugage was designed to be suitable as a target language also for other data-flow programming languages. The intermediate language is in fact a simple EDSL (embedded domain-specific programming language) for describing synchronous data circuits in C++. Programs in this domain specific language are produced by a dedicated compiler that will process Simulink designs.[1] The produced EDSL code can then be compiled using a standard C++ compiler, producing a shared object conforming to the *CESMI* specification (see Section 3.2).

This shared object can then be directly used as a verification subject, if we are interested in simple safety properties like presence of deadlocks. However, in the context of dataflow programs, this is not a very interesting question: more general safety and liveness properties can be expressed using LTL (see also Section 3.1).

To refer to “atomic” properties of the design in the specification of properties, the design needs to provide so-called “atomic propositions”. These are statements about individual system states, and are directly expressed in the EDSL translation of the model: in our case, the Simulink compiler is responsible for providing their definitions from a high level description. When these atomic propositions are specified, this is enough for the automated process of translating the LTL specification (which can refer to these atomic propositions, and derive).

The intermediate language provides support for creating blocks and connections between them. There are two main types of blocks: *atomic* and *composite*: implementation

[1] While the compiler which translates Simulink designs into our data-flow EDSL is proprietary (part of the ForReq extension of HiLiTE), the implementation of the intermediate EDSL itself is part of the DiVinE tool which is available for public download, including the source code, from `http://divine.fi.muni.cz`.

```
template< typename T >
struct Sum : Value< T > {
    InSet< T, 4 > in;

    virtual T get() {
        T acc = 0;
        typename InSet< T, 4 >::iterator i;
        for ( i = in.begin(); i != in.end(); ++i )
            acc += (*i)->get();
        return acc;
    }
};
```

Fig. 1. The data-flow EDSL in C++. An example (atomic) block with 4 inputs and a single output, implementing summation.

of atomic blocks needs to be done in terms of C++ code implementing the operation. An example of atomic block would be summation, product, minimum and the like: the source code for a summation atomic block is shown in Figure 1. Atomic blocks implement basic operations, and as such provide building blocks for *composite* blocks, which correspond to actual data-flow programs, or sub-programs. Since composite blocks can be naturally used as building blocks for other, more complex composite blocks, modular designs are easily expressible. In the Simulink translation, composite blocks are the natural counterpart of *sub-systems* as well as the entire *system* design.

The input program consists of a number of *blocks* with input and output *ports* and *connections*. The computation of the overall design is controlled by a global discrete clock. Every tick of the global clock triggers sampling on the system inputs (that are connected to the *environment*) and re-computation of all *port* values in the system based on data dependencies (represented by block *connections*).

While CTL is the common language for specifying synchronous systems, LTL is more commonly supported by model checkers geared towards asynchronous parallel systems, mainly software. Moreover, LTL model checkers more commonly use an explicit-state representation, which allows for parallel distributed-memory procesing. Since the application of parallel model checker was one of our primary goals, we needed to adapt the synchronous data-flow system for the purposes of explicit-state model checking. To this end, we needed to build an implicit representation of the state transition system corresponding to the behaviours encoded by the data-flow program.

We have so far only described data-flow programs on a very high level. To describe the transition system derived from a data-flow program, we need to look at the semantics of *blocks* in more detail. While it is an important feature of data-flow programming that most blocks contain no explicit memory – they operate solely on their inputs in a functional way – this is not a strict requirement. In fact, many designs require memoryful blocks: in the Simulink parlance, these are called *delay* and *Stateflow state machine* blocks. As for the delay block, the effect of the memory contained in a block is the introduction of a delay on the “signal” when it passes through a block. It is the state of this memory that introduces statefulness to data-flow programming (a memory-free data-flow program is stateless, and traditional LTL model checking cannot be applied, although it can still be meaningfully combined with a stateful environment – more on this in Section 4.2). Note that in current implementation our toolchain has no support for Stateflow blocks yet.

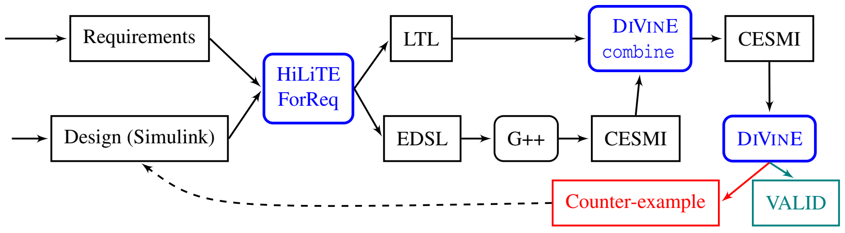


Fig. 2. The verification workflow based on the proposed toolchain

4.2 Environment

A reactive system, such as a data-flow program, is by definition an *open* system (it does not specify complete behaviours): the system reacts to stimuli from the *environment*: the environment comprises everything relevant that surrounds the reactive system in question. It could include other components of the entire system like sensors, actuators, other control units, and external factors such as weather, people and the like. In order to verify properties of the reactive system, we need to simulate this surrounding environment and its effects on the reactive system.

To this end, we create a simplified model of the environment and connect the inputs and outputs of the reactive data-flow program to the environment. To make a sensible environment model possible, we allow non-determinism in the actions originating in it: at any instant, a temperature rise or a drop might happen (although not both at the same time).

When the design and the environment model are connected, we obtain a *closed* system, and as such can be subject to model checking. However, the requirement to have a sensible environment can be an impediment to the verification process: it is another step that needs attention and manual work. Fortunately, for many purposes, the environment does not need to be very sophisticated, or even very sensible. In fact, a rather liberal environment puts more stringent requirements on the designed system itself. Additionally, it is easily possible to automatically generate an environment that is liberal in the extreme: one where *anything* might happen (we will say that this type of environment is *universal*). Such an environment is good enough surprisingly often, and can reveal overly optimistic assumptions the designer is making about the actual environment of the system.

4.3 Branching Explosion

Nevertheless, an universal environment has one major drawback in the presence of numeric inputs in the system: the environment actions corresponding to these inputs are of the form "value of input X has changed to Y", and in an universal environment, such an action will exist for each permissible Y. For inputs with large domains, this has the effect of creating extremely many actions, and consequently extremely many transitions in the resulting system. This in turn has disastrous effects on tractability of the model checking problem, since even simple systems can have millions of transitions between any their states. Since all model checking algorithms need to explore each transition to remain correct (unless they can prove a given transition to be redundant, an option that we will explore in the following section), this means that to make model checking practical in these situations, we need to limit the number of transitions (environment actions) that can happen.

One solution to this problem is to place simple constraints on the universal environment, without compromising our ability to generate such environments automatically. The universal environment is *stateless*: it remembers nothing and can do *anything* at any time. Especially when it comes to modelling variable quantities of the real world, this is overly pessimistic. If the environment is made *stateful*, it can remember the current values of the variables it models, and the limits can be in form "value X can change by at most Y in a single clock tick". Such limits can bring down the number of actions available at any given instant considerably. Unfortunately, this construction has the dual problem: now each environment variable with domain size n will cause existence of n distinct states, which again precludes efficient model checking. An approach akin to region construction could be used to address this latter problem though (this is a subject of future work; we expect this approach to be simpler than using a fully symbolic representation).

4.4 Data Abstraction

A more common, but more involved approach to deal with big domains is symbolic (abstract) representation. Instead of working with single values, the model checker manipulates entire sets of values. With this approach, the environment simply sets all its variables to carry their full domain as the symbolic value, which exactly corresponds to the notion of "any value". Then, it is the job of the model checker to refine the various sets adaptively: whenever a decision that depends on a variable value is done, and the answer will be different for different values in the current set, this set needs to be split (according to the predicate) and each part propagated along the control flow decision which has observed the respective part of the set. Since control flow decisions are comparatively rare in primarily data-flow based programs, this technique is likely to work very well. Nevertheless, efficient implementation of sets with the required operations is hard and laborious. In the domain of hardware design, application of BDDs (binary decision diagrams) in this role was a significant breakthrough. On the other hand, SAT and SMT solvers have been employed in software-oriented tools with some success, especially in the context of bounded model checking.

Nevertheless, none of these approaches is fully suitable for verification of generic, high-level data-flow programs. Arithmetic operations pose a challenge for BDD and SAT based representations, which are fundamentally bit-oriented. On the other hand SMT solvers are hard to implement, and scarce.

Fortunately, in many cases (especially in our case of data-flow oriented programming), a relatively simple static heuristic can be used to compute a set of abstract partitions that cover all behaviours of the program, by set-based, or rather symbolic, back-propagation. Using this partitioning, we can generate an abstract environment that will be substantially simpler. Additionally, we can entirely avoid set-based arithmetic in model checking, which would be required in a fully symbolic approach: instead, the abstract environment can directly use single concrete representatives from each abstract set of inputs, and the program can be interpreted concretely as usual.

5 Related Work

There are many types of tools for automated formal verification, each with their own strengths and weaknesses. Explicit state model checkers such as SPIN or DiVinE enumerate and store individual states. Implicit state (symbolic) model checkers such as NuSMV store sets of states using a compact representation (such as Binary Decision Diagrams). Alternatively, bounded model checkers such as CBMC use boolean formulae for symbolic representation and employ SAT solvers.

Satisfiability modulo theories (SMT) model checkers such as SAL and Prover use a form of induction to reason about models containing real numbers and unbounded arrays. Their properties need to be written in such a way that they can be proven by induction over an unfolding of the state transition relationship. For this reason, they tend to be more difficult to use than explicit and symbolic state model checkers.

Tools like SPIN, DiVinE, NuSMV, or SMC (a statistical model checker extension to UPPAAL) are not tightly integrated with a specific software design tool and can be used as stand-alone tools and possess a tool-specific input language. Application of the tool to models in "non-native" language usually requires either manual re-modelling of the system or, preferably, an automated translation between the two languages.

There is extensive work on translating Simulink/Stateflow models to modelling languages of specific model checkers. We will mention only some of those that we think are the most relevant. A translation into the native language of the symbolic model checker NuSMV is presented in [1]. The paper [11] suggests an analysis of Simulink models using the SCADE design verifier in the particular setting of system safety analysis. Another translation, this time from Stateflow to Lustre is described in [18]. Both share the interpretation of Simulink and Stateflow as a synchronous language, as mandated by the use of Lustre as the target language for the translations. Invariance checking of Simulink/Stateflow automotive system designs using a symbolic model checker is proposed in [20]. A first attempt to validate Stateflow designs using SPIN is presented in[16] (the described tool is however not publicly available).

More recently a team at Rockwell Collins has built a set of tools [22,9,15] that translate Simulink models into languages of several formal analysis tools, allowing direct analysis of Simulink models using model checkers and theorem provers. Among the tools considered are symbolic model checkers, bounded model checkers, model checkers using satisfiability modulo theories and theorem provers.

Unlike the above-mentioned tools, tools like SDL tool Telelogic TAU, the Statemate ModelChecker, SCADE Design Verifier [17], and the Simulink Design Verifier (an optional component of the Matlab Simulink tool set based on SAT technology) are tightly integrated with the design tool that they belong to. The design tools are typically domain specific. The Telelogic TAU tool, for instance, enjoys widespread popularity in telecommunications while Matlab Simulink is a de-facto standard for software design in automotive system engineering.

Our approach differs from the above-mentioned in using a parallel explicit-state LTL model checker to directly handle Simulink models and employing distributed memory to attack the state explosion problem. We are not aware of any other work with similar ambitions.

6 Use Case – Honeywell's Voter Core

The environment is automatically generated from a Simulink model. In each discrete time step, the environment ensures that the input blocks can assume any value within a certain range. For each input block the user specifies its permissible input range. Moreover the range can be further automatically reduced, as HiLiTE refines the actual range of each input and output of each block, using both forward and backward propagation of ranges of incident blocks using their semantics, through calculation of inverse functions for the analyzed blocks [3].

6.1 Voter Core Specification

Voter Core is the sub-system of the common avionics triplex sensor voter described in the paper [1]. However the Voter Core sub-system is not defined in the paper; therefore our engineer had to design it from scratch. The design shown in Figure 3 was given to the engineer with the task to design the highlighted Voter Core sub-system conforming to the following informal requirements:

1. If signal delta > 1 or signal is not valid, signal mismatch shall hold.
2. If signal mismatch held for 3 time units, permanent mismatch shall hold.
3. Once permanent mismatch holds it shall hold forever.

The picture in Figure 4 shows the Voter Core sub-system as designed by the engineer.

6.2 Formalizing Requirements

We will demonstrate how the ForReq tool guides the user to formalize requirements, by walking through the process of formalization of the last requirement mentioned above.

The user always starts with a scope and a specification and derives the complete requirement pattern step by step:

- ***scope*, *specification*.**

The scope of the requirement is after the "permanent mismatch holds" atomic proposition is valid. The partially formalized sentence will be:

- **After Q, *(order* $\vee$ *occurrence)*.**

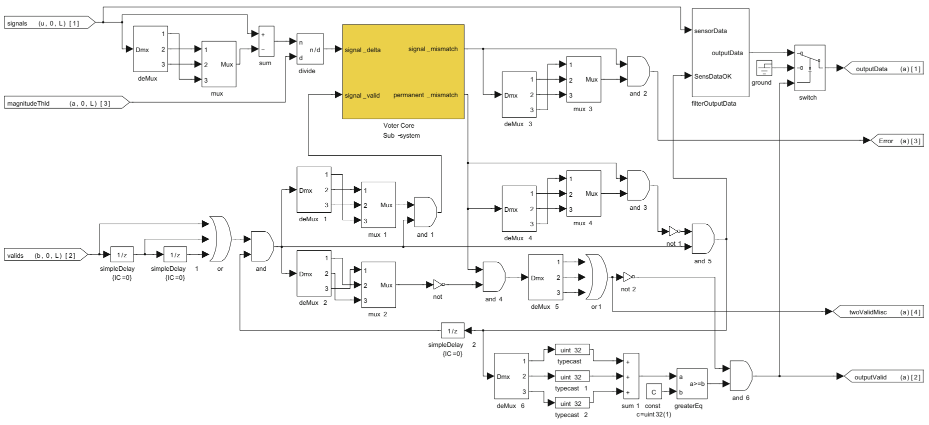


Fig. 3. The complete Voter subsystem, showing the context for the Voter Core component

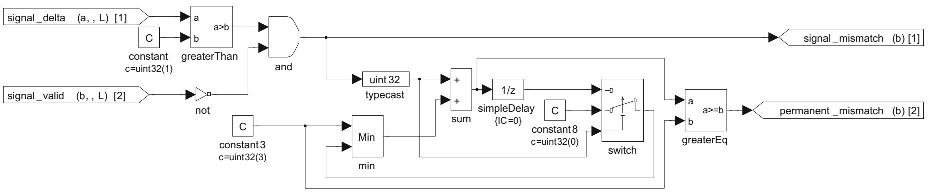


Fig. 4. The proposed design of the Voter Core

The user chooses the type of requirement pattern, occurence in this case. After that user chooses the universal requirement pattern:

– **After Q, it is always the case that P holds.**

ForReq then guides the user to assign the propositions Q and P with the logical formula with atomic propositions as variables that are directly mapped to Simulink model variables. In our use case the propositions $Q = P = \mathit{permanent\_mismatch}$ are valid only if *permanent_mismatch* signal is true.

The engineer eventually arrives at the following formalized requirements:

1. Globally, it is always the case that if (signal_delta > 1 $\vee\neg$signal_valid) holds, then (signal_mismatch) holds immediately.
2. Globally, it is always the case that if (signal_mismatch) holds for at least 3 time unit(s), then (permanent_mismatch) holds immediately.
3. After (permanent_mismatch), it is always the case that (permanent_mismatch) holds.

And the corresponding (automatically generated) LTL formulae:

1. $G((\text{signal\_delta} > 1 \vee \neg\text{signal\_valid}) \rightarrow (\text{signal\_mismatch}))$

2. $G(G_{<3}(\text{signal\_mismatch}) \rightarrow F_{=3}(\text{permanent\_mismatch}))$
3. $G(\text{signal\_mismatch} \rightarrow G(\text{signal\_mismatch}))$

The $G_{<i}$ and $F_{=i}$ operators are LTL extensions defined recursively as follows:

$$F_{=0}(\varphi) \equiv \varphi \tag{1}$$
$$F_{=i+1}(\varphi) \equiv X(F_{=i}\varphi) \tag{2}$$
$$G_{<0}(\varphi) \equiv true \tag{3}$$
$$G_{<i+1}(\varphi) \equiv \varphi \wedge X(G_{<i}(\varphi)) \tag{4}$$

6.3 Verification of Requirements

The Simulink model is translated to an intermediate XML representation and transformed, using XSLT, to C++ code conforming to CESMI (see Section 3.2).

Requirements in the form of LTL formulae and the corresponding CESMI design are then automatically sent to DiVinE model checker for verification. LTL formulae are stored in the following format (for example `VC.ltl`):

```
#property G(p0)
#property G(q1->G(p1))
#property G(p2*X(p2)*XX(p2)*XXX(p2)->XXXs2)
```

All atomic propositions (*p0*, *p1*, *p2*, and *s2*) are encoded as boolean C++ function so that the model checker can, in each state of the system, evaluate whether the atomic proposition is valid. Voter Core system together with abovementioned functions is compiled using `g++ VC.cpp -c` command.

Having both the system design and the specification, DiVinE is used to convert the LTL specification to the Büchi automaton and synchronize it with the system automaton by issuing `divine combine VC.o -f VC.ltl`.

For each LTL formula a `VC.prop*.so` file is created. DiVinE then reports, for each requirement, whether it is satisfied using the commands: `divine owcty VC.prop*.so`. In the negative case a counterexample that demonstrates the wrong behaviour is shown.

In the Voter Core use case only the last requirement was initially satisfied. The engineer actually made two fatal mistakes. The first requirement was not satisfied due to the engineer using an *and* logical block instead of *or*. The second requirement was not satisfied since the design did not treat the permanent mismatch as really permanent and the counterexample was that after 3 mismatch signals the permanent mismatch held; however after a few correct signals the permanent mismatch output was (wrongly) turned off.

In this use case the informal requirements ambiguity had not caused any defects. However in other cases, the user is forced – in order to successfully carry out the formalization – to choose among several options. This often helps to improve the overall quality of requirements by reducing ambiguity.

7 Conclusions and Future Work

We have presented a toolchain that allows engineers to employ explicit-state LTL model checking in the context of an established design verification framework, HiLiTE. The integration allows easy formulation of machine-readable temporal requirements (both safety and liveness) and exhaustive automated verification of such requirements in concrete Simulink designs. This in turn improves the verification process for mission-critical components designed in the data-flow programming environment of Simulink, which is frequently deployed in aerospace and automotive industries, both with stringent quality requirements. The improved process represents potential resource savings (by replacing expensive manual work with automated tools) and possible reductions in time-to-market, further increasing efficiency.

The toolchain as presented in this paper is already usable for verification of small to medium-sized stateful components, as illustrated by the *Voter Core* use case presented in Section 6. At the moment the tool chain is able to fully parse and process 31 types of Simulink blocks. On the other hand, there is room for improvements: the data abstraction process presented in Section 4.4 is still partially manual, but can be fully automated, and such automation is subject to a planned future extension of the toolchain. This will broaden the scope of the integration to more complex components, while at the same time reducing the amount of manual work required. We also intend to extend the toolchain to Stateflow diagrams.

References

1. Meenakshi, B., Bhatnagar, A., Roy, S.: Tool for Translating Simulink Models into Input Language of a Model Checker. In: Liu, Z., Kleinberg, R.D. (eds.) ICFEM 2006. LNCS, vol. 4260, pp. 606–620. Springer, Heidelberg (2006)
2. Barnat, J., Brim, L., Černá, I., Moravec, P., Ročkai, P., Šimeček, P.: DiVinE – A Tool for Distributed Verification (Tool Paper). In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 278–281. Springer, Heidelberg (2006)
3. Bhatt, D., Madl, G., Oglesby, D., Schloegel, K.: Towards Scalable Verification of Commercial Avionics Software (2010), `http://www.ics.uci.edu/~gabe/papers/BMOS_AIAA_2010.pdf`
4. Bhatt, D., Schloegel, K.: Effective Verification of Flight Critical Software Systems: Issues and Approaches. Presented at NSF/Microsoft Research Workshop on Usable Verification (November 2010)
5. Bingham, B., Bingham, J., de Paula, F.M., Erickson, J., Singh, M., Reitblatt, G.: Industrial Strength Distributed Explicit State Model Checking. In: Parallel and Distributed Methods in Verification and High Performance Computational Systems Biology (HiBi/PDMC), pp. 28–36. IEEE (2010)
6. Choi, Y.: From NuSMV to SPIN: Experiences with model checking flight guidance systems. Formal Methods in System Design 30, 199–216 (2007)
7. Ciardo, G., Zhao, Y., Jin, X.: Parallel symbolic state-space exploration is difficult, but what is the alternative? In: Parallel and Distributed Methods in Verification (PDMC). EPTCS, vol. 14, pp. 1–17 (2009)
8. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT press (1999)
9. Cofer, D.: Model Checking: Cleared for Take Off. In: van de Pol, J., Weber, M. (eds.) SPIN 2010. LNCS, vol. 6349, pp. 76–87. Springer, Heidelberg (2010)

10. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: A System of Specification Patterns (1998), http://www.cis.ksu.edu/santos/spec-patterns
11. Joshi, A., Heimdahl, M.P.E.: Model-Based Safety Analysis of Simulink Models Using SCADE Design Verifier. In: Winther, R., Gran, B.A., Dahll, G. (eds.) SAFECOMP 2005. LNCS, vol. 3688, pp. 122–135. Springer, Heidelberg (2005)
12. Kim, M., Choi, Y., Kim, Y., Kim, H.: Formal Verification of a Flash Memory Device Driver – An Experience Report. In: Havelund, K., Majumdar, R. (eds.) SPIN 2008. LNCS, vol. 5156, pp. 144–159. Springer, Heidelberg (2008)
13. Konrad, S., Cheng, B.H.C.: Real-time specification patterns. In: Proceedings of the 27th International Conference on Software Engineering, ICSE 2005, pp. 372–381. ACM, New York (2005)
14. Mathworks. Simulink, http://www.mathworks.com/products/simulink/
15. Miller, S.P.: Bridging the Gap Between Model-Based Development and Model Checking. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 443–453. Springer, Heidelberg (2009)
16. Pingree, P., Mikk, E., Holzmann, G., Smith, M., Dams, D.: Validation of mission critical software design and implementation using model checking. In: Proc. Digital Avionics Systems Conference, pp. 6A4-1–6A4-12. IEEE Computer Society (2002)
17. SCADE. Design verifier, http://www.esterel-technologies.com/products/scade-suite/add-on-modules/design-verifier
18. Scaife, N., Sofronis, C., Caspi, P., Tripakis, S., Maraninchi, F.: Defining and translating a "safe" subset of simulink/stateflow into lustre. In: EMSOFT, pp. 259–268. ACM (2004)
19. Schlenoff, C., Gruninger, M., Tissot, F., Valois, J., Road, T.C., Inc, S., Lubell, J., Lee, J.: The Process Specification Language (PSL) Overview and Version 1.0 Specification (1999)
20. Sims, S., Cleaveland, R., Butts, K., Ranville, S.: Automated validation of software models. In: ASE, pp. 91–102. IEEE Computer Society (2001)
21. Verstoep, K., Bal, H., Barnat, J., Brim, L.: Efficient Large-Scale Model Checking. In: 23rd IEEE International Parallel & Distributed Processing Symposium (IPDPS 2009). IEEE (2009)
22. Whalen, M., Cofer, D., Miller, S., Krogh, B.H., Storm, W.: Integration of Formal Analysis into a Model-Based Software Development Process. In: Leue, S., Merino, P. (eds.) FMICS 2007. LNCS, vol. 4916, pp. 68–84. Springer, Heidelberg (2008)

Range Analysis of Binaries with Minimal Effort

Edd Barrett and Andy King

School of Computing, University of Kent, CT2 7NF, UK

Abstract. COTS components are ubiquitous in military, industrial and governmental systems. However, the benefits of reduced development and maintainance costs are compromised by security concerns. Since source code is unavailable, security audits necessarily occur at the binary level. Push-button formal method techniques, such as model checking and abstract interpretation, can support this process by, among other things, inferring ranges of values for registers. Ranges aid the security engineer in checking for vulnerabilities that relate, for example, to integer wrapping, uninitialised variables and buffer overflows. Yet the lack of structure in binaries limits the effectiveness of classical range analyses based on widening. This paper thus contributes a simple but novel range analysis, formulated in terms of linear programming, which calculates ranges without manual intervention.

1 Introduction

Where once reverse engineering was the preserve of the black-hat, now binaries are routinely inspected members of the intelligence community, military organisations and employees of security firms. For these parties, an area of concern is the security of commercial off-the-shelf software (COTS) such as linkable code libraries [30]. COTS is increasingly deployed since it reduces development times, but such code is written by third-parties, typically with an eye towards functionality rather than security and reliability [11]. COTS could corrupt the system on which it is running or, more insidious still, introduce a trojan horse. The threat posed by COTS is significant motivating security audits which, since the source code is unavailable, are necessarily conducted at the binary level.

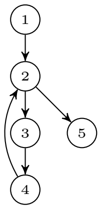
Surprisingly, buffer overflow deficiencies are still very popular targets for cyber-criminals [1]. Programs with buffer overflow deficiencies may fall victim to (amongst others) privilege escalation or code injection attacks. Often range information can help in identifying such deficiencies by, for example, asserting that an array index may be out of bounds. Whilst it is recognised that range information can aid the security engineer in the auditing process [12], industrial decompilers do not currently infer ranges for the values stored in basic data-types (though one commercial tool vendor recently mentioned this on its wish list). Range analysis is the pedagogical example that is used to illustrate the need for the widening and narrowing in program comprehension [10]. Even for finite-precision integers, the domain of ranges (also known as intervals) $D = \{\emptyset\} \cup \{[l, u] \mid -2^{31} \leq l \leq u \leq 2^{31} - 1\}$ admits long ascending chains such as

$d_0 = \emptyset$ and $d_{n+1} = [0, n]$ where $n \in [0, 2^{32} - 1]$. The force of this is that fixpoint acceleration aka. widening need be applied to compute an over-approximation of the ranges for registers that arise in loops. The idea behind widening is to accelerate convergence by leap frogging over intermediate points in the chain. To illustrate, observe that the lower bound in the chain d_0, d_1, d_2 is stable after d_1 whilst the upper bound is strictly increasing. Widening would typically enlarge, literally widen, d_3 to $[0, 2^{31} - 1]$ to preserve the lower bound of 0 whilst relaxing the upper bound to the maximum representable signed integer. This side-steps the generation of the intermediates $d_4, \ldots, d_{2^{31}-2}$.

One does not need to relax an unstable bound to the largest, or conversely the smallest, representable number in a single step. Instead, one can prescribe a set of increasing thresholds which are widened to in a series of steps. If relaxing a bound to one threshold is not sufficient for stability then, at the next step, the bound is related the next threshold, and so on. This is called widening with thresholds [6] yet it requires the thresholds to be specified a priori. With a view towards automation, widenings [14,25] have been suggested that infer the thresholds based on the structure of a program, in particular, where a transition in a chain from one interval to the next flips an inequality from unsatisfiable to satisfiable. The inequalities in question are those that occur in a control structure such as a conditional branch or a loop condition, the intuition being that the larger interval enables a new path through the program to be reached. However, quite apart from reasoning about the satisfiability of systems of inequalities, such widenings rely on extracting inequalities from the program, a problem that is straightforward for a source program, but difficult for its binary counterpart.

Iterative methods based on widening are sound in that they infer ranges which enclose any value that can reside in a register. Such analyses actually compute a post-fixpoint, though the most desirable solution is the least fixpoint which presents the best over-approximation of the intervals. This raises the question of whether least fixpoint can be found directly, dispensing with the need for iteration and widening. Different responses to this question are represented in the works of [15,22,26] that compute ranges by, respectively, mixed integer programming, parametric linear programming, and a mixture of transformation and chaotic iteration. The latter approach, in effect, proposes a constraint solver for a class of range constraints that can be solved in polynomial time. The former approaches, attempt to exploit existing mathematical or linear programming packages, though this presents the problem of how to express the fixpoint as an optimisation problem. This is not straightforward and indeed the way branching conditions are encoded in [22] is unsound for some classes of loop. (In the spirit of the call for papers, this shortcoming is discussed in Sect. 5 as well as its relationship to follow-on work [29]). Whether by design or by accident, the mathematical programming formulation, which is subsequently linearised, seems to avoid this problem but the encoding is not straightforward and it is not easy to validate the method due to its conceptual complexity. In this paper we extend existing techniques by, paradoxically, stripping them down. In doing so, we make the following contributions:

```
    assume (m ∈ {5, 20});
(1) i := 10;
(2) while (i ≥ m)
(3)       m := m + 1;
(4) end
(5)
```



$$
\begin{aligned}
S_1 &= \{\langle i, m\rangle \mid -2^{31} \leq i \leq 2^{31}-1 \wedge m \in \{5, 20\}\}\\
S_2^\star &= \{\langle 10, m\rangle \mid \langle i, m\rangle \in S_1\}\\
S_2 &= S_2^\star \cup S_4\\
S_3 &= \{\langle i, m\rangle \mid \langle i, m\rangle \in S_2 \wedge i \geq m\}\\
S_4 &= \{\langle i, \min(m+1, 2^{31}-1)\rangle \mid \langle i, m\rangle \in S_3\}\\
S_5 &= \{\langle i, m\rangle \mid \langle i, m\rangle \in S_2 \wedge i < m\}
\end{aligned}
$$

Fig. 1. (a) Program code (b) CFG and (c) collecting semantics

- We show how range analysis can be formulated, in the words of the title "with minimal effort", using systems of min and max constraints;
- We show how systems of such constraints can be solved by repeatedly calling a linear programming package;
- We show how the number of calls to the package can be significantly reduced by solving the linear programs in a propitious order.

The structure of the remainder of this paper is as follows. Sect. 2 shows a worked example of our analysis over a program, then in Sect. 3 we detail how we solve systems of inequalities containing disjunctions using repeated linear programming (LP). Experimental results of our analysis applied to several small programs are presented in Sect. 4 and in Sect. 5 we discuss some shortcomings in existing work that influenced the design of our analysis. Related work is discussed in Sect. 6 before we conclude the main body of the paper in Sect. 7.

2 Worked Example

In this section we explain how ranges can be derived without resorting to widening. Although our work is targeted at the binary level, we will introduce the ideas in terms of a generic high-level program so as to aid comprehension.

2.1 Collecting Semantics

Fig. 1a shows a small program with the program points annotated (1) through to (5). Our problem is how to summarise the program state at each of these points without actually running the program. We start by considering a natural set representation of all possible values of i and m at each program point. We aim to compute the smallest hyper-rectangle (a tuple of intervals) summarising all possible i and m combinations for each program point. To this end, the state at a single program point is expressed as a 2-dimensional vector $\langle i, m\rangle$, thus the states that can possibly arise at these 5 program points is described by 5 sets of vectors, namely $S_k \subseteq [-2^{31}, 2^{31}-1]^2$. Each set S_k is finite, though possibly large, since we suppose that i and m are represented by 32-bit signed integers.

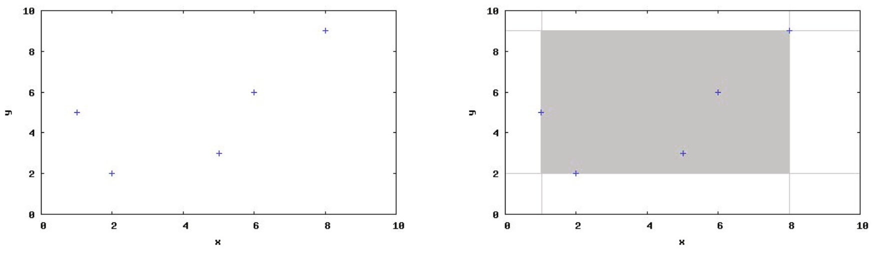


Fig. 2. (a) $S = \{\langle 2,2\rangle, \langle 5,3\rangle, \langle 1,5\rangle, \langle 6,6\rangle, \langle 8,9\rangle\}$ (b) $\alpha(S) = [1,8] \times [2,9]$

Fig. 1c presents a system of recursive equations that define and relate the sets $S_1, \ldots, S_5$. The dependencies between the program points, hence the sets, are illustrated in Fig. 1b. Note S_2 is defined in terms of S_4 and S_2^* which, in turn, is defined in terms of S_1. This is because control passes from (1) and (4) to program point (2). The set S_2^* is merely introduced as a calculational device (an intermediate set) that is used to decompose S_2 into an update operation and a merge operation, that define S_2^* and S_2 respectively. Note too that the increment operation at line (3) can potentially overflow, though it does not in this example. Instead of separately modelling the two modes of the increment: the exact mode when the increment does not wrap, and the overflow mode, and then distinguishing between these two modes with a guard, we simplify the presentation by modelling the overflow with a min operation. Together these equations can be considered as defining a collecting semantics [9] for the program; a semantics over sets that provides a basis for abstraction.

2.2 Abstract Semantics

Every set $S \subseteq [-2^{31}, 2^{31}-1]$ can be described by an interval drawn from the abstract domain $D = \{\emptyset\} \cup \{[l,u] \mid -2^{31} \leq l \leq u \leq 2^{31}-1\}$. Moreover, an n-ary tuple of intervals can describe a set of n-ary vectors, an idea that is formalised with an abstraction α mapping and a concretisation γ mapping [9]. The latter map explains how to interpret an n-tuple of intervals and the former specifies how best to describe a set $S \subseteq [-2^{31}, 2^{31}-1]^n$. These maps are defined thus:

$$\begin{array}{ll}
\gamma : D^n \to \wp([-2^{31}, 2^{31}-1]^n) & \alpha : \wp([-2^{31}, 2^{31}-1]^n) \to D^n \\
\quad \gamma(\emptyset) = \emptyset & \qquad \alpha(\emptyset) = \emptyset \\
\quad \gamma(S') = S' & \qquad \alpha(S) = \cap\{S' \in D^n \mid S \subseteq S'\}
\end{array}$$

Note how an n-tuple of intervals $\langle d_1, \ldots, d_n\rangle \in D^n$ is interpreted as its cartesian product $d_1 \times \ldots \times d_n$ which defines an hyper-rectangle in n-dimensional space. Thus the subset ordering on D naturally lifts to D^n by $\langle d_1 \ldots d_n\rangle \subseteq \langle e_1 \ldots e_n\rangle$ iff $d_i \subseteq e_i$ for all $i \in \{1, \ldots, n\}$. Observe too how $\alpha(S)$ is defined as the least hyper-rectangle that encloses S. Fig. 2 illustrates $\alpha(S)$ for a set S that is planar.

Abstraction and concretisation relate sets of vectors to hyper-rectangles. With this relationship in place, we can relax the collecting semantics given previously, to a system of recursive equations that operate over hyper-rectangles rather then arbitrary sets. Each S'_k is designed to faithfully characterise S_k in that $S_k \subseteq \gamma(S'_k)$. This relationship can be shown to hold inductively when:

$$\begin{aligned}
S'_1 &= \{\langle i, m\rangle \mid -2^{31} \leq i \leq 2^{31} - 1 \wedge 5 \leq m \leq 20\} \\
S'^{*}_2 &= \{\langle 10, m\rangle \mid \langle i, m\rangle \in S'_1\} \\
S'_2 &= \alpha(\ S'^{*}_2 \cup S'_4\ ) \\
S'_3 &= \alpha(\ \{\langle i, m\rangle \mid \langle i, m\rangle \in S'_2 \wedge i \geq m\}\ ) \\
S'_4 &= \alpha(\ \{\langle i, \min(m+1, 2^{31}-1)\rangle \mid \langle i, m\rangle \in S'_3\}\ ) \\
S'_5 &= \alpha(\ \{\langle i, m\rangle \mid \langle i, m\rangle \in S'_2 \wedge i < m\}\ )
\end{aligned}$$

When incrementing m (S'_4) the resulting upper bound of may saturate. It is possible to obtain a more faithful model of integer overflow using integer linear programming, but in the interest of brevity we refrain from presenting this idea here. Since the above semantics is derived as an abstraction of the collecting semantics, henceforth it will be referred to as the abstract semantics.

2.3 Direct Calculation of the Abstract Semantics

The abstract semantics can be evaluated by iteratively applying the above equations, with widening, until stability is achieved. This does not necessarily give the least (best) solution due to the approximation introduced by widening. However, the hyper-rectangles can be found directly by solving systems of equations. Let $S'_1 = [l_{i,1}, u_{i,1}] \times [l_{m,1}, u_{m,1}], \ldots, S'_5 = [l_{i,5}, u_{i,5}] \times [l_{m,5}, u_{m,5}]$. The solution to the following reformulation (as an optimisation problem) is the least fixed point of the abstract semantics:

$$\text{Minimise}: \sum_{j=1}^{5}(u_{i,j} - l_{i,j}) + \sum_{j=1}^{5}(u_{m,j} - l_{m,j})$$

subject to the (non-linear) constraints:

$$\begin{array}{rclcrcll}
l_{i,1} &=& -2^{31} & \wedge & u_{i,1} &=& 2^{31} - 1 & \wedge \\
l_{i,2^*} &=& 10 & \wedge & u_{i,2^*} &=& 10 & \wedge \\
l_{i,2} &=& \min(l_{i,2^*}, l_{i,4}) & \wedge & u_{i,2} &=& \max(u_{i,2^*}, u_{i,4}) & \wedge \\
l_{i,3} &=& \max(l_{i,2}, l_{m,2}) & \wedge & u_{i,3} &=& u_{i,2} & \wedge \\
l_{i,4} &=& l_{i,3} & \wedge & u_{i,4} &=& u_{i,3} & \wedge \\
l_{i,5} &=& l_{i,2} & \wedge & u_{i,5} &=& \min(u_{i,2}, u_{m,2} - 1) & \wedge \\
l_{m,1} &=& 5 & \wedge & u_{m,1} &=& 20 & \wedge \\
l_{m,2^*} &=& l_{m,1} & \wedge & u_{m,2^*} &=& u_{m,1} & \wedge \\
l_{m,2} &=& \min(l_{m,2^*}, l_{m,4}) & \wedge & u_{m,2} &=& \max(u_{m,2^*}, u_{m,4}) & \wedge \\
l_{m,3} &=& l_{m,2} & \wedge & u_{m,3} &=& \min(u_{i,2}, u_{m,2}) & \wedge \\
l_{m,4} &=& \min(l_{m,3} + 1, 2^{31} - 1) & \wedge & u_{m,4} &=& \min(u_{m,3} + 1, 2^{31} - 1) & \wedge \\
l_{m,5} &=& \max(l_{m,2}, l_{i,2} + 1) & \wedge & u_{m,5} &=& u_{m,2} &
\end{array}$$

The cost function asserts that the desired solution is the least (best) hyper-rectangle that satisfies all of the constraints. Of particular note are the constraints $l_{i,2} = \min(l_{i,2^*}, l_{i,4})$ and $u_{i,2} = \max(u_{i,2^*}, u_{i,4})$ which assert that

$[l_{i,2}, u_{i,2}]$ is the smallest interval that encloses both $[l_{i,2^*}, u_{i,2^*}]$ and $[l_{i,4}, u_{i,4}]$. Likewise $l_{m,2} = \min(l_{m,2^*}, l_{m,4})$ and $u_{m,2} = \max(u_{m,2^*}, u_{m,4})$ assert tight bounds on m. In combination, these four constraints symbolically define S'_2 as the merge of the hyper-rectangles S'_1 and S'^*_2. Modelling the loop condition $i \geq m$ is a particular subtlety. Note how $l_{i,3} = \max(l_{i,2}, l_{m,2})$ and $u_{i,3} = u_{i,3}$ strengthen (not weaken) the lower bound of i but preserve its upper bound. Conversely $l_{m,3} = l_{m,2}$ and $u_{m,3} = \min(u_{i,2}, u_{m,2})$ refine the upper bound of m but preserve its lower bound. An analogous construction is used to model the loop exit condition.

Solving the above (with the technique outlined in the following section) we find the following ranges:

$$\begin{array}{ll} S'_1 = [-2^{31}, 2^{31} - 1] \times [5, 20] & \\ S'_2 = [10, 10] \times [5, 20] & S'_4 = [10, 10] \times [6, 11] \\ S'_3 = [10, 10] \times [5, 10] & S'_5 = [10, 10] \times [11, 20] \end{array}$$

3 Solving Minimum and Maximum Constraints

The min and max terms in our system of inequalities are non-convex, yet convexity is a prerequisite of classical linear programming. We overcome this through repeated linear programming, which we overlay with heuristics.

3.1 Constraint Decomposition

First we decompose our system of constraints into a set of linear constraints L and a vector of non-convex complementary constraints $\boldsymbol{C}$. Note that the constraints in $\boldsymbol{C}$ must be disjunctions of linear terms and not arbitrary non-convex terms. Constraints containing min or max terms are rewritten using the following equivalence:

$$x = \min(y, z) \equiv (x \leq y) \wedge (x \leq z) \wedge (x = y \vee x = z)$$
$$x = \max(y, z) \equiv (x \geq y) \wedge (x \geq z) \wedge (x = y \vee x = z)$$

For example, the constraint $u_{m,3} = \min(u_{i,2}, u_{m,2})$ is decomposed into the linear system $L = \{u_{m,3} \leq u_{i,2}, u_{m,3} \leq u_{m,2}\}$ which is complemented with the system $\boldsymbol{C} = \langle (u_{m,3} = u_{i,2} \vee u_{m,3} = u_{m,2}) \rangle$. The decomposed constraints for the worked example are show in figure 3.

3.2 Constraint Solving

Although the disjuncts of $\boldsymbol{C}$ preclude LP from being directly applied, the complementary constraints can be supported by repeatedly solving LPs. To see this, observe that the complementary constraint $l_{m,6} = l_{m,5} \vee l_{m,6} = l_{i,5} + 1$ has one of two states, according to whether the first or the second equality holds. The disjuncts of $\boldsymbol{C}$ thus prescribe a search space of $2^{|\boldsymbol{C}|}$ combinations. In principle each of these combinations could be enumerated and combined with the linear

$$\begin{array}{lllll}
L = \{ & l_{i,1} = -2^{32}, & u_{i,1} = 2^{31}-1, & & \\
& l_{i,2*} = 10, & u_{i,2*} = 10, & & \\
& l_{i,2} \leq l_{i,2*}, & l_{i,2} \leq l_{i,4}, & u_{i,2} \geq u_{i,2*}, & u_{i,2} \geq u_{i,4} \\
& l_{i,3} \geq l_{i,2}, & l_{i,3} \geq l_{m,2}, & u_{i,3} = u_{i,2}, & \\
& l_{i,4} = l_{i,3}, & u_{i,4} = u_{i,3} & & \\
& l_{i,5} = l_{i,2}, & u_{i,5} \leq u_{i,2}, & u_{i,5} \leq u_{m,2}-1, & \\
& l_{m,1} = 5, & u_{m,1} = 20, & & \\
& l_{m,2*} = l_{m,1}, & u_{m,2*} = u_{m,1}, & & \\
& l_{m,2} \leq l_{m,2*}, & l_{m,2} \leq l_{m,4}, & u_{m,2} \geq u_{m,2*}, & u_{m,2} \geq u_{m,4}, \\
& l_{m,3} = l_{m,2}, & u_{m,3} \leq u_{i,2}, & u_{m,3} \leq u_{m,2} & \\
& l_{m,4} \leq l_{m,3}+1, & l_{m,4} \leq 2^{31}-1 & , u_{m,4} \leq u_{m,3}+1 & , u_{m,4} \leq 2^{31}-1 \\
& l_{m,5} \geq l_{m,2}, & l_{m,5} \geq l_{i,2}+1, & u_{m,5} = u_{m,2} & \}
\end{array}$$

$$\begin{array}{lll}
\boldsymbol{C} = \langle & (l_{i,2} = l_{i,2*} \vee l_{i,2} = l_{i,4}), & (u_{i,2} = u_{i,2*} \vee u_{i,2} = u_{i,4}) \\
& (l_{i,3} = l_{i,2} \vee l_{i,3} = l_{m,2}), & (u_{i,5} = u_{i,2} \vee u_{i,5} = u_{m,2}-1) \\
& (l_{m,2} = l_{m,2*} \vee l_{m,2} = l_{m,4}), & (u_{m,2} = u_{m,2*} \vee u_{m,2} = u_{m,4}) \\
& (u_{m,3} = u_{i,2} \vee u_{m,3} = u_{m,2}), & (l_{m,4} = l_{m,3}+1 \vee l_{m,4} = 2^{31}-1) \\
& (u_{m,4} = u_{m,3}+1 \vee u_{m,4} = 2^{31}-1), & (l_{m,5} = l_{m,2} \vee l_{m,5} = l_{i,2}+1 \quad \rangle
\end{array}$$

Fig. 3. Worked example constraints decomposed

component L to form an LP. Each LP could then be independently solved and then compared to find the least value of the objective function overall. However, we suggest an alternative strategy.

The boilerplate of our algorithm is shown in Algorithm 1. Before the algorithm commences, we augment L with:

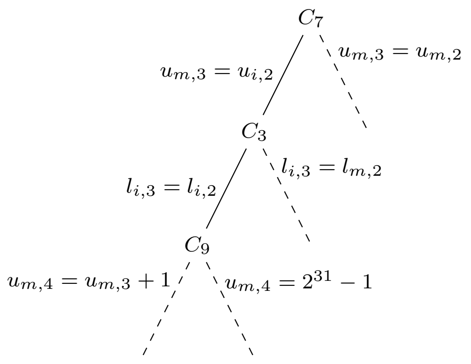
$$\bigwedge_{1 \leq k \leq 5} (-2^{31} \leq l_{i,k} \wedge u_{i,k} \leq 2^{31}-1) \wedge (-2^{31} \leq l_{m,k} \wedge u_{m,k} \leq 2^{31}-1)$$

so as to ensure that all the LPs are bounded. This augmented system will henceforth be denoted $\bar{L}$. The search starts at the root node of the search space with $\tau = \mathit{true}$. At each stage in the search $\bar{L} \wedge \tau$ is tested for satisfiability with a solver, where τ is the conjunction of equalities selected thus far from $\boldsymbol{C}$ (as illustrated in Fig. 4). If $\bar{L} \wedge \tau$ is unsatisfiable, then there is no solution for this choice of τ. Furthermore, augmenting τ with additional equalities from $\boldsymbol{C}$ would further constrain the LP rather than relax it. However, if $\bar{L} \wedge \tau$ is satisfiable, then another equality is selected from $\boldsymbol{C}$ from a disjunct that has not already been considered. This is the role of ChooseNextDecision. If exactly one equality has been selected from each disjunct of $\boldsymbol{C}$ and $\bar{L} \wedge \tau$ is still feasible, then a solution is recorded. The search terminates when the search space is exhausted, at which point the solution with the least objective function value is reported as the overall minimum.

The benefit of this strategy is that if inconsistency is detected when τ contains relatively few equalities from $\boldsymbol{C}$ then many branches through the search space can be discarded simultaneously. The effectiveness of this pruning strategy is dependent upon the ordering of decisions, and in particular the equalities that

Algorithm 1. Binary search algorithm

```
1: function BinSearch($\bar{L}$, $F$, $\boldsymbol{C}$, $\tau$)
2:     $r \leftarrow$ MinimizeLp($F$, $\bar{L} \wedge \tau$)
3:     if $\neg$ Sat(r) then
4:         return []            // No solutions here, prune.
5:     else if AllDecisionsMade($\boldsymbol{C}$, $\tau$) then
6:         return [(r, $\tau$)]       // Found a leaf with a solution
7:     end if
8:     $(e_1 \vee e_2) \leftarrow$ChooseNextDecision($\boldsymbol{C}$, $r$, $\tau$)
9:     $s_l \leftarrow$ BinSearch($\bar{L}$, $F$, $\boldsymbol{C}$, $\tau \wedge e_1$)
10:    $s_r \leftarrow$ BinSearch($\bar{L}$, $F$, $\boldsymbol{C}$, $\tau \wedge e_2$)
11:    return Append($s_l$, $s_r$)
12: end function
```



$\bar{L}_x$	Relaxation ($\bar{L} \wedge \tau$)
1	$\bar{L}$
2	$\bar{L} \wedge (u_{m,3} = u_{i,2})$
3	$\bar{L} \wedge (u_{m,3} = u_{i,2}) \wedge (l_{i,3} = l_{i,2})$
⋮	⋮

Fig. 4. First three linear relaxations of the worked example program

are selected from $\boldsymbol{C}$. For the search to be effective, inconsistencies need to be found early in the search, at a shallow depth in the tree, in order to maximise the effect of pruning. If an inconsistency is found later, then it is likely to be duplicated down alternative paths, nullifying the effect of pruning. Like many combinatoric search problems, the worst case complexity is high (worst case number of linear programs is $2^{|\boldsymbol{C}|+1} - 1$), but in practice performance can be significantly improved with the use of heuristics.

3.3 Heuristics

In order to improve upon the worst case complexity of our search space, we implement the following heuristics:

H1: Prune Inconsistencies Early. This heuristic suggests which disjunct $C_n \in \boldsymbol{C}$ is a good candidate from which an equality should be selected. Suppose solving $\bar{L} \wedge \tau$ returns a solution for which $(u_{m,3} = -2^{31}) \wedge (u_{i,2} = 10) \wedge (u_{m,2} = 20)$. Observe that the disjunct $C_7 = (u_{m,3} = u_{i,2} \vee u_{m,3} = u_{m,2})$ is unsatisfiable under

Algorithm 2. Heuristic 1

```
1: function ChooseNextDecision($\boldsymbol{C}$, $r$, $\tau$)
2:     $v \leftarrow$ GetViolatedCCs($\boldsymbol{C}$, $\tau$)
3:     if $|v| > 0$ then
4:         let $n \in v$
5:     else
6:         $n \leftarrow$ ChooseArbitraryNextDecision($\boldsymbol{C}$, $r$)
7:     end if
8:     return n
9: end function
```

this assignment. Then the heuristic suggests that τ should next be extended with an equality from C_7. If all complementary constraints are satisfied, then an arbitrary $C_n \in \boldsymbol{C}$ is chosen for selecting an equality; the selected C_n is literally chosen at random, thus introducing non-determinism into the algorithm. The intuition behind this selection strategy is that if C_7 is unsatisfiable for one solution to the LP, then extending τ with one of its equalities is likely to detect an inconsistency thereby pruning the search space. Algorithm 2 provides an implementation of ChooseNextDecision using this heuristic.

