Charles Pecheur
Michael Dierkes (Eds.)

Formal Methods for Industrial Critical Systems

LNCS 8187

18th International Workshop, FMICS 2013
Madrid, Spain, September 2013
Proceedings

Springer

Lecture Notes in Computer Science 8187

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

Charles Pecheur Michael Dierkes (Eds.)

Formal Methods for Industrial Critical Systems

18th International Workshop, FMICS 2013
Madrid, Spain, September 23-24, 2013
Proceedings

Springer

Volume Editors

Charles Pecheur
Université catholique de Louvain
Louvain-la-Neuve, Belgium
E-mail: charles.pecheur@uclouvain.be

Michael Dierkes
Rockwell Collins France
Blagnac, France
E-mail: mdierkes@rockwellcollins.com

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-41009-3 e-ISBN 978-3-642-41010-9
DOI 10.1007/978-3-642-41010-9
Springer Heidelberg New York Dordrecht London

Library of Congress Control Number: Applied for

CR Subject Classification (1998): D.2.4, F.3.1, D.2, C.3, F.4.1, 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 2013, the 18th International Workshop on Formal Methods for Industrial Critical Systems, taking place September 23–24, 2013, in Madrid, Spain. 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), Trento (August 2011), and Paris (August 2012). The FMICS 2013 workshop was co-located with the 11th International Conference on Software Engineering and Formal Methods (SEFM 2013).

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 25 submissions. Papers had to pass a rigorous review process in which each paper received three reports. The international Program Committee of FMICS 2013 decided to select 13 papers for presentation during the workshop and inclusion in these proceedings. The workshop also featured a first invited talk by Alessandro Fantechi (Università degli Studi di Firenze), jointly with the FM-RAIL-BOK workshop, and a second invited talk by Benjamin Monate (TrustMySoft SA).

Following a tradition established over the past few years, the European Association for Software Science and Technology (EASST) offered an award to the best FMICS paper. This year, the reviewers selected the contribution by Adrien Champion, Rémi Delmas, Michael Dierkes, Pierre-Loic Garoche, Romain Jobredeaux, and Pierre Roux titled "Methods for the Analysis of Critical Control Systems Models: Combining Non-Linear and Linear Analyses."

We would like to thank the SEFM general chair Manuel Nuñez and workshop chair Steve Counsell for taking care of all the local arrangements in Madrid, the ERCIM FMICS working group coordinator Radu Mateescu (INRIA Grenoble) for his help, 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 their submissions, and all attendees of the workshop. Thanks to all for your contribution to the success of FMICS 2013.

September 2013

Charles Pecheur
Michael Dierkes

Organization

Program Committee

Maria Alpuente	Universitat Politecnica de Valencia, Spain
Jiri Barnat	Masaryk University, Czech Republic
Eckard Böde	Offis, Germany
Jean-Louis Colaço	Prover Technology, France
Michael Dierkes	Rockwell Collins, France
Cindy Eisner	IBM, Island
Alessandro Fantechi	DSI - Università di Firenze, Italy
Andrew Gacek	Rockwell Collins, USA
Maria Del Mar Gallardo	University of Málaga, Spain
Stefania Gnesi	ISTI-CNR, Italy
Gordon Haak	Daimler AG, Germany
Holger Hermanns	Saarland University, Germany
Stefan Kowalewski	RWTH Aachen, Germany
Juliana Küster Filipe Bowles	University of St Andrews, UK
Frederic Lang	Inria, France
Diego Latella	ISTI-CNR, Italy
Odile Laurent	Airbus SAS Opérations, France
Stefan Leue	University of Konstanz, Germany
Amel Mammar	Telecom SudParis, France
Tiziana Margaria	University of Potsdam, Germany
Radu Mateescu	Inria, France
Pedro Merino	University of Málaga, Spain
David Parker	University of Birmingham, UK
Corina Pasareanu	CMU/NASA Ames Research Center, USA
Charles Pecheur	Université catholique de Louvain, Belgium
Jan Peleska	Universität Bremen, Germany
Ralf Pinger	Siemens AG, Germany
Andreas Podelski	University of Freiburg, Germany
Christophe Ponsard	CETIC, Belgium
Marco Roveri	FBK-irst, Italy
Cristina Seceleanu	Mälardalen University, Sweden
Marielle Stoelinga	University of Twente, The Netherlands
Jaco Van De Pol	University of Twente, The Netherlands

Additional Reviewers

Ahmad, Waheed
Ciancia, Vincenzo
Escobar, Santiago
Ferrari, Alessio
Franke, Dominik
Guck, Dennis
Itria, Massimiliano
Kacimi, Omar
Kupferman, Orna
Polini, Andrea
Seceleanu, Tiberiu
Serwe, Wendelin
Ujma, Mateusz
Van Glabbeek, Rob
Wiechowski, Nobert

Keynote Speakers

Twenty-Five Years of Formal Methods and Railways: What Next?