H2: Block Weak and Duplicate Solutions. A solution is found once an equality is selected from each disjunct of $\boldsymbol{C}$ such that possible to satisfy that $\bar{L} \wedge \tau$ remains satisfiable. There is only one minimal solution, but it is possible for other solutions to exist which, whilst they satisfy the *min*/*max* constraints, yield less tight intervals. It is also possible for both sides of the disjunct of a complimentary constraint to evaluate true (eg. $(l_{i,3} = l_{i,2} \vee l_{i,3} = l_{m,2})$ where $l_{i,2} = l_{i,3} = l_{m,2} = 1$), thereby introducing ineffectual decisions in C and ultimately duplicate solutions. Because there is no value in finding a solution if it does not improve the objective, we propose adding an extra linear constraint to the system that ensures that any solution that is subsequently found improves on the least value of the objective. Suppose that we analyse the worked example program and a solution is found whose objective function value we call o_{min}. Subsequent linear programs are solved in conjunction with an additional blocking constraint: $\sum_{j=1}^{5}(u_{i,j} - l_{i,j}) + \sum_{j=1}^{5}(u_{m,j} - l_{m,j}) < o_{min}$. Through this construction only solutions yielding a strictly smaller objective are feasible, thus further pruning the search space and in turn the number of LPs the analysis must perform.

4 Experimental Results

Our tooling, given a control flow graph and a description of CFG edge operations, generates $\langle \bar{L}, \boldsymbol{C} \rangle$ and proceeds to perform the binary search as described in Sect. 3. The binary search uses the `lpsolve` solver which we interface using Python language bindings. Individual search heuristics (H1 and H2) may be switched on and off, allowing performance comparisons to be drawn under different heuristics configurations. Evaluation of the complimentary constraints

Interval	$m_1 \in$		
	$[2, 2]$	$[63, 71]$	$[5, 20]$
m_{2*}	$[2, 2]$	$[63, 71]$	$[5, 20]$
m_2	$[2, 11]$	$[63, 71]$	$[5, 20]$
m_3	$[2, 10]$	$[63, 10]$	$[5, 10]$
m_4	$[3, 11]$	$[64, 11]$	$[6, 11]$
m_5	$[11, 11]$	$[63, 71]$	$[11, 20]$
i_1	$[0, 255]$	$[0, 255]$	$[0, 255]$
i_{2*}	$[10, 10]$	$[10, 10]$	$[10, 10]$
i_2	$[10, 10]$	$[10, 10]$	$[10, 10]$
i_3	$[10, 10]$	$[63, 10]$	$[10, 10]$
i_4	$[10, 10]$	$[63, 10]$	$[10, 10]$
i_5	$[10, 10]$	$[10, 10]$	$[10, 10]$

(a) Intervals determined by the analysis.

$m_1 \in$	H1	H2	Mean #LPs	Mean Time (s)
$[2, 2]$	✗	✗	208	0.2
	✓	✗	183	0.9
	✗	✓	152	0.1
	✓	✓	38	0.2
$[63, 71]$	✗	✗	200	0.2
	✓	✗	125	0.6
	✗	✓	105	0.1
	✓	✓	45	0.2
$[5, 20]$	✗	✗	207	0.2
	✓	✗	211	1.0
	✗	✓	143	0.1
	✓	✓	44	0.2

(b) Mean number of LPs and time required to find the best solution (sample size of 10, H1/H2 show heuristics enabled).

Fig. 5. Experimental results for the worked example program shown in Fig. 1

(for heuristic 1) is performed by SymPy, a computer algebra library for Python. Experiments were run on a 3GHz 64-bit Intel machine running OpenBSD.

The tables in Fig. 5 show some experimental results for the worked example (Fig. 1) with varying initial values of m_1 and heuristics configurations. Because the algorithm is non-deterministic, each experiment configuration was run 10 times and averages were taken. We show the intervals inferred by our analysis, the mean number of linear programs required (out of a possible worst case number of $2^{10+1} - 1 = 2047$) to find the best solution and the average amount of time spent finding the solution (in seconds). The intervals of the best solution are precise and in all cases, our heuristics reduced the number of LPs required to find the best solution. Further, when $m_1 \in [63, 71]$, the loop body is not entered and this is reflected in our results by the empty intervals at program points 3 and 4. Interestingly run-times appear to be longer when heuristic 1 is enabled. Profiling revealed that the evaluation of complimentary constraints (GetViolatedCCs) accounts for a large portion of solving time for this small example.

A second program was analysed by our analysis, this time at the binary level. Fig. 6 shows the disassembly of a defective implementation of `memcpy(3)` for the x64 architecture. The function takes a pointer to a buffer to write to (`rdi`), a buffer to read from (`rsi`) and a length argument (`rdx`). The `r15` register is used as both a loop counter and as an index into the source and destination buffers. Let $r15_1 \in [l_{r15,1}, u_{r15,1}]$ be the interval representing `r15` at the program point marked `p1`, where bytes are written into the destination buffer. In order to apply our conditional semantics to binary programs, high-level predicates are extracted from pairs of assembler instructions which define and use boolean flags within the status register [5]. For example, `cmp r15, rdx; jg return` causes a control flow despatch if `r15` $>$ `rdx`.

```
memcpy: xor r15, r15                        # loop counter
loop:   cmp r15, rdx
        jg return
        mov byte ptr cl, [rsi+r15]          # read out src
p1:     mov byte ptr [rdi+r15], cl          # write in dest
        inc r15
        jmp loop
return: mov rax, rdi                        # return ptr to dest
        ret
```

Fig. 6. A function to copy buffers

`rdx` $\in$	$r15_1$	H1	H2	MLP	MT
$[8, 8]$	$[0, 8]$	✗	✗	25143	75
		✓	✗	18596	178
		✗	✓	11940	45
		✓	✓	69	1
$[8, 4096]$	$[0, 4096]$	✗	✗	31045	116
		✓	✗	18963	198
		✗	✓	8989	45
		✓	✓	62	1
$[31, 66]$	$[0, 66]$	✗	✗	28639	107
		✓	✗	18963	194
		✗	✓	13885	55
		✓	✓	68	1

(a) `memcpy(3)`

`rdi` $\in$	rax_2	H1	H2	MLP	MT
$[8, 8]$	$[1, 8]$	✗	✗	36621	219
		✓	✗	20342	302
		✗	✓	7891	34
		✓	✓	85	1
$[7, 13]$	$[1, 13]$	✗	✗	35856	151
		✓	✗	19977	258
		✗	✓	8701	37
		✓	✓	99	1
$[4, 128]$	$[1, 128]$	✗	✗	40352	166
		✗	✓	19696	252
		✓	✗	7948	34
		✓	✓	105	1

(b) Endian swap

Fig. 7. Results for the second and third experiments (MLP is the mean number of linear programs required and MT is the mean time in seconds). Means calculated from a sample of 10 runs.

Fig. 7a shows the results of our analysis upon the `memcpy(3)` implementation. If a buffer size of between 8 and 4096 is passed to this function, then our analysis indeed infers $r15_1 \in [0, 4096]$, thereby indicating that one byte is potentially written outside of the allocated buffer. Again, the number of LPs the analysis is required to solve is improved through the use of heuristics. Evaluation of complimentary constraints is especially expensive when heuristic 1 alone is enabled, but the overall time spent searching is vastly improved through the use of heuristics 1 and 2 combined. This experiment utilises 18 complimentary constraints, so the theoretical worst case number of LPs required is $2^{18+1} - 1 = 524287$.

Fig. 8 shows an algorithm to byte-swap 16-bit words in a memory buffer. The function takes a buffer length (`rdi`) and a pointer to a buffer to swap (`rsi`). The register `rax` is being used as an index into the buffer pointed to by `rsi`. Let $rax_1 \in [l_{rax,1}, u_{rax,1}]$ and $rax_2 \in [l_{rax,2}, u_{rax,2}]$ be the intervals of `rax` at marked points `p1` and `p2` respectively. The results of the analysis of this program (Fig. 7b) highlight an interesting deficiency in our analysis. If the function is called with an odd buffer size argument, then the function indeed writes one byte outside its allocated buffer. Yet if we pass our analysis a a buffer size argument of 8, then we infer $rax_2 \in [1, 8]$. This would suggest that a byte was written outside of the

```
endswap: xor r15, r15   # loop counter
loop:    cmp r15, rdi
         jge return
         mov rax, r15   # rax is used as a write index
         mov byte ptr dl, [rsi+r15]
         inc r15
         mov byte ptr cl, [rsi+r15]
         inc r15
p1:      mov byte ptr [rsi+rax], cl
         inc rax
p2:      mov byte ptr [rsi+rax], dl
         jmp loop
return:  ret
```

Fig. 8. A 16-bit byte swap

allocated buffer, however, in reality this is untrue. Our analysis is unable to take into account the strided nature of the loop count and thus over-approximates the upper bound of `rax` upon entry to the loop. Nevertheless, the solution safely over-approximates all possible register values. The solution is found quickly and in a fraction of the worst case number of linear programs ($2^{22+1} - 1 = 8388607$).

5 Discussion

The analysis presented in this paper was mostly inspired by the pioneering work by Rugina et al. [22]. Our extension to Rugina's work diverges in some aspects with response to some shortcomings that are not mentioned in the literature. In this section we will discuss these aforementioned shortcomings thus providing an insight into some of the design decisions of our analysis.

5.1 Conditional Semantics

It would appear that Rugina's branching semantics are unable to model a class of loop constructs correctly. One such example is the program:

`assume(m < 10);` B_1`: int i = 10;` B_2`: while (i >= m)`$\{B_3$`: m = m + 1;`$\}$

Following Rugina's constraint generation scheme we reduce this program to the following constraints, which are infeasible:

$$\begin{array}{lllllllll}
l_{i,2} \leq 10 & \wedge & 10 \leq u_{i,2} & \wedge & l_{m,2} \leq l_{m,1} & \wedge & u_{m,1} \leq u_{m,2} & \wedge \\
l_{i,3} \leq l_{m,2} & \wedge & u_{i,2} \leq u_{i,3} & \wedge & l_{m,3} \leq l_{m,2} & \wedge & \mathbf{u_{m,2} \leq u_{m,3}} & \wedge \\
l_{i,2} \leq l_{i,3} & \wedge & u_{i,3} \leq u_{i,2} & \wedge & l_{m,2} \leq l_{m,3} + 1 & \wedge & \mathbf{u_{m,3} + 1 \leq u_{m,2}} &
\end{array}$$

5.2 Junk Propagation

In both our and Rugina's analysis unreachable code causes the existence of empty intervals (ie. an interval, $[l, u]$, where $l > u$). Consider the following program snippet, in which B_3 is unreachable:

B_1: `int i = 12;` B_2: `while (i <= 10)` $\{B_3$: `i = i + 1`$\}$

If we analyse this program via Rugina's method, we infer the following intervals: $i_2 \in [12, 15], i_3 \in [12, 10]$. The interval at i_3 is empty, correctly indicating that this program point is unreachable. Unfortunately, the loop propagates bounding information from B_3 back to B_2, thereby compromising the precision of the upper bound of i_2. We call this phenomenon "junk propagation".

We overcome imprecision incurred through junk propagation by treating the false branch of the loop check (or any conditional for that matter) as a conditional whose predicate is a negation of the predicate of the true branch. For the counter-example we have just presented, we insert a loop exit block B_4 which is a conditional edge asserting that $i > 11$, thereby retaining the precision of the upper bound of i_2.

6 Related Work

Range analysis has a long history in compilation and verification, dating back to the seminal work of Harrison [16]. This work resonates with ideas in the widening and narrowing approach to abstract interpretation [9], for instance, "this bound may be fed back into the range analysis to revise the ranges" is reminiscent of narrowing which classically following widening so as to tighten ranges. In this work, "each range description describes an arithmetic sequence with a lower bound, upper bound and an increment", and thus the descriptions are actually strided intervals [20], abstractions that are considered to be a recent invention. Widening and narrowing is a research topic within its own right [6,14,25]; a topic that is not confined to abstract interpretation either. Indeed, widening has been applied in conjunction with SMT solving [18], to pose successively weaker candidate loop invariants to the solver until an invariant is found that holds across every iteration of the loop. Our paper, together with [15,26], offers an alternative way of handling loops that aspire to directly compute a fixpoint.

As already discussed, range analysis can be expressed as mathematical integer programming [15], which, in turn, can be reduced to integer linear programming. In this approach binary decision variables are used to indicate the reachability of, among other things, guarded statements. This idea could be developed by making use of the finite nature of machine arithmetic, and encoding a branch condition $x \leq y$ as two inequalities $x \leq y + M(1 - \delta)$ and $x > y + M\delta$ where δ is the binary decision variable and M is a sufficiently large number [29].

Ideally ranges need be combined with the relational domain of congruences [7] since then a range on one variable can be used to trim the range on another and vice versa. Congruence relations, that is, linear constraints that respect the modulo nature of computer arithmetic, can be computed prior to range analysis which leaves the problem of how to amalgamate them into a system of linear constraints. However the congruence $x = y \mod 2^k$ holds iff there exists an integer variable n such that $x = y + n2^k$. This suggests that integer linear programming could be the right medium for marrying congruences with ranges.

Iterative value set analyses have been proposed for binary code [5]. The work, like ours, is predicated on extracting high level predicates from low level conditional jumps. For example the instruction `cmp eax, edx` followed by a `ja` instruction causes a jump if `eax` > `edx`. These authors argue that the predicates can be extracted by pattern matching, a topic that is discussed elsewhere [7].

Interpolation has recently come to the fore in model checking [19] and techniques have now emerged for constructing interpolants in linear arithmetic [23]. Such techniques could be applied with range analysis to find combinations of range constraints that are inconsistent and hence diagnose unreachable code.

7 Conclusions

With an eye towards simplicity, we have shown how range analysis can be computed, not as the solution to a system of recursive equations, but as the solution of a system of constraints over min and max expressions. We have demonstrated how such constraints can be reduced to linear constraints, augmented with complementary constraints, and thus solved by repeated linear programming. The method can be implemented with an off-the-shelf linear programming package which can be used as a black-box. Furthermore, we have shown how the number of calls to the black-box can be reduced by using search heuristics. The result is an analysis that does not depend on classical fixpoint acceleration methods such as widening since it is designed to compute the fixpoint directly.

References

1. National Vulnerability Database, `http://nvd.nist.gov`
2. Andersen, H.R.: An Introduction to Binary Decision Diagrams. Lecture notes, available online, IT University of Copenhagen (1997), `http://www.itu.dk/courses/AVA/E2005/bdd-eap.pdf`
3. Appel, A.W.: Modern Compiler Implementation in Java, Cambridge (2002)
4. Balakrishnan, G., Reps, T.: DIVINE: DIscovering Variables IN Executables. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 1–28. Springer, Heidelberg (2007)
5. Balakrishnan, G., Reps, T.W.: WYSINWYX: What You See Is Not What You eXecute. TOPLAS 32(6) (2010)
6. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A Static Analyzer for Large Safety-Critical Software. In: PLDI, vol. 38, pp. 196–207. ACM (2003)
7. Brauer, J., King, A., Kowalewski, S.: Range Analysis of Microcontroller Code Using Bit-Level Congruences. In: Kowalewski, S., Roveri, M. (eds.) FMICS 2010. LNCS, vol. 6371, pp. 82–98. Springer, Heidelberg (2010)
8. Chen, L., Miné, A., Wang, J., Cousot, P.: Linear Absolute Value Relation Analysis. In: Barthe, G. (ed.) ESOP 2011. LNCS, vol. 6602, pp. 156–175. Springer, Heidelberg (2011)
9. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL, pp. 238–252. ACM (1977)

10. Cousot, P., Cousot, R.: Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation. In: Bruynooghe, M., Wirsing, M. (eds.) PLILP 1992. LNCS, vol. 631, pp. 269–295. Springer, Heidelberg (1992)
11. Doan, D.: Commercial Off the Shelf (COTS) Security Issues and Approaches. Master's thesis, Naval Postgraduate School, Monterey, California (2006), http://www.dtic.mil/cgi-bin/GetTRDoc?AD=ADA456996
12. Durden, T.: Automated Vulnerability Auditing in Machine Code. Phrack Magazine 64 (2007)
13. Godefroid, P., Levin, M.Y., Molnar, D.A.: Automated Whitebox Fuzz Testing. In: NDSS. The Internet Society (2008)
14. Gopan, D., Reps, T.: Lookahead Widening. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 452–466. Springer, Heidelberg (2006)
15. Goubault, E., Le Roux, S., Leconte, J., Liberti, L., Marinelli, F.: Static Analysis by Abstract Interpretation: A Mathematical Programming Approach. ENTCS 267(1), 73–87 (2010)
16. Harrison, W.H.: Compiler Analysis for the Value Ranges of Varibles. IEEE Transactions on Software Engineering SE-3(3), 243–250 (1977)
17. Kapur, D.: Automatically Generating Loop Invariants using Quantifier Elimination. In: International Conference on Applications of Computer Algebra (2004)
18. Leino, K.R.M., Logozzo, F.: Using Widenings to Infer Loop Invariants Inside an SMT Solver, Or: A Theorem Prover as Abstract Domain. In: WING, pp. 70–84 (2007)
19. McMillan, K.L.: Applications of Craig Interpolants in Model Checking. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 1–12. Springer, Heidelberg (2005)
20. Reps, T.W., Balakrishnan, G., Lim, J.: Intermediate-Representation Recovery from Low-Level Code. In: PEPM, pp. 100–111. ACM (2006)
21. Rodríguez-Carbonell, E., Kapur, D.: An Abstract Interpretation Approach for Automatic Generation of Polynomial Invariants. In: Giacobazzi, R. (ed.) SAS 2004. LNCS, vol. 3148, pp. 280–295. Springer, Heidelberg (2004)
22. Rugina, R., Rinard, M.C.: Symbolic bounds analysis of pointers, array indices, and accessed memory regions. TOPLAS 27, 185–235 (2005)
23. Rybalchenko, A., Sofronie-Stokkermans, V.: Constraint Solving for Interpolation. Journal of Symbolic Computation 45, 1212–1233 (2010)
24. Schlich, B.: Model Checking of Software for Microcontrollers. ACM Transactions in Embedded Computing Systems 9, 1–27 (2010)
25. Simon, A., King, A.: Widening Polyhedra with Landmarks. In: Kobayashi, N. (ed.) APLAS 2006. LNCS, vol. 4279, pp. 166–182. Springer, Heidelberg (2006)
26. Su, Z., Wagner, D.: A class of polynomially solvable range constraints for interval analysis without widenings. TCS 345(1), 122–138 (2005)
27. Weissenbacher, G.: Program Analysis with Interpolants. PhD thesis, Magdalen College (2010), http://ora.ouls.ox.ac.uk/objects/uuid:6987de8b-92c2-4309-b762-f0b0b9a165e6
28. Wille, R., Fey, G., Drechsler, R.: Building Free Binary Decision Diagrams Using Sat Solvers. Facta Universitatis-series: Electronics and Energetics 20(3), 381–394 (2007)
29. Zaks, A., Yang, Z., Shlyakhter, I., Ivancic, F., Cadambi, S., Ganai, M.K., Gupta, A., Ashar, P.: Bitwidth Reduction via Symbolic Interval Analysis for Software Model Checking. IEEE TACAD 27(8), 1513–1517 (2008)
30. Zhong, Q., Edward, N.: Security Control COTS Components. IEEE Computer Society 31, 67–73 (1998)

Combining Analyses for C Program Verification

Loïc Correnson and Julien Signoles*

CEA LIST, Software Safety Lab.,
PC 174, 91191 Gif-sur-Yvette, France
firstname.lastname@cea.fr

Abstract. Static analyzers usually return partial results. They can assert that some properties are valid during all possible executions of a program, but generally leave some other properties to be verified by other means. In practice, it is common to combine results from several methods manually to achieve the full verification of a program. In this context, Frama-C is a platform for analyzing C source programs with multiple analyzers. Hence, one analyzer might conclude about properties assumed by another one, in the same environment. We present here the semantical foundations of validity of program properties in such a context. We propose a correct and complete algorithm for combining several partial results into a fully consolidated validity status for each program property. We illustrate how such a framework provides meaningful feedback on partial results.

1 Introduction

Validating a program consists in exhibiting evidence that it will not fail during any of its possible executions. From an engineering point of view, this activity generally consists in manual reviews, testing and formal verifications. Static analyzers can be used to *prove* properties about programs. More precisely, given the source code of a program, an analyzer states a property of *all* of its possible executions. However, analyzers are generally *partial*: they assert some program properties, but leave other ones unverified. Let us illustrate this point of view with some examples of verification techniques.

Abstract Interpretation [1]. This technique computes over-approximations of possible values of each memory location during program execution. When all values in the over-approximation of the memory entail a property, then the property holds during any concrete execution of the program. Otherwise, nothing can be claimed about the property. When such a property is required to hold for the analysis to proceed, the analyzer generally assumes its validity. Hence, the analyzer makes an *assumption* to be verified by other means.

Deductive Verification [2]. This modular technique explicitly proves that a property holds after the execution of a small piece of code, whenever some other property holds before it. We generally say that the *pre*-condition of the verified code entails its *post*-condition. These small theorems can then be chained with each others in order to prove that, whenever some initial pre-condition holds on initial states, the

* This work has been supported by the French ANR project U3CAT and the French FUI9 project Hi-Lite.

desired properties hold on all concrete executions of the program. Generally, not all these elementary steps can be proved, and there remain some properties to be asserted by other means.

Testing. In some sense, testing still falls into the depicted category of analyzers. Executing a test actually asserts that, for all possible executions, if the considered execution corresponds to the test case, then the property defined by the oracle of the test holds. This point of view is of particular interest when we aggregate a collection of tests that covers some criteria. Then, one might claim that the verified properties might only be invalid outside of the covered criteria. Last but not least, testing is also used to exhibit properties that *do not* hold, an activity of major interest during the verification engineering process.

A general practical approach is then to combine several analyzers to increase the coverage of verified properties. Thus there is a need for ensuring the consistency of several partial results. The purpose of this article is to give a semantical foundation to this problem and to provide an algorithm to combine several partial results from different analyzers. A salient feature of our approach is the use of a *blocking semantics*, which is pivotal in ensuring the correctness of the aforementioned algorithm. It allows the correctness to be independent from the hypotheses that the analyzers use to establish their results. These claims remain nevertheless essential for the completeness of the algorithm. The proposed framework is language independent, although it is instantiated in Frama-C [3], a platform dedicated to the verification of critical embedded software written in C, typically in the domain of avionics and energy industries.

Related Work. Combining analysis techniques (in particular static and dynamic ones) is a quite recent but not new idea [4]. However only very few of these works tackle the goal of formally verifying a program by combining these techniques in a consistent way. Heintze *at al.* [5] proposes a framework by equational reasoning to combine an abstract interpreter with a deductive verification tool to enhance verification of user assertions. As in our work, it does not depend on specific analyzers and is correct modulo analyzer's correctness. However, instead of focusing on merging analyzer's results, it implements a new analyzer which operates on the results of the analyzers which it is based on. This analyzer is incomplete in the sense that it not does always provide the more precise result. More recently, the Eve verification environment for Eiffel programs combines a deductive verification tool and a testing tool in order to make software verification practical and usable [6]. Eve reports the separated results obtained from this tool. Since tools which Eve is based upon are not supposed to be correct, Eve computes a so-called *correctness score* for each property. This score is a metrics indicating a level of confidence in its correctness. That is quite different from our approach where we suppose that analyzers are correct but can use other properties as hypotheses to build a proof. Comar *et al.* [7] also aim to integrate a deductive verification tool with testing to verify Spark[1] programs. As in our work, proofs are based on a notion of hypotheses, called *assumptions* in their context. However, to avoid consistency issues, they split the program in several chunks: in each chunk, the same analysis techniques must be applied. In our approach, we allow the user to verify each property by different means.

[1] Spark is a subset of Ada dedicated to the development of critical software.

Outline. The article is structured as follows. First Section 2 presents the problem and the key concepts thanks to a simple C program and a set of properties to verify on it. Section 3 introduces the semantic framework where key concepts are precisely defined. Section 4 presents the algorithm to compute consolidation statuses of properties in a logical way. Section 5 finally focuses on practical usages of the proposed framework: we explain the large variety of user feedbacks that can be obtained after consolidation.

2 Key Concepts

This section introduces all the concepts presented in this article through a running example. It consists of a short program written in C depicted in Figure 1. The program initializes an array with values returned by some external function `f` for which the code is not provided, but only some property `P` on its result is known. We are interested in proving different categories of properties on this short program:

- the program should never produce *runtime errors*, which are situations where the program's behavior is explicitly undefined by the ISO Specification of the C programming language, such as divisions by zero or accesses to uninitialized variables and invalid memory cells;
- once initialized, the values of the array satisfy the property `P` as expected.

Properties in the first category implicitly follow from the language semantics. The second category needs to be expressed explicitly by the developer in order to be verified. The Frama-C platform supports the ACSL language [8] for this purpose. We do not get into details of ACSL here: it is a first-order logical language designed to expressing properties of a C program during its execution.

```
/*@ axiomatic A { predicate P(int x); } */

/*@ ensures P(\result);
  @ assigns \nothing; */
int f(int);

void main(void) {
  int i, n = 0, a[100];
  for(i = 1; i <= 10; i++) n += i;
  // Have n = Sum {1..10}
  for(i = 0; i < n; i++) a[i] = f(i);
  //@ assert \forall integer k; 0 <= k < n ==> P(a[k]);
}
```

Fig. 1. Annotated Code Example

We now comment the source code of Figure 1 in more details. ACSL constructs are inserted into `@`-comments. The predicate `P` is abstractly defined in the pure logic world (axiomatic clause). The external function `f` is declared to have no visible side-effect (assigns clause), and to have its results satisfying `P` (ensures clause). The code to be verified lies in function `main`. It consists in two loops: the first one computes the sum of integers from 1 to 10 and stores the result in local variable `n`; the second loop initializes the `n`-th first indices of array `a` with function `f`. Finally, the ACSL clause assert states the additional property we want to verify for indices less than `n`. The set of properties to be verified for this simple program is then:

Overflows and Runtime Errors: three potential arithmetic overflows and one potential invalid memory access.
User Property: one user assertion to prove.
External Properties: the specification of function `f`.

Two static analyzers distributed with Frama-C address those properties:

`Value` [9] uses a context sensitive forward abstract interpretation to compute an over-approximation of possible values of variables at each program point. This analysis verifies the absence of *any* runtime error and can also handle simple ACSL assertions, like quantifier-free assertions.
`Wp` [10] implements deductive verification. This modular analysis is able to verify complex logical annotations using external automated or interactive provers, but requires extra code annotations to carry function contracts and loop invariants.

We intend here to use `Value` for proving the absence of runtime errors, and `Wp` to prove the assertion, which is not in the scope of `Value`. The external specification of `f` will be trusted here. In the rest of this section, we first report on an incremental study for verifying this program. Then, we introduce our key concepts of local and consolidated statuses for properties managed by Frama-C.

It would also be possible to use `Wp` or other analyzers to prove the absence of runtime errors thanks to the `RTE` Frama-C's plug-in, which generates standard ACSL assertions for any potential runtime error in a source code. More generally, using `RTE` promotes runtime errors to standard properties that smoothly integrate with our framework. However, even small C programs reveal many potential runtime errors, and generating all assertions produces a lot of noise compared to user-defined assertions. When `Value` can be used, it is then much more preferable to rely on it for runtime errors.

2.1 Verifying Properties in Practice

Running `Value` alone with its default configuration on this program gives poor results: variable `i` is not tied enough and the over-approximation of `n` contains overflowing values that become negative. Hence the memory access to `a[i]` in the second loop may be invalid and an alarm is generated. Running `Value` a second time with option `-slevel 100` makes the analyzer more precise during the first loop[2]. This time, all potential errors are discarded, and we get the following interesting properties on the final memory state: `n` is equal to `55`, `a[0..54]` takes any `int` value, and `a[55..99]` remains uninitialized.

Running `Wp` to prove the quantified assertion requires additional annotations from the developer, especially on loops. As illustrated in the `Wp` tutorial [11], a canonical way of proving such a property is to insert the loop invariants of Figure 2. The results of running `Wp` alone are quite encouraging: all annotations are discharged by the Alt-Ergo theorem prover [12], except the first loop invariant `0<=i<=n`. `Wp` proves the preservation of this

[2] The option `-slevel` N of `Value` makes the analyzer works over N different over-approximations in parallel. On our running example, the maximum of precision is obtained for $N \geq 55$. $N \geq 10$ is sufficient to prove the intended properties.

```
/*@ axiomatic A { predicate P(int x); } */

/*@ ensures P(\result);
  @ assigns \nothing; */
int f(int);

void main(void) {
  int i, n = 0, a[100];
  for(i = 1; i <= 10; i++) n += i;
  /*@ loop invariant 0 <= i <= n ;
    @ loop invariant \forall integer k; 0 <= k < i ==> P(a[k]);
    @ loop assigns i,a[0..n-1]; */
  for(i = 0; i < n; i++) a[i] = f(i);
  //@ assert \forall integer k; 0 <= k < n ==> P(a[k]);
}
```

Fig. 2. Annotated Code Example for `Wp`

invariant over loop iterations, but fails to establish it at the very beginning of the loop, because there is no invariant on the first loop establishing that `0<=n`. Of course, it is possible to complete the verification with `Wp` on the first loop, but these range properties over `n` are simple enough to be verified by `Value`. Running both `Value` with option `-slevel 100` and `Wp` on the completely annotated code of Figure 2, we obtain the following results:

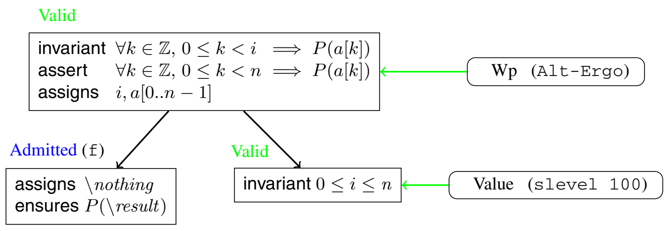
Runtime Errors: *all* potential runtime errors are discharged by `Value`.
Loop Annotations: `Wp` proves two of the three, but leaves the first invariant unverified. `Value` proves only this range invariant.
User Property: `Wp` proves it, but under the hypothesis of the range invariant.
External Properties: they are assumed here, but should be verified later against both the definition of `P` and the actual code of `f`.

Intuitively, the verification task is now complete: everything has been discharged by at least one analyzer. But formal practitioners would notice that it is not clear whether such a verification is conclusive. Indeed, complex dependencies between properties might interfere with each others.

2.2 Soundly Merging Results

A presentation of the results obtained during our verification process can be represented by a graph. With a node for each property, we can represent assumptions by edges from the proved property towards its hypotheses. We also represent analyzers as nodes, with edges towards the properties they established. To increase readability, it is convenient to merge isomorphic nodes into a single one. On our running example, the associated *final* graph is represented in Figure 3.

Such a report is actually accessible through the report plug-in and from the graphical user interface of Frama-C. Let us now present how Frama-C is able to perform this consolidation and report about this verification process. In the example presented above, we have collected different results at different times by using two analyzers with various parameters. Hence, it is not possible to build efficiently and incrementally the desired graph of Figure 3. Instead, it is easy to register each verification experiment with their parameters in a database and to build the consolidation on demand.



Valid
invariant $\forall k \in \mathbb{Z}, 0 \le k < i \implies P(a[k])$
assert $\forall k \in \mathbb{Z}, 0 \le k < n \implies P(a[k])$
assigns $i, a[0..n-1]$
Wp (Alt-Ergo)
Admitted (f)
assigns \nothing
ensures $P(\backslash result)$
Valid
invariant $0 \le i \le n$
Value (slevel 100)

Fig. 3. Consolidated Graph of Properties Validity

This is the key idea behind *local* versus *consolidated* statuses of properties. We introduce the concept of *emitter* to identify an *analyzer* with all of its *parameters*. The partial results provided by an analyzer are registered in a Frama-C database. Each entry of the database precisely consists of:

- an emitter made of an *analyzer* with concrete *parameters' values*;
- a target *property*;
- a *local status*, ranging over *True*, *False* or *Dont_know*;
- a list of *properties* notably used by the analyzer to claim this local status.

The entries obtained after many verification rounds can be very complex to represent. The graph in Figure 4 shows an extract from the full data collected during the verification of Example 2. Two kinds of nodes distinctly represent properties and emitters. Edges are added when analyzers emit local statuses. For instance, three edges are added when `Wp` (using Alt-Ergo as prover) emits *True* for the user assertion A: one from `Wp` to A labeled by the status, and two from A to the loop invariants representing the hypotheses under which this status holds.

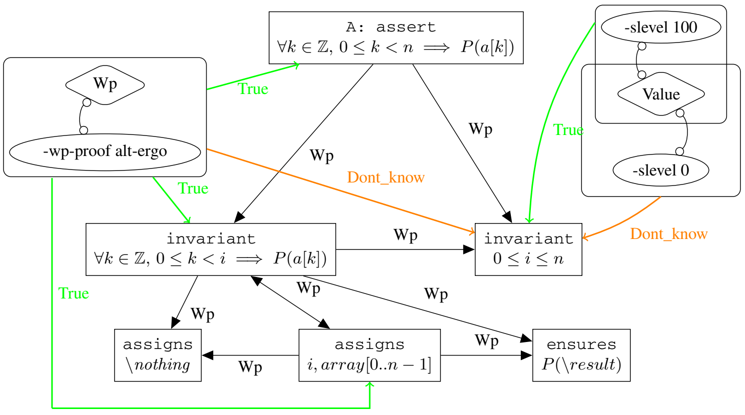


Fig. 4. Graph of Local Status (extract)

Let us illustrate how we obtain the *consolidated* status for this user assertion. Consider all the dependencies that were emitted in conjunction with a local status *True*.

All these paths either end at the range invariant, which is locally *True* with no more assumptions, or at the function contract of `f`. The internal cycle between loop invariants represents `Wp`'s internal inductive scheme. Hence the consolidation algorithm of Frama-C concludes that everything is proven, except the admitted properties of `f`. As we will see, consolidation can be quite challenging on more involved programs. For instance, a property may be false but only over unreachable traces. The general algorithm is complex enough for a semantical approach to be necessary.

3 Semantics

This section formalizes the semantics of annotated programs and property statuses. Our formalisation is independent from both the programming language and the formal specification language: we only suppose that the programming language is imperative, based on a set of *instructions*, and admits a specification language based on a set of *predicates*.

Property. A *property* $\pi = \phi \blacktriangleright \iota$ is a predicate ϕ attached to the program point just before the instruction ι. A predicate which does not depend on a program point (*e.g.* a mathematical lemma required to prove the program) is supposed to be attached to an arbitrary instruction ι_0 without any effect and put just before the first instruction of the program. We note Φ_P the finite set of properties of a program P.

Evaluation. The programming language being imperative, we suppose that there is a notion of *state* in which instructions are evaluated consistently with the operational semantics of the programming language. This notion of evaluation can be extended to predicates, as presented for instance in Herms' works [13]: a state ς validates ϕ, denoted by $\varsigma \models \phi$, if and only if the predicate ϕ is valid in the state ς.

Trace. We now consider that the underlying programming language comes with a trace semantics [14,15] keeping all intermediate instructions and states during execution. Thus a trace $\sigma = (\varsigma_i \rhd \iota_i)_i$ is a (potentially infinite) sequence of instructions, each of them coming with the state in which it is evaluated. Traces begin at the early program entry point $\varsigma_0 \rhd \iota_0$ and are consistent with the small step operational semantics of the program: at each step k, the transition $\varsigma_k \xrightarrow{\iota_k} \varsigma_{k+1}$ holds in the operational semantics of the program. A finite trace σ does not contain the final state ς of a finite execution. But it is still possible to extend it with $(\varsigma \rhd \texttt{skip})$ where `skip` is the identity instruction which does not modify ς. We note $\sigma_1 \prec \sigma_2$ if and only if σ_1 is a strict trace prefix of σ_2. Also, we say that the trace σ ends at instruction ι in state ς, and we note $\sigma \hookrightarrow \varsigma \rhd \iota$, if and only if σ is a finite trace of length n such that $\varsigma_n = \varsigma$ and $\iota_n = \iota$. By extension, for a property π attached at instruction ι, we note $\sigma \hookrightarrow \pi$ if and only if $\sigma \hookrightarrow \varsigma \rhd \iota$ for some state ς.

Trace Validity. We also extend the notation $\models$ for predicates to traces and properties. With $\pi = \phi \blacktriangleright \iota$, we say that σ validates π, and we note $\sigma \models \pi$, the fact: if σ ends at $\varsigma \rhd \iota$ then $\varsigma \models \phi$.

Trace Invalidity. The converse notation, $\sigma \not\models \pi$, is used for the logical negation of $\sigma \models \pi$. Remark it is *not* equivalent to $\sigma \models \neg\pi$, however, we still have $\sigma \not\models \pi \Rightarrow \sigma \models \neg\pi$.

Blocking Semantics. The correctness of our algorithm (see Theorem 1) requires a *blocking semantics* which is usual in semantics of annotated programs (see for instance [13,16]). In our theoretical framework, it can be expressed as follows.

Assumption 1 (Blocking Semantics). *If a trace leads to an invalid property, then the program stops and does not evaluate the following properties in the execution flow. More formally:*

$$\forall \textit{ traces } \sigma \textit{ and } \sigma', \forall \textit{ properties } \pi \textit{ and } \pi', \textit{ if } \sigma' \prec \sigma \textit{ and } \sigma' \not\models \pi', \textit{ then } \sigma \not\hookrightarrow \pi.$$

If all properties are valid, the blocking semantics coincides with the non-blocking one.

Reachability. An associated concept is the reachability of some instruction of the program. More precisely, we are interested in the reachability of instructions to which given properties are attached. In our framework, this concept is represented by global predefined (meta) properties of program properties, attached to the initial state ι_0 of the program:

$$\mathrm{reach}(\pi) \triangleq (\exists \sigma, \sigma \hookrightarrow \pi) \blacktriangleright \iota_0.$$

Local Validity. We say that a property π is *locally valid* under a finite set of hypotheses ξ, and we note $\xi \models \pi$, if and only if:

$$\forall \text{ trace } \sigma, \text{ if } (\forall \pi_i \in \xi, \forall \text{ trace } \sigma_i, \text{ if } \sigma_i \prec \sigma, \text{ then } \sigma_i \models \pi_i), \text{ then } \sigma \models \pi.$$

Informally, a property is locally valid if it is validated by each trace σ ending at it, assuming that each hypothesis π_i is itself validated by all subtraces of σ ending at π_i.

Local Invalidity. A property π is *locally invalid* under a finite set of hypotheses ξ, and we note $\xi \not\models \pi$, if and only if:

$$\text{if } (\forall \pi_i \in \xi, \forall \text{ trace } \sigma_i, \text{ if } \sigma_i \prec \sigma, \text{ then } \sigma_i \models \pi_i), \text{ then } \exists \text{ trace } \sigma, \sigma \not\models \pi.$$

A property is locally invalid if there is a trace σ ending at it but does *not* validate it, but still assuming each hypothesis π_i is valid on any subtrace of σ ending at π_i. These notions of local validity and local invalidity correspond to statuses emitted by Frama-C analyzers as we will see in assumptions 2 and 3 in the next section. Note that being locally invalid is not equivalent to not being locally valid: $\xi \not\models \pi \not\Leftrightarrow \neg(\xi \models \pi)$. Moreover, none of these predicates is equivalent to $\xi \models \neg\pi$.

Cycles. Statements $\{\pi\} \models \pi$ and $\xi \models \pi_i$ with $\pi_i \in \xi$ are not tautologies in loops. Instead they exactly correspond to proofs by induction, as committed by the strict prefix relation on traces. These statements are actually valid if and only if we can prove $\sigma \models \pi$ (resp. π_i), for any trace σ, under the hypotheses that $\sigma_j \models \pi$ for any *strict* subtrace σ_j of σ (resp. $\sigma_j \models \pi_j$ for any $\pi_j \in \xi$).

Global Validity. Last but not least, a property π is *valid*, and we note $\models \pi$, if and only if $\sigma \models \pi$ for each trace σ. We say that π is invalid, and we note $\not\models \pi$, if π is not valid, that is $\neg(\models \pi)$. Once again, $\not\models \pi \not\Leftrightarrow \models \neg\pi$. These notions of validity and invalidity correspond to the consolidated statuses computed by our algorithm from all the local validity statuses emitted by the analyzers.

4 Consolidation Algorithm

This section presents a high-level view of the so-called consolidation algorithm implemented in Frama-C. From the *local statuses* of a property π computed by each emitter under hypotheses, each of them corresponding to the local validity or invalidity of π, this algorithm computes its *consolidated status* corresponding to $\models \pi$.

4.1 Local Statuses

As already mentioned, an emitter can emit three different local statuses, namely *True*, *False* and *Dont_know*. The third one indicates that it is not able to conclude. Let Λ be the set of these local statuses. Local statuses emitted by analyzers are collected into a database, and we denote $\mathcal{L}_P$ the lookup function that returns them for each property:

$$\mathcal{L}_P : \Phi_P \to \mathscr{P}(\Lambda \times \mathscr{P}(\Phi_P))$$

If an emitter put a local status λ to the property π with hypotheses ξ, then $(\lambda, \xi) \in \mathcal{L}_P(\pi)$. We expect that analyzers are correct and emit local statuses consistently with the underlying annotated program semantics, in particular local validities and local invalidities of annotations. Furthermore, when emitting *False* for a property π, our algorithm also requires that the only possible hypothesis is $\mathrm{reach}(\pi)$. The following assumption formalizes this restriction.

Assumption 2 (Strong Correctness of Analyzers). *We assume that each analyzer is strongly correct: it emits the local status True (resp. False) only for locally valid (resp. invalid) properties under the hypotheses really used (and limited to reachability in case of invalidity). More formally, for each property π of a program P:*

$$\forall(\lambda,\xi) \in \mathcal{L}_P(\pi), \begin{cases} \textit{if } \lambda = \textit{True}, & \textit{then } \xi \models \pi; \\ \textit{if } \lambda = \textit{False}, & \textit{then } \xi \not\models \pi \textit{ and } \forall \pi_i, \pi_i = \mathrm{reach}(\pi). \end{cases}$$

However, in practice, it may be complicated or inefficient to compute the exact set of hypotheses which is used to compute a local status. Actually, in presence of a blocking semantics, the correctness of the consolidation algorithm does not rely on these hypotheses, as explained by Theorem 1 (correctness of the algorithm). They are useful for Theorem 2 (completeness of the algorithm) and to compute more precise informations for the end-user in the unconclusive cases. Thus, for correctness, the following weaker assumption is enough.

Assumption 3 (Weak Correctness of Analyzers). *Analyzers are assumed to be weakly correct. They emit the local status True (resp. False) only for locally valid (resp. invalid) properties under some unknown hypotheses (resp. reachability). More formally, for each property π of a program P:*

$$\forall(\lambda,\xi) \in \mathcal{L}_P(\pi), \begin{cases} \textit{if } \lambda = \textit{True}, & \textit{then } \exists \xi' \subseteq \Phi_P, \xi' \models \pi; \\ \textit{if } \lambda = \textit{False}, & \textit{then } \xi \not\models \pi \textit{ and } \forall \pi_i, \pi_i = \mathrm{reach}(\pi). \end{cases}$$

Ensuring correctness of analyzers in practice is out of the scope of this paper[3] and is strongly related to the qualification of verification tools for an operational use in a certified industrial process, for instance with respect to norms like aeronautic's DO-178C. Although none of the Frama-C analyzers is qualified at this time, efforts have already been made in this direction [13,17].

4.2 Invalidity and Reachability

Unfortunately there is a practical issue with the previous local status *False*: proving local invalidity of a property π requires to prove two different properties: (1) it exists a trace σ which ends at π and (2) this trace does not validates π. Let us examine what tools are able to assess.

– Deductive methods based on weakest precondition calculus are usually not able to prove invalidity. They are only able to prove validity: in practice, they never emit *False*.
– Testing tools usually prove together properties (1) and (2) by exhibiting a test case which invalidates the property: all is fine.
– Abstract interpreters only reason with an over-approximation of all possible traces of the program. Thus, when a property is invalidated for all these over-approximated traces, it means the property is invalid *if* the program point is reachable. But, abstract analyzers are usually not able to prove reachability.

For solving this issue, Frama-C allows emitters to emit either the local status *False_and_reachable* or the local status *False_if_reachable*. So, instead of working with $\mathcal{L}_P$, our algorithm uses the function $\mathcal{L}_P^\star : \Phi_P \to \mathscr{P}(\Lambda^\star \times \mathscr{P}(\Phi_P))$, where $\Lambda^\star$ is the set of emittable statuses defined by:

$$\Lambda^\star \triangleq \{\textit{True}, \textit{Dont\_know}, \textit{False\_if\_reachable}, \textit{False\_and\_reachable}\}.$$

For any program P, thanks to the reach operator, we can automatically compute $\mathcal{L}_P$ from $\mathcal{L}_P^\star$ as follows:

$$\begin{aligned}\forall\pi \in \Phi_P, \mathcal{L}_P(\pi) \triangleq\ & \{(\lambda,\xi) \mid \lambda \in \{\textit{True}, \textit{Dont\_know}\} \text{ and } (\lambda,\xi) \in \mathcal{L}_P^\star(\pi)\} \\ \cup\ & \{(\textit{False},\xi) \mid (\textit{False\_and\_reachable},\xi) \in \mathcal{L}_P^\star(\pi)\} \\ \cup\ & \{(\textit{False},\xi \cup \{\mathrm{reach}(\pi)\}) \mid (\textit{False\_if\_reachable},\xi) \in \mathcal{L}_P^\star(\pi)\}\end{aligned}$$

Emitting *False_and_reachable* is changed into emitting *False*, and emitting *False_if_reachable* is modified into emitting *False* under the additional hypothesis $\mathrm{reach}(\pi)$. Emitting *True* and *Dont_know* is left unchanged. This definition of $\mathcal{L}_P$ preserves both the strong and the weak correctness of analyzers (assumptions 2 and 3).

To avoid an inconsistency of our algorithm in a corner case leading to uncorrectness, we also introduce the following assumption for any $\mathrm{reach}(\cdot)$ property.

Assumption 4 (Do not prove unreachability with reachability). *We assume that no analyzer tries to prove unreachability of a property π by using its reachability. More formally, for a given program P:*

$$\forall\pi \in \Phi_P, \forall(\lambda,\xi) \in \mathcal{L}_P(\mathrm{reach}(\pi)), \textit{if } \lambda = \textit{False}, \textit{ then } \mathrm{reach}(\mathrm{reach}(\pi)) \notin \xi.$$

[3] See the small discussion about the status *Inconsistent* latter in this section however.

4.3 Algorithm

We now introduce the consolidation algorithm itself. Applied on a given program P, it may be seen as a function $\mathcal{S}_P : \Phi_P \to \Sigma$ in which the set Σ is defined by:

$$\Sigma \triangleq \{\mathit{Valid}, \mathit{Invalid}, \mathit{Unknown}, \mathit{Inconsistent}\}.$$

The third status is returned by the algorithm when it is not able to conclude, while the last one is returned when there is both a proof of validity and a proof of invalidity: in such a case, we can conclude that one emitter is not (strongly) correct[4]. Before the formal definition of $\mathcal{S}_P$, we present an informal sketch of the algorithm:

1. abort if assumption 4 is violated;
2. compute the most precise local status λ;
3. for each emitter which emits λ, compute the conjunction of the consolidated statuses of its hypotheses;
4. compute the most precise conjunction γ computed above;
5. compute the status of $\gamma \implies \lambda$;
6. check for inconsistencies.

Step 1 of the algorithm is a simple structural check.

Computing the most precise local status in Step 2 relies on the operator $\bigvee^L$ based on $\vee^L$ which ensures local validity and is defined below. It is mosly equivalent to a logical disjunction in a tri-valued boolean logic. But, in the case where an emitter emits *True* and another one emits *False*, we do not choose yet a status, even if it would be correct to choose *True*: in order to be complete, we wait Step 5 of the algorithm to select the one which is possible to fully consolidate.

$\vee^L$	*True*	*Dont_know*	*False*
True	{ *True* }	{ *True* }	{ *True*, *False* }
Dont_know	{ *True* }	{ *Dont_know* }	{ *False* }
False	{ *True*, *False* }	{ *False* }	{ *False* }

$$\bigvee^L \{\lambda_n\}_n = \begin{cases} \lambda_1 \vee^L \ldots \vee^L \lambda_n & \text{if } n > 0 \\ \mathit{Dont\_know} & \text{otherwise} \end{cases}$$

Computing the conjunction of the statuses of the hypotheses in Step 3 of the algorithm is done by the operator $\bigwedge^H$ based on $\wedge^H$ and defined below. This operator exactly is the standard conjunction of a tri-valued boolean logic. We omit the case *Inconsistent* which is treated as *Unknown* here, and still returns *Unknown*.

$\wedge^H$	*Valid*	*Unknown*	*Invalid*
Valid	*Valid*	*Unknown*	*Invalid*
Unknown	*Unknown*	*Unknown*	*Invalid*
Invalid	*Invalid*	*Invalid*	*Invalid*

$$\bigwedge^H \{\lambda_n\}_n = \begin{cases} \lambda_1 \wedge^H \ldots \wedge^H \lambda_n & \text{if } n > 0 \\ \mathit{Valid} & \text{otherwise} \end{cases}$$

[4] In practice, other origins are possible like inconsistent user-defined ACSL axiomatics.

Computing the most precise consolidated status as performed in Step 4 of the algorithm is done by the operator $\bigvee^H$ based on $\vee^H$ and defined below. It exactly corresponds to the standard disjunction of a tri-valued boolean logic. We omit the case *Inconsistent* which never occurs here, since $\wedge^H$ returns *Unknown* instead.

$\vee^H$	*Valid*	*Unknown*	*Invalid*
Valid	*Valid*	*Valid*	*Valid*
Unknown	*Valid*	*Unknown*	*Invalid*
Invalid	*Valid*	*Invalid*	*Invalid*

$$\bigvee^H \{\lambda_n\}_n = \begin{cases} \lambda_1 \vee^H ... \vee^H \lambda_n & \text{if } n > 0 \\ \textit{Unknown} & \text{otherwise} \end{cases}$$

The implication operator $\overset{\text{HL}}{\Longrightarrow}$ involved in Step 5 of the algorithm is defined below (left part in row, right part in column). Like for $\vee^H$, *Inconsistent* is omitted and treated as *Unknown*.

$\overset{\text{HL}}{\Longrightarrow}$	*True*	*Dont_know*	*False*
Valid	*Valid*	*Unknown*	*Invalid*
Unknown	*Unknown*	*Unknown*	*Unknown*
Invalid	*Unknown*	*Unknown*	*Valid*

This operator corresponds to the standard implication of a tri-valued boolean logic, but most cases remain unknown: remember that $\not\models \pi$ means that π is incorrect *for some trace* σ. Thus it would be wrong to assume than an hypothesis π_i being incorrect for some trace σ_i leads to a correct goal *for any trace*: it is still possible to have another trace σ independent of σ_i ($\sigma_i \not\prec \sigma$) which ends at π and invalidates it. It is possible to conclude *Valid* in the case *Invalid* $\overset{\text{HL}}{\Longrightarrow}$ *False* since the only possible hypothesis is $\mathrm{reach}(\pi)$ (assumption 4): if it is invalid, π is unreachable, hence valid.

Step 6 detects inconsistency when it is possible to consolidate a property to both *Valid* and *Invalid*, thanks to the operator $\bigvee^I$ which is equivalent to $\bigvee^H$ except that:

$$\textit{Valid} \vee^I \textit{Invalid} = \textit{Invalid} \vee^I \textit{Valid} = \textit{Inconsistent}.$$

With all the operators now introduced, we can formally define our algorithm as the function $\mathcal{S}_P$ in the following way:

$$\mathcal{S}_P(\pi) \triangleq \mathcal{S}_P^{\emptyset}(\pi)$$

$$\text{with } \mathcal{S}_P^{\Psi}(\pi) \triangleq \bigvee^I_{\lambda_\pi \in \Lambda_\pi} \left(\left(\bigvee^H_{\xi \in \Xi_{\lambda_\pi}} \bigwedge^H_{\pi_\xi \in \xi \setminus \Psi} \mathcal{S}_P^{\Psi \cup \{\pi\}}(\pi_\xi) \right) \overset{\text{HL}}{\Longrightarrow} \lambda_\pi \right)$$

$$\text{and } \Lambda_\pi = \bigvee^L \{ \lambda \mid (\lambda, \_) \in \mathcal{L}_P(\pi) \}$$

$$\text{and } \Xi_{\lambda_\pi} = \{ \xi \mid \mathcal{L}_P(\pi) = (\lambda_\pi, \xi) \}.$$

In this definition, the set Ψ used in $\mathcal{S}_P^{\Psi}$ stores the properties already visited in order to handle cycles in a well-founded way. This algorithm is correct with respect to the trace semantics of Section 3, as stated by the following theorem[5].

[5] Proofs are provided in appendix.

Theorem 1 (Correctness). *Under assumptions 1 (blocking semantics) and 3 (weak correctness of analyzers), if the consolidation algorithm returns Valid (resp. Invalid) for a property π, then π is valid (resp. invalid). If it returns* Inconsistent*, then both $\models \pi$ and $\not\models \pi$ hold (*i.e.* logical inconsistency). More formally, for a given program P:*

$$\forall \pi \in \Phi_P, \begin{cases} \text{if } \mathcal{S}_P(\pi) = \textit{Valid}, & \text{then } \models \pi; \\ \text{if } \mathcal{S}_P(\pi) = \textit{Invalid}, & \text{then } \not\models \pi; \\ \text{if } \mathcal{S}_P(\pi) = \textit{Inconsistent} & \text{then } \models \pi \text{ and } \not\models \pi. \end{cases}$$

The algorithm is also *complete* when the analyzers are strongly correct, in the following sense: if a property is assigned a local status of validity (resp. invalidity), and if recursively, all its dependencies are *globally valid*, then our algorithm computes a valid (resp. invalid) consolidated status. The notion of recursively valid hypotheses for property π is captured the following definition:

$$\mathcal{D}(\pi) \triangleq \exists \lambda \neq \textit{Dont\_know}, (\lambda, \xi) \in \mathcal{L}_P(\pi) \text{ and } \forall \pi_i, \mathcal{D}(\pi_i) \text{ and } \models \pi_i;$$

Theorem 2 (Completeness). *Under assumptions 1 (blocking semantics) and 2 (strong correctness of analyzers), if a property is valid (resp. invalid) and an emitter emits a local status different from Dont_know under recursively valid hypotheses, then the consolidation algorithm returns Valid (resp. Invalid). More formally, for a given program P:*

$$\forall \pi \in \Phi_P, \begin{cases} \text{if } \mathcal{D}(\pi) \text{ and } \models \pi, \text{ then } \mathcal{S}_P(\pi) = \textit{Valid}; \\ \text{if } \mathcal{D}(\pi) \text{ and } \not\models \pi, \text{ then } \mathcal{S}_P(\pi) = \textit{Invalid}. \end{cases}$$

5 Consolidated Partial Statuses

The previous formalization provides correctness and completeness results when every property is consolidated to valid or invalid. While this is perfect for the success of a verification campaign under strong qualification requirements, there is no way to know the origin of partial results, in particular in case of *Unknown* statuses.

In Frama-C, there is actually a variety of 11 consolidated statuses that can be synthesized for a property. These statuses provide feedback to the engineer from three complementary points of view: validity of the property, completeness with respect to its recursively valid hypotheses, and reachablity. A color is assigned to each point of view, and each of the 11 statuses of Frama-C has one or two colors for a fully detailed feedback on any property status. This variety of statuses can be simply understood as *refinements* for the four basic consolidated statuses presented in Section 4.

Refinements of Valid. As illustrated in the running example with external functions, it is sometimes impossible to complete a verification process inside the verification tool. It is then useful to consolidate admitted results like valid ones, while tagging those admitted results for manual reviews outside the tool. As seen in the previous section, another important case of validity is a locally invalid but unreachable property. To avoid confusing the user by presenting a *Valid* status on a locally invalid status *False*, we use a special “Invalid but dead” consolidated status in this situation. There is no interesting refinement for *Inconsistent* and *Invalid* statuses.

● **Valid**	● Admitted	● **Inconsistent**
	● Invalid but dead	● **Invalid**

Refinements of Unknown. The most versatile situations are related to the status *Unknown*. We distinguish several cases by taking into account whether the local status is *True* or *False*, or whether some hypothesis is surely *Invalid*. In the first category of cases, we want to retain the local status feedback although nothing can be claimed since assumptions are missing. In the second category of cases, we want to mark the property as irrelevant since there is an *Invalid* property previously in the control flow graph which may impact the status of this property.

○ No local status	No analyzer tried.
● Unknown	No analyzer succeeded.
● Locally Valid	Hypotheses are not yet consolidated (*Unknown*).
● Locally Invalid	
● Valid but irrelevant	One hypothesis is surely *Invalid*.
● Unknown but irrelevant	

Extension of the Algorithm. Extending the consolidation algorithm of Section 4 with this full variety of statuses is quite straightforward. Roughly, an extended status is treated like the status it refines. For instance, *Locally Valid* is treated as *Unknown*. This extension may be synthetized in the modified table of the $\overset{\text{HL}}{\Longrightarrow}$ operator below, which is responsible for consolidating the best local status with respect to the consolidated statuses of its hypotheses. The refined statuses are marked with a star (*).

$\overset{\text{HL}}{\Longrightarrow}$	*True*	*Dont_know*	*False*
Valid	*Valid*	*Unknown*	*Invalid*
Unknown	*Locally Valid* *	*Unknown*	*Locally Invalid* *
Invalid	*Valid but irrelevant* *	*Unknown but irrelevant* *	*Invalid but unreachable* *

The extended table for this operator is sound: a status λ is only replaced by a *refinement* of λ. Hence, we still benefit from correctness and completeness theorems.

6 Conclusion

We have presented a consolidation algorithm for verifying program properties by combining results from several program analyzers. This algorithm is proved to be correct and complete with respect to a generic blocking semantics of annotated programs, as long as analyzers are correct. Its correctness does not rely on hypotheses emitted by the analyzers: these hypotheses are only required for completness. We have also presented how to refine results to provide more informative feedback to the end-user.

This algorithm is fully implemented in Frama-C, a platform gathering several static analysis techniques in a single collaborative framework. It has successfully been used on a confidential 50-kloc case study which is representative of real-life software of systems important to safety in nuclear power plants. Here `Value` is primarily used, while `Wp` helps it to prove assertions on which `Value` is unconclusive. Other collaborations between different set of analyses are currently under way, in particular between test generation tools and static verifiers [18].

Acknowledgement. We would like to thank the anonymous referees for their meaningful comments. We also thank Patrick Baudin, Pascal Cuoq, Florent Kirchner, Benjamin Monate, Dillon Pariente, Virgile Prevosto and Boris Yakobowski for the intensive discussions about property statuses, their semantics and their implementation in Frama-C.

References

1. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL (1977)
2. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Communications of the ACM 18(8) (1975)
3. Correnson, L., Cuoq, P., Kirchner, F., Prevosto, V., Puccetti, A., Signoles, J., Yakobowski, B.: Frama-C User Manual (October 2011), `http://frama-c.com`
4. Elberzhager, F., Münch, J., Nha, V.T.N.: A systematic mapping study on the combination of static and dynamic quality assurance techniques. Information & Software Technology 54(1), 1–15 (2012)
5. Heintze, N., Jaffar, J., Voicu, R.: A framework for combining analysis and verification. In: POPL 2000 (2000)
6. Tschannen, J., Furia, C.A., Nordio, M., Meyer, B.: Usable Verification of Object-Oriented Programs by Combining Static and Dynamic Techniques. In: Barthe, G., Pardo, A., Schneider, G. (eds.) SEFM 2011. LNCS, vol. 7041, pp. 382–398. Springer, Heidelberg (2011)
7. Comar, C., Kanig, J., Moy, Y.: Integrating formal program verification with testing. In: ERTSS 2012 (2012)
8. Baudin, P., Filliâtre, J.C., Hubert, T., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language (February 2011), `http://frama-c.cea.fr/acsl.html`
9. Canet, G., Cuoq, P., Monate, B.: A Value Analysis for C Programs. In: SCAM 2009 (2009)
10. Correnson, L., Dargaye, Z.: WP Plug-in Manual, version 0.5 (January 2012)
11. Baudin, P., Correnson, L., Hermann, P.: WP Tutorial, version 0.5. (January 2012)
12. Bobot, F., Conchon, S., Contejean, E., Iguernelala, M., Lescuyer, S., Mebsout, A.: The Alt-Ergo Automated Theorem Prover, `http://alt-ergo.lri.fr`
13. Herms, P., Marché, C., Monate, B.: A Certified Multi-prover Verification Condition Generator. In: Joshi, R., Müller, P., Podelski, A. (eds.) VSTTE 2012. LNCS, vol. 7152, pp. 2–17. Springer, Heidelberg (2012)
14. Grall, H.: Deux critères de sécurité pour l'exécution de code mobile. PhD thesis, École Nationale des Ponts et Chaussées (December 2003) (in French)
15. Leroy, X., Grall, H.: Coinductive big-step operational semantics. Information and Computation 207(2) (2009)
16. Giorgetti, A., Groslambert, J., Julliand, J., Kouchnarenko, O.: Verification of class liveness properties with Java Modeling Language. IET Software 2(6) (2008)
17. Cuoq, P., Monate, B., Pacalet, A., Prevosto, V., Regehr, J., Yakobowski, B., Yang, X.: Testing Static Analyzers with Randomly Generated Programs. In: Goodloe, A.E., Person, S. (eds.) NFM 2012. LNCS, vol. 7226, pp. 120–125. Springer, Heidelberg (2012)
18. Delahaye, M., Kosmatov, N., Signoles, J.: Towards a common specification language for static and dynamic analyses of C programs (submitted)

A Proofs of Theorems

This appendix contains the proofs of the correctness and the completeness theorems of the paper. First we introduce a lemma which links local validity to validity.

Lemma 1 (Local validity implies validity). *Under assumption 1 (blocking semantics), if a property is locally valid (resp. invalid), then it is valid (resp. invalid). More formally, for a given program P:*

$$\forall \pi \in \Phi_P, \forall \Pi \subseteq \Phi_P, \textit{ if } \Pi \models \pi, \textit{ then } \models \pi.$$

Proof. Let π be a property, Π be a finite set of properties such that $\Pi \models \pi$ and σ be a trace. We have to show that $\sigma \models \pi$.

Case $\forall \pi_i \in \Pi, \forall$ **trace** $\sigma_i,$ **if** $\sigma_i \prec \sigma,$ **then** $\sigma_i \models \pi_i$**.** By definition of local validity, $\sigma \models \pi$.

Case $\exists \pi_i \in \Pi, \exists \sigma_i, \sigma_i \prec \sigma$ **and** $\sigma_i \not\models \pi_i$**.** By assumption 1 (blocking semantics), as $\sigma_i \prec \sigma$ and $\sigma_i \not\models \pi_i$, $\sigma \not\rightarrow \pi$. Thus, by definition of $\models$ (for trace), $\sigma \models \pi$.

Now, we introduce a well-founded relation which explains why the consolidation algorithm terminates. It is actually not so trivial: informally, our algorithm performs a topological iteration over a graph where vertices are properties and each edge indicates that a property is used as hypothesis of another one. But, this graph may contain cycles, while topological iteration is not well defined for such graphs. That is the *raison d'être* of the set of already visited properties in the algorithm. Thus this set must be taken into account in our proof. So, for any program P, let us introduce the following relation $\ll_P$ over $\Phi_P \times \mathscr{P}(\Phi_P)$ as the transitive closure of $\ll_P^1$ defined as follows:

$$\begin{aligned}(\pi_1, \Psi_1) \ll_P^1 (\pi_2, \Psi_2) &\iff \pi_1 \lessdot_P^1 \pi_2 \text{ and } \pi_1 \notin \Psi_1 = \Psi_2 \cup \{\pi_2\} \\ \text{with } \pi_1 \lessdot_P^1 \pi_2 &\iff \exists (\lambda, \xi_2) \in \mathcal{L}_P(\pi_2), \pi_1 \in \xi_2\end{aligned}$$

Informally, $\pi_1 \lessdot_P^1 \pi_2$ says that π_1 is used as hypothesis of π_2 (or there is an edge from π_2 to π_1 in the graph), while $(\pi_1, \Psi_1) \ll_P (\pi_2, \Psi_2)$ indicates that there is a path from π_2 to π_1 in the graph. It also requires that π_2 is the property currently visited (thus being included in the set of already visited properties) and π_1 is not already visited (in order to break cycles).

Lemma 2 ($\ll_P$ is a well-founded relation). *For any program P, $\ll_p$ is a well founded relation. More precisely: $\ll_P$ is a strict partial order (*i.e. *an anti-reflexive, antisymmetric and transitive relation) and every non-empty subset of $\Phi_P \times \mathscr{P}(\Phi_P)$ has a $\ll_P$-minimal element. Furthermore, the set of all these $\ll_P$-minimals is:*

$$\aleph_P \triangleq \{(\pi, \Psi) \,|\, \forall (\lambda, \xi) \in \mathcal{L}_P(\pi), \xi \setminus \Psi = \emptyset\}.$$

Proof. Consider a program P.

Anti-reflexivity. Since, for any $\pi \in \Phi_P$ and $\Psi \in \mathscr{P}(\Phi_P)$, $\pi \in \Psi \cup \{\pi\}$, $(\pi, \Psi) \not\ll_P (\pi, \Psi)$ by definition of $\ll_P$. Hence $\ll_P$ is anti-reflexive.

Antisymmetry. Let π_1 and π_2 being in Φ_P and Ψ_1 and Ψ_2 being in $\mathscr{P}(\Phi_P)$ such that $(\pi_1, \Psi_1) \ll_P (\pi_2, \Psi_2)$ and $(\pi_2, \Psi_2) \ll_P (\pi_1, \Psi_1)$. By definition of $\ll_P$:

$$\begin{aligned}
\pi_1 &\notin \Psi_1 && \text{since } (\pi_1, \Psi_1) \ll_P (\pi_2, \Psi_2) \\
&\notin \Psi_2 \cup \{\pi_2\} && \text{since } \Psi_1 = \Psi_2 \cup \{\pi_2\} \text{ because of } (\pi_1, \Psi_1) \ll_P (\pi_2, \Psi_2) \\
&\notin \Psi_1 \cup \{\pi_1\} \cup \{\pi_2\} && \text{since } \Psi_2 = \Psi_1 \cup \{\pi_1\} \text{ because of } (\pi_2, \Psi_2) \ll_P (\pi_1, \Psi_1).
\end{aligned}$$

The last line is a contradiction since $\pi_1 \in \{\pi_1\}$. Hence $\ll_P$ is antisymmetric.

Transitivity. Trivial by definition of transitive closure.

Elements of $\aleph_P$ are minimal. Let $(\pi, \Psi) \in \aleph_P$ and $(\pi', \Psi') \in \Phi_P \times \mathscr{P}(\Phi_P)$. Let us prove by contradiction that $(\pi', \Psi') \not\ll_P (\pi, \Psi)$. So let us suppose that $(\pi', \Psi') \ll_P (\pi, \Psi)$.

By definition of $\ll_P$, since $(\pi', \Psi') \ll_P (\pi, \Psi)$, $\pi' \lessdot_P \pi$, it exists π'' such that $\pi' \ll_P \pi''$ and $\pi'' \ll^1_P \pi$. Thus, $\pi'' \lessdot^1_P \pi$ and, by definition of $\lessdot^1_P$, it exists $(\lambda, \xi) \in \mathcal{L}_P(\pi)$ such that $\pi'' \in \xi$. By definition of $\aleph_P$, since $\pi \in \aleph_P$, $\pi'' \in \Psi$. However, by definition of $\ll^1_P$, $\pi'' \notin \Psi$, leading to a contradiction.

Only elements of $\aleph_P$ are minimal. Let $(\pi, \Psi) \notin \aleph_P$. Let us prove that it exists a couple $(\pi', \Psi') \in \aleph_P$ such that $(\pi', \Psi') \ll_P (\pi, \Psi)$. Consider the set Σ of sequences $(\pi_n, \Psi_n)_n$ such than $\pi_0 = (\pi, \Psi)$ and $(\pi_i, \Psi_i) \gg_P (\pi_{i+1}, \Psi_{i+1})$. Since, Φ_P is a finite set and, forall $i \geq 0$, $\Psi_i \subset \Psi_{i+1}$ by definition of $\gg_P$, any $(\pi_n, \Psi_n)_n$ is a finite sequence. If σ is a trace of Σ, π is a property and Ψ is a set of properties, we note $\sigma \gg_P (\pi, \Psi)$ the extension of σ with (π, Ψ) such that the resulting trace belongs to Σ. Thanks to this notation, we define a distance δ over traces of Σ as follows:

$$\begin{aligned}
\delta((\pi_n, \Psi_n)_n) &= 0 \qquad \text{if } \forall (\pi, \Psi) \in \Phi_P \times \mathscr{P}(\Phi_P), (\pi_n, \Psi_n) \not\gg {}_P(\pi, \Psi) \\
\delta((\pi_n, \Psi_n)_n) &= \min \left\{ k \in \mathbb{N} \,\middle|\, \begin{array}{l} \exists (\pi, \Psi) \in \Phi_P \times \mathscr{P}(\Phi_P), \\ \quad \delta((\pi_n, \Psi_n)_n \gg_P (\pi, \Psi)) = k - 1 \end{array} \right\} \text{ otherwise.}
\end{aligned}$$

Informally, δ measures the minimal distance of a trace to a Σ-sequence of maximal length. Now let us show by induction over δ that any element of any sequence of Σ is $\gg_P$-smaller than, or equal to, some $(\pi', \Psi') \in \aleph_P$: that will prove our goal. Let $\sigma = (\pi_0, \Psi_0) \gg_P \cdots \gg_p (\pi_{n-1}, \Psi_{n-1})$ be a sequence of Σ.

Case $\delta(\sigma) = 0$. By definition of δ, there is no (π_n, Ψ_n) such that $(\pi_{n-1}, \Psi_{n-1}) \gg_P (\pi_n, \Psi_n)$. Hence, by definition of $\ll_P$, either forall $(\lambda, \xi_{n-1}) \in \mathcal{L}_P(\pi_{n-1})$, ξ_{n-1} is the empty set or $\psi_{n-1} = \Phi_P$. In both cases, we can trivially conclude than $(\pi_{n-1}, \Psi_{n-1}) \in \aleph_P$. So the $\ll_P$-smallest element of σ belongs to $\aleph_P$: by transitivity and antisymmetry of $\gg_P$ any other element of σ is $\aleph_P$ $\gg_P$-bigger than (π, Ψ).

Case $\delta(\sigma) > 0$. By definition of δ, there exists (π_n, Ψ_n) such that σ is a prefix of the sequence $\sigma' = (\pi_0, \Psi_0) \gg_P \cdots \gg_P (\pi_n, \Psi_n)$ of length $n + 1$. Since $N - (n + 1) < N - n$, we can apply the induction hypothesis on σ': any element of σ' is $\ll_P$-bigger or equal to some couple $(\pi', \Psi') \in \aleph_P$. Hence, σ too by transitivity of $\gg_P$.

We now introduce the last definition before proving both theorems of the paper. Informally, it restricts the sets of already visited properties to those verifying the implicit invariants of our algorithm. Let P be a program and π be a property of P. We note Υ_π^n the set of finite sequences $\Psi_n = (\pi_n)_n$ of length n inductively defined by (consider that $\pi = \pi_{n+1}$):

$$\begin{aligned}
\Psi_0 &= \emptyset \\
\Psi_{i+1} &= \Psi_i \cup \{\pi_{i+1}\} && 0 \le i < n \\
\text{with } (\pi_{i+1}, \Psi_i) &\ll_P (\pi_i, \Psi_{i-1}) && 1 \le i \le n \\
\text{and } \exists(\lambda_i, \xi_i) \in \mathcal{L}_P(\pi_i), \lambda_i \neq \textit{Dont\_know} &\text{ and } \pi_{i+1} \in \xi_i && 1 \le i \le n.
\end{aligned}$$

With these preliminary definitions and properties, we are now able to prove both theorems of the paper.

Theorem 1 (Correctness of the Consolidation Algorithm). *Under assumptions 1 (blocking semantics) and 3 (weak correctness of analyzers), if the consolidation algorithm returns Valid (resp. Invalid) for a given property π, then π is valid (resp. invalid). In case of inconsistency, we can deduce both $\models \pi$ and $\not\models \pi$,* i.e. *an inconsistency. More formally, for a given program P:*

$$\forall \pi \in \Phi_P, \begin{cases} \textit{if } \mathcal{S}_P(\pi) = \textit{Valid}, & \textit{then } \models \pi; \\ \textit{if } \mathcal{S}_P(\pi) = \textit{Invalid}, & \textit{then } \not\models \pi; \\ \textit{if } \mathcal{S}_P(\pi) = \textit{Inconsistent} & \textit{then } \models \pi \textit{ and } \not\models \pi. \end{cases}$$

Proof. We actually prove the following more general result:

$$\forall n \in \mathbb{N}, \forall \pi \in \Phi_P, \forall \Psi_n \in \Upsilon_\pi^n, \begin{cases} \text{if } \mathcal{S}_P^{\Psi_n}(\pi) = \textit{Valid}, & \text{then } \models \pi; \\ \text{if } \mathcal{S}_P^{\Psi_n}(\pi) = \textit{Invalid}, & \text{then } \not\models \pi; \\ \text{if } \mathcal{S}_P(\pi) = \textit{Inconsistent} & \text{then } \models \pi \text{ and } \not\models \pi. \end{cases}$$

Let $n \in \mathbb{N}$. We prove the expected property by $\ll_P$-induction over (π, Ψ_n) (possible by lemma 2, well-foundedness of $\ll_P$):

Case $(\pi, \Psi_n) \in \aleph_P$**.** Let us prove separately the three expected properties.

Case $\mathcal{S}_P^{\Psi_n}(\pi) =$ ***Valid*****.** We have to prove $\models \pi$. According to the definitions of $\bigvee^I$ and $\overset{\text{HL}}{\Longrightarrow}$, there are two cases.

Case *True* $\in \Lambda_\pi$**.** By definition of $\bigvee^L$ and $\vee^L$, $(\textit{True}, \_) \in \mathcal{L}_P(\pi)$. Then by assumption 3 (weak correctness of analyzers), it exists $\Pi \subseteq \Phi_P$ such that $\Pi \models \pi$. Hence the expected result by lemma 1 (local validity implies validity).

Case *False* $\in \Lambda_\pi$**.** By definition of $\bigvee^L$, $(\textit{False}, \_) \in \mathcal{L}_P(\pi)$. Then:

$$\begin{aligned}
&\bigvee_{\xi \in \Xi_{\lambda_\pi}}^{H} \bigwedge_{\pi_\xi \in \xi \setminus \Psi_n}^{H} \mathcal{S}_P^{\Psi_n \cup \{\pi\}}(\pi_\xi) = \textit{Invalid} && \text{definition of } \overset{\text{HL}}{\Longrightarrow} \\
&\exists \xi \in \Xi_{\lambda_\pi}, \bigwedge_{\pi_\xi \in \xi \setminus \Psi_n}^{H} \mathcal{S}_P^{\Psi_n \cup \{\pi\}}(\pi_\xi) = \textit{Invalid} && \text{definition of } \vee^H \text{ and } \bigvee^H.
\end{aligned}$$

However, since $(\pi, \Psi_n) \in \aleph_P$, $\xi \setminus \Psi_n$ is empty. Hence, by definition of $\bigwedge^H$:

$$\bigwedge_{\pi_\xi \in \xi \setminus \Psi_n}^{H} \mathcal{S}_P^{\Psi_n \cup \{\pi\}}(\pi_\xi) = \mathit{Valid}.$$

Absurd since we previously demonstrated that this conjunction is invalid.

Case $\mathcal{S}_P^{\Psi_n}(\pi) = $ ***Invalid*****.** We have to prove $\not\models \pi$. According to the definitions of $\bigvee^I$ and $\overset{\text{HL}}{\Longrightarrow}$, *False* $\in \Lambda_\pi$. Then:

$$\bigvee_{\xi \in \Xi_{\lambda_\pi}}^{H} \bigwedge_{\pi_\xi \in \xi \setminus \Psi_n}^{H} \mathcal{S}_P^{\Psi_n \cup \{\pi\}}(\pi_\xi) = \mathit{Valid} \qquad \text{definition of } \overset{\text{HL}}{\Longrightarrow}$$

$$\exists \xi \in \Xi_{\lambda_\pi}, \bigwedge_{\pi_\xi \in \xi \setminus \Psi_n}^{H} \mathcal{S}_P^{\Psi_n \cup \{\pi\}}(\pi_\xi) = \mathit{Valid} \qquad \text{definition of } \vee^H \text{ and } \bigvee^H.$$

By assumption 3 (weak correctness of analyzers) which restricts the hypotheses of π when emitting *False* to at best $\{\text{reach}(\pi)\}$, ξ is either empty or $\{\text{reach}(\pi)\}$. Furthermore, since $(\pi, \Psi_n) \in \aleph_P$, $\xi \setminus \Psi_n$ is empty. So there only remains two cases.

Case $\xi = \emptyset$**.** Trivial by assumption 3 (weak correctness of analyzers).

Case $\text{reach}(\pi) \in \Psi_n$**.** Let us first prove that it exists $(\lambda, \xi) \in \mathcal{L}_P(\text{reach}(\pi))$ such that $\lambda = $ *True*. Since $\text{reach}(\pi) \in \Psi_n$, and by definition of Ψ_n, there exists $(\lambda, \xi) \in \mathcal{L}_P(\text{reach}(\pi))$ and π' such that $\lambda \neq$ *Dont_know* and $\pi' \in \xi$. Following assumption 3 (weak correctness of analyzers), π' must be $\text{reach}(\text{reach}(\pi))$. But, if $\lambda = $ *False*, that contradicts assumption 4 (do not prove unreachability with reachability). Hence $\lambda = $ *True*. So, by assumption 3 (weak correctness of analyzers) followed by lemma 1 (local validity implies validity), $\models \text{reach}(\pi)$. Furthermore, by assumption 3 (weak correctness of analyzers) again, since $($*False*$, \{\text{reach}(\pi)\}) \in \mathcal{L}_P(\pi)$, $\{\text{reach}(\pi)\} \not\models \pi$. Hence $\not\models \pi$ by definition of local invalidity.

Case $\mathcal{S}_P^{\Psi_n}(\pi) = $ ***Inconsistent*****.** Both *True* and *False* belong to Λ_π. So, following both cases which we just proved, we can trivially deduce $\models \pi$ and $\not\models \pi$ which is the expected result.

Case $(\pi, \Psi_n) \notin \aleph_P$**.** Let us prove separately the three expected properties.

Case $\mathcal{S}_P^{\Psi_n}(\pi) = $ ***Valid*****.** We have to prove $\models \pi$. According to the definitions of $\bigvee^I$ and $\overset{\text{HL}}{\Longrightarrow}$, there are two cases.

Case ***True*** $\in \Lambda_\pi$**.** This case is exactly equivalent to the same subcase of the basic case of the induction.

Case ***False*** $\in \Lambda_\pi$**.** Similarly to the same subcase of the basic case of the induction, we get:

$$\exists \xi \in \Xi_{\lambda_\pi}, \bigwedge_{\pi_\xi \in \xi \setminus \Psi_n}^{H} \mathcal{S}_P^{\Psi_n \cup \{\pi\}}(\pi_\xi) = \mathit{Invalid}.$$

By assumption 3 (weak correctness of analyzers) which restricts the hypotheses of π when emitting *False* to at best $\{\mathrm{reach}(\pi)\}$, ξ is either empty or $\{\mathrm{reach}(\pi)\}$. If $\xi \setminus \Psi_n$ is empty, then the case is absurd (see basic case of the induction). Otherwise, by definition of $\bigwedge^H$, we get

$$\mathcal{S}_P^{\Psi_n \cup \{\pi\}}(\mathrm{reach}(\pi)) = \textit{Invalid}.$$

As $\mathrm{reach}(\pi) \lessdot_P^1 \pi$ and $\mathrm{reach}(\pi) \notin \Psi_n \cup \{\pi\}$, $(\mathrm{reach}(\pi), \Psi_n \cup \{\pi\}) \ll_P (\pi, \Psi_n)$. Furthermore $\Psi_n \cup \{\pi\} \in \Upsilon^n_{\mathrm{reach}(\pi)}$. So we can apply the induction hypothesis on $(\mathrm{reach}(\pi), \Psi_n)$ to deduce $\not\models \mathrm{reach}(\pi)$. So, by definition of $\mathrm{reach}(\pi)$, no trace ends at π. Hence $\models \pi$ by definition of $\models$.

Case $\mathcal{S}_P^{\Psi_n}(\pi) = $ *Invalid*. We have to prove $\not\models \pi$. According to the definitions of $\bigvee^I$ and $\overset{\mathrm{HL}}{\Longrightarrow}$, *False* $\in \Lambda_\pi$. Then:

$$\bigvee^H_{\xi \in \Xi_{\lambda_\pi}} \bigwedge^H_{\pi_\xi \in \xi \setminus \Psi_n} \mathcal{S}_P^{\Psi_n \cup \{\pi\}}(\pi_\xi) = \textit{Valid} \qquad \text{definition of } \overset{\mathrm{HL}}{\Longrightarrow}$$

$$\exists \xi \in \Xi_{\lambda_\pi}, \bigwedge^H_{\pi_\xi \in \xi \setminus \Psi_n} \mathcal{S}_P^{\Psi_n \cup \{\pi\}}(\pi_\xi) = \textit{Valid} \qquad \text{definition of } \vee^H \text{ and } \bigvee^H.$$

By assumption 3 (weak correctness of analyzers) which restricts the hypotheses of π when emitting *False* to at best $\{\mathrm{reach}(\pi)\}$, ξ is either empty or $\{\mathrm{reach}(\pi)\}$. If ξ is empty or $\{\mathrm{reach}(\pi)\} \in \Psi_n$, then the proof is the same as the one of the basic case of the induction. Thus the remaining case is $\mathrm{reach}(\pi) \in \xi \setminus \Psi_n$. So $(\mathrm{reach}(\pi), \Psi_n \cup \{\pi\}) \ll_P (\pi, \Psi_n)$. Furthermore $\Psi_n \cup \{\pi\} \in \Upsilon^n_{\mathrm{reach}(\pi)}$: we can apply the induction hypothesis on $(\mathrm{reach}(\pi), \Psi_n \cup \{\pi\})$ to deduce $\models \mathrm{reach}(\pi)$. By definition of $\mathrm{reach}(\pi)$, there exists a trace σ which ends at π. Furthermore, by assumption 3 (weak correctness of analyzers), since $($*False*$, \{\mathrm{reach}(\pi)\}) \in \mathcal{L}_P(\pi)$, $\mathrm{reach}(\pi) \not\models \pi$. Hence $\not\models \pi$ by definition of $\not\models$.

Case $\mathcal{S}_P^{\Psi_n}(\pi) = $ *Inconsistent*. This case is similar to the same subcase of the basic case of the induction.

Theorem 2 (Completeness). *Under assumptions 1 and 2 (strong correctness of analyzers), if a property is valid (resp. invalid) and an emitter emits a local status different from Dont_know under recursively valid hypotheses, then the consolidation algorithm returns Valid (resp. Invalid). More formally, for a given program P:*

$$\mathcal{D}(\pi) \triangleq \exists \lambda \neq \textit{Dont\_know}, (\lambda, \Pi) \in \mathcal{L}_P(\pi) \textit{ and } \forall \pi_i, \mathcal{D}(\pi_i) \textit{ and } \models \pi_i.$$

then

$$\forall \pi \in \Phi_P, \begin{cases} \textit{if } \mathcal{D}(\pi) \textit{ and } \models \pi, \textit{ then } \mathcal{S}_P(\pi) = \textit{Valid}; \\ \textit{if } \mathcal{D}(\pi) \textit{ and } \not\models \pi, \textit{ then } \mathcal{S}_P(\pi) = \textit{Invalid}. \end{cases}$$

Proof. Let P be a program. Let us note, for any property π and set of properties Ψ:

$$\mathcal{D}^{\Psi}(\pi) \triangleq \exists \lambda \neq \textit{Dont\_know}, (\lambda, \Pi) \in \mathcal{L}_P(\pi) \text{ and } \forall \pi_i, \mathcal{D}^{\Psi \cup \{\pi\}}(\pi_i) \text{ and } \models \pi_i.$$

We actually prove the following more general result:

$$\forall n \in \mathbb{N}, \forall \pi \in \Phi_P, \forall \Psi_n \in \Upsilon_\pi^n, \forall \pi \in \Phi_P,$$
$$\begin{cases} \text{if } \mathcal{D}^{\Psi_n}(\pi) \text{ and } \models \pi, \text{ then } \mathcal{S}_P^{\Psi_n}(\pi) = \textit{Valid}; \\ \text{if } \mathcal{D}^{\Psi_n}(\pi) \text{ and } \not\models \pi, \text{ then } \mathcal{S}_P^{\Psi_n}(\pi) = \textit{Invalid}. \end{cases}$$

Let $n \in \mathbb{N}$. We prove the expected property by $\ll_P$-induction over (π, Ψ_n) (possible by lemma 2, well-foundness of $\ll_P$).

Case $(\pi, \Psi_n) \in \aleph_P$**.** Let λ be a local status different of *Dont_know*, and $\xi = \{\pi_i\}_i$ such than $(\lambda, \xi) \in \mathcal{L}_P(\pi)$ (if no such λ and ξ exist, the expected property is trivially true). We split the proof in two cases according to the value of λ.

Case $\lambda =$ ***True*****.** By definition of $\bigvee^L$ and $\bigvee^I$, $\lambda \in \Lambda_\pi$. By assumption 2 (strong correctness of analyzers), $\xi \models \pi$. Thus, by lemma 1 (local validity implies validity), $\models \pi$. So we have to prove $\mathcal{S}_P^{\Psi_n}(\pi) = \textit{Valid}$. According to the definition of $\aleph_P$, $\xi \setminus \Psi_n = \emptyset$. Then:

$$\bigwedge^H_{\pi_\xi \in \xi \setminus \Psi_n} \mathcal{S}_P^{\Psi_n \cup \{\pi\}}(\pi_\xi) = \textit{Valid} \qquad \text{definition of } \bigwedge^H$$
$$\bigvee^H_{\xi \in \Xi_{\lambda_\pi}} \bigwedge^H_{\pi_\xi \in \xi \setminus \Psi_n} \mathcal{S}_P^{\Psi_n \cup \{\pi\}}(\pi_\xi) = \textit{Valid} \qquad \text{definition of } \vee^H \text{ and } \bigvee^H$$
$$\bigvee^H_{\xi \in \Xi_\lambda} \bigwedge^H_{\pi_\xi \in \xi \setminus \Psi_n} \mathcal{S}_P^{\Psi_n \cup \{\pi\}}(\pi_\xi) \overset{\text{HL}}{\Longrightarrow} \lambda = \textit{Valid} \qquad \text{definition of } \overset{\text{HL}}{\Longrightarrow}.$$

By definition of $\bigvee^I$, $\mathcal{S}_P^{\Psi_n}(\pi)$ is either *Valid* or *Inconsistent*. In the former case, we directly get the expected result. In the latter case, by Theorem 1[6], we get an inconsistency from which we can trivially deduce the expected result.

Case $\lambda =$ ***False*****.** By definition of $\bigvee^L$ and $\bigvee^I$, $\lambda \in \Lambda_\pi$. By assumption 2 (strong correctness of analyzers), $\xi \not\models \pi$. Furthermore, according to the definition of $\aleph_P$, $\xi \setminus \Psi_n = \emptyset$. It follows:

$$\bigwedge^H_{\pi_\xi \in \xi \setminus \Psi_n} \mathcal{S}_P^{\Psi_n \cup \{\pi\}}(\pi_\xi) = \textit{Valid} \qquad \text{definition of } \bigwedge^H$$
$$\bigvee^H_{\xi \in \Xi_{\lambda_\pi}} \bigwedge^H_{\pi_\xi \in \xi \setminus \Psi_n} \mathcal{S}_P^{\Psi_n \cup \{\pi\}}(\pi_\xi) = \textit{Valid} \qquad \text{definition of } \vee^H \text{ and } \bigvee^H)$$
$$\bigvee^H_{\xi \in \Xi_\lambda} \bigwedge^H_{\pi_\xi \in \xi \setminus \Psi_n} \mathcal{S}_P^{\Psi_n \cup \{\pi\}}(\pi_\xi) \overset{\text{HL}}{\Longrightarrow} \lambda = \textit{Invalid} \qquad \text{definition of } \overset{\text{HL}}{\Longrightarrow}.$$

[6] Actually that is not precisely what Theorem 1 says : it expresses a statement for $\mathcal{S}_P$ and not $\mathcal{S}_P^{\Psi_n}$, but the proof of this theorem encloses the proof of the same property for $\mathcal{S}_P^{\Psi_n}$.

By definition of $\bigvee^{I}$, $\mathcal{S}_P^{\Psi_n}(\pi)$ is either *Inconsistent* or *Invalid*. In the former case, by Theorem 1[7], we get an inconsistency from which we can trivially deduce the expected result. So let us suppose $\mathcal{S}_P^{\Psi_n}(\pi) =$ *Invalid* and prove $\not\models \pi$ to be able to conclude. By assumption 2 (strong correctness of analyzers) which restricts the hypotheses of π when emitting *False* to at best $\{\mathrm{reach}(\pi)\}$, ξ is either empty or $\{\mathrm{reach}(\pi)\}$. Furthermore Since $(\pi, \Psi_n) \in \aleph_P$, $\xi \setminus \Psi_n$ is empty. So it only remains two cases.

Case $\xi = \emptyset$. Immediate by assumption 2 (strong correctness of analyzers).

Case $\mathrm{reach}(\pi) \in \Psi_n$. Let us first prove that it exists $(\lambda', \xi) \in \mathcal{L}_P(\mathrm{reach}(\pi))$ such that $\lambda' =$ *True*. Since $\mathrm{reach}(\pi) \in \Psi_n$, and by definition of Ψ_n, it exists $(\lambda', \xi) \in \mathcal{L}_P(\mathrm{reach}(\pi))$ and π' such that $\lambda' \neq$ *Dont_know* and $\pi' \in \xi$. Following assumption 2 (strong correctness of analyzers), π' must be $\mathrm{reach}(\mathrm{reach}(\pi))$. But, if $\lambda' =$ *False*, that contradicts assumption 4 (do not prove unreachability with reachability). Hence $\lambda' =$ *True*. So, by assumption 2 (strong correctness of analyzers) followed by lemma 1 (local validity implies validity), $\models \mathrm{reach}(\pi)$. Furthermore, by assumption 2 (strong correctness of analyzers) again, $\{\mathrm{reach}(\pi)\} \not\models \pi$. Hence $\not\models \pi$ by definition of local invalidity.

Case $(\pi, \Psi_n) \notin \aleph_P$. Let λ be a local status different of *Dont_know*, and $\xi = \{\pi_i\}_i$ such than $(\lambda, \xi \in \mathcal{L}_P(\pi)$ and, for each $\pi_i \in \xi$, $\mathcal{D}^{\Psi_n}(\pi_i)$ and $\models \pi_i$ (if no such λ and ξ exist, the expected property is trivially true). We split the proof in two cases according to the value of λ.

Case $\lambda =$ *True*. By definition of $\bigvee^{L}$ and $\bigvee^{I}$, $\lambda \in \Lambda_\pi$. By assumption 2 (strong correctness of analyzers), $\xi \models \pi$. Thus, by lemma 1 (local validity implies validity), $\models \pi$. So we have to prove $\mathcal{S}_P^{\Psi_n}(\pi) =$ *Valid*. For each $\pi_i \in \xi \setminus \Psi_n$, $(\pi_i, \Psi_n \cup \{\pi\}) \ll_P (\pi, \Psi_n)$. Furthermore $\Psi_n \cup \{\pi\} \in \Upsilon_{\pi_i}^{n}$: we can apply the induction hypothesis to each $\pi_i \in \xi \setminus \Psi_n$ to deduce $\mathcal{S}_P^{\Psi_n \cup \{\pi\}}(\pi_i) =$ *True* (since $\models \pi_i$ and $\mathcal{D}^{\Psi_n \cup \{\pi\}}(\pi_i)$). Then:

$$\bigwedge^{H}_{\pi_\xi \in \xi \setminus \Psi_n} \mathcal{S}_P^{\Psi_n \cup \{\pi\}}(\pi_\xi) = \textit{Valid} \quad \text{definition of } \wedge^{H} \text{ and } \bigwedge^{H}$$

$$\bigvee^{H}_{\xi \in \Xi_{\lambda_\pi}} \bigwedge^{H}_{\pi_\xi \in \xi \setminus \Psi_n} \mathcal{S}_P^{\Psi_n \cup \{\pi\}}(\pi_\xi) = \textit{Valid} \quad \text{definition of } \vee^{H} \text{ and } \bigvee^{H}$$

$$\bigvee^{H}_{\xi \in \Xi_{\lambda}} \bigwedge^{H}_{\pi_\xi \in \xi \setminus \Psi_n} \mathcal{S}_P^{\Psi_n \cup \{\pi\}}(\pi_\xi) \overset{\text{HL}}{\Longrightarrow} \lambda = \textit{Valid} \qquad \text{definition of } \overset{\text{HL}}{\Longrightarrow}.$$

By definition of $\bigvee^{I}$, $\mathcal{S}_P^{\Psi_n}(\pi)$ is either *Valid* or *Inconsistent*. In the former case, we directly get the expected result. In the latter case, by Theorem 1[8], we get an inconsistency from which we can trivially deduce the expected result.

[7] See footnote 6.

[8] See footnote 6.

Case $\lambda =$ *False*. By definition of $\bigvee^L$ and $\bigvee^I$, $\lambda \in \Lambda_\pi$. By assumption 2 (strong correctness of analyzers), $\xi \not\models \pi$ and ξ is either empty or $\{\text{reach}(\pi)\}$. We have to show $\mathcal{S}_P^{\Psi_n}(\pi) = $ *Invalid*. If ξ is empty or $\{\text{reach}(\pi)\} \in \Psi_n$, then the proof is the same as the one of the basic case of the induction. Thus the remaining case is $\text{reach}(\pi) \in \xi \setminus \Psi_n$. So $(\text{reach}(\pi), \Psi_n \cup \{\pi\}) \ll_P (\pi, \Psi_n)$. Furthermore $\Psi_n \cup \{\pi\} \in \Upsilon^n_{\text{reach}(\pi)}$: we can apply the induction hypothesis on $(\text{reach}(\pi), \Psi_n \cup \{\pi\})$ to deduce $\mathcal{S}_P^{\Psi_n \cup \{\pi\}}(\text{reach}(\pi)) = $ *Valid* (since $\models \text{reach}(\pi)$ and $\mathcal{D}^{\Psi_n \cup \{\pi\}}(\text{reach}(\pi))$). Then:

$$\bigwedge^H_{\pi_\xi \in \xi \setminus \Psi_n} \mathcal{S}_P^{\Psi_n \cup \{\pi\}}(\pi_\xi) = \textit{Valid} \qquad \text{definition of } \wedge^H \text{ and } \bigwedge^H$$

$$\bigvee^H_{\xi \in \Xi_{\lambda_\pi}} \bigwedge^H_{\pi_\xi \in \xi \setminus \Psi_n} \mathcal{S}_P^{\Psi_n \cup \{\pi\}}(\pi_\xi) = \textit{Valid} \qquad \text{definition of } \vee^H \text{ and } \bigvee^H$$

$$\bigvee^H_{\xi \in \Xi_{\lambda}} \bigwedge^H_{\pi_\xi \in \xi \setminus \Psi_n} \mathcal{S}_P^{\Psi_n \cup \{\pi\}}(\pi_\xi) \overset{\text{HL}}{\Longrightarrow} \lambda = \textit{Invalid} \qquad \text{definition of } \overset{\text{HL}}{\Longrightarrow}.$$

By definition of $\bigvee^I$, $\mathcal{S}_P^{\Psi_n}(\pi)$ is either *Invalid* or *Inconsistent*. In the former case, we directly get the expected result. In the latter case, by Theorem 1[9], we get an inconsistency from which we can trivially deduce the expected result.

[9] See footnote 6.

Model Checking the FlexRay Startup Phase[*]

Sjoerd Cranen

Eindhoven University of Technology

Abstract. The FlexRay protocol is an upcoming standard in automotive industry. Its specification is finalised and maintained by ISO. It is a time-triggered protocol that uses a fault-tolerant clock synchronisation mechanism. During a startup phase that should be resilient to certain faults, the clocks in the network are synchronised and the protocol is initialised. This paper presents a model of the startup phase of the protocol in the mCRL2 modelling language, and shows how model checking techniques can be used to check that the startup protocol fulfills the requirements. A previously unknown scenario is uncovered in which a single failing node can cause another node, or even the entire network, not to start up.

1 Introduction

In the year 2000, a consortium was established with the goal to design a new, time-triggered communication protocol for use in the automotive industry that would outperform CAN and TTP in both speed and reliability. At the end of 2009, the consortium was disbanded, leaving a final version of a time-triggered protocol called FlexRay. The final protocol definition, a 336 page document, became available in 2011, and is currently being transformed into an ISO standard.

Already in 2006, the first commercially available cars were equipped with FlexRay networks, enabling new algorithms for vehicle control because of its higher bandwidth.

Since FlexRay will be the basis for communication in many vehicles to come, we would like to establish that the protocol is correct, *i.e.*, that implementing a system according to the latest specification leads to a system that behaves predictably and that shows no undesirable behaviour. We base our notion of correctness on the requirements document [8] that was composed by the FlexRay consortium.

The FlexRay protocol requires that nodes are synchronised in order to communicate. The procedure of starting up a FlexRay network therefore is of particular interest, because it involves a distributed algorithm that should reach such a synchronised state in a reasonable amount of time. This procedure should work for any given startup scenario, and should be to some extent fault-tolerant.

We choose to formalise the FlexRay protocol by means of a model written in the mCRL2 specification language. This language has extensive support for the use of data in models, and allows us to create a concise model that stays close to the specification. Fault tolerence is checked by explicitly modelling a number of faults that should

[*] The work in this paper is supported by the Dutch HTAS-VERIFIED project, see `http://www.htas.nl/index.php?pid=154.`

be allowed to occur, according to the requirements document, and requiring the same properties to hold as for fault-free scenarios.

We have created a model that captures the details of that part of a node that orchestrates its operation while the node is starting up. Analysis of the model reveals a scenario in which the network does not correctly deal with a single failing node. To our knowledge, this scenario was not documented before.

We start by giving a brief overview of the FlexRay protocol, and describe how the startup procedure works. We then briefly discuss the requirements that the startup procedure should satisfy. After that, we show how we arrived at our model: we discuss the abstractions that we applied and we demonstrate how mCRL2 fragments are related to the protocol specification. We then describe the method used to verify the presented model, and subsequently present the verification results. We discuss related research, before wrapping up with some conclusions and suggestions for future work.