Alessandro Fantechi

Università di Firenze
Dip. di Ingegneria dell' Informazione
Via S. Marta 3, Firenze, Italy

Abstract. Railway signaling is now since more than 25 years the subject of successful industrial application of formal methods in the development and verification of its computerized equipment.
However the evolution of the technology of railways signaling systems in this long term has had a strong influence on the way formal methods can be applied in their design and implementation. At the same time important advances had been also achieved in the formal methods area.
The evolution of railways signaling systems has seen railways moving from a protected market based on national railway companies and national manufacturers to an open market based on international standards for interoperability, in which systems of systems are providing more and more complex automated operation, but maintaining, and even improving, demanding safety standards.
The scope of the formal methods discipline has enlarged from the methodological provably correct software construction of the beginnings to the analysis and modelling of increasingly complex systems, always on the edge of the ever improving capacity of the analysis tools, thanks to the technological advances in formal verification of both qualitative and quantitative properties of such complex systems.
In spite of these advances, the verification of complex railway signalling systems is still a main challenge and an important percentage of the cost in the development of these systems. We will discuss a few examples of such systems that witness these difficulties.
The thesis we will put forward in this talk is that the complexity of future railway systems of systems can be addressed with advantage only by a higher degree of distribution of functions on local interoperable computers - communicating by means of standard protocols - and by adopting a multi-level formal modelling suitable to support the verification at design time and at different abstraction levels of the safe interaction among the distributed functions.

TrustInSoft: Industrial Formal Methods to Protect Security-Sensitive Systems

Benjamin Monate

TrustInSoft
Bâtiment Erable, 86 rue de Paris, Orsay, France

Abstract. TrustInSoft is a start-up founded in 2013. It provides formal method-based solutions to help software publishers and integrators develop secure systems. In this talk I will explain the challenges we are facing as the software publisher of the Open Source platform Frama-C. I will discuss a few questions:

- Can theoretically well-founded formal methods be applied without compromises in non-regulated industrial domains?
- Who in the industry is ready to pay for formal security?
- Can a start-up afford an ambitious research roadmap in formal methods?

Table of Contents

Formal Methods for the Analysis of Critical Control Systems Models: Combining Non-linear and Linear Analyses

Adrien Champion[1,2], Rémi Delmas[1], Michael Dierkes[2], Pierre-Loïc Garoche[1], Romain Jobredeaux[4], and Pierre Roux[1,3]

[1] Onera – The French Aerospace Lab.
[2] Rockwell Collins France
[3] ISAE, University of Toulouse
[4] Georgia Institute of Technology, United States

Abstract. Critical control systems are often built as a combination of a control core with safety mechanisms allowing to recover from failures. For example a PID controller used with triplicated inputs and voting. Typically these systems would be designed at the model level in a synchronous language like Lustre or Simulink, and their code automatically generated from these models. We present a new analysis framework combining the analysis of open-loop stable controllers with safety constructs (redundancy, voters, ...). We introduce the basic analysis approaches: abstract interpretation synthesizing quadratic invariants and backward analysis based on quantifier elimination and convex hull computation synthesizing linear invariants. Then we apply it on a simple but representative example that no other available state-of-the-art technique is able to analyze. This contribution is another step towards early use of formal methods for critical embedded software such as the ones of the aerospace industry.

1 Control-Command Software Focused Analyses to Address V&V and Certification Needs

The aerospace industry is notoriously faced with highly critical issues. The safety of systems should be guaranteed even if the cost of ensuring safety is important. In development costs of the Boeing 777 [8], software accounts for a third of all costs. In this third, 70% consists in verification costs while only 30% are devoted to software development. Other aircraft manufacturers have similar figures.

The software specific certification regulatory document, *ie.* the recently updated DO 178-C, characterizes different levels of criticality from level A - the most critical - to level E - the less critical. Depending on the identified level, verification and validation activities are more or less intensive and therefore costly. This certification document has recently been updated and it also provides a formal methods supplement, identified as RTCA DO 333. This supplement explicitly enables the use of formal methods for critical embedded software.