2 FlexRay

We base our analysis on the 3.0.1 version of the protocol [9]. A FlexRay network consists of a number of nodes that are each connected to one or two communication mediums. Such a medium may itself consist of a number of infrastructure components, but can be as simple as a pair of copper wires. FlexRay is a *time-triggered* protocol, which is to say that a clock that is synchronised across the network dictates which node has the right to write messages to a communication medium. This in contrast to for instance CAN, which is *event-triggered*: whenever the event occurs that a node wishes to send a message, the CAN protocol decides at that moment whether that node is allowed to so so, based on some priority scheme.

A *schedule* records which node has access to the medium at what time. The schedule is an access scheme that defines for a finite period (which the protocol specification calls a *cycle*) the allocation of bandwidth to network nodes. Indefinite repetition of the schedule allows any node to decide at any moment in time which node is allowed to write to the medium.

This scheme is only strictly followed in that part of the schedule that is called the *static segment* (although we should note that there are features that allow a user to slightly deviate from the scheme). The FlexRay requirements document [8] states that the aim of the protocol is to provide both 'deterministic' communication and 'on-demand' communication. To this end, the schedule may leave part of the schedule undecided; this part is called the *dynamic segment*. A priority based selection scheme is implemented on top of a time-triggered access scheme to dynamically allocate bandwidth to nodes that need it in the dynamic segment. We will not consider the use of a dynamic segment, and will therefore not describe the details of its implementation.

In the static segment of the schedule, time is divided into *slots*, and each slot is allocated to a network node that is allowed to send data in that slot. Data is sent in structured packets called *frames*. Some frames play a special role in the protocol: they can be marked as *sync frame* or as *startup frame*. A sync frame is a frame that is used by all nodes in the network to adjust the local view on the global clock. This is done using a variant of the clock synchronisation algorithm of Lundelius and Lynch [15]. A startup frame is a sync frame that is allowed to be sent during startup of the network, which we describe in more detail in the next section.

We distinguish three phases in setting up communication over a FlexRay network: *wake-up*, *startup* and *communicating*. The first phase is in place to wake up any nodes that are in low-power mode. Nodes that are awake listen to the medium and can participate in the startup phase. The startup phase is then entered, in which the nodes in the network try to establish a globally synchronised clock, thus agreeing on the current position in the schedule. At the end of the startup phase, all nodes should be aware of the current position in the schedule, and their clocks are (and are kept) synchronised. Communication can now proceed according to the schedule.

2.1 The Startup Phase

We study the behaviour of FlexRay networks during the startup phase of the protocol. In this phase, a distinction is made between *coldstart* nodes and regular nodes. Coldstart nodes are the only nodes that are allowed to start sending data on the bus if there is no activity on the bus yet. A FlexRay network can be configured to have any number of coldstart nodes, with a minimum of three (or two if the network consists of only two nodes).

When the network is awake and a coldstart node is requested by a client application to start communication, the node will start by listening to the bus for a duration that is equal to the length of two schedule cycles, to detect ongoing traffic. Note that we cannot yet speak of real cycles, because there is no shared time base. We will however not always be this strict and will sometimes say 'cycle' when we mean 'the local estimation of the duration of a cycle', and we will similarly speak of 'bits' and 'slots'.

If no communication is detected during this first listening phase, the coldstart node will decide to send a *collision avoidance symbol*, or CAS. The CAS is a signal to the other nodes to indicate that some node is trying to initiate communication: if another (coldstart) node sees a CAS, it will wait for another two cycles, expecting to hear more from the node that sent the CAS.

It may be the case that two or more nodes simultaneously decide to send a CAS. To resolve this situation, a leader is selected based on the schedule. The first node scheduled to send a startup frame will become the leader. This is implemented as follows: as soon as a coldstart node has started sending the CAS, it moves to a collision resolution stage.

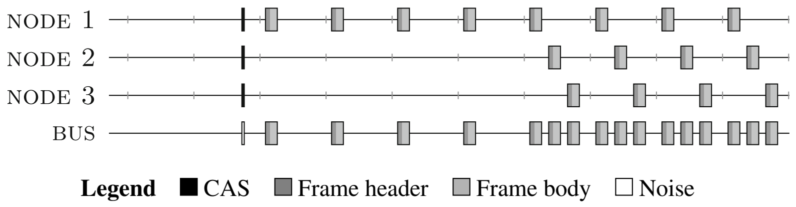


Fig. 1. Three nodes starting up. Black is CAS, dark grey is frame header and light grey is frame body. The BUS line is the combined signal of the three nodes.

During four cycles, it sends its startup frames according to the schedule, but if it sees a frame header on the bus, it aborts its startup attempt and returns to the listening stage. The result is that the first to send a frame after sending the CAS is the node that will initiate communication. Because the CAS was sent simultaneously, the clocks of the competing nodes are synchronized (within a certain error margin), which guarantees that the frames they send do not overlap and are therefore decoded correctly.

After four cycles, the initiating node checks during two consecutive cycles that it sees a frame from another node. If this is the case, then the startup phase ends for this node. If only one frame is decoded, then the initiating node considers the startup attempt failed, and it goes back to the listening stage. If no frames were decoded, then apparently no other nodes followed the initiative, so it is assumed that there simply were no other nodes ready to start communication yet. The initiating node waits for one cycle, and then resends the CAS and repeats the procedure.

If, in the listening phase, ongoing communication was detected, then a node will attempt to join in by first waiting for two consecutive frame headers from the same node to synchronise the clock with. It then checks during two cycles that either the node it synchronised on is still sending frames, or that at least two nodes are sending frames each cycle. If this is the case, it enters normal operation, if it is not, then it aborts its startup attempt and returns to the listening stage.

An example run of a fault-free startup is shown in figure 1, where nodes 1 and 2 start simultaneously, and node 3 joins in a little later.

2.2 Requirements

The FlexRay protocol defines a startup procedure by specifying the local behaviour of a FlexRay node. It is therefore not immediately clear how the startup phase can be defined at the level of a FlexRay network, and what this startup phase should ensure. We take inspiration from the FlexRay requirements document [8] in which we find requirements on a more global level. Regarding startup of networks, the requirements document specifies a number of faults that the system must be able to deal with when starting up:

> The wake-up and start-up of the 'communication system', the integration of 'nodes' powered on later and the reintegration of failed 'nodes' shall be fault-tolerant against: the temporary/permanent failure of one or more 'communication modules' (down to one module sending in the static part for mixed or pure static configurations), the temporary/permanent failure of one or more communication 'channel(s)' in a redundant configuration, and the loss of one or more 'frames'. *([8], page 84)*

A distinction is then made between faults associated to channels, faults associated to nodes and transient faults. In this paper, we only look at the latter two types of faults. A list of faults is provided, including for instance "A node (e.g. coldstart node) cannot receive any communication element on all its attached channels" (a node-related fault), and "A single bit of a communication element on one channel flips" (a transient fault, caused by electromagnetic interference).

For these types of faults, it is required that all fault-free nodes still reach—within finite time—a state in which they communicate according to schedule.

This leaves quite some room for interpretation. One of the faults that the system must be robust against, is for instance that a faulty node keeps disturbing the bus every once in a while. While it is doing so, the other nodes can obviously be prevented from communicating according to schedule. We therefore reinterpret successful startup to mean that the startup procedure has terminated successfully, and during one cycle in which none of the non-faulty nodes are executing their startup procedure, every frame that is sent by a non-faulty node is received by all other non-faulty nodes, unless a faulty node or transient fault prevents reception.

3 Model

The protocol specification defines the FlexRay protocol in terms of SDL (Specification and Description Language, [12]) diagrams and accompanying text, which, as stated in the introduction, is intended to provide "a reasonably unambiguous description of the mechanisms and their interactions" involved in the protocol. Furthermore, "an actual implementation should have the same behavior as the SDL description, but it need not have the same underlying structure or mechanisms". We are interested in this behaviour, and therefore construct a model that we can directly relate to the SDL description.

We discuss the most important aspects of our model, but we do not go into the details of specifying in mCRL2. We refer the interested reader to our technical report, which includes the full model [5].

A single FlexRay node consists of 12 concurrently running, interacting processes, called *controller host interface* (CHI), *protocol operation control* (POC), *macrotick generation* (MTG), *clock synchronization startup* (CSP-A, CSP-B), *clock synchronization processing* (CSP), *media access control* (MAC-A, MAC-B), *frame and symbol processing* (MAC-A, MAC-B) and *coding/decoding* (CODEC-A, CODEC-B). Some of these processes are dedicated to serve a single channel (A or B), and are hence duplicated. The discrete behaviour during the startup phase is governed by the POC process. We therefore aim to model this process in detail, while abstracting away as much as possible from the other processes.

3.1 Abstractions

We are, as stated before, interested in the discrete behaviour of a FlexRay network during the startup phase. This means we do not want to take into account timing aspects such as propagation delays, clock speed and so on. Like time, data also influences the behaviour of the protocol, but again we desire a limited level of detail: only features of the data structures used in FlexRay are modelled that influence the decisions made in the POC process. We describe the abstractions we use in more detail below.

Environment. During startup, the CHI process is used to feed back information to the client application. It may also influence the POC process by enabling the so-called *coldstart inhibit mode*, which causes a coldstart node to not actively start communication by sending a CAS. We assume none of the nodes are ever put in this mode, and hence leave out the CHI altogether.

The FSP process performs validity checks on the symbols that are decoded by the coding / decoding process. The only way in which FSP influences the POC process is by emitting a *fatal protocol error* signal, which happens when FSP detects that the node that it is part of is sending across a boundary in the schedule. We make the assumption that CODEC and MAC processes function correctly, which should prevent this error from occurring. We therefore do not model FSP.

Communication and calculation. Component interactions are modelled in the SDL description as signals being sent from one component to another. We make the assumption that messages are received the moment they are sent. This eliminates the need for (possibly unbounded) queues to model interactions between processes. Similarly, we assume that calculations take no time to complete.

Time. We make the assumption that all nodes in a network are always synchronised, eliminating any effect that clock synchronisation might have. This means that the CSS, MTG and CSP processes need not be modelled. This approach is similar to that of [19].

We implement a discrete clock by means of a synchronisation barrier: all processes synchronise every clock tick. The resolution of the clock is chosen to match the duration of a single bit (`gdBit` in [9]). In the mCRL2 modelling language, this amounts to allowing some actions to occur only in combination with a matching action from every other concurrently running process. There is no need for a separate clock process.

Data. Although we are modelling time at a resolution suitable to model every bit that is communicated in the protocol as-is, doing so leads to a statespace that is much too large. For example, a frame containing 16 bits of data requires 80 bits on the bus. For a network of three nodes, each sending a single frame, not taking into account any time in which the bus should be idle, simply summing up the initial phasing of nodes (assuming they start within one cycle of eachother) would already take millions of states, and modelling every frame that could possibly be sent would be out of the question.

We therefore try to compress the bit patterns into a minimal form that preserves some of the properties of real bit patterns on the bus. We no longer require the amount of bits to be realistic: we allow frames consisting of a one-bit frame header and a one-bit frame body, and the CAS (which is usually at least 11 bits long) can be only two bits long. We design the model such that we can choose the size of our symbols to be arbitrarily large.

Four types of symbols are of interest during startup: the CAS, frame headers, frame bodies and the *channel idle recognition point* (CHIRP). The latter is not a pattern that is actively sent, but a pattern that is decoded when no node sends data for a certain amount of time. Whenever a CHIRP is decoded by a node, it considers the bus to be idle until it detects that data is sent over the bus. We have chosen the symbols such that we can model the events that POC responds to (for instance, we need to be able to model the event that a frame header has been decoded from the bus).

In our model, each bit on the bus can have one of six values: H_{id}, D_{id}, B_{id}, *CAS*, *None* or *Noise*. The encoding of symbols is shown schematically in Figure 2. For frame data, every bit also carries an *id* field that identifies its sender. This is used to model the

CRC checks in the frame header and frame body: we assume these checks are perfect, so a CRC check passes if and only if the series of bits is equal to the sequence that was originally sent. We model this check by checking that all bits in the sequence are sent by the same sender, and make sure that each *id* is only used by one node in our model.

We assume that the simultaneous sending of two bits that are not of value *None* will always result in *Noise*. In existing implementations, the CAS is a sequence of 'dominant' bits, so two simultaneously sent CAS signals result in a valid CAS symbol again. The protocol does however not require this. In this paper we choose to not take into account this notion of dominance, so nodes have maximal potential to disturb eachothers transmissions. We have performed verification on models where we have taken dominance into account, but this did not yield different results.

3.2 Structure

We model a network consisting of three coldstart nodes that are all attached to a single channel in a bus topology. The channel is modelled by a process called *Bus*, which runs in parallel with three *Node* processes, each representing one coldstart node. The bus and the nodes synchronise on every clock tick. In between clock ticks, the bus goes through a writing phase, in which every node may write data to the bus using a *Put* action, and a reading phase, in which every node retrieves the combined signal using a *Get* action.

Figure 3 shows the structure of a node and how it is connected to the bus. Each node process consists of three communicating processes running in parallel: the *POC* process, which models the *Process Operation Control* SDL process, the *MAC* process, which (coarsely) models the *Media Access Control* SDL process and the *CODEC* process, which is a coarse model of the *Coding/Decoding* SDL process for one channel. The arrows indicate communication, and are labelled by the actions in the model that implement this communication. The dotted lines show which actions are synchronised every clock tick (the *bus, bit, wait* and *Encode* actions can only occur on a clock tick).

The MAC process is responsible for dispatching encoding requests to the CODEC. It can be ordered by POC to send a CAS, after which it starts sending frames periodically, or, in the case of integration into existing traffic, it can be requested to start sending frames immediately (but according to the schedule). When a startup attempt fails, it can be requested to go back to an inactive mode again.

The CODEC is modelled as a process that either reads from or writes to the bus. When in reading mode, it processes bits it reads from the bus, and does not write anything to the bus (which is implemented as writing silence to the bus). When it is in

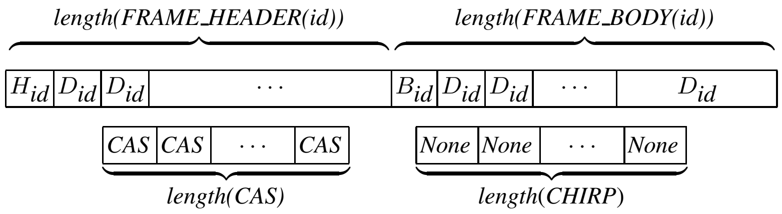


Fig. 2. Encoding of symbols on the bus

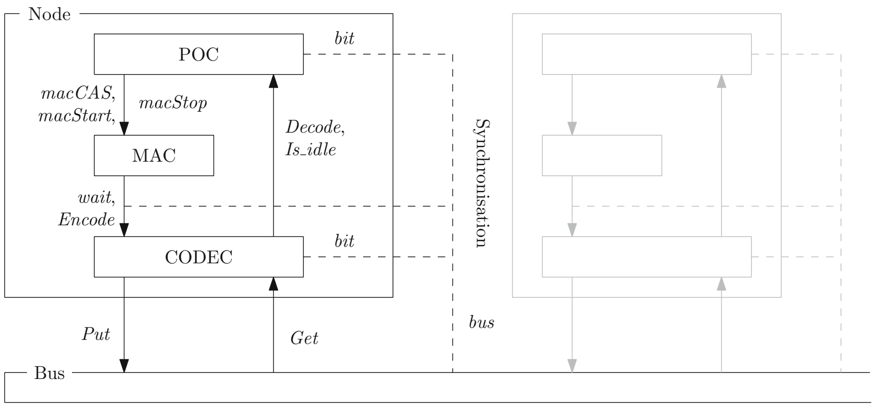


Fig. 3. Detail of the `Node` and `Bus` processes

writing mode, bits it reads from the bus are ignored, and it writes an encoding of the last requested symbol to the bus. This can be seen from Figure 3-18 in [9], where decoding is explicitly halted when an encoding request is received by the CODEC process.

The POC process is defined by a number of SDL procedures that call eachother. We follow this structure by modelling each SDL procedure by its own mCRL2 process. Every process can be seen as a superstate that is input enabled with respect to the signals coming from MAC and CODEC. Only SDL procedures that are executed by coldstart nodes are modelled.

Decisions in these procedures that involve signals coming from the clock synchronisation mechanism—POC for instance uses timers and signals that are generated at the start of a cycle—are emulated using minimal local administration (usually a counter that is incremented after every clock tick).

3.3 Verification

Our goal is to verify that our model satisfies the requirement from Section 2.2, *i.e.*, that the network starts up correctly in the presence of certain faults. The faults that are within the scope of our investigations can all be seen as instances of a few general problems: either a node is not able to send anything, a node is not able to receive anything, or the bus misbehaves in such a way that symbols are not always transmitted correctly. Only the periodic resetting of a node requires the node to display slightly more complicated behaviour. Two faults described in [8] comprise a node sporadically disturbing the bus; from the perspective of the correctly functioning nodes, however, this is not different to having a noisy bus.

Since for the POC a noisy signal is observably equal to no signal at all (the CODEC simply does not generate events), we model a limited set of scenarios. Each of the descriptions below describes two scenarios: one in which the node with the lowest identifier is the faulty node, and one in which another node is faulty. This is necessary because the protocol relies on a leader election mechanism that is not quite symmetric: although the process descriptions for startup are the same for every node, the leader that will be elected depends on the configuration of the nodes. The candidate configured with the lowest identifier will be elected as leader. At least one failure scenario (*viz.* the resetting node scenario below) [18] is only possible if the node with the lowest identifier is the faulty node.

In this manner, the following categories of scenarios are modelled.

Two nodes. A faulty node does not switch on at all, so effectively there are only two nodes present in the network.

Silent node. A faulty node is not able to send anything. Although we do model this separately, we note that this scenario is equivalent to the two-node scenario if we are not interested in the behaviour of the faulty node. We include this scenario because it shows that the silent node is still able to integrate into the communication correctly, albeit in a read-only mode.

Deaf node. A faulty node does not receive anything.

Resetting node. A node resets itself periodically.

Noisy channel. Signals sent by nodes are corrupted on the channel. We use a noise model that consists of a burst length and a maximum backoff time. The burst length determines the maximum number of sequential bits that are corrupted, the maximum backoff time determines the maximum number of sequential bits that pass through the channel unaltered. Due to practical limitations, we were only able to model this scenario in a two-node scenario.

For each of these scenarios, we check that the correctly functioning nodes start up. We do this by checking three properties. The first is absence of deadlock; a reachable deadlock would indicate an error in the model, rather than in the FlexRay protocol, for the construction of the model is such that time is always allowed to progress.

Absence of deadlock is checked while traversing the statespace. The other two properties are formulated in the first order modal μ-calculus (see, *e.g.*, [11]). For brevity, we use mathematical syntax rather than concrete mCRL2 μ-calculus syntax, and extra statements to help the mCRL2 toolset (*e.g.*, to prevent quantifiers from being expanded forever) are left out. It is important to note that these formulae only represent the intended properties correctly if the system they are checked on is deadlock free, as otherwise the $[\mathbf{true}]\varphi$ subformulae might trivially hold.

The second property asserts that eventually all correctly functioning nodes enter normal operation exactly once, an event that is flagged by the *enter_operation* action. It is expressed by the following formula, in which N is the total number of nodes and C is the set of correctly functioning nodes:

$$\begin{array}{l}
\mu X(r{:}\, 2^{\mathbb{N}} = C)\,.\,(\\
\qquad r \neq \emptyset \\
\quad\; \wedge\ (\forall i{:}\,\mathbb{N}\,.\,(i \in r \Rightarrow [\mathit{enter\_operation}(i)]X(r \setminus \{i\}))) \\
\quad\; \wedge\ [\neg\exists i{:}\,\mathbb{N}\,.\,\mathit{enter\_operation}(i)]X(r) \\
)\vee (r = \emptyset \wedge [\mathbf{true}^*][\exists i : \mathbb{N}\,.\, i \in C \wedge \mathit{enter\_operation}(i)]\mathbf{false})
\end{array}$$

The set r keeps track of which correctly functioning nodes are still running their startup procedure. Least fixpoint X denotes that all paths along which $r \neq \emptyset$ are finite, and the two quantified conjuncts remove i from r when along such a path the *enter_operation*(i)

action is encountered. From states in which X holds, we can effectively reach a state in which every node from C has executed its corresponding *enter_operation* action once and in which the second disjunct holds, which says that no node from C will ever do an *enter_operation* action again, but have done one once.

The last property says that eventually all correctly functioning nodes will keep receiving each others messages. Even though our model is not intended to model the ongoing traffic after startup, we have constructed our model in such a way that this property should hold. If this property does not hold, then it is likely that the nodes did not synchronise correctly. It is characterized by the following formula:

$$
\begin{array}{l}
\mu X \,.\, [\mathbf{true}]X \;\vee (\\
\nu Y(s\colon Symbol = \mathit{firstsymbol}) \,.\, (\\
\;\mu Z(r\colon 2^{\mathbb{N}} = C \setminus \mathit{sender}(s)) \,.\, ((\\
\qquad r \neq \emptyset\\
\quad\;\; \wedge\; (\forall i\colon \mathbb{N}, s'\colon Symbol \,.\\
\qquad\quad [\mathit{Decode}(i,s')](\,(i \in C \wedge s = s' \wedge i \in r \wedge Z(r \setminus \{i\})) \vee\\
\qquad\qquad\qquad\qquad\quad (i \notin C \wedge Z(r)))\\
\qquad)\\
\quad\;\; \wedge\; [\neg\exists i\colon \mathbb{N}, s'\colon Symbol \,.\, \mathit{Decode}(i,s')]Z(r))\\
\;\;) \vee (\\
\qquad r = \emptyset\\
\quad\;\; \wedge\; Y(\mathit{nextsymbol}(s))\\
\;)) \,) \,)
\end{array}
$$

Fixpoint X holds in every state where always eventually Y will hold. We assume that *firstsymbol* and *nextsymbol* are mappings that define the FlexRay schedule, *i.e.*, they define a repetitive pattern of frame headers and frame bodies that we expect to see on the bus. Then Y is true in states from which all correct nodes will decode *firstsymbol* first, followed by *nextsymbol*(*firstsymbol*), etcetera: it represents an infinite repetition of finite paths along which the currently scheduled symbol is decoded. The sender of a symbol is excluded from the set of recipients. The subformula

$$[\mathit{Decode}(i,s')]((i \in C \wedge s = s' \wedge i \in r \wedge Z(r \setminus \{i\})) \vee (i \notin C \wedge Z(r)))$$

makes sure that correct nodes can only decode the right symbol, and can do so only once (by removing them from r), but allows faulty nodes to decode arbitrary symbols.

Verification of these properties is done by linearising the mCRL2 specification and combining it with the formulae to form parameterised Boolean equation systems. These are instantiated to Boolean equation systems, which are in turn reduced modulo stuttering equivalence on parity games. The resulting smaller equation systems are then solved. A description of this procedure can be found in [6]. We use the July 2011 release of the mCRL2 toolset.

We note that it is also possible to check eventual startup and eventual communication by manual inspection. By hiding all actions but *enter_operation* and then reducing the statespace using branching bisimulation, the first property can be checked. The second property can be checked manually by hiding all but *Decode*, reducing the statespace using divergence preserving branching bisimulation and then manually inspecting all strongly connected components.

4 Results

All three properties hold for the 'no faulty nodes' and 'two nodes' scenarios. The other fault scenarios we discuss seperately. The figures that illustrate each scenario are generated from traces in our model.

Silent Node. All three properties hold on the system. Manually inspecting the branching-bisimulation reduced statespace reveals that the failing node can in this case enter normal operation using the wrong schedule (see Figure 4.a). The clock synchronisation process will allow this scenario, and frame and symbol processing will also not detect the mistake while the startup protocol has not finished. The mistake is harmless, however, because the silent node cannot disturb ongoing communication, and does not

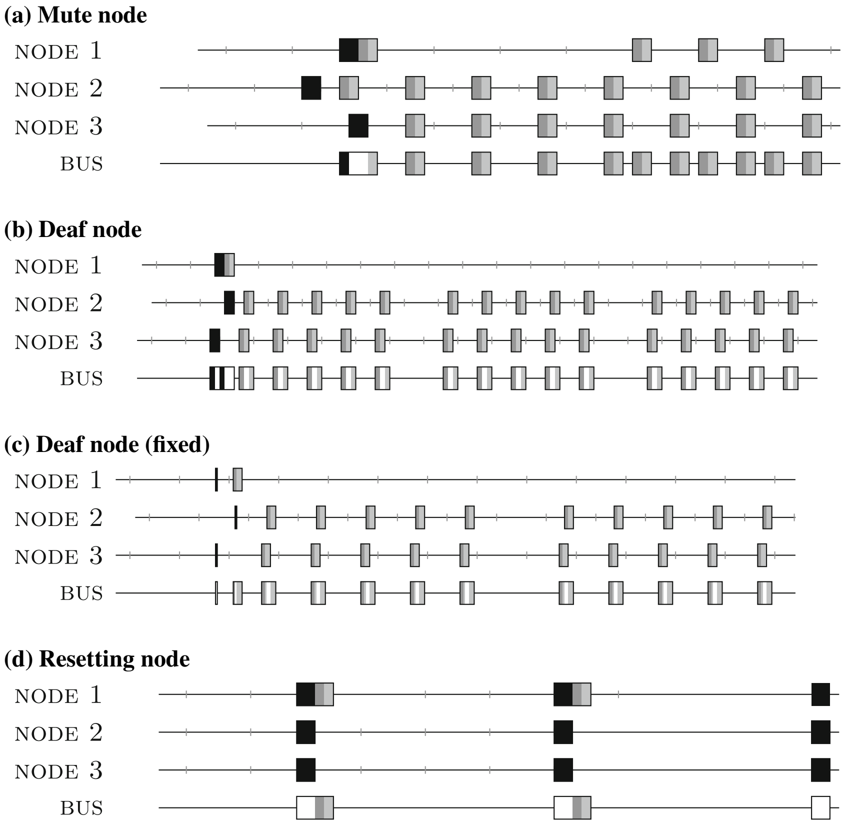


Fig. 4. Node 2 is mute, and can therefore start operation out of sync.

violate any requirement because it is the failing node that suffers the consequences. As soon as normal operation is entered, the clock correction process or the frame and symbol processing process of the faulty node will notice the error. A next attempt to integrate will succeed, because there is then already ongoing traffic.

Deaf Node. The statespace is deadlock free, but neither of the μ-calculus properties hold, because there is a possibility of the network not starting at all. This violates requirement 2111 nr. 6 in [8]. Figure 4.b shows such a scenario.

The deaf node can choose to align its frames with those of another startup node, causing only the headers of the other node to be readable on the bus. The non-faulty node that is broadcasting startup frames will not detect that every sent frame is corrupted by the faulty node. Because the non-faulty node's frame headers are untouched, all other nodes will wait until it gives up after the maximum number of startup attempts.

Although this scenario is a valid trace in our model, it exposes an inaccuracy in the model: because we did not model the non-coldstart behaviour, Node 3 simply stops after it spent its maximum number of coldstart attempts (three in this case). In reality, it would switch to an integrating mode, and would still be able to start up the network together with Node 1.

The scenario can however still be reproduced by changing the parameters somewhat: if each node starts with two remaining coldstart attempts (this is the minimal allowed configuration value), then Node 1 has spent one attempt after this scenario, and has one attempt left. The FlexRay protocol demands that at least two coldstart attempts are available in order to initiate a coldstart, which results in Node 1 switching to integration mode just like Node 2 and 3. We note that in the case of three or more coldstart attempts for at least two non-faulty nodes, the network will always start up because at least one node will be left with enough attempts to initiate a coldstart again.

The scenario was reproduced in our model, taking into account the ratios of the lengths of symbols and idle times on the bus (also taking into account overhead like frame / byte start sequences), and taking into account bus idle time that is enforced by the protocol (more specifically, the 'action point offset' and the idle time between the frame end sequence and the next slot boundary). The result is shown in Figure 4.c, where the following key configuration values are used.

`gdSampleClockPeriod`	0.0125 μs	`pSamplesPerMicrotick`	2
`pMicroPerMacroNom`	240	`gdTSSTransmitter`	9 gdBit
`gdStaticSlot`	4 MT	`gPayloadLengthStatic`	3

Resetting Node. The statespace is deadlock free, but neither of the μ-calculus properties hold. This violates requirement 2111 nr. 9 in [8]. Although this scenario was already known (it was described in [19]), the emergence of the trace in Figure 4.d gives us confidence that our model is correct. The trace shows that the leading node may cause startup of the network to fail by resetting itself every time it has sent a frame. In fact, it would just have to send the frame header, but the way we modelled our reset behaviour does not allow this.

It should be noted that in this scenario it is required that node 1 be the faulty node, which is not necessary in the scenarios for deaf and mute nodes.

Noisy Channel. The results depend very much on the parameters of the noise model. For an arbitrary noise pattern, it is obvious that the system will not start up. The channel could simply decide to corrupt all traffic going through it. The noise model we chose guarantees that some information will come through. Checking exactly for which values of maximum burst size and maximum backoff period the system starts correctly is too big a task, but simply trying a few settings soon gives an idea of how robust the system is. We made the following observation.

If there is noise on the channel for too long while nodes are trying to commence the startup procedure, then obviously startup may fail. The interesting scenarios are those in which some information can be communicated. However, if the minimum backoff time is less than the time needed for fault-free startup, then one of the sync frames of the leading coldstart nodes can always be corrupted, causing either the schedule initialisation or the consistency check of the other nodes to fail. If the presence of noise is the only anomaly in the system, then the minimum backoff time being at least the time required for fault-free startup is enough to guarantee that the system will come up.

It is interesting to note that in the verification of these properties, memory usage was not the bottleneck; the largest model used in our verification was that in which the resetting node was modelled, which consisted of around 26 million states and 76 million transitions. Generating this statespace is rather time-consuming however, most likely due to the multi-way communication used to model the clock tick, which can give rise to rather large guard expressions.

The verification of properties on the network via instantiation of parameterised boolean equation systems suffers from a similar problem. Although solving the generated equation system can be done quickly, generation takes a lot of time. This is currently preventing us from performing verification on models of networks with more than three nodes.

5 Related Work

The FlexRay protocol has been studied quite extensively, from numerous different perspectives. In this section we give a brief overview of previous studies known to us, and describe how they relate to our investigations.

Kühnel et al. aim to provide a framework in which distributed applications can be verified if they use a combination of an OSEK compliant (real-time) operating system, FTCOM (a fault-tolerant communication layer for OSEK operating systems) and FlexRay communication [13,14,17]. They use a specification language called FOCUS, for which refinement checks are carried out with the Isabelle/HOL theorem prover [4]. Their model is based on the 2.0 version of the FlexRay protocol specification, and they do not model start-up behaviour [13]. Their main aim is to verify applications built on top of a FlexRay network. This work is further extended by Botaschanjan et al. [3,2].

Zhang uses theorem proving to prove three functional properties under the assumption of synchronised clocks [21]. He takes into account local bus guardians [7].

The start-up behaviour of FlexRay networks was analysed by Malinsky in [16]. He uses UPPAAL to create a timed-automata representation of a system consisting of two *coldstart* nodes and one non-coldstart node. Using a few different settings for a number of FlexRay parameters, this system is checked for deadlock, and it is checked that the system starts up normally. It should be noted that this setup is not a valid FlexRay setup, because in a network with three nodes all nodes must be coldstart nodes, according to [9]. However, the requirements document [8] does state that startup must succeed when, due to a fault, only two coldstart nodes are active.

In our work, we do not model the FlexRay clock synchronisation, which is a modification of a clock synchronisation protocol described by Lundelius and Lynch [15]. Barsotti et al. have verified (amongst other protocols) the latter [1,10], although Zhang notes that the correctness of the FlexRay clock synchronisation protocol does not trivially follow from these results [20].

Steiner uses the SAL model checker to find failures in the startup protocol [18,19]. He identifies a scenario in which the system does not start up due to a single fail-silent node. The approach here is very similar to ours, and indeed the scenario found here is also detected by our model. Our model contains more detail, however, allowing us to find more subtle errors.

6 Conclusion

We have modelled a 3-node FlexRay network during communication start-up, using the mCRL2 modelling language. The core of the model was constructed by translating SDL specifications of the *process operation control* process on every node, which implements most of the discrete behaviour of a node during start-up. We formulated two properties on the model in the first-order modal μ-calculus.

Analysis of the model revealed two violations of the FlexRay requirements, one of which is a scenario that was not known before. This error could be detected because our model captures more details of the specification than the models used in [18].

We intend to check the same properties on a 4-node network, but currently the verification of such a network is too time consuming. The culprit seems to be the multi-way communication that is used to implement our assumption of synchronously running nodes. We consider it future work to see if this problem can be avoided by choosing a different synchronisation method, or if it can be remedied by preprocessing the specifications that are currently processed directly by the mCRL2 toolset.

If the aforementioned scaling issues can be overcome, it would be interesting to also extend the model by adding more detail to the MAC, FSP and CODEC processes defined in the protocol specification, in the same manner as was now done for the POC process.

Acknowledgements. I would like to thank to Alexander Kordes from the University of Siegen for noticing that the scenario in Figure 4.b was flawed. I would also like to thank Tim Willemse, Bas Luttik and Jeroen Keiren from the Eindhoven University of Technology, and Abhijit Deb and Bart Vermeulen from NXP Semiconductors for valuable feedback on previous versions of this document.

References

1. Barsotti, D., Nieto, L., Tiu, A.: Verification of Clock Synchronization Algorithms: Experiments on a combination of deductive tools. Formal Aspects of Computing 19(3), 321–341 (2007)

2. Botaschanjan, J., Broy, M., Gruler, A., Harhurin, A., Knapp, S., Kof, L., Paul, W., Spichkova, M.: On the correctness of upper layers of automotive systems. Formal Aspects of Computing 20(6), 637–662 (2008)
3. Botaschanjan, J., Gruler, A., Harhurin, A., Kof, L., Spichkova, M., Trachtenherz, D.: Towards Modularized Verification of Distributed Time-Triggered Systems. In: Misra, J., Nipkow, T., Karakostas, G. (eds.) FM 2006. LNCS, vol. 4085, pp. 163–178. Springer, Heidelberg (2006)
4. Broy, M., Stølen, K.: Specification and development of interactive systems: focus on streams, interfaces, and refinement. Springer (2001)
5. Cranen, S.: Model checking the flexray startup phase. Technical Report 12-01, Eindhoven University of Technology (2012)
6. Cranen, S., Keiren, J.J.A., Willemse, T.A.C.: Stuttering Mostly Speeds Up Solving Parity Games. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 207–221. Springer, Heidelberg (2011)
7. Flexray consortium: FlexRay Preliminary Node Local Bus Guardian Specification v2.0.9 (2005)
8. Flexray consortium: FlexRay Requirements Specification v2.1 (2005)
9. Flexray consortium: FlexRay Protocol Specification v3.0.1 (2010)
10. Fontaine, P., Gupta, K., Marion, J., Merz, S., Nieto, L., Tiu, A.: Towards a combination of heterogeneous deductive tools for system verification: A case study on clock synchronization. In: APPSEM Workshop, Citeseer (2005)
11. Groote, J., Mateescu, R.: Verification of temporal properties of processes in a setting with data. In: AMAST, pp. 74–90 (1998)
12. ITU-T: Recommendation Z.100: Specification and Description Language (SDL) (1999)
13. Kühnel, C., Spichkova, M.: FlexRay und FTCom: Formale Spezifikation in FOCUS. Technical report, Technische Universität München (2006)
14. Kühnel, C., Spichkova, M.: Upcoming Automotive Standards for Fault-Tolerant Communication: FlexRay and OSEKTime FTCom. In: EFTS 2006 International Workshop on Engineering of Fault Tolerant Systems. Universite du Luxembourg, CSC: Computer Science and Communication (2006)
15. Lundelius, J., Lynch, N.: A new fault-tolerant algorithm for clock synchronization. In: Proc. 3rd ACM Symposium on Principles of Distributed Computing, p. 88. ACM (1984)
16. Malinský, J., Novák, J.: Verification of flexray start-up mechanism by timed automata. Metrology and Measurement Systems XVII(3), 461–480 (2010)
17. Spichkova, M.: FlexRay: Verification of the FOCUS Specification in Isabelle/HOL. A Case Study. Technical report, Technische Universität München (2006)
18. Steiner, W.: An assessment of FlexRay 2.0 (safety aspects). Technical Report 45/2005, Technische Universität Wien (2005)
19. Steiner, W.: Model-checking studies of the FlexRay startup algorithm. Technical Report 57/200, Technische Universität Wien (2005)
20. Zhang, B.: On the Formal Verification of the FlexRay Communication Protocol. In: Automatic Verification of Critical Systems (AVoCS), pp. 184–189 (2006)
21. Zhang, B.: Specifying and Verifying Timing Properties of a Time-triggered Protocol for In-vehicle Communication. In: Proceedings of the 2008 Ninth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, pp. 467–472. IEEE Computer Society (2008)

Model-Based Risk Assessment Supporting Development of HSE Plans for Safe Offshore Operations*

Rainer Droste[1], Christoph Läsche[2], Cilli Sobiech[2], Eckard Böde[2], and Axel Hahn[1]

[1] Carl von Ossietzky Universität Oldenburg
{droste,hahn}@wi-ol.de
[2] OFFIS, Institute for Information Technology
{laesche,boede,sobiech}@offis.de

Abstract. The commercial installation of offshore wind farms is still far from having established standards or procedures and puts high demands on employees who deal with uncertainty and risks. We present a model-based risk assessment approach to support the development of health, safety, and environment (HSE) plans for safe offshore operations. For this purpose, a process model is used to integrate all aspects of these complex and safety-critical operations which involve many different actors, resources, and environmental conditions. On the basis of this model, we are able to identify and precisely describe hazards, quantify their safety impact, and develop risk mitigation means. To this end, we developed methods and tools to support this process, resulting in a formalization of hazardous events that can be used to unambiguously describe the risks of a given offshore operation model. We will demonstrate the feasibility of our approach on a specific offshore scenario.

1 Introduction

The radical change in the energy market towards renewable energy production, initiated by politics, causes a high demand for installation of offshore wind farms. The European Union set a mandatory target of at least 20 percent of produced energy originating from renewable sources by 2020 [1]. Yet, with merely 12 years of experience in the commercial installation of offshore wind farms, the industry is still in its infancy. In Germany, 24 wind parks in the North Sea have been approved so far [2]. However, the construction of many of these wind parks is delayed. As such a huge change in a short time can only be realized by a large amount of companies constructing multiple facilities concurrently, a lot new players rush into the offshore wind energy market. Not all of these companies have extended experience in the maritime or offshore sector and are familiar with the required health, safety, and environment (HSE) procedures. Development

* This work was partially supported by the European Regional Development Fund (ERDF) within the project *S*afe *O*ffshore *OP*erations (SOOP),
http://soop.offis.de/

and implementation of the necessary practices and processes is a highly complex task. “The average time needed to register and apply for permits [...] can take as long as five or seven years to navigate the process [3],” yet varying for different EU member states. For each permit, a substantial management plan considering health, safety, and environmental aspects is necessary to guarantee workplace safety. Recent events have shown that profound assessments are essential to protect personnel as well as the environment [4–7].

The SOOP project[1] aims at supporting planning and execution of safe offshore operations for construction and maintenance of offshore wind turbines. A special focus is set on human behavior. To analyze an operation, a model-based approach is used, also for representing the behavior of the involved persons as described in [8]. Thus, a conceptual model is build and maintained that describes the interaction of systems and persons as well as the evolution of the system. The architecture of the system and thus the one of the conceptual model will be changing over time as new needs might arise during the project period. Another aspect of the SOOP project is the identification and mitigation of possible risks during the planning process. Its results will also be used for an online assistant system that monitors the mission (e.g. the location of crew and cargo, cf. [9]) and warns if a hazardous event is emerging. This is intended as a further way to avoid risks during an offshore operation.

In this paper, we will focus on model-based planning and the risk assessment aspects of the project. Though the project is still in an early phase and the results might not be fixed and also not fully elaborated, we will discuss our currently planned approach and the developed methods.

2 Health, Safety, and Environment Aspects for Offshore Operations

The permit for the installation of offshore wind farms is accompanied by a substantial management plan considering health, safety, and environmental aspects [3]. HSE plans cover and improve workplace safety, health, and environment performance of the company in charge and the respective contractors. International guidelines for HSE management have also been developed in the offshore oil and gas industry, e.g. the OGP guidelines on HSE management [10, 11]. Since the 1970s to 1980s, the usage of Quantitative Risk Assessment (QRA) studies for offshore oil and gas operations in the North Sea has become a key issue in the management of HSE [12]. So far, the HSE situation in the offshore wind energy industry is very different from the situation for other offshore operations, e.g. due to lacking or at least incomplete HSE procedures, not defined use of protective equipment, and non-standardized safe working practices [13].

HSE plans are addressed to the particular wind farm projects and are developed according to the (national) guidelines and regulations of the country the project is located in. Yet, the project must also comply with additional requirements besides the rules and regulations of the country, e.g. maritime rules,

[1] http://soop.offis.de/

construction regulations, port working statutes, or operation-specific guidelines [3]. In many European countries, such as Germany, minimum requirements for workplace safety issues with regard to offshore wind farms have been defined [14]. In the UK, for example, the association RenewableUK — formerly known as BWEA — issued health and safety guidelines focusing on offshore and onshore wind farms. A summary of further UK Health and Safety legislation relevant to wind farm development is given in [15].

In order to install and implement a comprehensive HSE management system in a company, commonly used standards, such as ISO 9001 (Quality Management), ISO 14001 for environmental management, and OHSAS 18001 for occupational health and safety management systems development, are important. A HSE plan constitutes a description of the means of achieving health, safety, and environmental objectives [10], that is it includes the responsibilities, practices, procedures, processes, and resources. Furthermore, a successful HSE management encompasses the following key elements: a clear policy, organization, planning, implementation, measurement of performance, as well as auditing and review [15, 16]. The key elements policy and organization set the overall aims of the HSE management and identify responsibilities within the company. The latter key elements consider how these are put into practice and are continually improved by means of planning and implementation tools.

During the planning and implementation process of HSE management, necessary plans are drawn up and performance standards are set with the overall aim of eliminating and controlling risks [15]. Typical inputs are OHSAS 18002 [16] legal and other requirements concerning HSE, information on best practices, and incidents/accidents having occurred in similar organizations. Furthermore, information on facilities and environmental data of the workplace, processes, routine, and non-routine activities is prepared in process flow-charts, site plans, or working procedures descriptions. The competency requirements and the training needs for the personnel pose another significant task and challenge for HSE management.

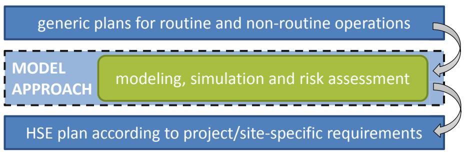

generic plans for routine and non-routine operations
MODEL APPROACH
modeling, simulation and risk assessment
HSE plan according to project/site-specific requirements

Fig. 1. Purpose of model-based approach for HSE planning of offshore operations

Besides these commonly used standards, further specified operation-specific guidelines are provided, for example by NOGEPA [17] for helideck operations or by IMCA with regard to dynamically positioned ships [18] and lifting operations [19]. A guideline by GL [20] specifies the process for qualitative and quantitative risk assessments in offshore wind farms. Still, the processes considered in these guidelines merely facilitate the development of generic plans for example for routine and non-routine lifting operations or the selection of suitable risk assessment methods. Further information has to be gathered for the company's HSE plan in order to meet the site/project-specific requirements and detailed working procedures and instructions have to be prepared [3]. This systematical assessment of the work processes enables the organization to identify hazards and carry out risk assessment. By formulating necessary risk control as well as monitoring measures, it finally leads to relevant operations planning and operational safety. Our model-based approach bridges the gap between generic routine and non-routine plans (e.g. such as guidelines) and the development of project/site-specific HSE plans (cf. Fig. 1) by providing a systematic, model-based approach.

3 Model Based Planning of Offshore Operations

The HSE plan provides a detailed description of work processes, involved actors (e.g. a crane operator), and equipment of the planned offshore operations. Our approach transfers this step for HSE management into model-based planning. The approach is intended to analyze all maritime operations (e.g. bunker, loading or even nautical maneuvers) for which we take generic plans and guidelines for critical routine and non-routine operations as a starting point, enrich them with project/site-specific information and perform a risk assessment to analyze and evaluate risks for personnel and equipment. For the process model, we have derived the main concepts from BPMN [21]. Thus, it provides a model-based planning solution that supports domain experts in the development of the HSE plan for offshore operations or other maritime maneuvers. To describe the behavior of the active entities, a conceptual model for planning offshore operation is described in the following. It represents the behavior of individual agents, the elements of the environment, and its dynamics. Fig. 2 provides a more detailed view of our conceptual model supporting the development of offshore operations. By using the conceptual model, we can represent processes by describing all necessary activities, participants, and events occurring in offshore operations. The respective processes of an operation are structured into so-called lanes in order to map different participants representing different agents interacting in offshore operations (cf. Fig. 2). A *Lane* is the graphical representation of a *Participant* in a *Process* and will extend its entire length.

A *Process* can be composed of different *Flow Objects* and *Connecting Objects* in order to achieve a sequential description (cf. Fig. 2). *Flow Objects* are *Events*, *Gateways*, and *Activities* (including *Participants*). An *Event* affects the sequence or timing of the process flow and usually has a trigger or an effect. There are three types of *Events*, depending on when they affect the flow: *Start Events*, *Intermediate Events*, and *End Events*. An *Activity* is an abstract term for working procedures performed during an Operation. It can either be one *Task* or might be further divided into *Sub-Processes* through a set of sub-activities. A *Task* is an atomic *Activity* that cannot be broken down to a finer level of Process detail. Also, *Tasks* can be further specified into different types. In our model, the set of *Tasks* can be extended, meaning that it can be further extended according to the considered operation. For example, a *Send Task* is designed to send a message from one participant to another or a *Receive Task* is designed to receive a message. Another benefit of the open task definition is the possibility to represent for example cognitive behavior as a *Procedure Task*. By application in a cognitive architecture [22] these tasks can be interpreted to enrich the model through non-normative behavior of humans.

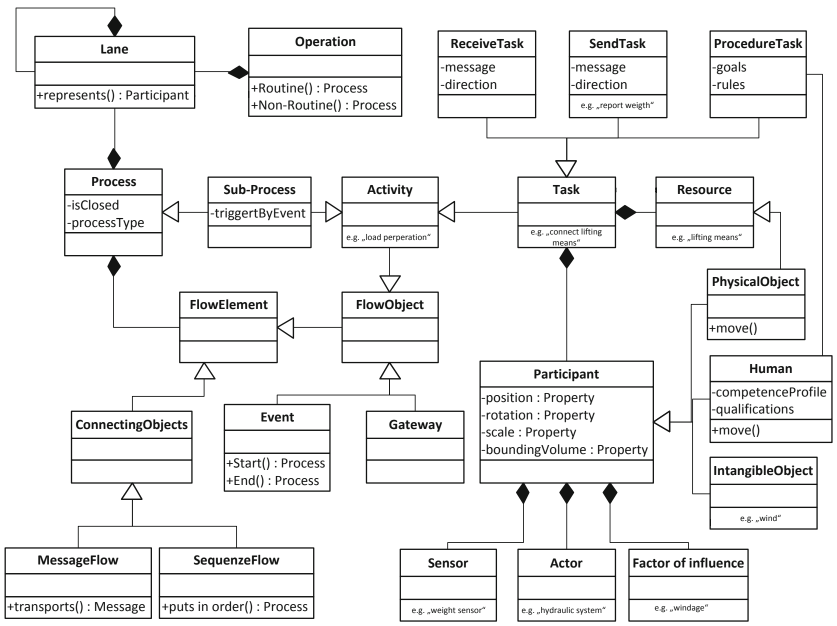


Fig. 2. Conceptual model supporting the description of offshore operations

Participants (cf. Fig. 2) represent specific entities performing certain *Tasks* in a *Process*. Every *Task* has only one *Participant*, a *Send Task* with a sender, and a *Receive Task* with a receiver. A *Participant* has *Sensors* and *Actors*, allowing it to perceive and respond to *Events* in its environment. *Human* participants are abstractions of individual employees, i.e. the personnel involved in a certain offshore operation. Hence, it can be characterized by several aspects: its responsibilities, the required competencies, skills depending on its role, etc. *Physical Objects* include all the equipment involved in a *Process* (e.g. vessels, turbines or cranes). *Intangible Objects* are without a physical manifestation. They describe the behavior of the environment directly related to the flow of a *Process*, such as wind or currents. A *Gateway* is used to control the divergence and convergence of the flow in a process. Thus, it determines branching, forking, merging, and joining of paths. *Connecting Objects* are used in the conceptual model to arrange the process in a sequential order and to include necessary messages for communication. A *Sequence Flow* is used to show the schedule of *Activities* performed in a *Process*. Each *Sequence Flow* has only one source and one target according to the set of *Flow Objects*. A *Message Flow* is used to show the flow of messages between two *Participants* that are prepared to send and receive them.

With the conceptual model, all necessary objects (human, physical, or intangible) and activities can be systematically described in a sequential order. To test and validate the model, it is applied to a specific offshore operation. In order to analyze a scenario described by using our extended process model, it needs to be transformed into a common representation of risk assessment techniques.

4 Model Based Risk Assessment of Offshore Operations

To assess the created process model regarding risks, a risk assessment process has to take place. Oil and gas companies have collected a lot of experience in the offshore sector. We investigated existing concepts currently used in the offshore sector and additionally took the practice of the automotive sector into consideration.

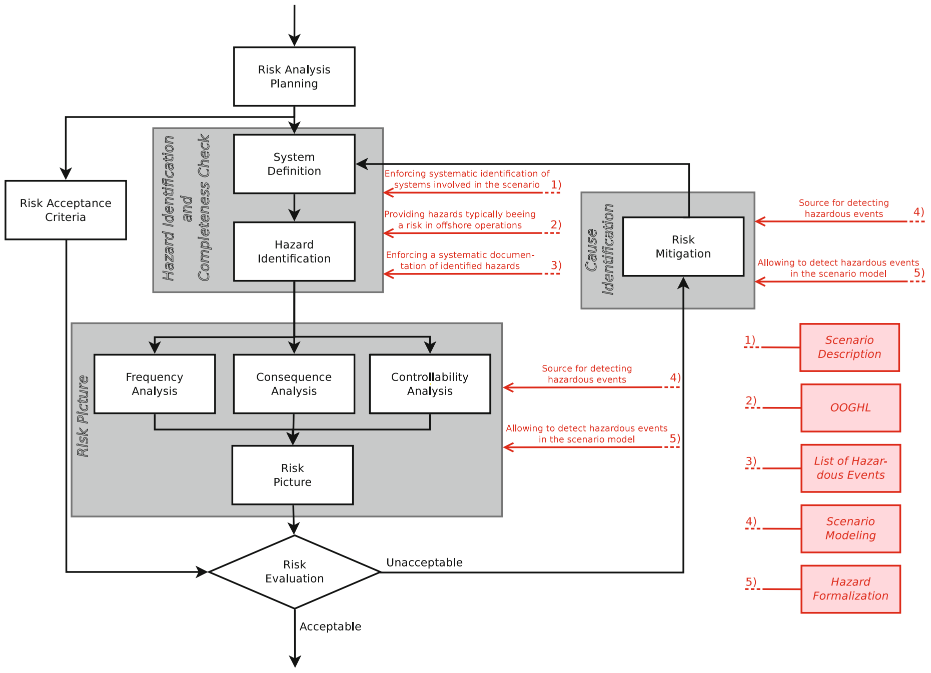

Risk Analysis Planning
Risk Acceptance Criteria
Hazard Identification and Completeness Check
System Definition
Hazard Identification
Enforcing systematic identification of systems involved in the scenario 1)
Providing hazards typically beeing a risk in offshore operations 2)
Enforcing a systematic documentation of identified hazards 3)
Cause Identification
Risk Mitigation
Source for detecting hazardous events 4)
Allowing to detect hazardous events in the scenario model 5)
Risk Picture
Frequency Analysis
Consequence Analysis
Controllability Analysis
Risk Picture
Source for detecting hazardous events 4)
Allowing to detect hazardous events in the scenario model 5)
Risk Evaluation
Unacceptable
Acceptable
1) Scenario Description
2) OOGHL
3) List of Hazardous Events
4) Scenario Modeling
5) Hazard Formalization

Fig. 3. Overview over the risk assessment steps. Enhanced version the QRA approach from [12]

The experience in the offshore sector is mainly derived from oil and gas rigs. Thus, the knowledge cannot directly be applied to offshore wind turbine operations as, although some similarities exist, most of the risks differ substantially. For example, there may be a lot of risks regarding fire and explosion when considering oil and gas rigs, as both of them handle ignitable compounds. Those are not primary risks when talking about offshore wind turbines, neither are blowouts or leakage. Besides these differences, some operations are common between both types of offshore operations. Therefore, Vinnem[12] has been taken into consideration as a source of the current state of practice in risk analysis. In detail, it addresses the steps of QRA, which is frequently applied to offshore operations. Its approach is based on the standards IEC 61508[23] and IEC 61511[24].

A further approach is *Formal Safety Analysis* which is also used for offshore safety assessment[25]. It is based on assigning risks to three levels: intolerable, *As Low As is Reasonably Practicable* (ALARP), and negligible. Risk assigned to the ALARP level are only accepted if it is shown that serious hazards have been identified, associated risks are below a tolerance limit and are reduced “as low as is reasonably practicable”. Because this concept does not rely on quantification and rather uses an argumentative method for assessing risks, it is not suited for usage with our model-based approach.

In addition to this approach, the current automotive standard is of particular interest. This is due to the strong competition between different manufacturers in this industry and the large amount of sold units. As a consequence, the processes in the automotive sector have to be highly time and cost efficient. To achieve a cost efficient process of risk assessment, a specialized approach is being used, defined in ISO 26262 [26].

One of the concepts originating in the automotive sector that is used in our approach are hazardous events. Their usage enables for us to further differentiate hazards by specifying the situations in which they occurs. This allows us to assess the impact of a hazard in a specific operational situation, as the impact might be dependent on it.

The automotive industry also considers controllability as a factor for the risk assessment. Controllability of a hazard reflects the ability to avoid harm or damage by timely reacting to a hazardous event. This could be realized by alerting persons that a risk might emerge, hence they are aware of it and have the possibility to deploy preventive measures.

We use the controllability of a risk as a further assessment factor in our approach, which will support the risk mitigation by introducing measures raising the awareness of a risk, thus allowing the reduction of its consequence. This results in shorter cycles of risk assessment and risk mitigation. The resulting assessment, based on Vinnem[12], weights the consequence of the hazardous event by also considering the frequency (for each independent cause i, thus for the hazardous event) and the controllability:

$$\text{Risk(Hazardous Event)} = \underbrace{\sum_i \left(\text{Probability of independent cause}_i\right)}_{\text{Probability of Hazardous Event}} \times \text{Consequence} \times \text{Controllability}. \quad (1)$$

This risk definition requires a quantification of the consequences as well as of the controllability. As a first step, we can use the same classification procedure as defined in risk assessment standards, such as *Risk Graphs*, *Risk Assessment Table*, or *Risk Matrix*. A further ranking and scaling of these classes (cf. Vinnem[12]) leads to the desired quantification. Again following the automotive approach of the ISO 26262, the controllability can similarly be categorized into classes, thus providing a quantification of the corresponding reduction of the consequence.

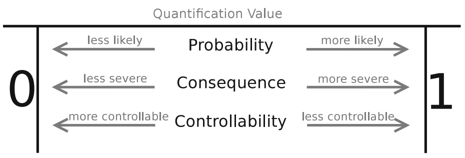
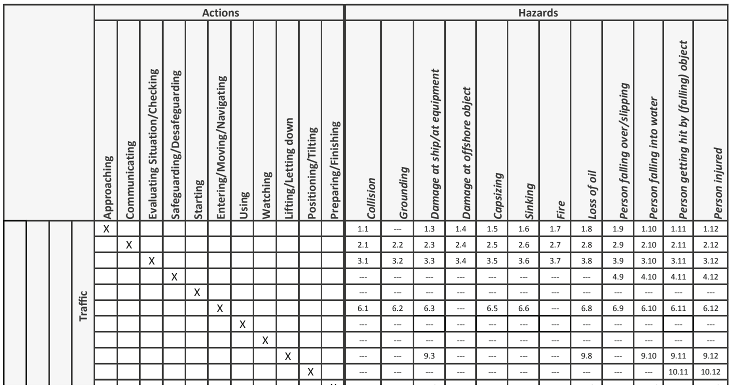


Fig. 4. Our planned quantification

Our modified and extended approach for assessing the risk of an offshore operation is depicted in Fig. 3 of which we introduce every step as well as the supporting methods in the next sections.

4.1 Hazard Identification and Completeness of Identified Hazards

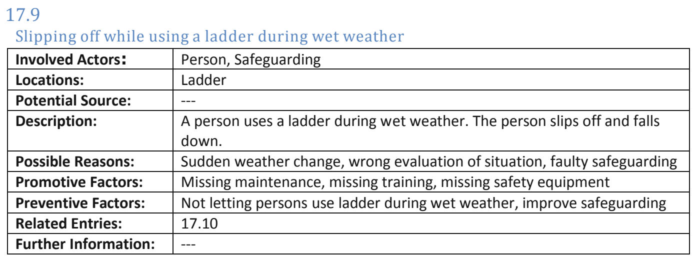
The base for the assessment of risks are the hazardous events. To create this base, it is necessary to identify all possible hazards including the related faults, environmental conditions, and operational situations that constitute a hazardous event. We introduce three steps that result in a list of hazardous events and the corresponding causes.



	Actions											Hazards											
	Approaching	Communicating	Evaluating Situation/Checking	Safeguarding/Desafeguarding	Starting	Entering/Moving/Navigating	Using	Watching	Lifting/Letting down	Positioning/Tilting	Preparing/Finishing	*Collision*	*Grounding*	*Damage at ship/at equipment*	*Damage at offshore object*	*Capsizing*	*Sinking*	*Fire*	*Loss of oil*	*Person falling over/slipping*	*Person falling into water*	*Person getting hit by (falling) object*	*Person injured*
Traffic	X											1.1	---	1.3	1.4	1.5	1.6	1.7	1.8	1.9	1.10	1.11	1.12
		X										2.1	2.2	2.3	2.4	2.5	2.6	2.7	2.8	2.9	2.10	2.11	2.12
			X									3.1	3.2	3.3	3.4	3.5	3.6	3.7	3.8	3.9	3.10	3.11	3.12
				X								---	---	---	---	---	---	---	---	4.9	4.10	4.11	4.12
					X							---	---	---	---	---	---	---	---	---	---	---	---
						X						6.1	6.2	6.3	---	6.5	6.6	---	6.8	6.9	6.10	6.11	6.12
							X					---	---	---	---	---	---	---	---	---	---	---	---
								X				---	---	---	---	---	---	---	---	---	---	---	---
									X			---	---	9.3	---	---	---	---	9.8	---	9.10	9.11	9.12
										X		---	---	---	---	---	---	---	---	---	---	10.11	10.12

Fig. 5. Excerpt from the OOGHL to depict its structure

The first step is obtaining a detailed ***Scenario Description*** out of which possible hazards have to be identified in the next step. The description can be obtained from the previously mentioned process model. With the *Automotive Generic Hazard List* (AGHL) as described by Reuss[27] and Beisel[28], an approach of systematic hazard detection exists. The AGHL is optimized for automotive assistance systems and thus cannot be transfered directly to the offshore sector. The nature and the complexity of interactions and hazards differ in an extensive manner. Because of this, we have developed the ***Offshore Operation Generic Hazard List*** OOGHL that is specifically adjusted to offshore related interactions and hazards. Its data is derived from accident reports (e.g. *Lessons Learned for Seafarers*[29]), guidance documents (e.g. by IMCA [18, 19]), and expert interviews. Combining the scenario description and the OOGHL, a step by step walk trough the scenario is possible. To perform a hazard lookup systematically, the OOGHL comprises of all possible actions that might be part of an offshore operation. It also contains points of interaction (e.g. other traffic, fixed installations, or other resources) and the potential hazards that might occur in the analyzed scenario. The structure of the OOGHL is depicted in Fig. 5. For each combination of the possible actions and all feasible points of interaction (indicated by an `X`) a list of references to potential hazards is given in the corresponding row. An example for a referenced entry is visible in Fig. 6. Using the OOGHL to identify possible hazards has the advantage that only one source has to be taken into consideration, although we cannot guarantee neither consistency nor completeness as it is based on empirically derived data. But using it leads to less expense in processing hazard descriptions and therefore takes less time than traditional approaches that consider several sources. Additionally, the OOGHL is a more systematic approach in detecting potential hazards, thus leading to a more complete list of the ones that might occur during a scenario.



17.9

Slipping off while using a ladder during wet weather

Involved Actors:	Person, Safeguarding
Locations:	Ladder
Potential Source:	---
Description:	A person uses a ladder during wet weather. The person slips off and falls down.
Possible Reasons:	Sudden weather change, wrong evaluation of situation, faulty safeguarding
Promotive Factors:	Missing maintenance, missing training, missing safety equipment
Preventive Factors:	Not letting persons use ladder during wet weather, improve safeguarding
Related Entries:	17.10
Further Information:	---

Fig. 6. Example entry of the OOGHL

To assess the risks associated with the identified hazards, the hazards have to be documented. This can be achieved by creating a ***List of Hazardous Events*** that comprises of all hazards. We extend the list with Hazardous Events as well as with all their dependencies and the dependencies for the hazards. To distinguish between the entries, we mark the type of every entry of the table.

This list consisting of all events and conditions allows us to create a dependency structure for each hazardous event by using the causes of each event. Thus, a fault tree can automatically be generated. A similar approach is described by Peikenkamp[30]. Further to this, the dependency structure can be used to formalize the hazardous events listed in the table for analyzing the scenario, as we will demonstrate in the next section.

4.2 Risk Picture

After all potential hazards of the scenario are defined, a risk picture can be created. This can be achieved by modeling the scenario as well as to formalize the hazardous events to analyze their occurrence. During the creation of the risk picture, the risks associated with the hazardous events are assessed by evaluating the frequency, consequence, and controllability. This can be realized by investigating the underlying causes for a hazardous event regarding their frequency of occurrence. Additionally, the hazardous events themselves have to be assessed regarding their consequences (i.e. harm caused to humans and to the environment) and their controllability (i.e. if the event can be controlled if it occurs, thus mitigating its impact). Beforehand, a risk acceptance value has to be defined which represents the maximum quantified risk value that is tolerable.

Using a model of the scenario, it can be checked if it is possible to reach a hazardous event. How such a ***Scenario Model*** can be obtained is described in section 5. To automatically check whether a hazardous event might occur, a model checker or simulation of the model can be used. For this, an observer can be utilized to check if a state that might constitute a hazardous event has been reached. One way to create such an observer is to use a ***Formalization of Hazardous Events*** describing the hazardous events from the List of Hazardous Events. This can be realized by using a formal language. As we are currently developing a specialized language, we cannot yet give concrete examples for the formalization process. The formalization allows assessing each state of the simulation regarding if a hazardous event has occurred. By this means, an automatic detection of all hazardous events happening in a modeled scenario is possible.

The dependency structure of the List of Hazardous Events allows to use the list as a source for formalizing a hazardous event. Resolving the dependencies in the List of Hazardous Events, a formula can be developed stepwise. An example for this can be found in section 5. The formula also contains the faults and environmental conditions that are necessary for the hazardous event to occur. This list of faults can be used as a source for the causes that might lead to the hazardous event. Because of our model-based approach they can be injected into the model to trigger a hazardous event. Of course, the possibility for injecting faults requires the model to be prepared. A detailed methodology for fault injection and model checking has been developed in the ESACS and ISAAC projects (cf. [30, 31]) and will not be discussed further in this paper.

After having detected all possible causes of a hazardous event and assessing the frequency, consequence, and controllability, a quantification of the risk for each hazardous event exists (cf. equation 1). If the quantified risk value is higher than the risk acceptance value risk mitigation measures need to be developed that reduce the actual risk to an acceptable level.

4.3 Risk Mitigation

In order to minimize the risk of an offshore scenario, risk mitigation measures for hazardous events that have a risk quantification value higher than the acceptable risk have to take place. This can be realized by developing measures to prevent certain faults, thus lowering the probability of occurrence of a hazardous event. Another way to minimize the risk is to raise the controllability of the hazardous event. To reach this, the awareness of potential hazards has to be raised so that proper reaction to the hazardous event can happen. A third option is to minimize consequence on a hazardous event if it occurs. To minimize the risk in the example scenario, possible causes for risks can be excluded (e.g. not allowing an operation during particular weather conditions). Another way is to add additional safety measures.

The approach and its risk assessment process is demonstrated by an example in the next section.

5 Application to an Example Scenario

To demonstrate our approach, we use an example to apply and verify our conceptual model approach on an example offshore scenario. The application scenario includes a jack-up vessel performing a crane operation to lift a quadpod, a foundation type used for offshore wind turbines, from the vessel to its final installation position in the sea. In the scenario, the crane operation begins after the jack-up barge has been positioned and is lifted up to create a stable operational platform. Before the lifting equipment and crane are prepared, it has to be confirmed that the personnel involved (e.g. lift supervisor, banksman, and crane operator) is physically able for and has understood the intended lifting activity as well as their respective roles. Besides, personnel should be equipped with its protective equipment. The process step "cargo preparation and cargo lifting" is described in further detail in Fig. 7.

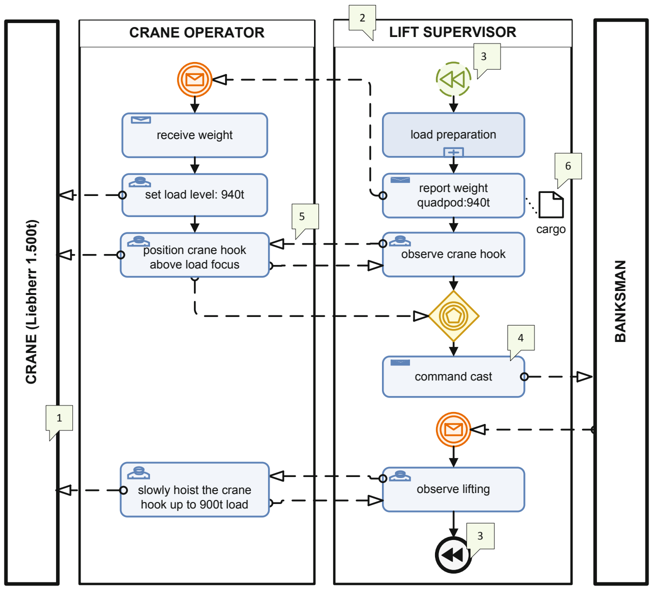


Fig. 7. Extract of the process diagram of a lifting operation

Fig. 7 shows the cooperative activities of the process step "cargo preparation and cargo lifting" involving four participants represented in four different lanes. It allows representing the crane as a Physical Object (Fig. 7: 1) and three Human Participants taking part in the process (Fig. 7: 2). The process step starts and ends with Events (Fig. 7: 3); Sequence Flows facilitate a scheduling of the different Activities performed by the Participants. As described in the conceptual model, different types of activities can be further distinguished. Send and Receive Tasks (Fig. 7: 4) (e.g. report weight or warn personnel) initiate communication/interaction between different Participants. Message Flows

(Fig. 7: 5) describe the destination of messages in order to synchronize the process activities. Other Physical Objects used as Resources (e.g. cargo) can be associated to an Activity (Fig. 7: 6).

The process diagram provides a detailed description of the cooperative activities, participants, and communication. As in other guidelines [19], these essential components and steps can support the development of HSE plans. Due to the underlying profound conceptual model, the diagram can be further enriched with site and project-specific data. The modeling and notation of the process steps allow the transformation into a common representation for the risk assessment.

To perform a model-based risk assessment of a scenario defined by this process model, the process model has to be converted into a model type that enables analysis regarding risks. Because the behavior of the involved actors the model is highly dynamic, we decided to use a graph transformation model which allows us to reflect this behavior in a way that is not possible when using, for instance, finite state machines. Graph transformation gives us the possibility to dynamically add or remove actors and enable changing the way of interaction and the relations between actors during execution, as well as to add or remove actors.

We selected GROOVE[2] as a graph transformation tool. It has the advantage that it is open source software and thus can be extended to fit our custom needs. Additionally, the used data format is XML based and therefore can be generated automatically.

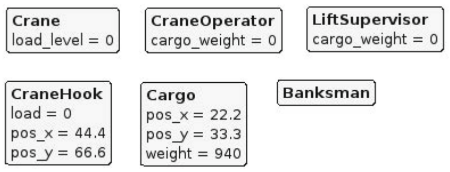


Fig. 8. Start graph of a GROOVE model

A graph transformation model consists of a start graph that represents the start situation of the scenario. Additionally, there are rules that match the current state of the graph and transforms the graph. It is possible to add, remove, or modify edges or nodes by applying the rules. Combining the start graph and the rules it is possible to automatically generate the state space of the scenario and to perform queries to check whether a specific graph configuration can be reached. Rensink [32] gives an overview about how graph transformation works and what its advantages are. See also Kastenberg[33] for the formal aspects of graph transformation techniques.

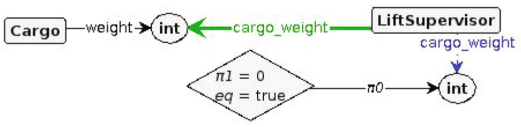


Fig. 9. Example of a graph transformation rule

The challenge is to convert the process model into a graph transformation model and the corresponding rules. An example for this conversion can be seen in Fig. 8. In this start graph, the lanes from the process model and all their points of interaction have been created as nodes. They are attributed by the parameters that change during the execution of the described scenario.

[2] http://groove.cs.utwente.nl/

The interactions of the process model are described by graph transformation rules. One of these rules can be seen in Fig. 9. The rule describes the process of the cargo preparation of the lift supervisor who reads the weight of the cargo (if he has no knowledge about the cargo weight) and thus knows the correct weight.

With this converted model it is possible to check if a hazardous state can be reached. For this, faults have to be injected to simulate wrong behavior of humans, machines, or materials. As a preparation for this, hazards have been analyzed as described in section 4. The resulting list contains the possible events that might lead to a hazardous event. Fig. 10 outlines a possible fault tree generated from the list. With having this information, it is possible for us to add possible faults to the model to support fault injection. This can be done by introducing additional rules that represent the dysfunctional behavior or by annotating the transition rules with possible incorrect behavior. Of course, for the latter the simulation environment has to be adjusted to support annotations of this kind.

Using the modified model, we are able to formalize the hazardous event as described in section 4. In the following we sketch how this formalization can be used to check the reachability of hazardous states. More specifically, Linear Temporal Logic (LTL, cf. [34]) can be used as formalization language. For illustration purposes, (part of) the LTL formula checking the occurence of “Person Injured by Cargo” (cf. Fig. 10) can be expanded as follows:

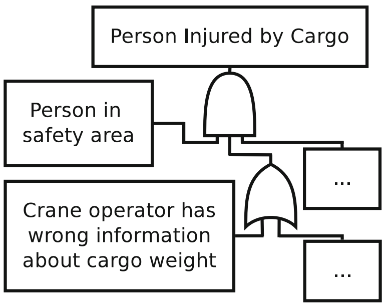


Fig. 10. Example of a fault tree generated from the List of Hazardous Events

$$\pi^i \models \text{Person injured by cargo}$$
$$\pi^i \models F(\text{Person in safety area} \wedge O(\text{Cargo wrongly lifted} \vee \ldots))$$
$$\pi^i \models F(\text{Person in safety area} \wedge O((\text{Wrong information about cargo weight} \vee \ldots) \vee \ldots))$$

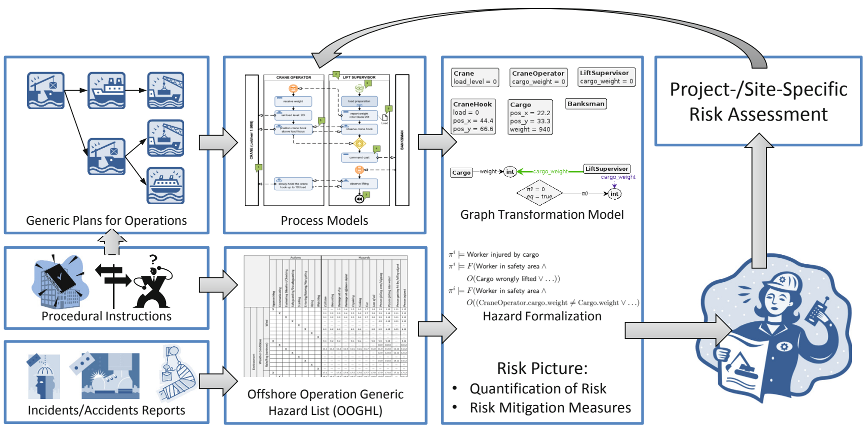

Generic Plans for Operations
Process Models
CRANE OPERATOR
LIFT SUPERVISOR
BANKSMAN
Crane load_level = 0
CraneOperator cargo_weight = 0
LiftSupervisor cargo_weight = 0
CraneHook load = 0 pos_x = 44.4 pos_y = 66.6
Cargo pos_x = 22.2 pos_y = 33.3 weight = 940
Banksman
Cargo
weight
int
cargo_weight
LiftSupervisor
Graph Transformation Model
Project-/Site-Specific Risk Assessment
Procedural Instructions
π^i ⊨ Worker injured by cargo
π^i ⊨ F(Worker in safety area ∧ O(Cargo wrongly lifted ∨ ...))
π^i ⊨ F(Worker in safety area ∧ O((CraneOperator.cargo.weight ≠ Cargo.weight ∨ ...)
Hazard Formalization
Incidents/Accidents Reports
Offshore Operation Generic Hazard List (OOGHL)
Risk Picture:
• Quantification of Risk
• Risk Mitigation Measures

Fig. 11. Schematic illustration of our approach

If these states are reachable, the risk represented by the simulation trace has to be analyzed. That is to say, the possible faults that occur have to be evaluated with respect to their frequency. These factors have to be accumulated and further the consequence and the controllability for the hazardous event have to be assessed. Multiplying these factors we receive a quantitative assessment for the hazardous event. If the value is too high, mitigation measures have to be developed as described in section 4.3.

6 Conclusion and Outlook

In this paper, we presented a model-based risk assessment approach, supporting the development of HSE plans for offshore operations. Thus, our approach allows bridging the gap between generic operation plans and the project/site-specific risk assessment for HSE development (cf. Fig. 11). Current HSE regulations, guidelines, and incident/accident reports have been taken into account to develop a systematic assessment of the processes, hazards, and risks during offshore operations. Different model types, notations, the concept of an Offshore Operation Generic Hazard List (OOGHL), and a formalization of hazardous events have been introduced in order to integrate all aspects of these complex and safety-critical operations, and evaluate risks for personnel and equipment.

By performing a quantitative assessment — including the frequency, severity, and controllability of hazardous events — a quantification of risks can be created and necessary risk mitigation measures can be derived to improve the operational safety during offshore operations. In an example scenario, the feasibility of our approach has been demonstrated.

Our approach can be further extended to support the analysis of other maritime operations or even nautical maneuvers by adding generic plans and guidelines and enriching them with project/site-specific information to concluding perform risk assessment.

References

1. European Union: Directive 2009/28/EC of the European Parliament and of the Council of 23 April 2009 on the promotion of the use of energy from renewable sources and amending and subsequently repealing Directives 2001/77/EC and 2003/30/EC (2009)
2. Bundesamt für Seeschifffahrt und Hydrographie: Genehmigung von Offshore Windenergieparks (2012), `http://www.bsh.de/de/Meeresnutzung/Wirtschaft/Windparks/index.jsp`
3. Thomsen, K.E.: Offshore Wind: A Comprehensive Guide to Successful Offshore Wind Farm Installation. Elsevier Science & Technology, Amsterdam (2012)
4. NDR online: Retter setzen Suche in der Nordsee fort, January 28 (2012) `http://www.ndr.de/regional/niedersachsen/oldenburg/arbeitsunfall103.html`
5. Brandt, J.: Stundenlange Suche nach dem Vermissten auf See. Ostfriesen-Zeitung, January 26 (2012)

6. Vertikal.net: Fatal accident in Harwich, May 21 (2010), http://www.vertikal.net/en/news/story/10145/
7. Stromsta, K.E.: Firms fined for grisly accident at Scottish offshore project, September 28, (2010), http://www.rechargenews.com/energy/wind/article230665.ece
8. Lenk, J.C., Droste, R., Sobiech, C., Lüdtke, A., Hahn, A.: Towards Cooperative Cognitive Models in Multi-Agent Systems. In: International Conference on Advanced Cognitive Technologies and Applications (2012)
9. Wehs, T., Janssen, M., Koch, C., von Cölln, G.: System Architecture for Data Communication and Localization under Harsh Environmental Conditions in Maritime Automation. In: IEEE 10th International Conference on Industrial Informatics (2012)
10. E&P Forum: Guidelines for the Development and Application of Health, Safety and Environmental Management Systems, London, E&P Forum (1994)
11. OGP: HSE management - guidelines for working together in a contract environment, London, Int. Assoc. of Oil & Gas Producers (2000)
12. Vinnem, J.E.: Offshore Risk Assessment: Principles, Modelling and Applications of QRA Studies, 2nd edn. Springer, London (2007)
13. Tveiten, C.K., Albrechtsen, E., Heggset, J., Hofmann, M., Jersin, E., Leira, B., Norddal, P.K.: HSE challenges related to offshore renewable energy. Volume Report A18107, Trondheim, SINTEF Technology & Society (2011)
14. Deutscher Bundestag: Keine zusätzlichen Sicherheitsanforderungen bei Offshore-Windanlagen, Berlin, PuK 2 Parlamentskorrespondenz, Drucksache 17/5441 (2011)
15. RenewableUK: Guidelines for Onshore and Offshore Wind Farms - Health & Safety in the Wind Energy Industry Sector, London, RenewableUK (2010)
16. BSI: Occupational Health and Safety Management Systems: Guidelines for the Implementation of OHSAS 18001, London, British Standards Institution (2000)
17. NOGEPA: Helideck Operations and Procedures Manual, 's-Gravenhage, NOGEPA (2011)
18. IMCA: FMEA Management Guide, London, International Marine Contractors Association (2005), M 178
19. IMCA: Guidelines for Lifting Operations, London, International Marine Contractors Association (2007), SEL 019
20. GL: Richtlinie zur Erstellung von technischen Risikoanalysen für Offshore-Windparks, Hamburg, Germanischer Lloyd (2002)
21. OMG: Business Process Model and Notation (BPMN) Version 2.0. OMG (2011)
22. Lüdtke, A., Weber, L., Osterloh, J.P., Wortelen, B.: Modeling Pilot and Driver Behavior for Human Error Simulation. In: Duffy, V.G. (ed.) ICDHM 2009. LNCS, vol. 5620, pp. 403–412. Springer, Heidelberg (2009)
23. International Electrotechnical Commission: IEC 61508 (2010)
24. International Electrotechnical Commission: IEC 61511 (2003)
25. Wang, J.: Offshore safety case approach and formal safety assessment of ships. Journal of Safety Research 33(1), 81–115 (2002)
26. International Organization for Standardization: ISO/DIS 26262 - Road vehicles Functional safety (2011)
27. Reuß, C.: Automotive Generic Hazard List. Technische Universität Braunschweig (2009)
28. Beisel, D., Reuß, C., Schnieder, E.: Approach of an Automotive Generic Hazard List. In: Proceedings of European Safety and Reliability, ESREL (2010)

29. IMO: Lessons Learned from Casualties for Presentation to Seafarers (2004/2006/2010), `http://www.imo.org/blast/mainframe.asp?topic_id=800`
30. Peikenkamp, T., Cavallo, A., Valacca, L., Böde, E., Pretzer, M., Hahn, E.M.: Towards a Unified Model-Based Safety Assessment. In: Górski, J. (ed.) SAFECOMP 2006. LNCS, vol. 4166, pp. 275–288. Springer, Heidelberg (2006)
31. Åkerlund, O., et al.: ISAAC, a framework for integrated safety analyses of functional, geometrical and human aspects. ERTS (2006)
32. Rensink, A.: The Joys of Graph Transformation. Nieuwsbrief van de Nederlandse Vereniging voor Theoretische Informatica 9 (2005)
33. Kastenberg, H.: Graph-based software specification and verification. PhD thesis, University of Twente, Enschede (2008)
34. Latvala, T., Biere, A., Heljanko, K., Junttila, T.: Simple Is Better: Efficient Bounded Model Checking for Past LTL. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 380–395. Springer, Heidelberg (2005)

Modular Automated Verification of Flexible Manufacturing Systems with Metric Temporal Logic and Non-Standard Analysis

Luca Ferrucci, Dino Mandrioli, Angelo Morzenti, and Matteo Rossi

Politecnico di Milano, Dipartimento di Elettronica e Informazione,
Piazza Leonardo da Vinci 32, – 20133 Milano, Italy
{ferrucci,mandrioli,morzenti,rossi}@elet.polimi.it

Abstract. Industrial systems are made of interacting components, which evolve at very different speeds. This is often dealt with in notations used in the industrial practice, such as Stateflow, through the notion of "zero-time transitions". These have several drawbacks, especially when building complex models from basic components, whose coordination is complicated by the fact that each element is modeled to be in different states at the same time. We exploit a temporal logic formalism based on non-standard analysis to provide a natural formal semantics to the composition of modules described as Stateflow diagrams. The semantics has been implemented in a fully automated formal verification tool, which we apply to the formal verification of an example of robotic cell.

Keywords: metric temporal logic, formal verification, flexible manufacturing systems, micro- and macro-steps, non-standard analysis.

1 Introduction

Formal models to describe and verify Flexible Manufacturing Systems (FMS) are gaining widespread use. Many of them are based on the classical Stateflow notation, which is inspired from the Statecharts formalism [10]. Two distinguishing features of these formalisms are: (i) rather subtle and therefore difficult to precisely model temporal behavior and, (ii) the need to structure complex systems as the composition of several interacting modules. Not surprisingly, the difficulties of type (i) have a kind of "multiplying effect" with those of type (ii). Despite an abundant literature on Stateflow, Statecharts, Petri nets and other notations and the various ways to give them a fully formal semantics, some concerns mainly related with the two aspects above still remain. One of them refers to the way evolutions that occur at very different time scales, say milliseconds and seconds, are formalized. Quite often the problem is solved by introducing two different classes of transitions: *micro-steps* take – formally – no time to change the system state – therefore they are also called *zero-time transitions* –, whereas the only transitions that take a non-null time to complete are called *macro-steps* (and often are paired with a discrete time domain, say the naturals).

The notion of zero-time transition is a useful abstraction, but it entails some risks from the point of view of naturalness of modeling. In fact, a major consequence is that a system can be in different states at the same time, which is counterintuitive from the standpoint of the traditional dynamical systems view; this creates the risk of contradictory assertions about timing properties of the system.

In the past we have proposed a natural way to overcome this difficulty by exploiting non-standard analysis [7], where the domain of any variable is extended by introducing non-standard numbers, which, roughly speaking, include infinitesimal quantities. We exploited this idea by formalizing zero-time transitions of Petri nets in terms of transitions that take a non-null, infinitesimal time in the context of our metric temporal logic language TRIO [4]. In a companion paper [6] we applied this approach to formalize micro- and macro-steps in timed systems by introducing in TRIO the next-time operator typical of various temporal logics: the new state entered after a transition is at a time distance that can be a standard positive number, in the case of a macro-step, or a non-standard, infinitesimal one, in the case of a micro-step. This extension of TRIO, called X-TRIO, allowed us to describe in a natural way the formal semantics of the execution of a module, modeled as a state machine, in notations that are widely used in the industrial practice, such as the Stateflow notation.

The present paper further extends the approach by addressing the formalization of the semantics of a system modeled by several state machines that evolve independently and asynchronously through a sequence of micro-steps and synchronize by exchanging signals and data at each macro-step. The subtle semantic issues involved in the synchronization of such modules are addressed in a general and flexible way by means of suitable X-TRIO axioms.

Since the introduction of Statechart several different semantics have been defined for it. The three most classical ones, the fixpoint [13], STATEMATE [10], and UML [12] semantics, differ in the features adopted for step execution, and have been fully analyzed in [5]. In the present work we focus on Stateflow because of its widespread use in industrial settings. In [3] the authors introduce the "shallow synchronized" semantics to address the reachability problem for a network of Linear Hybrid Automata: the automata proceed autonomously, unless they perform a synchronizing transition, in which case they realign their absolute time. In [8] the authors give an operational semantics to a subset of Stateflow for efficiently compiling it into an input language of a model checker. With respect to [3,5,8], our approach is more general, flexible, and purely descriptive. As hinted at the end of Section 3.2, it can be adjusted to any of the semantics defined for Statecharts or other state-based formalisms that use the abstraction of micro- and macro-steps. In the companion paper [6] we give more references to other micro- and macro-step based formalisms, such as those based on Duration Calculus or super dense time, or papers that are only partially connected to ours, as for example related to different time scales or granularity.

Besides naturalness and generality, however, we pursue the goal of providing practitioners with fully automated tools supporting the analysis of the modeled systems. This is possible because a decidable fragment of the X-TRIO logic, one that is expressive enough to fully capture the semantics of the target notation, is translated into the Propositional Linear Temporal Logic with Both future and past operators (PLTLB) that is amenable to automated analysis by existing tools such as $\mathbb{Z}$ot [1].

The paper is structured as follows. In Section 2 we briefly introduce the X-TRIO logic. Then, in Section 3 we use X-TRIO to model the composition of Stateflow diagrams, and we show how the logic model can be exploited to perform automated verification of real-life industrial systems, using a Flexible Manufacturing System as example. Section 4 concludes and hints at possible extensions and enhancements of this work.

2 Syntax and Semantics of X-TRIO

In this section we briefly present the X-TRIO logic, with some necessary background about non-standard analysis. X-TRIO, and its X-TRIO$_{\mathbb{N}}$ subset that we exploit in this work to formalize the semantics of the composition of Stateflow diagrams, have been introduced in a companion work [6]. Here, we recall the main features of the logic.

The original TRIO language [4] is a general-purpose specification language suitable for modeling real-time systems. It is a temporal logic supporting a metric on time. TRIO formulae are built out of the usual first-order connectives, operators, and quantifiers, and the single basic modal operator, Dist: for any formula ϕ and term t indicating a time distance, the formula $\mathrm{Dist}(\phi, t)$ specifies that ϕ holds at a time instant whose distance is exactly t time units from the current instant. TRIO formulae can be interpreted both in discrete and dense time domains.

X-TRIO extends TRIO along two main lines. First, the temporal domain T is augmented with non standard numbers. Non-standard numbers formalize the concept of infinitesimal numbers within the theory of Non-Standard Analysis (NSA) founded by A. Robinson [14], which has already been exploited in TRIO [7] to deal with time-critical systems. Intuitively, for any numerical domain D, ϵ is infinitesimal in D if $\epsilon \geq 0$ and ϵ is smaller than any number in $D_{>0}$. The original values of D are classified as *standard* and are characterized by predicate st; that is, x is standard iff $st(x)$ holds. D is augmented with infinitesimal numbers and all numbers resulting from adding and subtracting infinitesimal non-zero numbers to and from standard ones. Predicate $ns(x)$ denotes that x is *non-standard*; for each x, $st(x)$ holds if and only if $ns(x)$ does not hold. Notice that 0 is the only infinitesimal standard number and that non-standard numbers are of the form $v \pm \epsilon$, where $st(v)$ holds, and ϵ is infinitesimal greater than 0. NSA provides an axiomatization that allows one to apply all arithmetic operations and properties of traditional analysis in an intuitive way: for instance the sum of two standard numbers is standard, the sum of two infinitesimal numbers is an infinitesimal and the sum of an infinitesimal with a standard number is a non-standard number. The theory of NSA introduced also the notion of infinite numbers, plus a rich set of results that make it an appealing framework for

Table 1. X-TRIO$_{\mathbb{N}}$ syntax

$$\phi \;:= p \,|\, \neg\phi \,|\, \phi_1 \wedge \phi_2 \,|\, \mathrm{Dist}(\phi, 1) \;|\, \mathrm{Dist}(\phi, -1) \;|\, \mathrm{Dist}(\phi, \epsilon)$$
$$\mathrm{Until}(\phi_1, \phi_2) \;|\, \mathrm{Since}(\phi_1, \phi_2) \;|\, \mathrm{X_{st}}(\phi) \;|\, \mathrm{X_{ns}}(\phi) \;|\, \mathrm{NowST}$$

reasoning on. In this paper we do not use the full power of NSA, for example we do not deal with the above mentioned infinite numbers. We denote as $\overline{T}$ the extension of the temporal domain T with non-standard infinitesimal numbers.

The second major novelty of X-TRIO is the introduction of the *next* operator X which is typical to describe the evolution of dynamical systems as a sequence of discrete steps. Unlike the traditional use of the operator in a metric setting, however, the time distance between two consecutive states is not implicitly assumed as a time unit; on the contrary it can be any standard or non-standard positive number. More precisely, we introduce two different types of X operator, namely $\mathrm{X_{st}}$ and $\mathrm{X_{ns}}$. Intuitively, formula $\mathrm{X_{st}}(\phi)$ is true in the current instant iff ϕ is true in the next state entered by the system and this occurs at a time instant that is a standard number; conversely, formula $\mathrm{X_{ns}}(\phi)$ is true iff in the next state ϕ is true and the time of occurrence is a non-standard number. We will use these two operators to distinguish between two typical ways of modeling system evolution: $\mathrm{X_{st}}$ will formalize *macro-steps* i.e. transitions that "consume real, tangible time", whereas $\mathrm{X_{ns}}$ will describe *micro-steps* which are often formalized as zero-time transitions.

In this paper we use a decidable fragment of the X-TRIO logic, one that is expressive enough to formalize the semantics of composition for Stateflow diagrams. This fragment, which we call X-TRIO$_{\mathbb{N}}$, uses natural numbers for time domain (i.e., $T = \mathbb{N}$), and corresponds to the syntax of Table 1.

In X-TRIO$_{\mathbb{N}}$, one can also write $\mathrm{Dist}(\phi, 1 + \epsilon)$, which is an abbreviation for $\mathrm{Dist}(\mathrm{Dist}(\phi, \epsilon), 1)$, while $\mathrm{Dist}(\phi, 2\epsilon)$ is an abbreviation for $\mathrm{Dist}(\mathrm{Dist}(\phi, \epsilon), \epsilon)$ ($\mathrm{Dist}(\phi, k\epsilon)$ is an easy generalization of the schemata above). This highlights that, in X-TRIO$_{\mathbb{N}}$, we consider numbers of $\overline{\mathbb{N}}$ to have the form $v + k\epsilon$, where $v, k \in \mathbb{N}$ and ϵ is an infinitesimal constant fixed *a priori*. In other words, standard numbers are identified by the coefficient $k = 0$, and infinitesimal numbers are multiples of the infinitesimal unit ϵ. To distinguish between standard and non-standard instants, X-TRIO$_{\mathbb{N}}$ introduces the operator NowST such that $S, i \models \mathrm{NowST}$ iff $st(i)$. We briefly discuss in Section 3.1 how this impacts our conceptual model of composition of Stateflow diagrams.

X-TRIO$_{\mathbb{N}}$ introduces also typical derived temporal operators such as "sometimes" ($\mathrm{SomF}(\phi) = \mathrm{Until}(\top, \phi)$) and "always" ($\mathrm{AlwF}(\phi) = \neg\mathrm{SomF}(\neg\phi)$).

A model-theoretic semantics for X-TRIO is defined by following a fairly standard path on the basis of a temporal structure $S = \langle \overline{T}, \beta, \sigma \rangle$, where:

- $\overline{T}$ is the time domain such that $\forall t \in \overline{T}$ it is $t \geq 0$.
- $\beta : \overline{T} \longmapsto 2^{AP}$ is an interpretation function that associates each instant of time t with the set of atomic propositions $\beta(t)$ that are true in t.
- $\sigma = \{\sigma_i | i \in \mathbb{N} : \sigma_i \in \overline{T} \wedge \sigma_0 = 0 \wedge \forall j \in \mathbb{N}(j < i \Rightarrow \sigma_j < \sigma_i) \wedge \forall t \in \overline{T}(\sigma_i < t < \sigma_{i+1} \Rightarrow \beta(\sigma_i) = \beta(t))\}$ is the distinguishing element of the

Table 2. Satisfaction relation for X-TRIO$_{\mathbb{N}}$

$S, i \models p$ iff $p \in \beta(i)$

$S, i \models \neg\phi$ iff $S, i \not\models \phi$

$S, i \models \phi_1 \wedge \phi_2$ iff $S, i \models \phi_1$ and $S, i \models \phi_2$

$S, i \models \mathrm{Dist}(\phi, 1)$ iff $S, i + 1 \models \phi$

$S, i \models \mathrm{Dist}(\phi, -1)$ iff $i - 1 \geq 0$ and $S, i - 1 \models \phi$

$S, i \models \mathrm{Dist}(\phi, \epsilon)$ iff $S, i + \epsilon \models \phi$

$S, i \models \mathrm{Until}(\phi, \psi)$ iff $\exists j \geq i$ s.t. $S, j \models \psi$ and $\forall l$ s.t. $i \leq l < j$ it is $S, l \models \phi$

$S, i \models \mathrm{Since}(\phi, \psi)$ iff $\exists j$, with $0 \leq j \leq i$, s.t. $S, j \models \psi$ and $\forall l$ s.t. $j < l \leq i$ it is $S, l \models \phi$

$S, i \models \mathrm{X}_{\mathrm{st}}(\phi)$ iff there is $j \in \mathbb{N}$ s.t. $\sigma_j \leq i < \sigma_{j+1}, st(\sigma_{j+1})$ and $S, \sigma_{j+1} \models \phi$

$S, i \models \mathrm{X}_{\mathrm{ns}}(\phi)$ iff there is $j \in \mathbb{N}$ s.t. $\sigma_j \leq i < \sigma_{j+1}, ns(\sigma_{j+1})$ and $S, \sigma_{j+1} \models \phi$

X-TRIO temporal structure; it is a (possibly infinite) sequence of time instants starting from the initial instant 0, called *History*. Intuitively, it represents the discrete sequence of instants when the system changes state; thus, the X operator represents a step moving from σ_i to σ_{i+1}.

Let us point out some features of sequence σ, which will be exploited in Section 3.2 to provide an elegant approach to the problem of synchronizing components that can make different numbers of micro-steps in the same macro-step. Given two elements, σ_i and σ_{i+1} of the history σ, if σ_{i+1} is a nonstandard number (i.e., $ns(\sigma_{i+1})$), then the distance between σ_i and σ_{i+1} is the infinitesimal ϵ (i.e., $\sigma_{i+1} = \sigma_i + \epsilon$). If, on the other hand, σ_{i+1} is standard ($st(\sigma_{i+1})$), then between σ_i and σ_{i+1} there is an infinite sequence of nonstandard numbers $\sigma_i + \epsilon, \sigma_i + 2\epsilon, \ldots$ such that, for all $k \in \mathbb{N}$, $\beta(\sigma_i + k\epsilon) = \beta(\sigma_i)$.

The satisfaction relation $\models$ of an X-TRIO$_{\mathbb{N}}$ formula ϕ w.r.t. a structure $S = \langle \overline{T}, \beta, \sigma \rangle$ at a time instant $i \in \overline{T}$ is defined as in Table 2. A formula ϕ is *satisfiable* in a structure $S = \langle \overline{T}, \beta, \sigma \rangle$ when $S, 0 \models \phi$.

Despite its limited syntax and the restriction of the time domain to $\mathbb{N}$, X-TRIO$_{\mathbb{N}}$ is undecidable [6]. For our purposes, however, we do not need its full expressive power. More precisely, if we restrict formulae of the form $\mathrm{Dist}(\phi, 1)$ and $\mathrm{Dist}(\phi, -1)$ to be evaluated only at standard instants, the logic becomes decidable, and effective decision procedures can be defined for it. Occurrences of $\mathrm{Dist}(\phi, 1)$ (and, in general, of $\mathrm{Dist}(\phi, v + k\epsilon)$, with $v \geq 1$) in X-TRIO$_{\mathbb{N}}$ formulae are to be read as abbreviations for $\mathrm{Dist}(\phi, 1) \wedge \mathrm{NowST}$. This is sufficient to achieve decidability of X-TRIO$_{\mathbb{N}}$ [6].

3 Using X-TRIO$_{\mathbb{N}}$ to Model and Analyze the Composition of Stateflow Diagrams

In this section we exploit X-TRIO$_{\mathbb{N}}$ to ascribe a formal semantics that includes a precise, metric notion of time to the Stateflow notation that is used in the industrial practice of the design of Flexible Manufacturing Systems (FMS). In particular, we focus on the issue of providing a general, abstract, hierarchical mechanism to compose Stateflow diagrams into complex specifications of components interacting with each other. We exploit an encoding into the PLTLB logic of the decidable subset of X-TRIO$_{\mathbb{N}}$ introduced in [6] to perform automated formal verification of some properties of interest of a simple, yet realistic FMS.

3.1 Composition of Stateflow Diagrams

We use the Stateflow notation, a variation of Statecharts [9], to describe single modules and Simulink graphs to describe the composition of concurrent modules.

In a nutshell, a Stateflow diagram is composed of: (i) a finite set of typed variables V partitioned into input (V_I), output (V_O), and local (V_L) ones; input and output events are represented, respectively, through Boolean variables of V_I and V_O; input and output variables constitute the public interface of the module, as shown in the Simulink graph of Figure 2; (ii) a finite set of states S that can be associated with *entry*, *exit* and *during* actions, which are executed, respectively, when the state is entered, exited, or during the permanence of the system in the state; (iii) a finite set of transitions, H, that may include guards (i.e., conditions) on the variables of V and actions. An *action* is the assignment of the value of an expression over constants and variables to a non-input variable. We assume all variables in V to take values in a finite domain, which we denote by D_V.

We illustrate the notation through the example of a robotic cell composed of a robot arm that loads and unloads various parts on two machines, M_1 and M_2. The cell, as shown in Figure 1(a), is served by a conveyor belt, which provides pallets to be processed. There are two types of pallets, A and B, which are processed, respectively, by machine M_1 and by machine M_2. After processing, the finished parts are discharged from the cell by means of the conveyor out belt.

At any time, the robot arm can switch from automatic to manual mode or from manual to automatic. The Stateflow diagram describing the behavior of the robot arm is reported in [6]. The M_1 component is presented in Figure 1(b), while Figure 2 shows a Simulink graph representing a part of the robotic cell.

A Simulink graph represents a *component* of the system, which can be *basic* or *composed*. A basic component has a public *interface*, that corresponds to the set of variables $V_{Int} = V_I \cup V_O$ of the module, and a *behavior* description that is represented by a Stateflow graph. The specification of a composed component is structured as follows: at the lowest level of the system description hierarchy, it is represented by a Simulink graph with two or more basic components. Its interface is the union of the Input and Output variables of its components; the behavior is described by the Stateflow graphs of its modules, plus a network of communication relations between components represented graphically by a set of *links*. Each link corresponds to a flow of messages (signals or data) sent from a component to another one. The communication is realized by the assignment of the value of an Output variable of the sending component to a corresponding Input variable of the receiving component. One or more Simulink graphs can be further composed to obtain a new higher-level component.