Among the various systems of an aircraft, and their associated software, one of the most critical is the flight control system of the aircraft. Addressing the issue of

C. Pecheur and M. Dierkes (Eds.): FMICS 2013, LNCS 8187, pp. 1–16, 2013.

verifying such specific software seems to be a pertinent goal: proposing new ways to validate it could both increase the trust we have in the released software and reduce the cost of V & V by providing more automatic (and exhaustive) analysis means.

These reactive system can be seen as the composition of two parts. The first is the computation core itself, achieving the main objective of the software: controlling the aircraft by receiving inputs from sensors and commanding the aircraft actuators. The second part tries to handle any possible failure of sensors or of the core system. This safety architecture is mainly based on information redundancy and fusion. These two parts are usually designed using a model based approach.

The approach of control system modeling as proposed by The MathWorks with MATLAB Simulink, by Esterel Technologies with the SCADE language or by the academic community with the Lustre language, is extensively used for reactive systems design and often allows the automatic generation of the embedded code. However, despite the existence of a few formal verification tools supporting these languages, few system builders actually rely on formal approaches to demonstrate safety properties of their software products.

Recent advances of formal methods, as well as the evolution of certification standards enable the deployment of formal methods in the industry to analyze such systems. Formal methods can thus be truly considered as a key technological advantage on the critical systems market. Thanks to long term research efforts, formal methods have matured up to the point they were found, by industrials, to be helpful in dealing with the difficulties arising from highly complex system designs, and enabled system providers to meet the requirements of certification which are to:

- provide evidence of the system safety, and
- master the overall product life-cycle.

Our goal is to support the verification and validation of such systems for all their specification, at the various stages of their development. This paper focus on a representative running example: a controller for a physical device together with inputs triplication and voting. We show how we propose to analyze such systems by composing new-generation formal methods.

The paper is structured as follows: Section 2 recalls the state of the art of formal methods in that context; Section 3 presents the running example in details; Sections 4 and 5 introduce our contribution, two new automatic analyses; Section 6 illustrates the use of these new techniques in combination on the example; and Section 8 presents the tools implementing these techniques.

2 State of the Art of Formal Methods

Since the early 60s, researchers have proposed multiple theoretical frameworks to analyze systems and programs. These techniques, based on formal foundations – mainly discrete mathematics, algebra – allow the exhaustive study of all the behaviors of some categories of systems, as opposed to test or simulation methodologies which only cover the system traces identified by a collection of tests scenarios.

We briefly focus here on relevant techniques for the analysis of functional specifications of control systems. These techniques, from early academic work driven by industry needs, to actual transferred technologies[18], are currently used in the aerospace industry, at different TRLs[1].

2.1 Abstract Interpretation

Abstract interpretation was first proposed in the 70s as a general framework to express static analyses. In practice, it has shown to be very efficient to compute numerical invariants over programs.

The basic principle of this static analysis technique is to automatically compute an over-approximation of the set of all behaviors of the program (i.e. its semantics). This over-approximation is computed thanks to *abstract domains*. The role of abstract domains is twofold: (1) they characterize the nature of the over-approximation which is performed, (2) they are equipped with a set of functions, the *abstract primitives*, which allow to compute the abstract semantics.

Each abstract domain is associated with a given trade off between precision (depending on the kind of properties that it can infer) and efficiency (related to the computational complexity of its abstract primitives). A wide literature addresses the definition of abstract domains, and more specifically numerical abstract domains, *eg.* intervals [5], polyhedra [6] or weaker but less costly domains such as zones [13] or octagons [14].

A success story of abstract interpretation is the complete analysis of the flight control systems of the Airbus A380 [12], which has proved that no runtime error (out of bounds memory accesses or arithmetic exceptions) can occur.

One of the domains used to perform this analysis is specifically focused on second order linear filters. This domain is able to precisely over-approximate their set of reachable states. We present in this paper another similar abstraction that outperforms it in terms of expressiveness.

2.2 SMT-Based Verification Approaches

Satisfiability Modulo Theory. Satisfiability Modulo Theories solvers are decision procedures for logical theories in which some atoms belong to certain decidable first order theories such as linear real/integer arithmetic, the theory of bit-vectors, the theory of arrays (with read over write axioms), etc. Roughly speaking, these procedures are usually built by extending Boolean satisfiability procedures with a combination of dedicated background theory solvers [20].

SMT-solvers are used as back-end reasoning engines in a wide range of formal verification applications, such as deductive methods, bounded model checking, k-induction, test case generation, etc. Recently, the SMT-lib initiative (`http://www.smtlib.org/`) has gathered major SMT-solver developers around a standardized formula and solver command language. The SMT-lib 2.0 standard introduced specific features easing the implementation of incremental verification approaches like BMC or k-induction.