The documentation provided by Mathworks presents the complete, although informal, specification of Stateflow diagrams, but it does not provide a precise

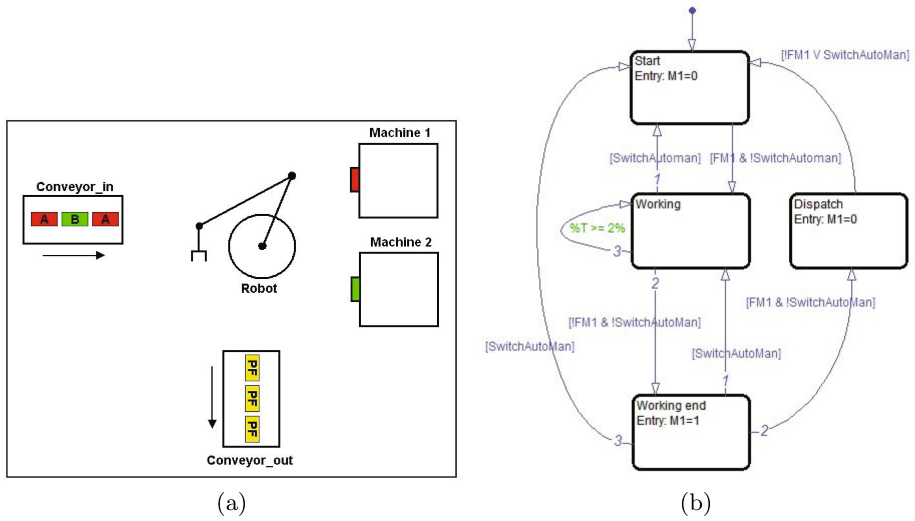


Fig. 1. Robotic Cell 1(a) and Stateflow graph of machine M_1 1(b)

definition of their semantics. Our semantics of Stateflow diagrams is based on the STATEMATE semantics of Statecharts ([10]). It includes a composition operator for building hierarchical, modular models from simpler ones.

The semantics of Stateflow hinges on the concept of *run*, which represents the reaction of the system to a sequence of input events. A run is a sequence of *configurations*; each configuration $c_i = \langle s, \nu \rangle$ pairs the current state $s \in S$ with an evaluation function $\nu : V \to D_V$ representing the current values of the variables. The configuration changes only when an enabled transition is executed.

The semantics of the evolution of time in Statecharts/Stateflow has proven difficult to pin down precisely, and different solutions have been proposed in the literature (e.g., [11,2]). Our model is of the so-called **run-to-completion** variety. In this model the system reacts to the input events by performing a sequence of *macro-steps*. Within every macro-step, a maximal set of *micro-steps* is executed based on the events generated in the previous macro-step. Micro-steps are executed infinitely fast, with time advancing only at macro-step boundaries, when the system reaches a *stable* configuration, in which no transition is enabled. As in the STATEMATE semantics of Statecharts, input events and data are sensed only at the beginning of macro-steps, while events and data are output to the environment, which also includes the other components, only at the end of macro-steps.

In the semantics outlined above each run identifies a sequence of time instants $\{T_i\}_{i \in \mathbb{N}}$, one for each macro-step, hence the time domain is discrete. This is consistent with the underlying physical model, as the PLCs on which FMS control solutions are built are governed by discrete clocks. Therefore, each macro-step corresponds to a *clock cycle* of the modeled PLC.

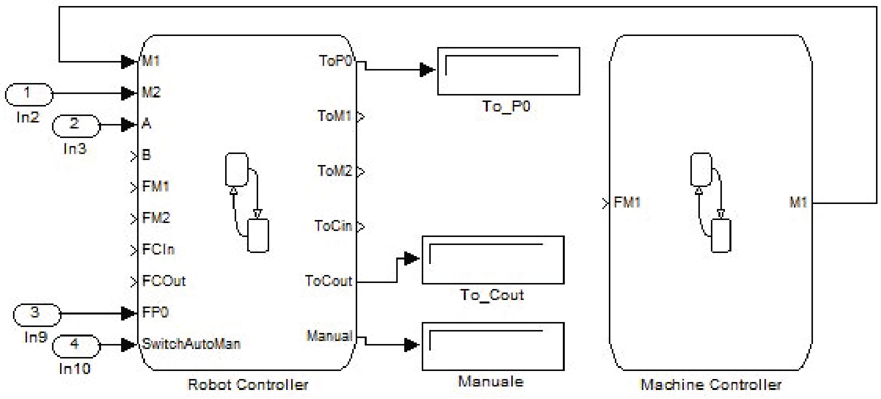


Fig. 2. Fragment of the Simulink graph of the robotic cell

For example, if, at the beginning of a macro-step, machine M_1 of Figure 1(b) is in state *Working end* and the robot arm signals that it is ready to dispatch the finished workpiece by setting input variable $FM1$, the transition between states *Working end* and *Dispatch* is enabled, so M_1 executes a micro-step and output variable $M1$ is set to false. At this point, the whole system has reached a stable state, since M_1 must wait for the robot arm to be free to deliver a new piece.

Let us now informally describe the semantics of the composition of two or more modules in a Simulink graph. Given two modules $G_1 = \langle V_1, S_1, H_1 \rangle$ and $G_2 = \langle V_2, S_2, H_2 \rangle$, we introduce a compositional binary operator $\|$ whose result is a new component $G = G_1 \parallel G_2$ with $V = V_1 \cup V_2$ and $S = S_1 \times S_2$. The transition relation of G is intuitively given by analyzing the example of Figure 3. The figure shows the runs of two modules A and B that are composed to obtain a component C. Each run is represented as a sequence of configurations and transitions, respectively described as rectangles labeled with the name of the current state and arrows labeled with the guard and the transition action. The figure represents a macro-step of the two runs: for component A the macro-step begins in state S_0 and ends in state S_3; similarly, component B goes from S_7 to S_9. The x axis represents the number of micro-steps executed from the beginning of the macro-step. The figure shows that when C is in a configuration where the two components have both an enabled transition (as in micro-step 0), the transitions are executed in parallel. After micro-step 1 component B reaches a stable configuration, using fewer micro-steps than component A. The two modules complete the macro-step in states S_3 and S_9, where a synchronization event occurs: all Output events and data are produced and sent as Input events and data to the corresponding receiving component according to the link network of the Simulink graph, and the “real” time advances.

It is easy to show that the parallel operator is associative, hence we can build a hierarchy of components.

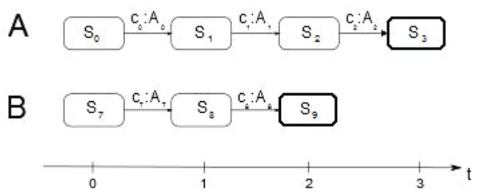


Fig. 3. Examples of runs of two composed modules A and B

3.2 X-TRIO$_{\mathbb{N}}$-Based Modular Semantics of Stateflow Diagrams

This section presents the formalization of the semantics of Stateflow diagrams using X-TRIO$_{\mathbb{N}}$ formulae. We first focus on single Stateflow diagrams, then deal with the issue of composing diagrams in a hierarchy.

As the domain D_V of variables is finite, given a $v \in V$ and a $k \in D_V$, $v = k$ is represented through a propositional letter v_k.

Given a Stateflow diagram representing the behavior of a module m, for each transition $H_{m,i} : s_{m,i} \overset{g_{m,i}/a_{m,i}}{\longrightarrow} s'_{m,i}$ originating from state $s_{m,i}$ and targeting state $s'_{m,i}$ with guard $g_{m,i}$ and action $a_{m,i}$, we introduce the following formula:

$$\text{AlwF}\Big((\gamma_{m,i} \wedge s_m = s_{m,i}) \rightarrow \text{X}_{\text{ns}}\big(s_m = s'_{m,i}\big) \wedge \alpha_{m,i} \wedge \alpha_{ex_{s_{m,i}}} \wedge \alpha_{en_{s'_{m,i}}}\Big) \quad (1)$$

where $\gamma_{m,i}$ is an X-TRIO$_{\mathbb{N}}$ formula encoding guard $g_{m,i}$, and $\alpha_{m,i}$, $\alpha_{ex_{s_{m,i}}}$ and $\alpha_{en_{s'_{m,i}}}$ are X-TRIO$_{\mathbb{N}}$ formulae encoding, respectively, the transition action $a_{m,i}$, the *exit* action of $s_{m,i}$, and the *entry* action of $s'_{m,i}$. Formula (1) formalizes the execution of a micro-step: it asserts that if the current state of module m is $s_{m,i}$ and the transition condition $\gamma_{m,i}$ holds, then in the next micro-step the active state is $s'_{m,i}$ and the *entry* actions of $s'_{m,i}$ and the *exit* actions of $s_{m,i}$ are executed. Thus, operator X_{ns} replaces a zero-time transition. To simplify the formalization of the Stateflow semantics, in this paper we restrict the distance between two consecutive non-standard instants to be ϵ, so each micro-step has a predefined fixed length. In principle, it would be possible to define different infinitesimal lengths for each basic component, but this is outside of the scope of the present work. Then, operator X_{ns} is related to the metric operator $\text{Dist}(\phi, \epsilon)$, as formula $\text{AlwF}(\text{X}_{\text{ns}}(\phi) \rightarrow \text{Dist}(\phi, \epsilon))$ holds. If no transition is enabled, the configuration does not change, as captured by the following formula:

$$\text{AlwF}\Big(\bigwedge_{i=1}^{|H_m|} \neg(\gamma_{m,i} \wedge s_m = s_{m,i}) \rightarrow \mathit{NOCHANGE}\Big) \quad (2)$$

where subformula $\mathit{NOCHANGE}$, which is not further detailed for space reasons, asserts that in the next micro-step the current state and the values of all output and local variables of module m do not change.

The "real" time advancement of our semantics is modeled through operator X_{st}: every time the system reaches a stable state (where no transition is enabled), the time advances to the next standard number. We restrict the distance between two consecutive standard instants (i.e. macro-steps) in a run to be exactly 1. The following formula captures the advancement of the "real" time in a single module:

$$\mathrm{AlwF}\left(\mathrm{X_{st}}(\top) \rightarrow \bigwedge_{i=1}^{|H_m|} \neg(\gamma_{m,i} \wedge s_m = s_{m,i})\right). \tag{3}$$

Formula (3) expresses a necessary condition for time advancement. A sufficient condition can be expressed at the level of the single module only after having introduced a pair of additional predicates that are used to coordinate the composition of different modules. This is unsurprising, as time advancement, as explained in Section 3.1, requires *all* modules to have reached a configuration where no further transitions are possible for any of them.

Finally, we introduce a formula asserting that input variables $V_{I,m}$ of module m change values only at the beginning of a macro-step, i.e. in a standard time instant. In other words, if the next time instant is non-standard, then the values of the input variables must be the same as those in the current instant:

$$\mathrm{AlwF}\Big(\mathrm{X_{ns}}(\top) \rightarrow (\textstyle\bigwedge_{v \in V_{I,m}, x \in D_V} v = x \rightarrow \mathrm{X_{ns}}(v = x))\Big) \tag{4}$$

The formula MOD_m encoding the behavior of a single component m is given by the conjunction of formulae $\bigwedge_{i=1}^{|H_m|}(1)_i$, (2-4), plus others not shown for brevity.

To build complex models from basic modules we employ a hierarchical approach where basic Simulink graphs are built from Stateflow diagrams, and they can then be in turn composed into Simulink components of higher level, and so on. To achieve this, for each module m, be it a simple Stateflow diagram, or a Simulink graph of any level, we introduce two X-TRIO$_{\mathbb{N}}$ predicates – and related X-TRIO$_{\mathbb{N}}$ formulae – that act as the interface of the module for the purpose of coordinating time advancement. More precisely, these predicates are used to guarantee that time advances only when all components are in a *stable* state, i.e., when none of their transitions is enabled. The first predicate, $stable_m$ is true when component m reaches a stable configuration. This is formalized by the following formula:

$$\mathrm{AlwF}\Big(\textstyle\bigwedge_{i=1}^{|H_m|} \neg(\gamma_i \wedge s_m = s_{m,i}) \leftrightarrow stable_m\Big) \tag{5}$$

The second predicate, $extST_m$, is used to coordinate different modules; more precisely, it is used to communicate to m when time advances, and its truth value is determined by the "environment" of m, i.e., by its enclosing module (if any). At the level of module m, time advancement obeys the following constraint:

$$\mathrm{AlwF}(\mathrm{X_{st}}(\top) \leftrightarrow stable_m \wedge extST_m) \tag{6}$$

Suppose now that module m is composed of n lower-level modules $m_1 \dots m_n$. Module m has its own predicates $stable_m$ and $extST_m$, where $stable_m$ is defined from the values of $stable_{m_1} \dots stable_{m_n}$, whereas $extST_m$ is defined by its environment. We have the following constraints, in addition to (5) and (6):

$$\text{AlwF}\left(\mathit{stable}_m \leftrightarrow (\bigwedge_{j=1}^{n} \mathit{stable}_{m_j})\right) \tag{7}$$

$$\text{AlwF}\left(\bigwedge_{j=1}^{n} (\mathit{extST}_m \leftrightarrow \mathit{extST}_{m_j})\right) \tag{8}$$

Formula (7) states that module m is stable only when all its components are, while (8) defines that the value of extST_m is passed on from m to its components.

A system is defined by an outermost Simulink diagram I that hierarchically integrates all components together. At the outermost level, time advances simply when all components are stable; in other words, for the overall module I, X_{st} in (6) depends only on predicate stable_m (which in turn is defined by formula (7)); to achieve this, extST_I is constrained to be always true, i.e. $\text{AlwF}(\mathit{extST}_I)$.

Finally, we formalize the relations between inputs and outputs of components of Simulink graphs such as those of Figure 2. If an output variable $v_{i,out}$ of module i is connected in the Simulink graph to input variable $v_{j,in}$ of module j, then the value of $v_{i,out}$ is synchronized with the value of $v_{j,in}$ at the beginning of each macro-step, i.e., when the time instant is standard. This is captured by the following constraint, which uses predicate NowST introduced in Section 2:

$$\text{AlwF}(\text{NowST} \rightarrow (v_{i,out} = v_{j,in})) \tag{9}$$

Our formalization of the compositional semantics is such that *stutter* steps, which are only implicit in the informal semantics described in Section 3.1, emerge explicitly. Figure 4 shows the runs of two concurrent components A and B that are part of a higher-level module C. The figure represents a macro-step executed by the system at time t: component A (resp. B) starts the macro-step in state S_0 (resp. S_7) and ends in state S_k (resp. S_{10}). The x axis represents the time instants of $\overline{T}$. As the figure shows, B reaches stable state S_9 (in which predicate stable_B holds) before A, in instant $t + 2\epsilon$. Since A has not yet reached a stable state at $t + 2\epsilon$, stable_A (hence also stable_C) is false at $t + 2\epsilon$ and, from formula 6, the next instant is non-standard. Finally, since formula 9 does not hold at $t + 3\epsilon$, component B remains in stable state S_9, thus creating a *stutter step*, a zero-time transition that does not change configuration.

Variations to the composition semantics. As mentioned in Section 1, many semantics exist for Statecharts and its variants such as Stateflow. The X-TRIO$_{\mathbb{N}}$-based approach pursued in this paper gives us great flexibility in adapting the formal semantics depending on the cases. In fact, changing semantics is as simple as changing X-TRIO$_{\mathbb{N}}$ formulae.

For example, in some semantics input and output variables are synchronized not only at the end of a macro-step, but also during it [5]. To allow for this behavior one would have to change constraint 9 with the following one (also, constraint 4 would have to be modified, but this is not shown here for brevity),

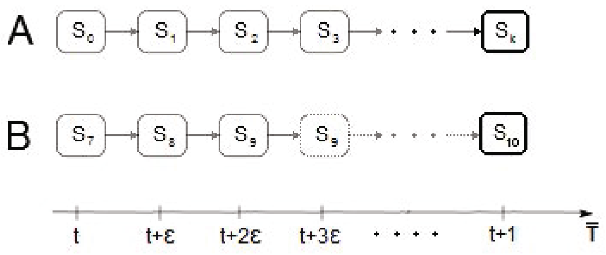


Fig. 4. Runs of two composed components A and B introducing stutter transitions

which prescribes that connected input and output variables have the same values, unless one of the two components has reached a stable state:

$$\text{AlwF}(\neg stable_i \wedge \neg stable_j \rightarrow (v_{i,out} = v_{j,in})) \tag{10}$$

Another possible variation could consist of imposing that there must be a maximum number K of micro-steps in a macro-step. This would reduce to forcing condition $stable_m \wedge extST_m$ (which, for formula (6), entails passing to a new macro-step) to occur within $K\epsilon$ instants from a standard one, which is simply formalized by the following formula (where $\text{WithinF}(\phi, K\epsilon)$ is an abbreviation for $\phi \vee \text{Dist}(\phi, \epsilon) \vee \ldots \vee \text{Dist}(\phi, K\epsilon)$):

$$\text{AlwF}(\text{NowST} \rightarrow \text{WithinF}(stable_m \wedge extST_m, K\epsilon)) \tag{11}$$

In a similar vein, another possible semantic variation might impose that a macro-step cannot last more than K micro-steps, even if the system has not yet reached a stable state. This semantics could be formalized, for example, by introducing an additional predicate, say $adv\_unstable_m$ which holds exactly at distance $K\epsilon$ from a standard instant t if at $t + K\epsilon$ the module is still not stable. Then, formula (6) should be modified as follows (formulae (1)-(3) would also have to be modified, but this is not shown here for brevity):

$$\text{AlwF}(\text{X}_{\text{st}}(\top) \leftrightarrow (stable_m \vee adv\_unstable_m) \wedge extST_m) \tag{12}$$

3.3 Verification of System Properties and Experimental Results

The formalization introduced in Section 3.1 has been implemented in the $\mathbb{Z}$ot [1] bounded satisfiability/model checker to perform the verification of some typical real-time properties of the example FMS system of Figure 1(a). In [6] we show how we have encoded X-TRIO$_{\mathbb{N}}$ operators in the input language of the $\mathbb{Z}$ot tool.

In this paper, we focus the attention on properties of the overall system, which depend on the interactions of the modules composing the robotic cell. An important property in a system of modules evolving concurrently is the presence of deadlock. To define the property, we use a variation of the intuitive notion of "until", called $\mathrm{Until}_{\mathrm{stable}}$, which takes into account only the last micro-step of each macro-step, when the system reaches a stable state. Informally, $\mathrm{Until}_{\mathrm{stable}}(\phi, \psi)$ holds if there is a future macro-step such that in its last micro-step ψ holds, and ϕ holds in the last micro-step of all macro-steps before that. It is defined by the following X-TRIO$_{\mathbb{N}}$ formula:

$$\mathrm{Until}_{\mathrm{stable}}(\phi, \psi) \overset{\mathrm{def}}{=} \mathrm{Until}(\mathrm{X}_{\mathrm{st}}(\top) \to \phi, \mathrm{X}_{\mathrm{st}}(\top) \wedge \psi) \tag{13}$$

where the last micro-step is identified by the fact that its next instant is standard. Our notion of deadlock is defined only over macro-steps, since we consider micro-steps to be *transient* states that are non-observable outside of a module. Then, we say that the system is in deadlock if *all* of its components are in a deadlock state. If M is the set of components of the system, where each $m \in M$ is described through a Stateflow diagram with state space S_m, the following X-TRIO$_{\mathbb{N}}$ formula captures this notion of deadlock:

$$\bigwedge_{m \in M} \bigvee_{x \in S_m} \mathrm{SomF}_{\mathrm{stable}}(\mathrm{AlwF}_{\mathrm{stable}}(s_m = x)) \tag{14}$$

where $\mathrm{SomF}_{\mathrm{stable}}(\phi)$ and $\mathrm{AlwF}_{\mathrm{stable}}(\phi)$ are, as usual, abbreviations for $\mathrm{Until}_{\mathrm{stable}}(\top, \phi)$ and $\neg\mathrm{SomF}_{\mathrm{stable}}(\neg\phi)$, respectively.

Other properties, that we have verified in [6], are: (P1) the presence of *Zeno runs*, which would make the modeled system unfeasible; (P2) real-time properties, for example whether it is possible to produce and deliver one processed workpiece of any kind within L time units from the system startup.

Some performance results obtained during the verification of properties above are shown in [6]. The verification was performed with a time bound of 70 time units, which is a user-defined parameter of the verification that corresponds to the maximum length of runs analyzed by $\mathbb{Z}$ot, as customary in bounded satisfiability checking. The absence of deadlocks (property P1 above) was determined in less than 90 seconds, using about 260MB of memory.[1] Similarly, the tool determined in around 400s that property P2 *does not hold* when $L = 15$, and in 90s that it holds for $L = 20$. On the other hand, $\mathbb{Z}$ot determined in 17991s (using about 270MB of memory) that property (14) does not hold, i.e., the system is deadlock-free. The long verification time is due to the fact that the sole Stateflow diagram of the controller of the robot arm of Figure 1(a) [6] has $12 \cdot 2^{18}$ possible configurations ($|S| \cdot 2^{|D_V|}$); the overall system model is of course bigger.

During the verification phase we also detected and corrected errors in an earlier version of the robotic cell design. More precisely, by checking X-TRIO$_{\mathbb{N}}$ formula (14) on an earlier model, the tool determined that deadlocks did exist, and it returned a case of deadlocked run. By studying this run, we discovered that there was a problem in the communication protocol between the *Robot* and machine M_1, which also affected the cell *Controller*. The problem was that the

[1] All tests have been performed on a 3.3GHz QuadCore PC with Windows 7 and 4GB of RAM. The verification engine was the `ae2zot` SMT-based $\mathbb{Z}$ot plugin using the z3 3.2 solver (`http://research.microsoft.com/en-us/um/redmond/projects/z3/`).

system remained forever in a configuration where the robot arm was waiting for machine M_1 to signal the end of the communication protocol. The termination event is signaled by M_1 by setting a variable called $FCIn$ to 1, but this had not been modeled in the earlier model. After correcting the error, a new check of property 14 showed that the modified system model was deadlock-free.

To conclude this section we remark that, thanks to the compositional nature of the semantics presented in Section 3.2, properties such as the absence of Zeno runs can be studied at the level of the single components instead of the whole system. Intuitively, a component has a Zeno behavior if, from a certain point on, the execution trace presents an infinite sequence of micro-steps (i.e. non-standard instants). Let us consider a single Stateflow diagram m. If no Zeno runs exist in m when predicate $extST_m$ is always true (i.e., when m is analyzed in isolation), then no Zeno runs of m can occur when the module is composed with others. Therefore, given a system I composed by modules $m_1 \dots m_n$, if none of these has Zeno runs, I does not have them, either, since input variables of the single components do not change value until the overall system I reaches a stable state. This suggests the possibility of performing that kind of verification on the (smaller) models of the single components, instead of on the full integrated model. On the other hand, if a Zeno runs exists in a component m taken in isolation, this might be triggered by a combination of inputs that does not appear in the system as a whole, so in this case a further analysis should be carried out at system level.

4 Conclusions and Future Work

We presented an approach to formally model and automatically analyze FMS specified as an integrated collection of Stateflow modules. We focused our attention on the semantic intricacies due to the separation between micro- and macro-steps (the only ones in which time elapses according to traditional literature); such intricacies become even more critical when moving from the semantics of a single module to that of the coordinated behavior of a collection thereof.

In a companion paper we exploited nonstandard analysis to deal with micro-steps by replacing zero-time transitions with ϵ-time ones, thus avoiding a few pitfalls typical of transitions consuming no time; in this paper we extended our approach – based on the X-TRIO$_{\mathbb{N}}$ metric temporal logic – to formalize the concurrent behavior of several Stateflow modules.

We deem that the distinguishing feature of our approach is its generality: for instance, in Section 3.2 we offered a sample of variations of the composition semantics which could be obtained by simple changes in the X-TRIO$_{\mathbb{N}}$ formulae formalizing it; other generalizations could involve the use of different standard and non-standard values to measure the duration of micro- and macro-steps in different modules. Our approach could also be easily adapted to other classical notations, such as activity diagrams, Petri nets, etc.

Finally, the logic language used in this paper was suitably restricted to support automated verification of systems and properties typical of FMS. An intriguing challenge for future research would be looking for (sub)optimal trade-offs between expressiveness and automated verification: e.g., how far could distances between micro- and macro-steps be generalized, still maintaining decidability?

Acknowledgments. We thank Emanuele Carpanzano and Mauro Mazzolini of CNR-ITIA for providing expertise, insight and examples of design of FMS.

References

1. The ℤot bounded model/satisfiability cheker, http://zot.googlecode.com
2. Alur, R., Henzinger, T.: Reactive modules. Formal Methods in System Design, pp. 15:7–15:48 (1999)
3. Bu, L., Cimatti, A., Li, X., Mover, S., Tonetta, S.: Model Checking of Hybrid Systems Using Shallow Synchronization. In: Hatcliff, J., Zucca, E. (eds.) FMOODS 2010. LNCS, vol. 6117, pp. 155–169. Springer, Heidelberg (2010)
4. Ciapessoni, C., Mirandola, P., Coen-Porisini, A., Mandrioli, D., Morzenti, A.: From formal models to formally-based methods: an industrial experience. In: ACM TOSEM, pp. 79–113 (1999)
5. Eshuis, R.: Reconciling statechart semantics. Sci. of Comp. Prog. 74, 65–99 (2009)
6. Ferrucci, L., Mandrioli, D., Morzenti, A., Rossi, M.: Non-null infinitesimal micro-steps: a metric temporal logic approach (2012), extended version, http://arxiv.org/abs/1206.0911
7. Gargantini, A., Mandrioli, D., Morzenti, A.: Dealing with zero-time transitions in axiom systems. Information and Computation 150(2), 119–131 (1999)
8. Hamon, G., Rushby, J.: An operational semantics for stateflow. Int. J. on Software Tools for Technology Transfer 9(5-6), 447–456 (2007)
9. Harel, D.: Statecharts: A visual formalism for complex systems. Sci. of Comp. Prog. 8(3), 231–274 (1987)
10. Harel, D., Naamad, A.: The STATEMATE semantics of statecharts. ACM TOSEM 5(4), 293–333 (1996)
11. Levi, F.: Compositional verification of quantitative properties of statecharts. J. Log. Comp. 11(6), 829–878 (2000)
12. Object Management Group: OMG Unified Modeling Language (OMG UML), Superstructure. Tech. rep., OMG (2010), formal/2010-05-05
13. Pnueli, A., Shalev, M.: What is in a Step: On the Semantics of Statecharts. In: Ito, T., Meyer, A.R. (eds.) TACS 1991. LNCS, vol. 526, pp. 244–264. Springer, Heidelberg (1991)
14. Robinson, A.: Non-standard analysis. Princeton University Press (1996)

Optimizing the Robustness of Software against Communication Latencies in Distributed Reactive Embedded Systems

Vlad Popa and Wolfgang Schwitzer

Institut für Informatik,
Technische Universität München,
Boltzmannstr. 3, D-85748 Garching, Germany
{popav,schwitze}@in.tum.de
http://www4.in.tum.de/

Abstract. This paper presents a formal approach of designing software robust against communication latencies that typically occur in distributed embedded systems. In this approach, the software's data-flow is retimed and scheduled in order to achieve the maximum robustness against possible communication latencies. This robustness is derived individually for a given software and its distribution on a platform's communication topology. Robustness is interpreted as the guaranteed amount of time, up to which the system does not change its externally observable behavior due to communication latencies. The software's data-flow is given as a data-flow graph with nodes representing tasks and edges representing communication channels. A linear problem approach is employed that transforms elements of data-flow into variables of linear expressions. An implementation of the approach in the tool CADMOS together with the application on a case example from the automotive software engineering domain shows its practicability.

Keywords: Distributed Systems, Embedded Systems, Reactive Systems, Data-Flow, Retiming, Linear Problem, Scheduling, Communication Latency Robustness.

1 Introduction

Embedded software systems realize critical functionality in industrial products like cars, airplanes, production automation systems, energy supply systems and medical devices. In these areas, embedded software systems are typically *reactive* [1]. This means, there is a permanent and time-sensitive interaction between the system and its physical environment. Sensors read the environment inputs, which are processed and sent as outputs to the actuators. In many cases, these embedded systems are built upon a distributed platform architecture that comprises embedded control units (ECUs) interconnected by bus systems. Hence, software tasks send and receive messages through these bus systems in order to communicate with each other. A bus system's message transmission time can deviate from an expected ideal transmission time, which can lead to unwanted behavior of a reactive and time-sensitive system. In this paper, we present a formal approach how software can be made robust against communication latencies that occur in distributed embedded systems.

It is a typical design goal for distributed reactive software systems to be as robust as possible against communication latencies while exploiting available parallel resources of the platform. Regarding deviations in transmission time between two ECUs, there are two possible cases. In the first case, a message arrives at its destination *earlier* than expected. This is commonly solved by buffering incoming messages at the receiver's side and is not within the scope of this paper. In the second case, a message arrives at its destination *later* than expected. This paper focuses on how to analyze and maximize the robustness against unwanted system behavior in the second case. In the approach presented in this paper, we want to harness any delays and waiting times in the software of the system, so that it is more robust against latencies on the platform.

It is common for industrial critical systems that the software has to produce (functionally) correct outputs, which additionally have to be delivered within an expected time interval. Different technical issues influence the expected transfer time on bus systems. Deviations are typically caused by electromagnetic compatibility (EMC) issues, resource conflicts on the MAC-layer or clock synchronization issues. In the context of this paper, we regard these deviations as inevitable and concentrate on how to make the software as robust as possible against it. By describing robustness against communication latencies as an "extra-functional" requirement, we present algorithms to determine whether this requirement can be fulfilled and how this can be done by retiming and scheduling the software.

This paper provides an interpretation of robustness as a formal property $\rho(S)$ of a system S in Sec. 2 and Sec. 3. Additionally a more coarse grain property $\rho^{DFG}(S)$ is provided, if execution times of tasks are not available, e.g., in early phases of development.

Related work. In this paper, we analyze and transform a software's *data-flow* represented by a *data-flow graph* (DFG). In a data-flow graph $G = (V, E, \delta)$ each *node* in V represents an executable *task* and each *edge* in E represents a directed communication *channel* between two nodes. General concepts of modeling systems with data-flow graphs are found in [2,3,4,5], for example. Data-flow modeling is supported by well-known tools like e.g. Simulink, Labview and Ptolemy. In this paper we will use an *iterative* version of data-flow [4], in which every node is executed exactly *once* in each iteration. In iterative data-flow, while a node is executed, it *consumes* one message from each of its *incoming* channels and *produces* one message on each of its *outgoing* channels.

Furthermore, data-flow channels may contain *unit delays* (or *delays*). The delay function $\delta \in E \rightarrow \mathbb{N}_0$ returns the number of delays for each channel. If a channel c contains $d = \delta(c)$ unit delays, then the output of the source node $src(c)$ in the i-th iteration will be processed by the destination node $dst(c)$ in the $(i + d)$-th iteration. Delays are present in data-flow models for different reasons:

– A cyclic path in a DFG (describing a recurrence equation) must contain at least one unit delay in order to be computable (see [6]).
– An algorithm requires delays (e.g.: digital FIR- and IIR-filters).
– Adding or moving delays modifies the critical path (the most time-consuming undelayed path) [7,8].

Leiserson, Rose and Saxe presented the *retiming* transformation [7,8] that allows for moving delays between edges in a DFG. While changing the internal structure of the DFG, retiming preserves the externally observable behavior of the data-flow program. Originally, retiming aims at modifying the critical path to enable better parallelization. In [7] a *linear programming* based approach is presented, how delays have to be retimed to achieve maximum parallelization. A similar approach based on linear programming is described in [9] for the computation of the minimum cycle period in a DFG. In this paper, we use retiming and scheduling to maximize robustness against communication latencies. The robustness maximization algorithms (see Sec. 3) determine the appropriate retiming of delays and the schedule based on linear problems.

We also make use of a *delay caclulus* [10] to describe some of the analysis and maximization methods. In this delay calculus, each output channel o in a data-flow graph G has a so-called *minimum guaranteed delay*, denoted by $gardelay(G, o)$. In a data-flow graph G there can be many paths from input channels $\{i_1, \ldots, i_n\}$ that lead to the same output channel o. For an output channel o the minimum guaranteed delay $gardelay(G, o)$ is equal to the minimum sum of delays on each of the paths leading from the $\{i_1, \ldots, i_n\}$ to o.

Outline. This paper is organized as follows. Section 2 presents methods to analyze the robustness against communication latencies. The following section 3 shows how a given DFG can be retimed and scheduled in order to achieve maximum robustness against communication latencies. Section 4 describes how the previous techniques can be implemented for automatic verification and optimization with the tool Cadmos [11]. The next section 5 presents an evaluation of the presented concepts. For this purpose, the data-flow model of an automotive *Adaptive Cruise Control* system is analyzed and optimized with the tool Cadmos. The last section 6 concludes with final remarks and future work.

2 Analyzing the Robustness against Communication Latencies

In a typical iterative development process, software is designed, analyzed and improved if necessary. Robustness against communication latencies can already be analyzed in early phases of development. In this paper, we regard a system as a tuple $S = (G, R, M)$ with data-flow graph $G = (V, E, \delta)$, platform resources $R = (P, B)$ (processors P, buses B) and a mapping M that defines the allocation of data-flow elements onto resources. This section is organized as follows. First, we define a coarse grain robustness $\rho^{DFG}(S)$ based on delay analysis only. This is followed by a refined notion of robustness $\rho(S)$ by analyzing a static schedule for a given system S. With the help of $\rho^{DFG}(S)$ and $\rho(S)$ we are able to decide, whether an improvement of the robustness is actually needed. In the scope of this paper, the following assumptions are made:

- All tasks of the system are executed once within an iteration.
- An iteration has a given *static* execution period T.
- The mapping M of data-flow elements onto resources is known (see Eq. 2).
- The precedence relation of tasks is defined (see [5]).
- Schedules are computed in an *offline* and *static* way.

Analyzing the data-flow graph: coarse grain robustness. First, we define a coarse grain robustness $\rho^{DFG}(S)$ by analyzing the delays of the data-flow graph $G = (V, E, \delta)$ of the system S. The function $\delta \in E \rightarrow \mathbb{N}$ returns the number of delays for each channel. By $\delta(X, Y)$ we denote the number of delays on the channel (X, Y). In order to analyze $\rho^{DFG}(S)$ we need to know, which channels are mapped onto the platform's bus systems. In more detail, this leads us to the subgraphs G_i which receive bus messages *and* produce outputs. However, input channels and communication channels that go out of G_i are ignored. For the robustness analysis, the channels which carry bus messages into G_i are the new input channels for G_i.

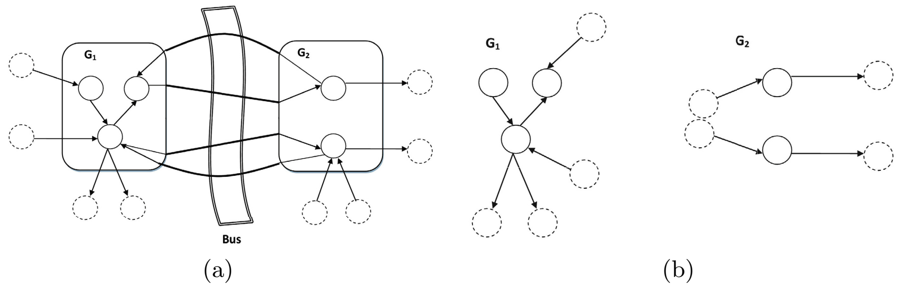


Fig. 1. Obtaining subgraphs G_i: (a) the original DFG; (b) the subgraphs G_1, G_2 relevant for analysis. Nodes with dotted lines are environmental input/output nodes.

After determining the subgraphs G_i and the respective output channels $Output(G_i)$, we define $\rho^{DFG}(S)$ with the help of the minimum guaranteed delay function $gardelay(G_i, o)$ [10] (see related work, page 178):

$$\rho^{DFG}(S) = min\{gardelay(G_i, o) \mid \forall o \in Output(G_i)\} \quad (1)$$

The function $\rho^{DFG}(S)$ expresses the communication latencies robustness in integer multiples of iterations. For example, $\rho^{DFG}(S) = 2$ means that S will not change its externally observable behavior if communication on any bus is up to two iterations late. The function $\rho^{DFG}_{opt}(S)$ calculates the maximum achievable robustness by retiming and is further explained in subsection 3.1.

Analyzing the schedule: refined robustness. In an offline and statically scheduled system S, we can derive a more accurate robustness $\rho(S)$. Let us regard the undelayed, feed-forward system together with its schedule on two processors in Fig. 2, where A_i indicates the i-th execution of node A. Using Eq. 1, the communication latency robustness is $\rho^{DFG}(S) = 0$. However, the schedule in 2(b) reveals the amount of latency which can occur after a bus transmission, without affecting the correctness and end-to-end latency of the system. This is indicated by the dashed areas ("Robust").

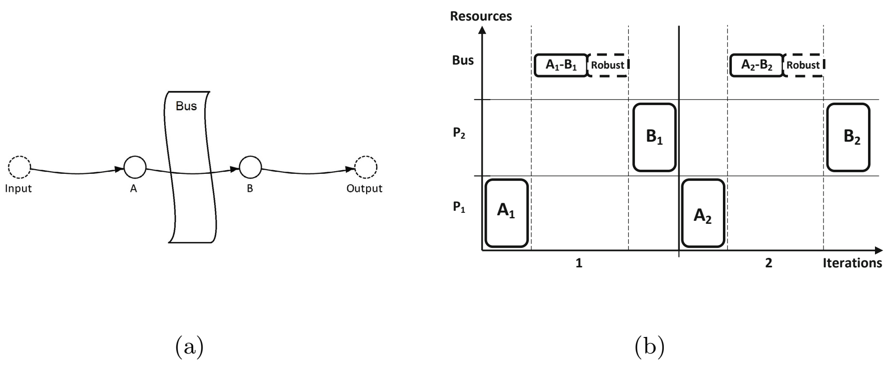


Fig. 2. Additional robustness by scheduling: (a) data-flow model of a feed-forward system; (b) its possible schedule

Knowing the schedule of a system, the following functions are computable: $s(X_i)$ returns a real number, representing the *starting time* in multiples of iterations of the i-th execution of node X; $e(X)$ returns a real number, representing the *execution time* in multiples of iterations of X; $tr(X, Y)$ returns a real number, representing the message *transmission time* in multiples of iterations on channel (X, Y). The mapping function M with signature

$$M \in V \cup E \rightarrow P \cup B \tag{2}$$

associates each DFG element (tasks V, channels E) of the DFG $G = (V, E, \delta)$ with a resource (processors P, buses B). Hence, regarding the scheduling of a system S the communication latency robustness is computed as follows:

$$\begin{aligned}\rho(S) = min\ \{\ & s(Y_j) - (s(X_i) + e(X)) - tr(X, Y) \\ & \mid \forall (X, Y) \in E \wedge M(X, Y) \in B \wedge \delta(X, Y) = j - i\}\end{aligned} \tag{3}$$

This refined robustness $\rho(s)$ returns real numbers describing complete iterations and their fractional parts. A computational node X_i communicates with node Y_{i+d} if and only if $\delta(X, Y) = d$. In Sec. 3.2, we present a method, which retimes the data-flow model and modifies the schedule in order to obtain the global maximal robustness of a system S against communication latencies.

3 Optimizing the Robustness against Communication Latencies

If the robustness of a system is insufficient, the communication latencies can cause the system to produce unexpected and unwanted outputs. Distributed reactive embedded systems are exposed to communication latencies and therefore, need to be analyzed carefully. Especially those tasks of a system, which produce safety-critical outputs, should be as robust as possible. If, after applying the analysis methods presented in Sec. 2, the communication latency is able to influence the functionality of a system, the robustness needs to be improved. Subsection 3.1 shows how to optimize the data-flow graph by retiming and subsection 3.2 explains how get more accurate results by optimizing the schedule.

3.1 Optimizing the Data-Flow Graph by Retiming

This section shows how to achieve more robustness against communication latencies by retiming the given DFG of a system. The following paragraphs first present two basic scenarios to illustrate the idea using simple graphs, then provide a general formalization and finally give a more complex example.

Basic scenarios. The first line of table 1 sketches the first scenario for improving the robustness against communication latencies. In the first column, a chain of three communicating nodes A, B and C is shown. The delay function is $\delta(A, B) = 3$ and zero otherwise. Moreover, the channel (B, C) is mapped on the bus. The robustness $\rho^{DFG}(S)$ of the system is equal to $\delta(B, C) = 0$. For an improvement, channel (B, C) has to get delays of channel (A, B) by retiming. The optimal robust system is seen in the second column with $\rho^{DFG}_{opt}(S) = \delta(B, C) = 3$.

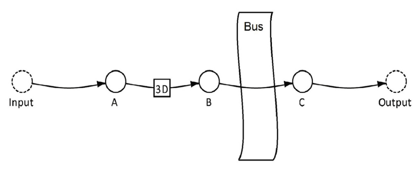
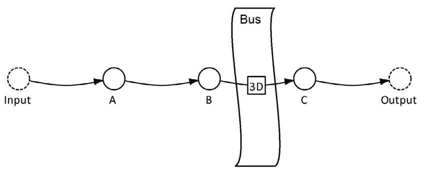
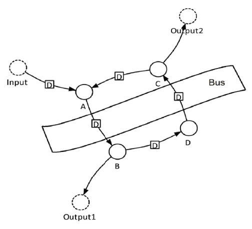
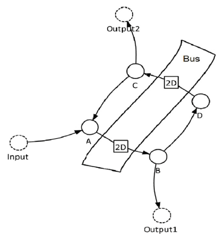
The DFG of the system in the second line of table 1 contains a cycle with four nodes A, B, C and D. The delay function has the non-zero values: $\delta(Input, A) = \delta(A, B) = \delta(B, D) = \delta(D, C) = \delta(C, A) = 1$. The channels (A, B) and (D, C) are mapped on the bus. If we apply Eq. 1, then $\rho^{DFG}(S) = 1$. The cycle contains four unit delays, which have to be equally distributed to the channels (A, B), (D, C), in order to maximize the robustness of the system (the number of delays on a cycle stays the same, see [7]). The second column contains the retimed system, in which the robustness is $\rho^{DFG}_{opt} = 2$.

Formal approach. The ρ^{DFG}_{opt} function computes the optimal achievable robustness in number of iterations for a system S. For maximizing the robustness against communication latencies, we have to consider a *max-min* problem.

$$\rho^{DFG}_{opt}(S) = maxmin\{\delta(X, Y) \mid \forall (X, Y) \in E \wedge M(X, Y) \in B\} \tag{4}$$

Eq. 4 suggests that the minimal number of delays of a channel mapped on the bus has to be maximal in the robust system. When the sender submits its message, the possible communication latency is compensated by unit delays. For computing the value $\rho^{DFG}_{opt}(S)$ we have to make some observations. If we

Table 1. Two basic data-flow graphs and their optimization: (1) a pipeline; (2) a cycle

Nr.	Data-Flow Graph	Robust Retimed Data-Flow Graph
1		
2		

regard an *elementary path* as a ordered set of edges, $P = \{e_1, \ldots, e_n\}$, in which $\forall i \neq j, e_i \neq e_j$, the following lemma holds.

Lemma 1. *The sum of all delays on every elementary path from an input to an output stays the same.*

Proof. Let P be an elementary path from input I to output O. According to [8] the number of delays in P after the retiming is $\delta_r(P) = \delta(P) + r(O) - r(I)$ ($r(X)$ is the number of delays used to retime X). Input/Output nodes can not be retimed and hence, $r(I) = r(O) = 0$.

$$\Rightarrow \delta_r(P) = \delta(P) \qquad \square$$

The optimization algorithm is divided into two parts. The first part involves finding a path coverage that includes all channels in elementary cycles and paths from inputs to outputs. A system of linear expressions is build out of the path coverage. In the second part this system of linear expressions is solved with the help of a linear problem solver. The final graph structure can be retraced from the variable occupancy of the linear expression result.

The algorithm for computing all paths and cycles in the DFG can be implemented with a depth-first-search and will not further be discussed in this paper. However, every elementary cycle and path of the resulting coverage $X = \{e_1, e_2, \ldots e_n\}$, $X \in \mathit{Coverage}$ contributes to the linear system with an equation:

$$a_{e_1} + \cdots + a_{e_{n-1}} + a_{e_n} = \sum_{i=1}^{n} \delta(e_i) \tag{5}$$

In Eq. 5, the a_{e_i} are variables that represent the number of delays that will be on channel e_i in the robust system S. On every cycle the number of delays stays the same (see [8]) and lemma 1 ensures that equations as in Eq. 5 will also hold after the system is retimed. The optimal solution is calculated by maximizing the value of the variables a_{e_i}, which correspond to graph channels e_i deployed on the Bus (see definition of ρ_{opt}^{DFG} in Eq. 4).

Example 1. A system containing 8 nodes is given in data-flow representation (see Fig. 3). We want to optimize the distribution of delays in order to obtain the optimal robustness against communication latencies. All end-to-end latencies (i.e. from inputs to outputs) of the original graph must be preserved. Furthermore, we assume that the channels mapped on the bus are (D, E) and (H, F). Regarding the definition of ρ_{opt}^{DFG} in Eq. 4, these channels are relevant for the communication latencies robustness. Therefore, we have to retime the system in such a way that $min\{\delta(D, E), \delta(H, F)\}$ is maximal. The analyzed system's DFG does not contain any cycles. The first step for optimizing the robustness against communication latencies is to compute the paths, which cover all channels of the graph. Such a path-coverage is realized by the two paths P_1 and P_2:

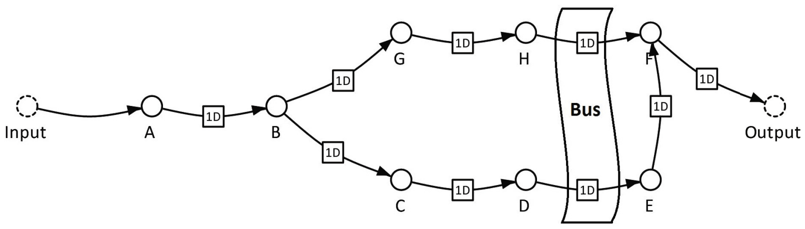


Fig. 3. The data-flow graph of a system containing 8 vertices. The channels (D, E) and (H, F) are mapped on a bus. This figure is based on a visualization by the tool CADMOS [11].

$$P_1 = \{(Input, A), (A, B), (B, C), (C, D), (D, E), (E, F), (F, Output)\}$$
$$P_2 = \{(Input, A), (A, B), (B, G), (G, H), (H, F), (F, Output)\}$$

Consequently, the linear system contains the two equations:

$$a_{IA} + a_{AB} + a_{BC} + a_{CD} + a_{DE} + a_{EF} + a_{FO} = 6$$
$$a_{IA} + a_{AB} + a_{BG} + a_{GH} + a_{HF} + a_{FO} = 5$$

This system of equations has to be solved so that the minimum between $a_{DE} = \delta(D, E)$ and $a_{HF} = \delta(H, F)$ is maximal. We employ a linear programming solver to compute this max-min problem. For example, the tool lp_solve offers a language to describe linear problems (see [12]) and we use the following script for lp_solve:

```
// Objective function.
max: robustness;
// Variable bounds.
ia + ab + bc + cd + de + ef + fo = 6;
ia + ab + bg + gh + hf + fo = 5;
// Minimization part, robustness = min{de, hf}.
robustness < de;
robustness < hf;
// Integer variables.
int ia, ab, bc, cd, de, ef, bg, gh, hf, fo;
```

The linear programming solver finds that the optimal solution for $\rho_{opt}^{DFG} = 5$. The following variable values lead to this optimum:

$$ia = 0, ab = 0, bc = 1, cd = 0, de = 5, bg = 0, gh = 0, hf = 5, ef = 0, fo = 0$$

3.2 Optimizing the Schedule

As presented in Sec. 2, the robustness against communication latencies can be determined more accurately if the schedule is additionally taken into account. In this section we explain how to find a schedule, which leads to optimal robustness. This involves finding the proper distribution of delays and the starting time of each task during scheduling. We employ a linear problem based approach similar to subsection 3.1. The function, which calculates the optimal robustness is defined as:

$$\rho_{opt}(S) = maxmin\ \{\ s(Y_j) - (s(X_i) + e(X)) - tr(X,Y) \\ \mid \forall (X,Y) \in E \wedge M(X,Y) \in B \wedge \delta(X,Y) = j - i\} \quad (6)$$

In the context of this paper, we obtain a schedule in an *offline* and *static* way. Hence, the statement $\forall j \geq i : s(X_j) = s(X_i) + j - i$ is valid. It suggests that a task X starts at the same point of time within each iteration. Consequently, Eq. 6 is transformed into:

$$\rho_{opt}(S) = maxmin\ \{\ s(Y_0) - (s(X_0) + e(X)) - tr(X,Y) + \delta(X,Y) \\ \mid \forall (X,Y) \in E \wedge M(X,Y) \in B\} \quad (7)$$

Note that this method of computing the robustness against communication latencies adds accuracy to the one defined by Eq. 4. Anyhow, it requires the worst case execution time of each task and the worst case transmission time on the bus to be known at scheduling time.

Now, that we have defined the objective function, the data-flow graph's precedence constraints are modeled as linear constraints. For every channel $(X,Y) \in E$ with $\delta(X,Y) = 0$, the constraint of X being executed before Y in each iteration needs to be satisfied.

Lemma 2. *For a data-flow channel* $(X, Y) \in E$ *the following inequality holds.*

$$s(Y_{i+\delta(X,Y)}) - (s(X_i) + e(X)) - tr(X, Y) \geq \begin{cases} \delta(X, Y) & \text{if } \delta(X, Y) = 0 \\ \delta(X, Y) - 1 & \text{if } \delta(X, Y) > 0 \end{cases} \quad (8)$$

This means, if $\delta(X, Y) = 0$, the starting execution time of the receiving node Y is greater or equal to the time when the sending node X finishes its computation and transmission. In this case the precedence constraint is fulfilled. If $\delta(X, Y) > 0$, inequality 8 shows that the gap between X_i and $Y_{i+\delta(X,Y)}$ is at least $\delta(X, Y) - 1$.

A branching if-construct is unsuitable for a linear program. Therefore, the linear expression 9 is used. In case of $\delta(X, Y) = 0$, it describes the precedence constraint. Otherwise if $\delta(X, Y) > 0$, it will be a true statement because $s(Y_0) - (s(X_0) + e(X)) - tr(X, Y) \in [-1, 1]$ and therefore, will not affect the variables.

$$s(Y_0) - (s(X_0) + e(X)) - tr(X, Y) \;\geq\; -\delta(X, Y) \quad (9)$$

Finally we need to introduce constraints that ensure that tasks running on the same processing unit do not overlap in their execution. For two tasks X, Y of the DFG, either

$$\begin{aligned} &s(X_0) - (s(Y_0) + e(Y)) \geq 0 \quad (Y \text{ finishes before } X) \text{ or} \\ &s(Y_0) - (s(X_0) + e(X)) \geq 0 \quad (X \text{ finishes before } Y) \end{aligned}$$

is true. This leads us to the binary variable b_{XY}, which can only take the values 0 or 1. If $b_{XY} = 1$, then X is executed before Y. Otherwise, Y is executed before X. The following statements written in lp-language guarantee that the tasks X and Y will not overlap if they are executed on the same processor:

$$s(X_0) - (s(Y_0) + e(Y)) \geq -b_{XY} \quad (10)$$

$$s(Y_0) - (s(X_0) + e(X)) \geq -(1 - b_{XY}) \quad (11)$$

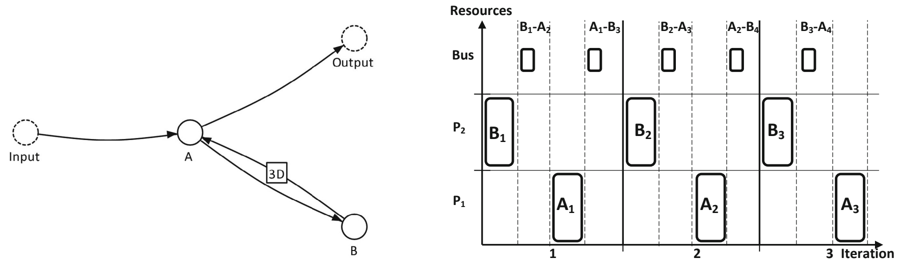
When $b_{XY} = 0$, statement 10 enforces Y to be executed before X. Consequently, statement 11 is true and does not affect the variables because $s(Y_0) - (s(X_0) + e(X)) \in [-1, 1]$. Analogue is the case when $b_{XY} = 1$.

Example 2. Regarding the DFG in Fig. 4(a), we add the necessary information for optimizing the schedule. The worst case execution time in iterations $e(X), \forall X \in V$ and the worst case transmission time are given. We define $e(A) = e(B) = 0.25$ iterations and $tr(A, B) = tr(B, A) = 0.1$ iterations. Similar as in the data-flow optimizing case, the equations as defined by Eq. 5 are computed. There is one path, which covers the graph:

$$a_{IA} + a_{AB} + a_{BA} + a_{AO} = 3 \quad (12)$$

The statements, which describe the precedence constraints of the graph are:

$$\begin{aligned} s(A_0) - (s(B_0) + e(B)) - tr(A, B) \;&\geq\; -\delta(A, B) \\ s(B_0) - (s(A_0) + e(A)) - tr(B, A) \;&\geq\; -\delta(B, A) \end{aligned}$$



(a) (b)

Fig. 4. The data-flow model of a system S and its optimal schedule: (a) original DFG; (b) optimal schedule regarding robustness $\rho_{opt}(S)$

Since there are no two tasks running on the same processor without precedence constraints, we do not have to deal with overlapping problems.

In order to optimize the robustness against communication latencies the objective function needs to maximize the expressions $s(B_0) - (s(A_0) + e(A)) - tr(A, B) + \delta(A, B)$ and $s(A_0) - (s(B_0) + e(B)) - tr(B, A) + \delta(B, A)$ (see [7]).

The optimal robust system schedule is shown in Fig. 4(b). It is the result of the previously presented linear expressions computed by lp_solve. The optimal robustness against communication latencies is 1.15 of an iteration. For achieving this robustness the system is retimed such that $\delta(A, B) = 2$ and $\delta(B, A) = 1$. Moreover, the linear system produces a solution for the starting times: $s(A_0) = 0.5$ and $s(B_0) = 0$. Hence, the starting times in each iteration compute as follows:

$$s(A_i) = s(A_0) + i = 0.5 + i$$
$$s(B_i) = s(B_0) + i = i$$

4 Automatic Analysis and Optimization with Cadmos

The previous sections provide the theoretical background to the concepts. This section is concerned with the implementation of the concepts in the tool Cadmos, which can be used to visualize, analyze and transform data-flow graphs and is developed at Chair IV for Software & Systems Engineering of the Technische Universität München (see [11]). The presented analysis and transformation techniques for robustness optimization are implemented in this tool. Tool-supported automatic analysis and optimization is useful for industrial relevant system scales where manual analysis is time-consuming, error-prone or impractical.

Analyzing the robustness against communication latencies. The first step when trying to improve the robustness of the system is to analyze whether the current robustness is sufficient. Cadmos allows to do this with the computing of the function $\rho(S)$ (see Sec. 2). Fig. 5(a) shows the DFG of a system with

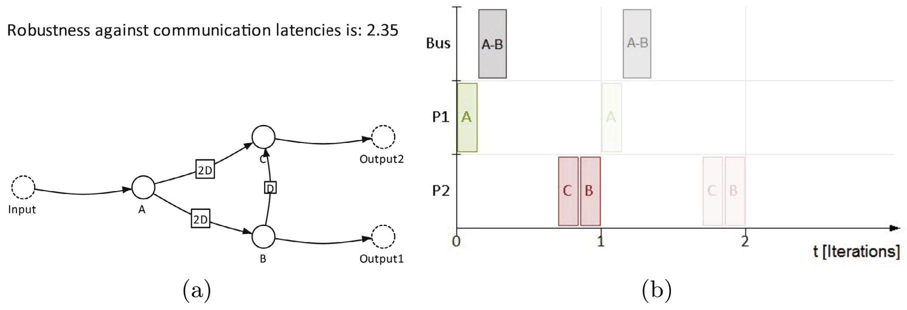


Fig. 5. The data-flow model of a system and its optimal schedule: (a) data-flow; (b) optimal schedule regarding robustness against communication latencies as generated by the tool Cadmos.

three computational nodes. The platform contains two processors $P1, P2$, which communicate through a bus. Task A is mapped on processor $P1$ and tasks B, C are mapped on $P2$. The robustness against communication latencies of this system can be read above the DFG. It is 2.35 in multiples of iterations. Hence, if all the messages on the bus are delayed less than $2.35 \cdot T$, the system will produce the correct outputs, where T represents the actual duration of an iteration in physical time. For example, with $T = 100ms$ we get a robustness of $2.35 \cdot 100ms = 235ms$.

Optimizing the robustness against communication latencies. If the analyzed robustness is not sufficient for the system, we have the possibility to optimize it regarding the communication latencies. Fig. 5 presents a system which has reached its optimal robustness against communication latencies. Again, the platform consists of two ECUs connected by a bus and A is executed on one ECU while B and C are executed on the other ECU. The worst case execution time of all nodes is $e(X) = 0.15$ iterations and the worst case transmission time of all channels on the bus is $tr(X, Y) = 0.2$ iterations. For solving the set of inequalities presented in 3.2, the linear problem solver lp_solve [12] is used. The result is a system with $\rho_{opt}(S) = 2.35$ iterations. A schedule with this robustness is illustrated in Fig. 5(b).

5 Evaluation

In this section, the concepts are evaluated by applying them to an industrial relevant system. We use a model of an *Adaptive Cruise Control* (ACC) which is a result of the Dentum research project of the Chair for Software and System Engineering of the Technische Universität München in cooperation with an international automotive supplier [13]. In this section, we evaluate the results of the communication latencies robustness analysis and optimization using a more complex model in order to show the practicability of these concepts.

ACC functionality and structure. An Adaptive Cruise Control is a system used in the automotive industry for vehicle speed control. It maintains a constant speed unless the distance to the next vehicle is to low. By deceleration or even active breaking, the ACC system preserves a required minimal distance depending on the current vehicle speed. The system also contains a *pre-crash safety* (*PCS*) node. If the *PCS* detects that a collision is about to happen, it suspends any acceleration, tights the seat belt and breaks actively. The *PCS* has the responsibility to suspend any acceleration by sending the value "true" over the channel *suspend* to the *OnOffArbiter* if a forthcoming collision is detected. Hence, the *suspend* channel is considered a *critical* channel of the system, which should have a high robustness against latencies on a bus. Fig. 6 shows the overall data-flow graph $G = (V, E, \delta)$ and emphasizes the data-flow from the *PCS*, over the *suspend* channel to the *OnOffArbiter* (as marked by text boxes).

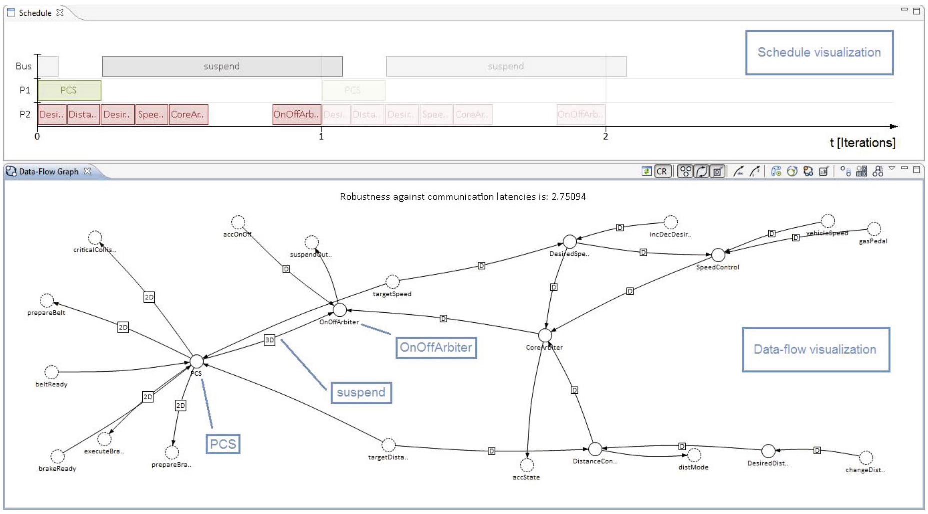

Robustness against communication latencies is: 2.75094

Fig. 6. Using the tool CADMOS to analyze and optimize the data-flow and the schedule of the Adaptive Cruise Control model. A robustness-optimal schedule of the ACC system is visualized on the upper side and the optimized data-flow graph is visualized on the lower side.

The platform resources are two ECUs of type MPC5554 ($P = \{P_1, P_2\}$) connected through a CAN-bus ($B = \{Bus\}$). The execution times of nodes on the ECUs as well as the transmission times of messages on the bus are known. The mapping M of this system contains the *PCS* node running on ECU P_1 while all the other nodes run on the other ECU P_2. The *suspend* channel, that forwards messages from the *PCS* to the *OnOffArbiter*, is the only channel mapped on the CAN-bus *Bus*. In the following paragraphs, we use the concepts presented in this paper to maximize the robustness of the *suspend* channel in the ACC system.

Optimizing the robustness against communication latencies. The robustness optimization of the ACC model is realized automatically with the implemented methods (see Sec. 4) in CADMOS. Fig. 6 shows the modified system with its optimal data-flow graph and schedule. The new DFG is the result of a series of retiming transformations. Here, the *suspend* channel, which is mapped on the bus, obtains the maximum number of three delays. Taking a closer look at the schedule in Fig. 6, we observe that inside of an iteration the sending *PCS* node is executed as *early* as possible, whereas the receiving *OnOffArbiter* node is executed as *late* as possible. CADMOS assures that this system is robust as long as a communication latency on the CAN-bus does not exceed 2.75094 multiples of an iteration. If we use an iteration period $T = 5ms$, the ACC system is guaranteed to be robust against latency caused by the CAN-bus of up to $2.75094 \cdot 5ms = 13.7547ms$.

CADMOS generates the respective lp-scipt automatically. The generated script of the ACC model, has 73 lines of code written in linear programming language and includes 48 variables. For the ACC model, which only comprises 7 nodes, a manual attempt to implement this linear program, using the algorithm presented in Sec. 3.2, would already be very time-consuming. The time required by lp_solve to find the optimal solution with the generated script is relatively low. For the ACC model, the optimal result is produced in 0.03 seconds on an Intel(R) Core(TM)2 Duo CPU with 2.4GHz. It is future work to evaluate how these results scale on more complex models with hundreds of nodes and channels.

6 Concluding Remarks and Future Work

Robustness against communication latencies is an important property of distributed reactive systems. After analysis of the robustness $\rho(S)$ as presented in Sec. 2, engineers can decide if the robustness of a system S needs to be improved. The optimization methods in Sec. 3 provide the maximum communication latency robustness $\rho_{opt}(S)$, which can be achieved for a system $S = (G, R, M)$ regarding its data-flow graph G, platform resources R and mapping M.

Threats to validity. One result of the optimization is a retimed DFG. Retiming also involves the re-computation of initial values of channels. Our approach currently requires that the initial values are manually defined, after the system has been retimed. Otherwise, computing the initial values from the original system can be challenging. If delays are moved from inputs towards outputs of a node X, the new initial values val'_i can be computed by applying the functionality of X on its old initial values val_i. ($val'_i = X(val_i)$). The difficult case is when delays are moved from outputs towards inputs. This means that for the computation of the new initial values the inverse function X^{-1} must be applied on the old initial values ($val'_i = X^{-1}(val_i)$). In general, finding the inverse function (if it exists) is not trivial.

Trade-offs for optimizing the communication latency robustness. Achieving the optimal robustness against communication latencies can have severe negative influence on other important properties of the system like performance or parallelism. It is future work to find a possibility to have a weighted optimization method with respect to these properties. Nevertheless, the optimization methods in Sec. 3 give insight, whether the system can ever be robust enough to the expected communication latencies of the platform. If even the optimal robustness is not sufficient to cover worst case scenarios, engineers should consider adding more delays or changing parts of the platform or changing the mapping of data-flow elements onto resources.

Acknowledgements. This work was partially funded within the project ARAMiS by the German Federal Ministry for Education and Research with the funding IDs 01IS11035.

References

1. Harel, D., Pnueli, A.: On the development of reactive systems, pp. 477–498. Springer-Verlag New York, Inc., New York (1985)
2. Davis, A., Keller, R.: Data flow program graphs. Computer 15(2), 26–41 (1982)
3. Lee, E., Messerschmitt, D.: Synchronous data flow. Proceedings of the IEEE 75(9), 1235–1245 (1987)
4. Parhi, K.: Algorithm transformation techniques for concurrent processors. Proceedings of the IEEE 77(12), 1879–1895 (1989)
5. Sriram, S., Bhattacharyya, S.S.: Embedded Multiprocessors: Scheduling and Synchronization, 1st edn. Marcel Dekker, Inc., New York (2000)
6. Brock, J., Ackerman, W.: Scenarios: A model of non-determinate computation. In: Díaz, J., Ramos, I. (eds.) Formalization of Programming Concepts. LNCS, vol. 107, pp. 252–259. Springer, Heidelberg (1981)
7. Leiserson, C., Rose, F., Saxe, J.: Optimizing synchronous circuitry by retiming. In: Third Caltech Conference on Very Large Scale Integration, pp. 87–116. Computer Science Press, Incorporated (1983)
8. Leiserson, C., Saxe, J.: Retiming synchronous circuitry. Algorithmica 6(1), 5–35 (1991)
9. Chao, L.F., Sha, E.H.M.: Scheduling data-flow graphs via retiming and unfolding. IEEE Trans. Parallel Distrib. Syst. 8, 1259–1267 (1997)
10. Broy, M.: Relating time and causality in interactive distributed systems. European Review 18(04), 507–563 (2010)
11. Schwitzer, W., Popa, V., Chessa, D., Weissenberger, F.: Cadmos - A concurrent architectures toolkit (2012), http://code.google.com/p/cadmos/ (accessed April 03, 2012)
12. LP_Solve Developers: LP_Solve Website (2012), http://lpsolve.sourceforge.net/ (accessed April 03, 2012)
13. Feilkas, M., Fleischmann, A., Hölzl, F., Pfaller, C., Scheidemann, K., Spichkova, M., Trachtenherz, D.: A Top-Down Methodology for the Development of Automotive Software. Technical report, Technische Universität München (2009)

A Formal Design of a Tool for Static Analysis of Upper Bounds on Object Calls in Java[⋆]

Konrad Siek and Paweł T. Wojciechowski

Poznań University of Technology
{Konrad.Siek,Pawel.T.Wojciechowski}@cs.put.poznan.pl

Abstract. This paper presents a formal design of a tool for statically establishing the upper bound on the number of executions of objects' methods in a fragment of object-oriented code. The algorithm that our tool employs is a multi-pass interprocedural analysis consisting of data flow and region-based analyses. We describe the formalization of each of stage of the algorithm. This rigorous specification greatly aids the implementation of the tool by removing ambiguities of textual descriptions. There are many applications for information obtained through this method including reasoning about concurrent code, scheduling, code optimization, compositing services, etc.We concentrate on using upper bounds to instrument transactional code that uses a synchronization mechanism based on versioning, and therefore benefits from *a priori* knowledge about the usage of shared objects within each transaction. To this end we implement a precompiler for Java that analyzes transactions, and injects generated source code to initialize each transaction.

Keywords: static analysis, data flow analysis, transactional memory.

1 Introduction

In this paper we present a tool for estimating the maximum of how many times methods of objects will be called within a fragment of object-oriented code. We report the tool's formalization and discuss its implementation. We see the formalization as one of the main contributions of this paper. The described algorithm is relatively simple: it is based on data flow analysis to establish information about values and paths and region analysis to tally method calls. We expand regions with additional properties so that the final, vital part of the analysis becomes straightforward. We also describe a use of a natural positive set extended by an absorbing value to count uncertain executions. In effect we manage to infer upper bounds (either concrete or infinite) that provide safety.

There are several possible applications for information about the upper bound on the number of objects' method calls obtained through our analysis. Among

⋆ This work has been partially supported by the Polish Ministry of Science and Higher Education within the European Regional Development Fund, Grant No. POIG.01.03.01-00-008/08.

others, such upper bounds may be used to analyze concurrent code to find relationships between threads: a thread accessing a shared object once, several times, or not at all may impact safety guarantees like isolation or performance in different ways. With this information prior to execution it can be treated differently, i.e., applied proper synchronization, delayed, or executed as-is without breaking guarantees. The upper bounds can also be applied in compile-time resource optimization. For instance, the amount of memory used by a given program or its influence on network traffic may be estimated from calls to particular objects if the interface is known and used to configure the environment appropriately or to optimize the analyzed program. Other uses may be found in code rewriting, automatic refactoring, etc. Apart from the work of [15] these applications seem largely unexplored.

Our particular interest lays with algorithms that require this type of information up front for efficient operation, e.g., those found in scheduling and synchronization via transactions. One such application is Atomic RMI [26,27]. It is a distributed transactional memory extension to Java RMI, an API for distributed programming using remote procedure calls that is well-established in business and industry. Atomic RMI uses versioning algorithms that need *a priori* information about shared object accesses to figure out whether an object may be released before a transaction commits (or aborts). Releasing objects early gives Atomic RMI a performance edge, so a precompiler that provides upper bounds automatically and precisely has significant practical value. We see the paper as contributing the application and the implementation of the precompiler as well as the analysis and its formalization. The technical documentation of the tool is available on the project web page [19].

The paper has the following structure. In Section 2 we present work similar to ours. We formalize the static analysis in Section 3— each subsection describes an individual constituent part of the analysis. Then, in Section 4 we discuss the precompiler implementation. Finally, we conclude with Section 5.

2 Related Work

There is a large body of research related to analysis of programs that aims at deriving information about execution patterns statically (we sketch out some of these below). However, we do not know examples of using this information for optimizing the execution of distributed transactions in the way we do. The largest body of work to which our static analysis bears resemblance has been done with regard to the Worst Case Execution Time (WCET) problem [24]—establishing upper bounds on the time code takes to run. However, most of this work is aimed at real-time systems, not transactional concurrency control which is the main concern of our work. A number of frameworks are available for WCET analysis, like aiT [4], Bound-T [10], SWEET [7], and SymTA/P [20]. A comprehensive survey of these tools and methods was done in [25]. Whereas our approach is based on region analysis, some work in WCET use symbolic analysis [13], path analysis [8], and abstract interpretation [10,5]. We also use the latter type of analysis for our value analysis algorithm (Section 3.2). In WCET emphasis is placed on the problem of evaluating loops in general and bounding loop iterations in particular. This is done, among others, by the use of Presburger arithmetic [17], path analysis (using integer linear programming) [21], or a combination of methods involving abstract interpretation [3]. Our work touches on those concerns in Section 3.2 where we use loop unfolding to establish their bounds roughly similar to that of SWEET [7] but simpler. WCET tools additionally often use the Implicit Path Enumeration technique [12] or single feasible paths [28] to establish worst-case paths and perform final timing analyses. While our application presents no need for the latter, we use region-based analysis (described in Section 3.4) to conservatively deduce worst-case paths. WCET tools also allow for manual declaration or correction of difficult-to-deduce information (e.g., loop bounds).

Our work has significant similarities to work on lock inference. Lock inference aims to determine which memory locations or shared objects in a program must be protected by locks and where these locks should be located. Thus, our work and lock inference share the same ultimate goal of providing concurrency control via static analysis albeit by different mechanisms. The authors of [2] employ backward data flow analysis to transform a program's control flow graph into a path graph which is then used to derive locks. In [9] the authors present a method for identifying shared memory locations using type-based analysis, points-to analysis, and label flow analysis [16]. In Autolocker [14] pessimistic atomic sections are converted into lock-guarded critical sections by analyzing dependencies among annotated locks based on a computation history derived from a transformation of the code using a type system.

In [15] the authors propose a tool for the automatic inference of upper bounds on the usage of user-specified resources. Rather than memory or execution time, these may be the number of open files, accesses to database, sent text messages, etc.This work and ours share the set of tools they use (Soot and Jimple [22]) and they both try to solve a similar problem. The tool presented by the authors performs a data flow analysis to derive data dependencies, then creates a set of equations from input-output parameter size relationships. Finally the equations are solved using a recurrence solver. Our approach differs most in that we perform region analysis to determine maximum paths and resource use where they construct and solve equations.

3 Upper Bound Prediction Analysis

In this section we describe an algorithm for deriving upper bounds (or suprema) on the number of method calls to objects via static analysis. The upper bounds for some specific objects—remote objects used in a transaction—are used for concurrency control by Atomic RMI. To derive the suprema the algorithm performs multiple passes over the input code in the form of an intermediate language (see Section 3.1). Three passes correspond to the three phases that form our algorithm: value analysis, region analysis, and call count analysis. In addition another pass is performed before value analysis to identify loops. Value analysis

Identifiers	$j \in Ident$	
Constants	$c \in Const$	
Labels	$l \in Lab$	
Types	$t \in Type$	
Fields	$f \in Field$	$::= j : t$
Immediates	$i \in Imed$	$::= j \mid c$
Right-hand values	$r \in Rval$	$::= i \mid i[i] \mid i.[f] \mid [f]$
Methods	$m \in Meth$	$::= \texttt{invoke}\ i.[j(j_1, ..., j_n)](i_1, ..., i_n)\{b_1, ..., b_n\}$
Conditions	$p \in Cond$	$::= i == i \mid i \geq i \mid i > i \mid i \leq i \mid i < i \mid i \neq i$
Expressions	$e \in Expr$	$::= i + i \mid i \mathbin{/} i \mid i * i \mid i\%i \mid -i \mid i - i \mid i \,\vert\, i \mid i \,\&\, i$
		$\mid i\ \texttt{xor}\ i \mid i \gg i \mid i \ll i \mid (t)i \mid i\ \texttt{instanceof}\ t$
		$\mid \texttt{new}\ t \mid \texttt{new}\ t[i_1]...[i_n] \mid \texttt{length}\ i \mid p$
Statements	$s \in Stmt$	$::= \texttt{switch}(i)\{\texttt{case}\ c_1 : l_1; ...; \texttt{case}\ c_n : l_n; \texttt{default}:\ l_0\}$
		$\mid \texttt{if}\ p\ \texttt{goto}\ l_1\ \texttt{else}\ l_2 \mid l \mid j = m \mid j = r \mid m$
		$\mid \texttt{goto}\ l \mid \texttt{return}\ i$
Blocks	$b \in Bloc$	$::= l : b_1; ...; b_n; \mid b_1; ...; b_n; \mid s$

Fig. 1. Jimple syntax (altered)

predicts possible values of variables in the code. It also identifies unfeasible or dead code, and unfolds loops. Region analysis uses the results of value analysis to convert the input code into regions. Finally, call count analysis examines these regions to produce the upper bounds on method call counts. We describe our use of Jimple and the phases of the algorithm in detail in the following subsections.

3.1 Translation to Jimple

In order to analyze a program in Java with Atomic RMI transactions we translate it into an intermediate representation called Jimple [23] using the Soot framework [22]. We use Jimple as an intermediate language because it is much better suited for analysis than either Java source code or bytecode. The reason for this is that Jimple is a 3-address code representation with a very limited instruction set consisting of 17 statements. In our earlier attempts to perform similar analyses using Java source code [18] we learned that such analyses become convoluted and the implementation costly in effort due to the number of constructs needing handling and the complexity of their semantics.

The part of Jimple syntax that is pertinent to our further discussion is presented in Fig. 1. The semantics are mostly straightforward, the reader is referred to [23] for details and the complete language. The constructs most important to us are the conditional statements, switch statements, method invocations, assignments, and labeled blocks. We introduce superficial alterations to the syntax to suit further description of the algorithm. We treat labels as statements and place them at the beginning of labeled blocks. We modify the conditional statement to define target labels for both outcomes instead of having a succeeding

```
 1  Transaction k = new Transaction(reg);
 2  a = k.accesses(a, 2); // generated, upper bound = 2
 3  b = k.accesses(b, 1); // generated, upper bound = 1
 4  k.start();
 5  int balance = a.getBalance();
 6  if (balance >= sum) {
 7      a.withdraw(sum); b.deposit(sum);
 8      k.commit();
 9  } else
10      k.rollback();
```

Fig. 2. Example Java code for a distributed transaction using Atomic RMI

```
 1    k = new soa.atomicrmi.Transaction;
 2    invoke k.[<init>(@parameter0)](reg){$b0};
 3    invoke k.[start()](){$b1};
 4    balance = invoke a.[getBalance()](){$b4};
 5    if balance < sum goto label1 else label0;
 6  label0:
 7    invoke a.[withdraw(@parameter0)](sum){$b5};
 8    invoke b.[deposit(@parameter0)](sum){$b6};
 9    invoke k.[commit()](){$b2}; return null;
10  label1:
11    invoke k.[rollback()](){$b3};
```

Fig. 3. Java code translated to altered Jimple

block of code called if the condition is false. We do not distinguish among different sorts of method invocations—interface, special, virtual, and static—and we remove type information from invocations while adding a direct definition of the methods' arguments and a set of possible bodies. We also fix method invocations nested in other statements by defining a separate assignment statement instead where the results of the invocation are assigned to an identifier. We show an example Java program using our Atomic RMI distributed transactions Fig. 2 translated to the altered form of Jimple in Fig. 3 (lines 2, 3 are omitted because they are generated from Jimple later—see Section 4.2 for details).

For the purposes of analysis the input program is represented as *Control Flow Graphs* (CFGs) and each method's body is a separate graph. Most statements in Jimple will have one incoming and outgoing edge. The conditional statement will have 2 outgoing edges, and the `switch` statement will have one more outgoing edge than it has conditions. Loop headers and labeled blocks will have more incoming edges. Invoke statements point to CFGs of other method bodies.

3.2 Value Analysis

As a preliminary to the value analysis we find loops in code. A loop consists of a head and a body. A loop head is a statement s that dominates any other statement s' (all paths from the start to s' lead through s [1], denoted s **dom** s') while simultaneously being the successor of s' (there is a path from s' to s). A loop body is a sequence of statements all of which are dominated by a loop head and have that loop head as their successor. We gather the heads in set $\mathbb{H}$

$$\mathbb{G}(s) \triangleq \mathbb{S} \triangleq (\mathbb{S}_V, \mathbb{S}_P, \mathbb{S}_D, \mathbb{S}_I)$$
$$\mathbb{G}(s) = \mathsf{eval}(\mathsf{join}(\{\mathbb{G}(p) \mid s\ \mathbf{succ}\ p\}), s)$$
$$\mathsf{eval}(\mathbb{S}, j = r) \triangleq (\mathbb{S}_V[j \mapsto \{\mathsf{val}(r, \mathbb{S}_V)\}], \mathbb{S}_P, \mathbb{S}_D, \mathbb{S}_I)$$
$$\mathsf{eval}(\mathbb{S}, j = m) \triangleq \mathbb{S}' = \mathsf{eval}(\mathbb{S}, m), (\mathbb{S}_V \oplus \mathbb{S}'_V[j \mapsto \{\mathsf{val}(m, \mathbb{S}_V)\}], \mathbb{S}_P, \mathbb{S}_D, \mathbb{S}_I)$$
$$\mathsf{eval}(\mathbb{S}, \texttt{invoke}\ \ i.[j(j_1, ..., j_n)](i_1, ..., i_n)\{b_1, ..., b_m\}) \triangleq$$
$$\quad \text{case } \mathsf{depth}(i.j) \to \mathbb{S}_V[k \mapsto \omega k \in \mathsf{defs}(b_1) \cup ... \cup \mathsf{defs}(b_m)], \mathbb{S}_P, \mathbb{S}_D, \mathbb{S}_I)$$
$$\quad \text{otherwise} \to \mathbb{S}' = (\mathbb{S}_V[j_1 \mapsto \mathsf{val}(i_1, \mathbb{S}_V), ..., j_n \mapsto \mathsf{val}(i_n, \mathbb{S}_V)], \mathbb{S}_P, \mathbb{S}_D, \mathbb{S}_I),$$
$$\qquad \mathsf{join}(\mathsf{eval}(\mathbb{S}', b_1), ..., \mathsf{eval}(\mathbb{S}', b_m))$$
$$\mathsf{eval}(\mathbb{S}, l) \triangleq (\mathbb{S}_V \oplus \mathbb{S}_P(l), \mathbb{S}_P, \mathbb{S}_D, \mathbb{S}_I)$$
$$\mathsf{eval}(\mathbb{S}, s : \texttt{return}\ i) \triangleq (\mathbb{S}_V, \mathbb{S}_P, \mathbb{S}_D \cup \{(s, s') \mid s\ \mathbf{pdom}\ s', s' \in \mathit{Stmts})\}, \mathbb{S}_I)$$
$$\mathsf{eval}(\mathbb{S}, s : \texttt{if}\ p\ \texttt{goto}\ l_1\ \texttt{else}\ l_2) \triangleq$$
$$\quad \text{case } \mathsf{pred}(p, \mathbb{S}) = \texttt{true} \to (\mathbb{S}_V, \mathbb{S}_P\big[l_1 \mapsto \mathbb{S}_P[p \mapsto \texttt{true}]\big], \mathbb{S}_D \cup \{(s, l_2)\}, \mathbb{S}_I)$$
$$\quad \text{case } \mathsf{pred}(p, \mathbb{S}) = \texttt{false} \to (\mathbb{S}_V, \mathbb{S}_P\big[l_2 \mapsto \mathbb{S}_P[p \mapsto \texttt{false}]\big], \mathbb{S}_D \cup \{(s, l_1)\}, \mathbb{S}_I)$$
$$\quad \text{case } \mathsf{pred}(p, \mathbb{S}) = \omega \to (\mathbb{S}_V, \mathbb{S}_P\big[l_1 \mapsto \mathbb{S}_P[p \mapsto \texttt{true}], l_2 \mapsto \mathbb{S}_P[p \mapsto \texttt{false}]\big], \mathbb{S}_D, \mathbb{S}_I)$$
$$\mathsf{eval}(\mathbb{S}, s : \texttt{switch}(i)\{\texttt{case}\ c_1 : l_1; ...; \texttt{case}\ c_n : l_n; \texttt{default:}\ \ l_0\}) \triangleq (\mathbb{S}_V,$$
$$\quad \mathbb{S}_P\big[l_1 \mapsto \mathbb{S}_P(l_1)[j = \mathsf{val}(c_1)], ..., l_n \mapsto \mathbb{S}_P(l_n)[j = \mathsf{val}(c_n)]\big], \mathbb{S}_D$$
$$\quad \cup\{(s, l_k) \mid \mathsf{pred}(c_k = j, \mathbb{S}) = \texttt{false} \vee \mathsf{pred}(c_r = j, \mathbb{S}) = \texttt{true}, k = 1, ..., n, r = 1, ..., k\}$$
$$\quad \cup\{l_0 \mid \mathsf{pred}(\exists k, c_k = j, \mathbb{S}) = \texttt{true}, k = 1, ..., n\}, \mathbb{S}_I)$$
$$\mathsf{eval}(\mathbb{S}, s \in \mathbb{H}) \triangleq \mathsf{evalloop}(s, \mathbb{G}, \mathbb{L}(\mathsf{id}(s)), 1, L)$$
$$\mathsf{eval}(\mathbb{S}, s \in \mathbb{B} \wedge s \notin \mathbb{H}) \triangleq \mathbb{S}$$
$$\mathsf{join}(\mathbb{S}^1, ..., \mathbb{S}^n) \triangleq \big(\{k \mapsto \mathbb{S}^1_V(k) \cup ... \cup \mathbb{S}^n_V(k) \mid k \in (\mathrm{dom}\ \mathbb{S}^1_V \cup \mathrm{dom}\ \mathbb{S}^n_V)\}, \{l \mapsto$$
$$\quad \{k \mapsto \mathbb{S}^1_P(l)(k) \cup ... \cup \mathbb{S}^n_P(l)(k)\} \mid l \in \mathrm{dom}\ \mathbb{S}^1_P \cup ... \cup \mathbb{S}^n_P, k \in \mathrm{dom}\ \mathbb{S}^1_P(l) \cup ... \cup \mathbb{S}^n_P(l)\},$$
$$\quad \mathbb{S}^1_D \cup ... \cup \mathbb{S}^n_D, \{k \mapsto \max_{i=1,...,n}(\mathbb{S}^i_I(k)) \mid k \in (\mathrm{dom}\ \mathbb{S}^1_I \cup \mathrm{dom}\ \mathbb{S}^n_I)\}\big)$$
$$\mathbb{S}'_V \oplus \mathbb{S}''_V \triangleq \{k \mapsto \mathbb{S}'_V(k) \cup \mathbb{S}''_V(k) \mid k \in (\mathrm{dom}\ \mathbb{S}'_V(k) \cup \mathrm{dom}\ \mathbb{S}''_V(k))\}$$

Fig. 4. Value analysis

and create map $\mathbb{L}$ which contains a unique identifier of each statement h from $\mathbb{H}$ as a key mapped to a set of statements whose elements are all dominated by h.

The first phase of the analysis is a forward data flow analysis performed on the CFG. Its main purpose is threefold: to establish the possible values of variables at each node of the CFG representing the program, to count the maximum number of loop iterations through loop unfolding, and to establish which nodes of the CFG are dead or unfeasible (will not be executed). There are two principal functions in value analysis, eval and join. These functions are used to compute members of global state $\mathbb{G}$, a data structure that results from the analysis. We present all of those elements in Fig. 4 and describe them below.

Global state $\mathbb{G}$ maps Jimple statements to states which apply to them. Global state is constructed during value analysis by constructing a state for each statement using a transfer function eval and an aggregation of states for the predecessors of a given statement using join. We designate individual states $\mathbb{S}$, such that $\mathbb{S}$ is a quadruple consisting of a value map $\mathbb{S}_V$, an inferred value map $\mathbb{S}_P$, a dead edge set $\mathbb{S}_D$, and a loop iteration map $\mathbb{S}_I$. $\mathbb{S}_V$ is a map of locals (identifiers and constants) to sets of values—it indicates what values a given variable or constant may take at this point in the program. $\mathbb{S}_P$ maps labels (names of blocks) to value maps and indicates assumptions about values of variables and constants inferred from conditions that will apply at a particular succeeding statement. $\mathbb{S}_D$ contains pairs of statements indicating edges that will definitely not be used in the execution of the program. $\mathbb{S}_I$ is a map of loop heads to numbers indicating the maximum estimated iteration count of the loop, or an unknown value. All components of the state are initially empty.

Transfer function eval is the key function of the analysis. It analyzes each Jimple statement and establishes the state of the program that holds after the statement is evaluated. The resulting state depends on the type of statement and the state before that statement.

When encountering an assignment of a right-hand side expression r to an identifier j, a new mapping is added to $\mathbb{S}_V$ that maps j to the set of possible values of expression r. When eval encounters an assignment of the results of method invocation m to identifier j, first m is evaluated separately and state after its evaluation $\mathbb{S}'$ is extended by the mapping of j to the result of m. A method invocation itself is analyzed by first extending the value map by parameter identifiers mapped to the values of arguments. Then all possible bodies are evaluated and the results are joined (the particular bodies are identified from the type hierarchy and arguments but we leave the details to Soot). But if recursion exceeds a depth L all the values defined within possible method bodies are set to unknown (this degrades precision but maintains safety). L must be tuned to a given application. A label l extends the value map with predictions from the inferred map. A `return` statement adds all other statements it dominates to $\mathbb{S}_D$.

When analyzing an `if` statement the expression that is the condition is checked. If the condition yields true then the edge in the CFG from the current statement to label l_2 is added to dead edges, and predictions about variables are made under the assumption that the condition will be true at label l_1. Conversely, if the condition yields false the edge from the statement to l_1 will be dead and predictions will be made for l_2 under the assumption that the condition is false. If the condition yields an unknown, no edges will be added to the dead edge set, but predictions for both l_1 and l_2 will be made. A `switch` statement is analyzed by creating a prediction for each constant $c_1, ..., c_n$ that the local i is equal to it at an appropriate label $l_1, ..., l_n$. Furthermore, if any of the constants c_k is definitely equal to i, edges from this statement to labels subsequent to that constant $l_{k+1}, ..., l_n$ and the default label l_0 are added to the dead edge set $\mathbb{S}_D$.

Function join (Fig. 4) is responsible for joining states and is used when a statement has two or more incoming edges. Each component of the state is joined with its counterpart in the second state. Sets $\mathbb{S}_D$ are added together. The keys and values are copied to a new map, and if a key is present in both maps, the values are added ($\mathbb{S}_V$, $\mathbb{S}_P$) or the higher one is selected ($\mathbb{S}_I$).

We use the following helper functions within eval. Function val substitutes values from a value map for identifiers and constants (where possible) in a given expression and evaluates it to establish a set of values that the expression may yield. The returned set may consist of a single value, any number of elements or contain the unknown value ω. We use the function pred in a similar manner,

$$
\begin{aligned}
&\mathsf{evalloop}(s, \mathbb{G}', \mathbb{U}, i, L) \triangleq \\
&\quad \mathbb{G}'' = \mathbb{G}', \quad \mathbb{G}''(u) = \mathsf{eval}(\mathsf{join}(\{\mathbb{G}''(u) \mid u \ \mathbf{succ} \ p \wedge u \in \mathbb{U}\})), \\
&\quad \mathbb{E} = \{e \mid s \ \mathbf{succ} \ e \wedge s \notin \mathbb{U} \wedge e \in \mathbb{U}\}, \\
&\quad \mathbb{E}' = \mathbb{E} \setminus \{d \mid \mathbb{G}''(d) = \mathbb{S}', \mathsf{unpredecessed}(\mathbb{S}'_D, d) \wedge d \in \mathbb{E}\}, \\
&\quad \mathbb{S}^e = \mathsf{join}(\{\mathbb{G}''(e) \mid e \in \mathbb{E}'\}), \\
&\quad \mathbb{Z} = \{(b, h) \mid h \ \mathbf{dom} \ b \wedge h \ \mathbf{succ} \ b \wedge \nexists s \in \mathbb{U}, h \ \mathbf{succ} \ s \ \mathbf{succ} \ b\} \\
&\quad \mathbb{S}^z = \mathsf{join}(\{\mathbb{G}''(b) \mid (b, h) \in \mathbb{Z}\}), \\
&\quad \text{case } \mathbb{Z} \subseteq \mathbb{S}^z_D \vee (\forall (b, h) \in \mathbb{Z}, \ \mathsf{unpredecessed}(\mathbb{S}^z_D, b)) \rightarrow (\mathbb{S}^e_V, \mathbb{S}^e_P, \mathbb{S}^e_D, \mathbb{S}^e_I[h \mapsto i]) \\
&\quad \text{case } i > L \rightarrow (\mathbb{S}^e_V[k \mapsto \omega, k \in \mathsf{defs}(\mathbb{U})], \mathbb{S}^e_P, \mathbb{S}^e_D, \mathbb{S}^e_I[h \mapsto \omega]) \\
&\quad \text{case } i \leq L \rightarrow \mathsf{evalloop}(h, \mathbb{G}'', \mathbb{U}, i + 1, L) \\
&\mathsf{unpredecessed}(\mathbb{S}_D, s) \triangleq \forall s \ \mathsf{succ} \ p, \ (p, s) \in \mathbb{S}_D \vee \mathsf{unpredecessed}(p) \\
&\mathsf{defs}(\mathbb{U}) \triangleq \{j \mid s \in \{j = m, j = r\} \wedge s \in \mathbb{U}\}
\end{aligned}
$$

Fig. 5. Loop unfolding within value analysis

except that only conditional expressions are evaluated and a single ternary value is returned—`true`, `false`, or ω. We use depth to find out the depth of a method's recursion. Function id produces a unique identifier of a statement. Operators **succ**, **dom** and **pdom** denote the succession, domination and post-domination relation of two statements in the CFG.

When encountering a statement that was identified as a head of a loop, function evalloop is used where the statements that form the body of the loop are taken from $\mathbb{L}$ and evaluated. During evaluation a collection of states $\mathbb{G}'$ is created and used to find those exit statements $\mathbb{E}'$ and back edges $\mathbb{Z}$ that may be executed during this iteration. If no back edge could be used during this iteration we know the loop exits, so we aggregate the states after all exit statements and finish evaluating the loop. It can also be deduced at this point that the loop will be executed at most as many times as we performed iterations. Otherwise, if we have not reached an arbitrary limit of iterations we conduct another iteration using evalloop. If the limit was reached we do not proceed but assume that this loop will continue indefinitely and set all the values that are defined within its body to unknown ω. Upon evaluation exit statements from the loop body are derived from the dead edge set of the resulting state. If there is only one exit from the loop then the loop exits in the current iteration and both the state of the variables and the number of iterations are added to $\mathbb{S}$. Otherwise another iteration is required and the evaluation is repeated. In order to manage infinite loops or those where the conditions of exiting are uncertain, an iteration limit L is given which, when reached, will cause the evaluation to cease and set all effects of the loop to unknown value ω. Setting values to ω preserves safety. We use two additional helper functions within evalloop. We define predicate unpredecessed which checks whether a statement's predecessors are all dead or the edge from them to it are unused. We also define function defs which returns the names of variables defined in a given statement.

Unit regions	$U \in \mathit{Units}$	$::=$ `unit`
Statement regions	$S \in \mathit{Statements}$	$::=$ `statement` s
Invocation regions	$I \in \mathit{Invocations}$	$::=$ `invoke` $j, R_1, ..., R_m, s$
Block regions	$B \in \mathit{Blocks}$	$::=$ `block` $[R_1, ..., R_n]$
Condition regions	$C \in \mathit{Conditions}$	$::=$ `condition` p, R_1, R_2
Loop regions	$L \in \mathit{Loops}$	$::=$ `loop` h, R
Regions	$R \in \mathit{Regions}$	$::= U \mid S \mid I \mid B \mid C \mid L$

Fig. 6. The region-based intermediate representation

3.3 Regions

The second phase of our analysis is concerned with preparing the input structure required by the third phase which is conducted using region-like structures. Thus we introduce a function to convert the CFG into a region graph. *Regions* [1,11] are areas of code with a single entry point, like code blocks. We extend each region with information about its rôle in the code. We distinguish unit regions, statement regions, invocation regions, block regions, condition regions, and loop regions. We show their definitions in Fig. 6.

Regions are converted from Jimple CFG by the analysis defined in Fig. 7. The analysis is performed on the root of the CFG using regf. The function then handles each node of the CFG by recursion and returns a tree of regions. It uses the loop header set $\mathbb{H}$ and a map of loop headers to their bodies $\mathbb{L}$ from the previous analysis, and a set of dead statements $\mathbb{D}$ whose all incoming edges or predecessors are dead (according to $\mathbb{S}_D$). For convenience, we also define function block which creates a block region from a sequence of statements by applying regf to each of them in succession and aggregating them into a single region.

3.4 Call Count Analysis

Call count analysis is performed on the region tree in order to establish the number of times each object's methods are called. It is depicted in Fig. 8. The analysis begins with the application of function ccount at the root of the region tree and proceeds depth-first through the subregions. In general, method calls on objects in the tree's leafs are counted and the counts are aggregated upwards, either by adding the call counts (with addjoin) in cases of sequences or by taking the highest count (using maxjoin) in cases of alternative program paths.

Function ccount takes three arguments—the global state $\mathbb{G}$, the maximum number of executions of the parent region n, and the region of appropriate type. The function returns a map of object identifiers to the number of times that particular object's method were called. Thus, when the function comes across statement or unit regions it returns empty sets. When it reaches an invoke region it notes the object owning the method and creates a mapping of that object to the number of times the parent region is to be executed; this mapping is then aggregated using function addjoin to the results of the evaluation of the joined bodies of the invoked method using ccount. If a block region is encountered its

$$\mathbb{D} \triangleq \{s \mid \mathbb{S} = \mathbb{G}(s), \mathsf{unpredecessed}(\mathbb{S}_D, s)\}$$
$$\mathsf{block}(\mathbb{H}, [s_1, ..., s_n]) \triangleq \texttt{block}\ \ [R_i \mid 1 < i < n, R_i = \mathsf{regf}(\mathbb{H}, s_i) \wedge (i = 1 \vee \neg s_i \in R_{i-1})]$$
$$\mathsf{regf}(\mathbb{H}, l : b_1; ...; b_n;) \triangleq \mathsf{block}(\mathbb{H}, [l, b_1, ..., b_n])$$
$$\mathsf{regf}(\mathbb{H}, b_1; ...; b_n;) \triangleq \mathsf{block}(\mathbb{H}, [b_1, ..., b_n])$$
$$\mathsf{regf}(\mathbb{H}, s \in \mathbb{H}) \triangleq \texttt{loop}\ \ s, \mathsf{block}(\mathbb{H} \setminus \{s\}, [s' | s' \in \mathbb{L}(s)])$$
$$\mathsf{regf}(\mathbb{H}, s \in \bigcup_{\forall h \in \mathbb{H}} \mathbb{L}(h) \vee s \in \mathbb{D}) \triangleq \texttt{unit}$$
$$\mathsf{regf}(\mathbb{H}, s : \texttt{if}\ p\ \texttt{goto}\ \ l_1\ \ \texttt{else}\ \ l_2) \triangleq$$
$$\quad \text{case } \nexists e \in \mathit{Stmt}, e \ \mathbf{pdom}\ s \rightarrow$$
$$\qquad \texttt{condition}\ \ p, \mathsf{block}(\mathbb{H}, [s' \mid l_1 \ \mathbf{dom}\ s']), \mathsf{block}(\mathbb{H}, [s' \mid l_2 \ \mathbf{dom}\ s'])$$
$$\quad \text{case } \exists e \in \mathit{Stmt}, \nexists e' \in \mathit{Stmt}, e \ \mathbf{pdom}\ s \wedge e' \ \mathbf{pdom}\ s \wedge e \ \mathbf{psdom}\ e' \rightarrow$$
$$\qquad \texttt{condition}\ \ p, \mathsf{block}(\mathbb{H}, [s' \mid l_1 \ \mathbf{dom}\ s' \wedge e \ \mathbf{pdom}\ s']),$$
$$\qquad\quad \mathsf{block}(\mathbb{H}, [s' \mid l_2 \ \mathbf{dom}\ s' \wedge e \ \mathbf{pdom}\ s'])$$
$$\mathsf{regf}(\mathbb{H}, s : \texttt{switch}(i)\{\texttt{case}\ c_1 : l_1; ...; \texttt{case}\ c_n : l_n; \texttt{default:}\ \ l_0\}) \triangleq$$
$$\quad \text{case } \nexists e \in \mathit{Stmt}, e \ \mathbf{pdom}\ s \rightarrow$$
$$\qquad \texttt{condition}\ \ (i = c_1), \mathsf{block}(\mathbb{H}, [s' \mid l_1 \ \mathbf{dom}\ s']), \big(, ...,$$
$$\qquad\ \big(\texttt{condition}\ \ (i = c_n), \mathsf{block}(\mathbb{H}, [s' \mid l_n \ \mathbf{dom}\ s']), \mathsf{block}(\mathbb{H}, [s' \mid l_0 \ \mathbf{dom}\ s'])\big)\big)$$
$$\quad \text{case } \exists e \in \mathit{Stmt}, \nexists e' \in \mathit{Stmt}, e \ \mathbf{pdom}\ s \wedge e' \ \mathbf{pdom}\ s \wedge e \ \mathbf{psdom}\ e' \rightarrow$$
$$\qquad \texttt{condition}\ \ (i = c_1), \mathsf{block}(\mathbb{H}, [s' \mid l_1 \ \mathbf{dom}\ s' \wedge e \ \mathbf{pdom}\ s']), \big(, ...,$$
$$\qquad\ \big(\texttt{condition}\ \ (i = c_n), \mathsf{block}(\mathbb{H}, [s' \mid l_n \ \mathbf{dom}\ s' \wedge e \ \mathbf{pdom}\ s']),$$
$$\qquad\quad \mathsf{block}(\mathbb{H}, [s' \mid l_0 \ \mathbf{dom}\ s' \wedge e \ \mathbf{pdom}\ s'])\big)\big)$$
$$\mathsf{regf}(\mathbb{H}, s : \texttt{invoke}\ \ i.[j(j_1, ..., j_n) : t](i_1, ..., i_n)\{b_1, ..., b_m\}) \triangleq$$
$$\qquad \texttt{invoke}\ \ i, \mathsf{regf}(\mathbb{H}, b_1), ..., \mathsf{regf}(\mathbb{H}, b_m), s$$
$$\mathsf{regf}(\mathbb{H}, s) \triangleq \texttt{statement}\ \ s$$

Fig. 7. Region finding analysis

subregions are evaluated first and the results of these evaluations are aggregated using addjoin. When ccount encounters a conditional region the condition is checked and one of the subregions is evaluated, if the condition is true or false or both conditions are evaluated and their results are aggregated using maxjoin if the condition is unknown. Finally, with loop regions the subregion that is the loop's body is processed using ccount, but the number of executions of the parent region is multiplied by the number of loop iterations (obtained from $\mathbb{S}_I$).