[1] Technology Readiness Levels as defined by NASA;
see `http://esto.nasa.gov/files/trl_definitions.pdf`

Quantifier Elimination. Assuming a first order formula over Boolean, real or integer variables $\exists x, \mathcal{F}(x, y_1, \ldots, y_n)$, where $\mathcal{F}$ is quantifier-free, quantifier elimination allows to generate a new formula

$$\mathcal{G}(y_1, \ldots, y_n) \equiv \exists x, \mathcal{F}(x, y_1, \ldots, y_n)$$

by *eliminating* the quantified variable x from $\mathcal{F}$. Slightly rephrased, quantifier elimination generates a condition $\mathcal{G}$ on variables $y_1, \ldots, y_n$ which, when satisfied, entails the existence of an x such that $\mathcal{F}(x, y_1, \ldots, y_n)$ is also satisfied.

Even though the theory of real closed fields admits quantifier elimination [4,19], general non-linear QE methods have extremely high computational costs, limiting their practical applications. This is why QE for linear fragments of integer and real theories has been a very busy research domain. The most recent advance for linear QE combines state of the art SMT-solving with polyhedral projection [15] for a great performance increase, the general idea of which is given in Algorithm 1.

Algorithm 1. $QE(\mathcal{F}, V)$ QE by Lazy Model Enumeration

Require: $\mathcal{F}$: a linear arithmetic formula.
Require: V: a collection of variables to eliminate from $\mathcal{F}$.
Ensure: $\mathcal{O}$: a formula in disjunctive normal form such that $\mathcal{O} \equiv \exists \mathcal{V}, \mathcal{F}$
  $\mathcal{O} \leftarrow \bot$
  while *isSatisfiable*$(\mathcal{F} \wedge \neg\mathcal{O})$ **do** ▷ check satisfiability using an SMT solver.
    $M \leftarrow$ *getModel*$(\mathcal{F} \wedge \neg\mathcal{O})$ ▷ get a model using an SMT solver.
    $P \leftarrow$ *extrapolate*$(\mathcal{F}, M)$ ▷ extrapolate M, yields a conjunction of literals which entail $\mathcal{F}$.
    $P' \leftarrow$ *project*(P, V) ▷ polyhedral projection.
    $\mathcal{O} \leftarrow \mathcal{O} \vee P'$
  end while
  return $\mathcal{O}$

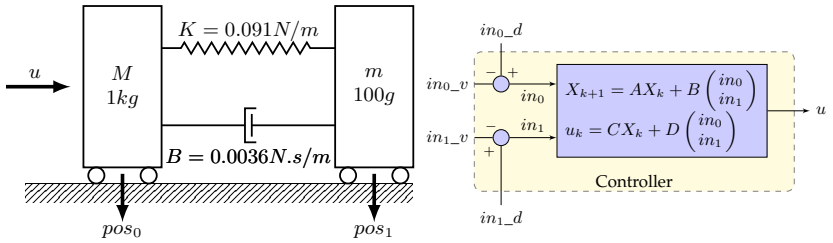
The *extrapolate* function generalizes the model M with respect to $\mathcal{F}$ and produces a conjunction of literals P, *i.e.* a polyhedron in geometric terms, such that $M \models P$ and $P \implies \mathcal{F}$. Polyhedral projection is then used to eliminate variables V from P and obtain another P' characterizing a polyhedron of lower dimension. The formula $\mathcal{O}$ resulting from this procedure can be viewed as a union of polyhedra over reals/integers.

Quantifier elimination enjoys many applications in formal verification: pre-image computation on transition systems, automatic program abstraction and even controller synthesis, to name only a few. Section 5 details our use of QE for pre-image computation and lemma generation.

3 Running Example: Coupled Mass System Linear Controller behind Triplicated Inputs with Saturations

Throughout this article, we consider the control of the coupled mass system[2] shown in Figure 1a. Such a coupling can be used to model numerous physical

[2] This system is extracted from [17].



(a) Coupled mass system (b) Controller node

Fig. 1. Coupled mass system and its linear controller

phenomena such as vibration propagation patterns in fluids and flexible structures, among others.