Function maxjoin is used for joining the results of evaluations of two or more subregions where it is unknown which ones will execute. It takes n maps of some keys to numerical values as arguments and returns a similar map. Out of all values that share a key across the maps the maximum one is inserted into the resulting map. Function addjoin is used for aggregating the results of evaluations of a sequence of subregions that will execute one after another. It takes n maps of some keys to numerical values as arguments and returns a similar map. All values that share a key across the maps will be added together and the sum will be inserted into the resulting map under that key.

$$\begin{aligned}
&\mathsf{ccount}(\mathbb{G}, n, \texttt{unit}\ ) \triangleq \varnothing\\
&\mathsf{ccount}(\mathbb{G}, n, \texttt{statement}\ s) \triangleq \varnothing\\
&\mathsf{ccount}(\mathbb{G}, n, \texttt{invoke}\ j, R_1, ..., R_m, s) \triangleq \mathbb{S} = \mathbb{G}(s),\\
&\quad \mathsf{addjoin}(\{\mathbb{S}_V(j) \mapsto n\}, \mathsf{maxjoin}(\mathsf{ccount}(\mathbb{G}, n, R_1), ..., \mathsf{ccount}(\mathbb{G}, n, R_m)))\\
&\mathsf{ccount}(\mathbb{G}, n, \texttt{block}\ [R_1, ..., R_n]) \triangleq \mathsf{maxjoin}(\mathsf{ccount}(\mathbb{S}_V, n, R_1), ..., \mathsf{ccount}(\mathbb{G}, n, R_n))\\
&\mathsf{ccount}(\mathbb{G}, n, \texttt{condition}\ p, R_1, R_2, s) \triangleq \mathbb{S} = \mathbb{G}(s),\\
&\quad \text{case } \mathsf{pred}(p, \mathbb{S}) = \texttt{true} \rightarrow \mathsf{ccount}(\mathbb{G}, n, R_1)\\
&\quad \text{case } \mathsf{pred}(p, \mathbb{S}) = \texttt{false} \rightarrow \mathsf{ccount}(\mathbb{G}, n, R_2)\\
&\quad \text{case } \mathsf{pred}(p, \mathbb{S}) = \omega \rightarrow \mathsf{maxjoin}(\mathsf{ccount}(\mathbb{G}, n, R_1), \mathsf{ccount}(\mathbb{G}, n, R_2))\\
&\mathsf{ccount}(\mathbb{G}, n, \texttt{loop}\ h, R) \triangleq \mathbb{S} = \mathbb{G}(h), \mathsf{ccount}(\mathbb{G}, n * \mathbb{S}_I(h), R)\\
&\mathsf{maxjoin}(\mathbb{M}_1, ..., \mathbb{M}_n) \triangleq \{k \mapsto \max(\mathbb{M}_1(k), ..., \mathbb{M}_n(k)) \ \mid\ k \in \text{dom } \mathbb{M}_1 \cup ... \cup \text{dom } \mathbb{M}_n\}\\
&\mathsf{addjoin}(\mathbb{M}_1, ..., \mathbb{M}_n) \triangleq \{k \mapsto \mathbb{M}_1(k) + ... + \mathbb{M}_n(k) \ \mid\ k \in \text{dom } \mathbb{M}_1 \cup ... \cup \text{dom } \mathbb{M}_n\}\\
&\omega + c = \omega,\ \omega * c = \omega,\ \max(\omega, c) = \omega
\end{aligned}$$

Fig. 8. Call count analysis

Functions at this stage of the analysis may need numerical values to be added or multiplied with the unknown value ω. If this happens, we treat it as an absorbing element, and the result of such an operation is always unknown. In a similar vein, the maximum of any set of numbers including ω is also unknown.

4 Precompiler Implementation

We implemented our precompiler as a tool for Atomic RMI using the Soot framework. The precompiler implementation consists of three elements: Jimple creation, upper bound analysis, and code generation (as shown in Fig. 9). The Jimple creator converts Java source code into the Jimple intermediate language—this is provided by Soot. The upper bound analysis deduces the information about remote object calls within Jimple. It is divided into four analyses, each responsible for one pass over the code. The code generator instruments the input source code with instructions based on the information obtained by the analysis. The two components are described in more detail below.

4.1 Upper Bound Analysis

The upper bound analysis consists of value analysis (VA), region finding (RF), transaction finding (TF), and object call analysis (OCA). These are forward flow analyses implemented in Soot. Each of them makes one pass over the input code in the form of a CFG from a particular starting point (the main method, for instance). The implementation of each analysis defines a transfer function applied to each node of the CFG, a join operator for joining result sets, and initial result sets (see Fig. 4, Fig. 7, and Fig. 8).

Value analysis is the most complex of the analyses. It is the implementation of the algorithm in Section 3.2 such that the transfer function and join implement

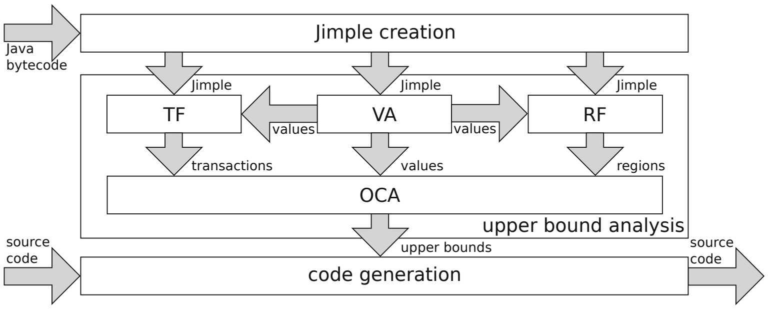


Fig. 9. Components and information flow in the precompiler

eval and join. The transfer function performs whatever action is needed for a given statement type (these are recognized via the type system). The result sets represent $\mathbb{S}_V$ and $\mathbb{S}_P$, $\mathbb{S}_I$, and $\mathbb{S}_D$ are passed via separate fields (for convenience). The implementation finds loops headers and bodies using Soot's built-in loop finder. Loops are processed by running the analysis repeatedly on a pruned copy of the CFG that contains only the statements from the loop and integrating the results into the original analysis. Recurrent calls are handled by finding all applicable method bodies, starting an analysis on each, and joining the results. A stack of calls keeps track of the depth of recursion and when to bound it.

The implementation of value analysis needs to take care of additional significant mechanisms that are obvious in the formalization and therefore glossed over. These include mechanisms for evaluating expressions. Expressions' arguments' types are recognized and the semantics appropriate to them is applied (i.e. `a + b` is addition if `a` and `b` are integers or concatenation if they are strings). All combinations of basic types (at least primitives and `Object`) and operators need to be implemented. We take the approach that operators are defined by classes and perform argument-dependent operations.

Region finder converts the CFG into a region graph in accordance with the algorithm in Section 3.3. The algorithm performs numerous graph searches like finding domination and post-domination relations within the graph (provided by Soot) and finding if particular paths exist within the CFG (e.g. whether all paths from a conditional expression leads to the end of the body or to a common post-dominator). RF creates a region hierarchy, were each region is characterized by its type and type-specific fields.

Transaction finder is a component that tracks Atomic RMI transactions and their components: it identifies the start and possible ends of transactions, remote objects used within, and transactions' preambles. These information are collected for use by OCA and marked in Jimple using the Soot tag system.

OCA is responsible for tallying remote objects calls as in Section 3.4. The implementation is straightforward: it accepts the data from the preceding analyses and uses them to traverse regions and identify those that make calls to remote objects. The number of executions of these regions is predicted and the counts are summed up with reference to particular remote objects.

The implementation must take into account the unknown values that may appear in the course of this analysis. These are implemented as a new type that allows any positive natural number or a value representing ω. The type also defines the maximum function and arithmetical operations using the unknown value (specifically addition and multiplication from Fig. 8).

4.2 Code Generation

The code generator for Atomic RMI modifies code on the lexical level using the suprema obtained from the execution of the upper bound analysis. Necessarily, in order for the code generator to modify the existing source code that source code must be available for analysis. The source is converted into tokens by the SableCC lexer [6] and then divided into lines.

The generator performs three passes over the collection of lines of tokens. In the first pass the generator locates transactions in the source code using the information provided by the transaction finding (TF) phase of code analysis. When found, all definitions in a transaction's preamble are marked for removal, with the exception of those which are followed by a comment string specifying them as manual overrides. The second pass inserts a line of code into each transaction's preamble for each identified remote object pertaining to that transaction (lines 2, 3 in Fig. 2). The insert contains a variable representing the remote object and a supremum on the number of method calls to that object, and it is built using on a simple template. All the inserts are marked for prepending to the beginning of the transaction. The final pass applies all the changes marked by the previous two passes to the tokens and they can then be written to the output stream.

5 Conclusion

Our work illustrates a static analysis for extracting the maximum number of times objects will be called in a fragment of code. Such information has a number of applications (we discuss them in Section 1) but we concentrate on using the upper bounds as input data for Atomic RMI. We have so far found that the analysis we implemented solves this problem satisfactorily for our purposes. The tree-like region-based intermediate representation allows to find all of the method calls within the code and the use of the absorbing unknown value produces conservative results when uncertain values are involved. Both of these guarantee that the statically derived upper bounds are correct, i.e. not lower than any actual number of method calls on a particular object. Apart from being conservative, the estimated upper bounds should also be as accurate as possible—as close to the actual number of executions as possible. For typical Atomic RMI transaction code, the analysis is able to handle most scenarios adequately.

The formalization of our algorithm and adherence to it simplified the implementation of the tool. The formalization was a blueprint for the join operators and transfer function of the individual data flow analyses which it defined their *modi operandi* and allowed us to concentrate on the details of the interfaces, data structures, etc. during implementation. Another advantage is that the correctness of the created tool is verifiable, extensible, and amendable by inspection and modification of the underlying algorithm, without an initial need to delve into the actual source code.

Our future work may include extending the current analysis with some additional refined analyses. In particular, there are ways to provide better identification of particular objects in the code, and more accurate ways to bound loops and recursion. The current algorithms may have trouble analyzing certain instances of input code accurately, especially when the depth of recursion or the number of loop iterations exceeds L. This may be resolved by following some of the approaches we list in Section 2. We also plan on exploring some elements of lock inference [2]. In particular, if an object's method were called an unknown number of times due to loops or recursion it would be possible to mark the last use of the remote object and to free it on that basis in run-time. These two methods could provide complementary mechanisms covering most scenarios. We also look forward to using our tool in new applications such as scheduling.

References

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: principles, techniques, and tools, 2nd edn. Addison Wesley (August 2006)
2. Cunningham, D., Gudka, K., Savani, R.: Keep Off the Grass: Locking the Right Path for Atomicity. In: Hendren, L. (ed.) CC 2008. LNCS, vol. 4959, pp. 276–290. Springer, Heidelberg (2008)
3. Ermedahl, A., Sandberg, C., Gustafsson, J., Bygde, S., Lisper, B.: Loop bound analysis based on a combination of program slicing, abstract interpretation, and invariant analysis. In: Proc. of the 7th Workshop on WCET Analysis (July 2007)
4. Ferdinand, C., Heckmann, R.: AiT: Worst-case execution time prediction by static program analysis. In: Proc. of IFIP WCC 2004 (2004)
5. Ferdinand, C., Heckmann, R., Langenbach, M., Martin, F., Schmidt, M., Theiling, H., Thesing, S., Wilhelm, R.: Reliable and Precise WCET Determination for a Real-Life Processor. In: Henzinger, T.A., Kirsch, C.M. (eds.) EMSOFT 2001. LNCS, vol. 2211, pp. 469–485. Springer, Heidelberg (2001)
6. Gagnon, É.M., Hendren, L.J.: SableCC, an object-oriented compiler framework. In: Proc. of TOOLS 1998 (August 1998)
7. Gustafsson, J., Ermedahl, A., Lisper, B.: Towards a flow analysis for embedded system C programs. In: Proc. of WORDS 2005 (September 2005)
8. Harmon, T., Schoeberl, M., Kirner, R., Klefstad, R.: A modular worst-case execution time analysis tool for Java processors. In: Proc. of RTAS 2008 (April 2008)
9. Hicks, M., Foster, J.S., Prattikakis, P.: Lock inference for atomic sections. In: Proc. of TRANSACT 2006 (June 2006)
10. Holsti, N., Långbacka, T., Saarinen, S.: Worst-case execution-time analysis for digital signal processors. In: Proc. of EUSIPCO 2000 (September 2000)
11. Lee, Y.-F., Ryder, B.G., Fiuczynski, M.E.: Region analysis: A parallel elimination method for data flow analysis. IEEE TSE 21, 913–926 (1995)
12. Li, Y.-T.S., Malik, S.: Performance analysis of real-time embedded software. Springer (November 1998)

13. Lundqvist, T., Stenström, P.: An integrated path and timing analysis method based on cycle-level symbolic execution. Real-Time Systems 17(2-3), 183–207 (1999)
14. McCloskey, B., Zhou, F., Gay, D., Brewer, E.: Autolocker: Synchronization inference for atomic sections. In: Proc. of POPL 2006 (January 2006)
15. Navas, J., Méndez-Lojo, M., Hermenegildo, M.V.: User-definable resource usage bounds analysis for Java bytecode. ENTCS 253(5), 65–82 (2009)
16. Pratikakis, P., Foster, J.S., Hicks, M.W.: Existential Label Flow Inference via CFL Reachability. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 88–106. Springer, Heidelberg (2006)
17. Pugh, W.: The Omega Test: a fast and practical integer programming algorithm for dependence analysis. Communications of the ACM 8, 4–13 (1992)
18. Siek, K., Wojciechowski, P.T.: Statically computing upper bounds on object calls for pessimistic concurrency control. In: Proc. of the EC^2 2010: Workshop on Exploiting Concurrency Efficiently and Correctly (July 2010), Brief Announcement
19. Siek, K., Wojciechowski, P.T., Mruczkiewicz, W.: Atomic RMI documentation (2011), `http://www.it-soa.pl/atomicrmi/`
20. Staschulat, J., Braam, J., Ernst, R., Rambow, T., Schlor, R., Busch, R.: Cost-efficient worst-case execution time analysis in industrial practice. In: Proc. of ISoLA 2006 (November 2006)
21. Theiling, H., Ferdinand, C., Wilhelm, R.: Fast and precise WCET prediction by separated cache and path analyses. Real-Time Syst. 18, 157–179 (2000)
22. Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., Sundaresan, V.: Soot - a Java optimization framework. In: Proc. of CASCON 1999 (November 1999)
23. Vallée-Rai, R., Hendren, L.J.: Jimple: Simplifying Java bytecode for analyses and transformations. Technical Report 1998-4, McGill University (July 1998)
24. Wilhelm, R.: Determining bounds on execution times. In: Handbook on Embedded Systems, ch. 14. CRC Press (2006)
25. Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D.B., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P.P., Staschulat, J., Stenström, P.: The worst-case execution time problem-overview of methods and survey of tools. ACM TECS 7(3) (April 2008)
26. Wojciechowski, P.T.: Language Design for Atomicity, Declarative Synchronization, and Dynamic Update in Communicating Systems. Poznań University of Technology Press (2007); Habilitation thesis
27. Wojciechowski, P.T., Siek, K.: Transaction concurrency control via dynamic scheduling based on static analysis. In: Proc. of WTM 2012 (April 2012)
28. Wolf, F., Ernst, R., Ye, W.: Path clustering in software timing analysis. IEEE Trans. Very Large Scale Integr. Syst. 9, 773–782 (2001)

Checking Properties Described by State Machines: On Synergy of Instrumentation, Slicing, and Symbolic Execution

Jiří Slabý, Jan Strejček, and Marek Trtík

Faculty of Informatics, Masaryk University,
Botanická 68a, 60200 Brno, Czech Republic
{slaby,strejcek,trtik}@fi.muni.cz

Abstract. We introduce a novel technique for checking properties described by finite state machines. The technique is based on a synergy of three well-known methods: instrumentation, program slicing, and symbolic execution. More precisely, we instrument a given program with a code that tracks runs of state machines representing various properties. Next we slice the program to reduce its size without affecting runs of state machines. And then we symbolically execute the sliced program to find real violations of the checked properties, i.e. real bugs. Depending on the kind of symbolic execution, the technique can be applied as a stand-alone bug finding technique, or to weed out some false positives from an output of another bug-finding tool. We provide several examples demonstrating the practical applicability of our technique.

1 Introduction

There are several successful formalisms for description of program properties. One of the most popular is a *finite state machine (FSM)*. This formalism is simple and still flexible enough to describe many often studied program properties including locking policy in concurrent programs, null-pointer dereferences, resource allocations, and resource leaks. FSM specification is therefore used in many static program analysis tools like XGCC [24], SLAM [4], SDV [3], BLAST [5], ESP [14], or STANSE [27]. All the mentioned tools produce *false positives*, i.e. they report errors that do not correspond to any real error. We now roughly explain the basic principle of static analysis implemented in XGCC, ESP, and STANSE.

1.1 Checking FSM Properties by Static Analysis

Let us consider the state machine *SM*(x) of Figure 1. It describes a lock manipulation including malign transitions. Intuitively, the state machine represents possible courses of states of a lock referenced by x along an execution of a program. The state of the lock is changed according to the program behavior. Whenever the program contains a statement syntactically subsuming the label of a transition, the transition is fired in the state machine. We would like to

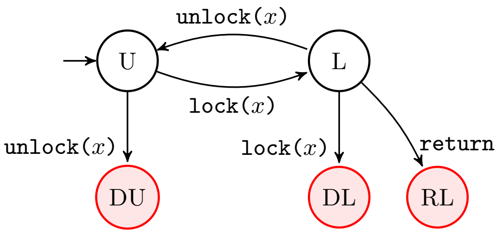


Fig. 1. State machine *SM*(x) describing errors in manipulation with lock x. The nodes U and L refer to states *unlocked* and *locked*, respectively. The other three nodes refer to error states: DU to *double unlock*, DL to *double lock*, and RL to *return in locked state*. The initial node is U.

decide whether there exists any program execution where an instance of state machine *SM*(x) reaches an error state for some lock in the program. Unfortunately, this is not feasible due to potentially unbounded number of executions and unbounded execution length. Hence, static analysis tools overapproximate the set of reachable state machine states.

Let us assume that we want to check the program of Figure 2 for errors specified by the state machine *SM*(x). First, we find all locks in the program and to each lock we assign an instance of the state machine. In our case, there is only one lock pointed to by `L` and thus only one instance *SM*(`L`). For each program location, we compute a set overapproximating possible states of *SM*(`L`) after executions leading to the location. Roughly speaking, we initialize the set in the initial location to $\{U\}$ and the other sets to $\emptyset$. Then we repeatedly update the sets according to the effect of individual program statements until the fixed point is reached. The resulting sets for the program of Figure 2 are written directly in the code listing as comments.

As we can see, the sets contain two error states: *double unlock* after the `unlock(L)` statement and *return in locked state* in the terminal location. If we

```
char *copy(char *dst, char *src, int n, int *L) {
   int i, len;                                    // {U}
   len = 0;                                       // {U}
   if (src != NULL && dst != NULL) {              // {U}
      len = n;                                    // {U}
      lock(L);                                    // {L}
   }                                              // {U,L}
   i = 0;                                         // {U,L}
   while (i < len) {                              // {U,L}
      dst[i] = src[i];                            // {U,L}
      i++;                                        // {U,L}
   }                                              // {U,L}
   if (len > 0) {                                 // {U,L}
      unlock(L);                                  // {DU,U}
   }                                              // {U,L}
   return dst;                                    // {U,RL}
}
```

Fig. 2. Function `copy` copying a source string `src` into a buffer `dst` using a lock `L` to prevent parallel writes

analyze the computation of the sets, we can see that the first error corresponds to executions going through lines 1,2,3,4,8, then iterating the `while`-loop and finally passing lines 13,14. These execution paths are not feasible due to the value of `len`, which is set to 0 at line 3 and assumed to satisfy `len > 0` at line 13. Hence, the first error is a *false positive*. The second error corresponds to executions passing lines 1,2,3,4,5,6,7,8, then iterating the `while`-loop and finally going through lines 13,16. All these paths are also infeasible except the one that performs zero iterations of the `while`-loop, which is the only real execution leading to the only real locking error in the program.

To sum up, static analysis tools like XGCC, ESP, and STANSE are highly flexible, fast and thus applicable to extremely large software projects (e.g. the Linux kernel). It examines all the code and finds many error reports. Unfortunately, many of the reports are false positives.[1] As manual sorting of error reports containing a pile of false positives is tedious work, the practical applicability of such tools is limited.

1.2 No False Positives with Symbolic Execution

In contrast to static analysis, test-generation tools based on *symbolic execution* do not suffer from false positives, since the checked program is actually executed (but on symbolic input instead of concrete one). A disadvantage of these tools is that they usually detect only low-level errors representing violations of programming language semantics, i.e. various types of undefined behavior or crashes. To detect violations of other program properties like locking policy, the program has to be modified such that the errors can be detected during the execution. This can be achieved, for example, by introducing a couple of `assert` statements to specific program locations. Another and even more important disadvantage of the tools is extreme computation cost of symbolic execution. In particular, programs containing loops or recursion have typically large or even infinite number of execution paths and cannot be entirely analysed by symbolic execution.

1.3 Our Contribution: A New Technique

In this paper, we introduce a new fully automatic program analysis technique offering flexibility of FSM property specification with zero false positive rate of symbolic execution. The technique symbolically executes only parts of the analysed program having impact on the checked property. The basic idea is very simple:

1. We get state machines describing some program properties. We instrument a given program with a code tracking behavior of the state machines.
2. The instrumented program is then reduced using method called *slicing* [33]. The sliced program has to meet the criterion to be equivalent to the instrumented program with respect to reachability of error states of tracked state

[1] We note that XGCC and ESP actually use many techniques for partial elimination of false positives (see [24,14] for details).

machines. Note that slicing may remove large portions of the code, including loops and function calls. Hence, an original program with an infinite number of execution paths may be reduced to a program with a finite number of execution paths.
3. Finally, we execute the sliced program symbolically to find violations of the checked property.

Our technique may be used in two ways according to the applied symbolic execution tool. If we apply a symbolic executor that prefers to explore more parts of the code (for example by exploring each program loop at most twice), we may use the technique as a general bug-finding technique reporting only real errors. On the contrary, if we use a symbolic executor exploring all execution paths, we may use our technique for classification of error reports produced by other tools (e.g. XGCC or STANSE). For each such an error report, we may instrument the corresponding code only with the state machine describing the reported error. If our technique finds the same error, it is a real one. If our technique explores all execution paths of the sliced code without detecting the error, it is a false positive. If our technique runs out of resources, we cannot decide whether the error is a real one or just a false positive.

We have developed an experimental tool implementing our technique. The tool instruments a program with a state machine describing locking errors (we use a single-purpose instrumentation so far), then it applies an interprocedural slicing to the instrumented code, and it passes the sliced code to symbolic executor KLEE [9]. Our experimental results indicate that the technique can indeed classify error reports produced by STANSE applied to the Linux kernel.

We emphasize the synergy of the three known methods combined in the presented technique.

- Instrumentation of a program with a code emulating state machines provides us with simple slicing criteria: we want to preserve values of memory places representing states of state machines. Hence, the sliced program contains only the code relevant to the considered errors specified by state machines.
- Slicing may substantially reduce the size of the code, which in turn may remarkably improve performance of the symbolic execution.
- Application of symbolic execution brings us another benefit. While in standard static analysis, the state machines are associated to syntactic objects (e.g. lock variables appearing in a program), we may associate state machines to actual values of these objects. This leads to a higher precision of error detection.

The rest of the paper is organized as follows. Sections 2, 3, and 4 deal with program instrumentation, program slicing, and symbolic execution, respectively. Experimental implementation of our technique and some experimental results are discussed in Section 5. Section 6 is devoted to related work while Section 7 indicates some directions driving our future research. Finally, the last section summarizes presented results.

2 Instrumentation

In our algorithm, the purpose of the instrumentation is to insert a code implementing a state machine into the analysed program. Nonetheless, the semantics of the program being instrumented must not be changed. A result of this phase is therefore a new program that still has the original functionality but it also simultaneously updates instrumented state machines. We show the process using the state machine $SM(x)$ of Figure 1 and a program consisting of two functions: `copy` of Figure 2 and `foo` of Figure 3. The function `foo` calls `copy` twice, first with the lock `L1` and then with the lock `L2`. The locks protect writes into buffers `buf1` and `buf2` respectively. The function `foo` is a so-called *starting function*. It is a function where the symbolic execution starts.

```
char *buf1, *buf2;
int L1, L2;

void foo(char *src, int n) {
   copy(src, buf1, n, &L1);
   copy(src, buf2, n, &L2);
}
```

Fig. 3. Function `foo` forms the analysed program together with function `copy`

The instrumentation starts by recognizing code fragments which manipulate with locks in the analysed program. More precisely, we look for all those code fragments matching edge labels of the state machine $SM(x)$ of Figure 1. The analysed program contains three such fragments, all of them in function `copy` (see Figure 2): the call to `lock` at line 6, the call to `unlock` at line 14, and the `return` statement at line 16.

Next we determine a set of all locks that are manipulated by the program. From the recognized code fragments, we find out that a pointer variable `L` in `copy` is the only program variable through which the program manipulates with locks. Using a points-to analysis, we obtain set $\{\texttt{L1}, \texttt{L2}\}$ of all possible locks the program manipulates with.

We introduce a unique instance of the state machine $SM(x)$ for each lock in the set. More precisely, we define two integer variables `smL1` and `smL2` to keep the current state of state machines $SM(\texttt{L1})$ and $SM(\texttt{L2})$, respectively. Further, we need to specify a mapping from locks to their state machines. The mapping is basically a function (preferably with constant complexity) from addresses of program objects (i.e. the locks) to addresses of corresponding state machines. Figure 4 shows an implementation of a function `smGetMachine` that maps addresses of locks `L1` and `L2` to addresses of corresponding state machines. We note that the implementation of `smGetMachine` would be more complicated if state machines are associated to dynamically allocated objects.

Besides `smGetMachine`, Figure 4 contains also many constants and a function `smFire` implementing the state machine $SM(x)$. Further, Figure 4 declares variables `smL1` and `smL2` and initializes them to the initial state of the state machine.

```
const int smU  = 0;      // state U
const int smL  = 1;      // state L
const int smDU = 2;      // state DU
const int smDL = 3;      // state DL
const int smRL = 4;      // state RL

const int smLOCK   = 0; // transition lock(x)
const int smUNLOCK = 1; // transition unlock(x)
const int smRETURN = 2; // transition return

int smL1 = smU, smL2 = smU;

int *smGetMachine(int *p) {
   if (p == &L1) return &smL1;
   if (p == &L2) return &smL2;
   return NULL;         // unreachable
}

void smFire(int *SM, int transition) {
   switch (*SM) {
   case smU:
      switch (transition) {
      case smLOCK:
         *SM = smL;
         break;
      case smUNLOCK:
         assert(false); // double unlock
         break;
      default: break;
      }
      break;
   case smL:
      switch (transition) {
      case smLOCK:
         assert(false); // double lock
         break;
      case smUNLOCK:
         *SM = smU;
         break;
      case smRETURN:
         assert(false); // return in locked
         break;
      default: break;
      }
      break;
   default: break;
   }
}
```

Fig. 4. Implementation of the state machine (`smFire`) and its identification (`smGetMachine`)

Note that we represent both states of the machine and names of transitions by integer constants. Also keep in mind that the pointer argument `SM` of `smFire` function points to an instrumented state machine, whose transition has to be fired.

It remains to instrument the recognized code fragments in the original program. For each fragment we know its related transition of the state machine and we also know what objects the fragment manipulates with (if any). Therefore, we first retrieve an address of state machine related to manipulated objects (if any) by using the function `smGetMachine` and then we fire the transition by calling the function `smFire`. The instrumented version of the original program consists of the code of Figure 4 and the instrumented version of the original functions

```
    char *buf1, *buf2;
    int L1, L2;

    char *copy(char *dst, char *src, int n, int *L) {
       int i, len;
       len = 0;
       if (src != NULL && dst != NULL) {
          len = n;
*         smFire(smGetMachine(L), smLOCK);
          lock(L);
       }
       i = 0;
       while (i < len) {
          dst[i] = src[i];
          i++;
       }
       if (len > 0) {
*         smFire(smGetMachine(L), smUNLOCK);
          unlock(L);
       }
*      smFire(smGetMachine(L), smRETURN);
       return dst;
    }

    void foo(char *src, int n) {
       copy(src, buf1, n, &L1);
       copy(src, buf2, n, &L2);
    }
```

Fig. 5. Functions `foo` and `copy` instrumented by calls of `smFire` function

`foo` and `copy` given in Figure 5, where the instrumented lines are highlighted by `*`. Note that in our example, the instrumented state variables `smL1` and `smL2` directly correspond to the program locks `L1` and `L2` respectively. In general, however, states of a state machine of a more complex property need not necessarily correspond to values of a particular program variable. Therefore, we apply this general approach to our example too.

3 Slicing

Let us have a look at the instrumented program in Figure 5. We can easily observe, that the main part of the function `copy`, the loop copying the characters, does not affect states of the instrumented state machines. Symbolic execution of such a code is known to be very expensive, moreover in this case it is yet unneeded. Therefore, we use the slicing technique from [33] to eliminate such a code from the instrumented program.

The input of the slicing algorithm is a program to be sliced and a so-called *slicing criteria*. A slicing criterion is a pair of a program location and a set of program variables. The slicing algorithm removes program statements that do not affect any slicing criterion. More precisely, for each input data passed to both original and sliced programs, values of the variable set of each slicing criterion at the corresponding location are always equal in both programs. Our analysis is interested only in states of the instrumented automata, especially in locations corresponding to errors. Hence, the slicing criterion is a pair of a

```
 1: char *buf1, *buf2;
 2: int L1, L2;
 3:
 4: char *copy(char *dst, char *src, int n, int *L) {
 5:    int len;
 6:    len = 0;
 7:    if (src != NULL && dst != NULL) {
 8:       len = n;
 9:       smFire(smGetMachine(L), smLOCK);
10:    }
11:    if (len > 0) {
12:       smFire(smGetMachine(L), smUNLOCK);
13:    }
14:    smFire(smGetMachine(L), smRETURN);
15:    return dst;
16: }
17:
18: void foo(char *src, int n) {
19:    copy(src, buf1, n, &L1);
20:    copy(src, buf2, n, &L2);
21: }
```

Fig. 6. Functions `foo` and `copy` after slicing

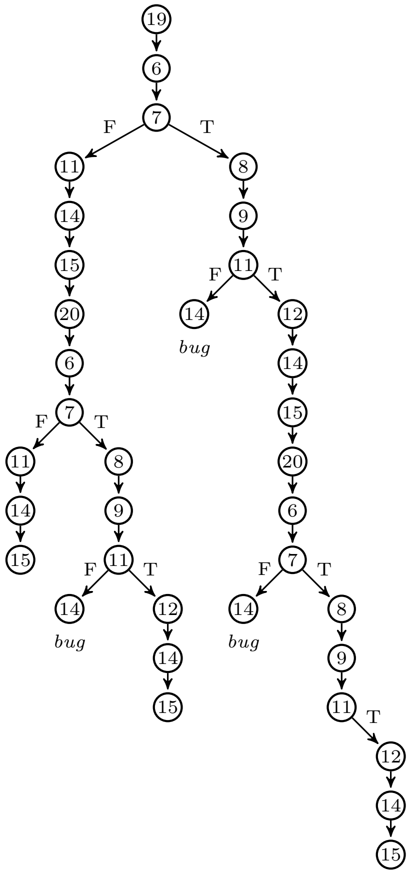


Fig. 7. Symbolic execution tree of the sliced program of Figure 6

location preceding an `assert` statement in `smFire` function and the set of all variables representing current states of the corresponding state machines. The slicing criteria then comprises all such pairs.

In the instrumented program of Figures 4 and 5, we want to preserve variables `smL1` and `smL2`. We put slicing criteria into the lines of code detecting transitions of state machines into error states. In other words, the slicing criteria for our running example are pairs (27,{`smL1`,`smL2`}), (35,{`smL1`,`smL2`}), and (41,{`smL1`,`smL2`}), where the numbers refer to lines in the code of Figure 4. The result of the slicing procedure is presented in Figures 4 and 6 (the code in the former Figure shall not be changed by the slicing). Note that the sliced code contains neither the `while`-loop nor the `lock` and `unlock` commands.

It is important to note that some slicing techniques, including the one in [33] that we use, do not consider inputs for which the original program does not halt. As a result, an input causing an infinite run of the original program can induce a finite run in the sliced program. Moreover, the finite run can contain locations not visited by the infinite run. This is the only principal source of potential false positives in our technique.

4 Symbolic Execution

This is the final phase of our technique. We symbolically execute the sliced program from the entry location of the starting function. Symbolic execution explores real program paths. Therefore, if it reaches some of the assertions inside function `smFire`, then we have found a bug.

Our running example nicely illustrates the crucial role of slicing to feasibility of symbolic execution. Let us first consider symbolic execution of the original program. It starts at the entry location of the function `foo`. The execution eventually reaches the function `copy`. Note that value of the parameter `n` is symbolic. Therefore, symbolic execution will fork into two executions each time we reach line 9 of Figure 2. One of the executions skips the loop at lines 9–12, while the other enters it. If we assume that the type of `n` is a 32-bit integer, then the symbolic execution of one call of `copy` explores more then 2^{31} real paths.

By contrast, the sliced program does not contain the loop, which generated the huge number of real paths. Therefore, a number of real paths explored by the symbolic execution is exactly 6. Figure 7 shows the symbolic execution tree of the sliced program of Figure 6. We left out vertices corresponding to lines in called functions `smGetMachine` and `smFire`. Note that although the parameter `n` has a symbolic value, it can only affect the branching at line 11. Moreover, the parameter `L` always has a concrete value. Therefore, we do not fork symbolic execution at branchings inside functions `smGetMachine` and `smFire`. Three of the explored paths are marked with the label *bug*. These paths reach the second assertion in function `smFire` (see Figure 4) called from line 14 of the sliced program. In other words, the paths are witnesses that we can leave the function `copy` in a locked state. The remaining explored paths of Figure 7 miss the assertions in the function `smFire`. It means that the original program contains only one locking error, namely *return in locked state.*

It might be the case for some program and checked property, that the sliced code still contains loops. Then the subsequent symbolic execution can be very costly due to the well-known path explosion problem. Fortunately, there have been done some work tackling the problem [18,19,22,30,34].

5 Implementation and Experimental Results

To verify applicability of the presented technique, we have developed an experimental implementation. Our experimental tool works with programs in C and, for the sake of simplicity, it detects only locking errors described by a state machine very similar to $SM(x)$ of Figure 1. The instances of the state machine are associated with arguments of `lock` and `unlock` function calls. Note that the technique currently works only for the cases where a lock is instantiated only once during the run of the symbolic executor, which is the most frequent case. However we plan to add a support even for the rest. The main part of our implementation is written in three modules for the Llvm framework [35], namely `Prepare`, `Slicer`, and `Kleerer`. The framework provides us with a C compiler clang. We also use an existing symbolic executor for Llvm called Klee [9].

Instrumentation of a given program proceeds in two steps. Using a C preprocessor, the original program is instrumented with function calls `smFire` located just above statements changing states of state machines. The program is then translated by clang into Llvm bytecode [35]. Optimizations are turned off as required by Klee. The rest of the instrumentation (e.g. adding global variables and changing the code to work with them) is done on the Llvm code using the module `Prepare`.

The module `Slicer` implements a variant of the inter-procedural slicing algorithm by Weiser [33]. To guarantee correctness and to improve performance of slicing, the algorithm employs points-to analysis by Andersen [2].

The module `Kleerer` performs a final processing of the sliced bytecode before it is passed to Klee. In particular, the module adds to the bytecode a function `main` that calls a *starting function*. The `main` function also allocates a symbolic memory for each parameter of the starting function. Size of the allocated memory is determined by the parameter type. Plus, when the parameter is a pointer, the size is multiplied by 4000. For example, 4 bytes are allocated for an integer and 16000 bytes for an integer pointer. Further, for the pointer case, we pass a pointer to the middle of the allocated memory (functions might dereference memory at negative index). The idea behind is explained in [28]. Finally, the resulting bytecode is symbolically executed by Klee. If a symbolic execution touches a memory out of the allocated area, we get a *memory error*. To remedy this inconvenience, we plan to implement the same on-demand memory handling UcKlee [28] does.

5.1 Experiments

We have performed our experiments on several functions of the Linux kernel 2.6.28, where the static analyzer Stanse reported some error. More precisely,

Table 1. Experimental results. The table presents running time of preprocessing and compilation (**Comp.**), instrumentation including points-to analysis (**Instr.**), slicing (**Slic.**), symbolic execution (**SE**), and the total running time. The column **Sliced** presents the ratio of instructions sliced away from the instrumented LLVM code and the exact number of instructions before/after slicing. The column **Result** specifies the result of our tool: BUG means that the tool found a real error, FP means that the analysis finished without error found (i.e. the original error report is a false positive), TO that the symbolic execution did not finish in time and ME denotes an occurrence of memory error. The last column specifies the factual state of the error report.

File	**Running Time** (s)					**Sliced**	**Result**	**Factual**
Function	**Comp.**	**Instr.**	**Slic.**	**SE**	**Total**			**State**
`fs/jfs/super.c` `jfs_quota_write`	1.25	0.18	0.15	5.09	6.67	67.8% 369/119	BUG	BUG
`drivers/net/qlge/qlge_main.c` `qlge_set_mac_address`	2.70	0.72	26.75	13.28	43.45	66.5% 1333/447	BUG	BUG
`drivers/hid/hidraw.c` `hidraw_read`	1.06	0.18	0.14	Timeout		67.0% 666/220	TO	BUG
`drivers/net/ns83820.c` `queue_refill`	1.76	0.29	1.72	0.62	4.39	72.9% 1212/329	FP	FP
`drivers/usb/misc/sisusbvga/sisusb_con.c` `sisusbcon_set_palette`	1.50	0.24	0.27	17.19	19.20	76.0% 2936/705	FP	FP
`fs/jffs2/nodemgmt.c` `jffs2_reserve_space`	1.04	0.18	0.22	Timeout		46.8% 677/360	TO	FP
`kernel/kprobes.c` `pre_handler_kretprobe`	0.32	0.09	0.51	2.43	3.35	66.3% 202/68	ME	FP

STANSE reported an error trace starting in these functions. We consulted the errors with kernel developers to sort out which are false positives and which are real errors. All the selected functions (and all functions transitively called from them) contain no assembler (in some cases, it has been replaced by an equivalent C code) and no external function calls after slicing.

We ran our experimental tool on these functions. All tests were performed on a machine with an Intel E6850 dual-core processor at 3 GHz and 6 GiB of memory, running Linux. We specified KLEE parameters to time out after 10 seconds spent in an SMT solver and after 300 seconds of an overall running time. Increasing these times brings no real effect in our environment. We do not pass optimize option for KLEE because it causes KLEE to crash for most of the input.

Table 1 presents results of our tool on selected functions. The table shows compilation, instrumentation, slicing, symbolic execution, and the overall running time. Further, the table presents the ratio of instructions that were sliced away from the instrumented LLVM code. The last two columns specify the results of our analysis and the real state confirmed by kernel developers. The table clearly shows that the bottleneck of our technique is the symbolic execution.

However, if we did not slice the code, the only function completely executed in time would be `sisusbcon_set_palette`, computed in 20.64s.

Although the results have no statistical significance, it is clear that the technique can in principle classify error reports produced by other tools like Stanse. If our technique reports an error, it is a real one. If it finishes the analysis without any error detected, the original error report is a false positive. The analysis may also not finish in a given time, which is usually caused by loops in the sliced code. Finally, it may report a memory error mentioned above.

6 Related Work

There are many tools checking properties described by finite state machines. They produce both kinds of reports, real error as well as false positives. The technique of xgcc presented in [11,12,16,24] found a thousands of bugs in real system code. It provides a language Metal for easy description of properties to be checked. xgcc suffers from false positives despite usage of false positive suppression algorithms like killing variables and expressions, synonyms, false path pruning, and others. Besides the suppression algorithms, bug-reports from the tool are further ranked according to their probability of being real errors. There are generic and statistical ranking algorithms ordering bug-reports. An extension introduced in [17] provides an automatic inference of some temporal properties based on statistical analysis of assumed programmer's beliefs. The ESP [14] technique uses a similar language to Metal for properties description. It implements an interprocedural dataflow algorithm based on [29] for error detection and an abstract simulation pruning algorithm for false positives suppression. Stanse [27], a static analysis tool also uses state machines for description of checked program properties. The description is based on parametrised abstract syntax trees. Finally, CEGAR [13] based tools like SLAM [4], SDV [3], or Blast [5], do not produce false positives, in theory. However, to achieve an appropriate efficiency and scalability for a practical use, the implementation of the CEGAR loop is typically unsound.

Program analysis tools based on symbolic execution [25] mainly discover low-level bugs like division by zero, illegal memory access, assertion failure etc. These tools typically do not have problems with false positives, but they have problems with scalability to large programs. There has been developed a lot of techniques improving the scalability to programs used in practice. Modern techniques are mostly hybrid: they usually combine symbolic execution with concrete one [20,21,31]. There are also hybrid techniques combining symbolic execution with a complementary static analysis [22,26]. Symbolic execution can be accelerated by a compositional approach based on function summaries [1,18]. Another approach to effective symbolic execution introduced in [8,9,10] is based on recording of already seen behavior and pruning its repetition. There is an orthogonal line of research which tries to improve the symbolic execution for programs with some special types of inputs. Some techniques deal with programs manipulating strings [7,34], and some other techniques reduce input space using a given input grammar [19,30].

The interprocedural static slicing was introduced by Weiser [33]. But nowadays, there are many different approaches to program slicing. They are surveyed by several authors [6,15,32]. Applications of slicing include program debugging, reverse engineering and regression testing [23].

7 Future Work

Our future work has basically three independent directions.

First, we plan to run our tool to classify all lock-related error reports produced by Stanse on the Linux kernel. The results should provide a better image of practical applicability of the technique. To get a relevant data, we should solve some practical issues like a correct detection of starting functions, automatic replacement of assembler, treatment of external function calls, etc. We should also implement an on-demand memory allocation to Klee as discussed in Section 5 or use a different executor.

The second direction is to adopt or design some convenient way for specification of arbitrary state machines. It may be a dedicated language similar to Metal [12]. Then we plan to implement an instrumentation treating these state machines. In particular, the instrumentation should correctly handle state machines associated with dynamically allocated objects.

Finally, we would also like to examine performance of our technique as a standalone error-detection tool. To this point, we have to use a symbolic executor aiming for maximal code coverage. In particular, such an executor has to suppress execution paths that differ from explored paths only in number of loop iterations. Unfortunately, we do not know about any publicly available symbolic executor of this kind. However, it seems that UcKlee [28] (which is not public as of now) has been designed for a similar purpose.

8 Conclusion

We have presented a novel technique combining three standard methods (instrumentation, slicing, and symbolic execution) to check program properties given in form of finite state machines. We have discussed a synergy of the three methods. Moreover, our experimental results indicate that the technique can recognize some false positives and some real errors in error reports produced by other error-detection tools.

Acknowledgements. The authors are supported by the Czech Science Foundation, project No. P202/12/G061.

References

1. Anand, S., Godefroid, P., Tillmann, N.: Demand-Driven Compositional Symbolic Execution. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 367–381. Springer, Heidelberg (2008)

2. Andersen, L.O.: Program analysis and specialization for the C programming language. PhD thesis, DIKU, University of Copenhagen, report 94/19 (1994)
3. Ball, T., Bounimova, E., Kumar, R., Levin, V.: SLAM2: Static driver verification with under 4% false alarms. In: Proceedings of FMCAD, pp. 35–42. IEEE (2010)
4. Ball, T., Levin, V., Rajamani, S.K.: A decade of software model checking with SLAM. Journal on Commun. ACM 54(7) (2011)
5. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker BLAST. Journal on Software Tools for Technology Transfer 9(5), 505–525 (2007)
6. Binkley, D.W., Gallanger, K.B.: Program slicing. Advances in Computers 43 (1996)
7. Bjørner, N., Tillmann, N., Voronkov, A.: Path Feasibility Analysis for String-Manipulating Programs. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 307–321. Springer, Heidelberg (2009)
8. Boonstoppel, P., Cadar, C., Engler, D.: RWset: Attacking Path Explosion in Constraint-Based Test Generation. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 351–366. Springer, Heidelberg (2008)
9. Cadar, C., Dunbar, D., Engler, D.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of OSDI, pp. 209–224. USENIX Association (2008)
10. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: EXE: Automatically generating inputs of death. ACM Trans. Inf. Syst. Secur. 12, 1–38 (2008)
11. Chelf, B., Hallem, S., Engler, D.: How to write system-specific, static checkers in metal. In: Proceedings of PASTE, pp. 51–60. ACM (2002)
12. Chou, A., Chelf, B., Engler, D., Heinrich, M.: Using meta-level compilation to check FLASH protocol code. ACM SIGOPS Oper. Syst. Rev. 34(5), 59–70 (2000)
13. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided Abstraction Refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000)
14. Das, M., Lerner, S., Seigle, M.: ESP: Path-sensitive program verification in polynomial time. In: Proceedings of PLDI. SIGPLAN, vol. 37(5). ACM Press (2002)
15. De Lucia, A.: Program slicing: methods and applications. In: Proceedings of SCAM, pp. 142–149. IEEE Computer Society (2001)
16. Engler, D., Chelf, B., Chou, A., Hallem, S.: Checking system rules using system-specific, programmer-written compiler extensions. In: Proceedings of OSDI, pp. 1–16. ACM (2000)
17. Engler, D., Chen, D., Hallem, S., Chou, A., Chelf, B.: Bugs as deviant behavior: A general approach to inferring errors in systems code. In: Proceedings of SOSP, pp. 57–72. ACM (2001)
18. Godefroid, P.: Compositional dynamic test generation. In: Proceedings of POPL, pp. 47–54. ACM (2007)
19. Godefroid, P., Kiezun, A., Levin, M.Y.: Grammar-based whitebox fuzzing. In: Proceedings of PLDI, pp. 206–215. ACM (2008)
20. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. In: Proceedings of PLDI, pp. 213–223. ACM (2005)
21. Godefroid, P., Levin, M.Y., Molnar, D.A.: Active property checking. In: Proceedings of EMSOFT, pp. 207–216. ACM (2008)
22. Godefroid, P., Nori, A.V., Rajamani, S.K., Tetali, S.D.: Compositional must program analysis: unleashing the power of alternation. In: Proceedings of POPL, pp. 43–56. ACM (2010)
23. Gupta, R., Harrold, M.J., Soffa, M.L.: An approach to regression testing using slicing. In: Proceedings of ICSM, pp. 299–308. IEEE (1992)

24. Hallem, S., Chelf, B., Xie, Y., Engler, D.R.: A system and language for building system-specific, static analyses. In: Proceedings of PLDI, pp. 69–82. ACM (2002)
25. King, J.C.: Symbolic execution and program testing. Communications of ACM 19(7), 385–394 (1976)
26. Nori, A.V., Rajamani, S.K., Tetali, S., Thakur, A.V.: The Yogi Project: Software Property Checking via Static Analysis and Testing. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 178–181. Springer, Heidelberg (2009)
27. Obdržálek, J., Slabý, J., Trtík, M.: STANSE: Bug-Finding Framework for C Programs. In: Kotásek, Z., Bouda, J., Černá, I., Sekanina, L., Vojnar, T., Antoš, D. (eds.) MEMICS 2011. LNCS, vol. 7119, pp. 167–178. Springer, Heidelberg (2012)
28. Ramos, D.A., Engler, D.R.: Practical, Low-Effort Equivalence Verification of Real Code. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 669–685. Springer, Heidelberg (2011)
29. Reps, T., Horowitz, S., Sagiv, M.: Precise interprocedural dataflow analysis via graph reachability. In: Proceedings of POPL, pp. 49–61. ACM (1995)
30. Saxena, P., Poosankam, P., McCamant, S., Song, D.: Loop-extended symbolic execution on binary programs. In: Proceedings of ISSTA, pp. 225–236. ACM (2009)
31. Sen, K., Marinov, D., Agha, G.: CUTE: a concolic unit testing engine for C. In: Proceedings of ESEC/FSE, vol. 30, pp. 263–272. ACM (2005)
32. Tip, F.: A survey of program slicing techniques. Journal of Programming Languages 3, 121–189 (1995)
33. Weiser, M.: Program slicing. IEEE Transactions on Software Engineering 10(4), 352–357 (1984)
34. Xu, R.G., Godefroid, P., Majumdar, R.: Testing for buffer overflows with length abstraction. In: Proceedings of ISSTA, pp. 27–38. ACM (2008)
35. LLVM, `http://llvm.org/`

Author Index