3.1 Controller Design

The continuous model for the plant, i.e. the physical system, is as follows:

$$\dot{x}_p = \begin{bmatrix} 0 & 1 & 0 & 0 \\ -K/m & -B/m & K/M & B/M \\ 0 & 0 & 0 & 1 \\ K/M & B/M & -K/M & -B/M \end{bmatrix} x_p + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1/M \end{bmatrix} u_p \qquad y_p = \begin{bmatrix} 1\,0\,0\,0 \\ 0\,0\,1\,0 \end{bmatrix} x_p$$

where the output y_p consists of the displacement of mass 1 and mass 2 with respect to their equilibrium position.

It is discretized with a period $T = 0.4s$ with a zero order hold, yielding:

$$x_{p,k+1} = \begin{bmatrix} 0.9285 & 0.3876 & 0.0715 & 0.0124 \\ -0.3516 & 0.9146 & 0.3516 & 0.0854 \\ 0.0071 & 0.0012 & 0.9929 & 0.3988 \\ 0.0352 & 0.0085 & -0.0352 & -0.9915 \end{bmatrix} x_{p,k} + \begin{bmatrix} 0.0013 \\ 0.0124 \\ 0.0799 \\ 0.3988 \end{bmatrix} u_{p,k} \quad y_{p,k} = \begin{bmatrix} 1\,0\,0\,0 \\ 0\,0\,1\,0 \end{bmatrix} x_{p,k}$$

Also assuming the presence of some sensor noise, the control theorist designs the following so called *linear quadratic Gaussian regulator*:

$$\begin{cases} x_{k+1} = Ax_{c,k} + By_{c,k} \\ u_{c,k} = Cx_{c,k} + Dy_{c,k} \end{cases}$$

with:

$$A = \begin{bmatrix} 0.6229 & 0.3871 & -0.117 & 0.0102 \\ -0.3363 & 0.9103 & -0.4062 & 0.06486 \\ 0.1134 & -0.02646 & -1.065 & 0.2669 \\ 0.2877 & -0.1298 & -1.333 & 0.3331 \end{bmatrix} \quad B = \begin{bmatrix} 0.3063 & 0.1866 \\ -0.009808 & 0.7406 \\ -0.07102 & 1.947 \\ -0.07649 & 0.7439 \end{bmatrix}$$

$$C = \begin{bmatrix} -0.1896 & -0.346 & 6.709 & -1.651 \end{bmatrix} \qquad D = \begin{bmatrix} 0.6311 & -8.098 \end{bmatrix}$$

The controller is parameterized by the positions of the two masses. Figure 1b illustrates the architecture of this linear controller, in which in_0_d and in_1_d correspond to the desired positions while in_0 and in_1 (denoted $y_{c,k}$ in the equations) are the differences between measured and desired positions.

Fig. 2. Simulation of the controlled system. $x_{p,i}$ correspond to actual plant states and $x_{c,i}$ to associated estimates in the controller. Simulation starts with an initial position of 1 for m_1.

This classic design uses a Kalman state estimator combined with a linear quadratic regulator. Typical high level properties guaranteed by this design include that the variance of the estimation error is minimized and convergence to zero in the absence of noise, as shown in Figure 2, obtained by simulation in the absence of noise and under $x_{p,0} = \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix}^{\mathrm{T}}$, $x_{c,0} = \begin{bmatrix} 0 & 0 & 0 & 0 \end{bmatrix}^{\mathrm{T}}$.

In the rest of this article, we focus on the simpler, yet essential bounded-input, bounded-output (BIBO) stability of the controller, as it is a high level property that can be established by inspecting the controller code alone. At the system level of description, inspecting the modulus (.84, .84, .0063 and .73) of the eigenvalues of the controller state matrix A is enough to guarantee that property. However, more elaborate techniques are required at code level since neither these eigenvalues nor their link with BIBO stability are readily available.

3.2 Safety Architecture

In real world systems, the controller will not be directly embedded on the target platform but rather used in conjunction with a safety architecture used to obtain a fault tolerant system.

In our case, we choose to rely on representative yet simple building blocks: validators and voters. We assume the sensors could be faulty and provide an erroneous data. Either the data could be out of a legal value range or it can vary within the legal range. First, a set of validators allows to correct the signals by saturating them to keep them within the legal range. Then, the triplication of each sensor allows to tolerate a faulty sensor. We rely on the triplex presented in [7].

This fault-tolerant system is depicted in Figure 3. To guard against execution hardware errors (CPU or RAM), one could pursue the experiment by triplicating the controller core and vote on its outputs, or rely on the command/monitor (COM/MON) architecture. An even more fault-tolerant and diagnosable system would also implement error detection flows with some alarm logic combining information about detected errors. However, in this paper we focus on numerical properties of control systems.

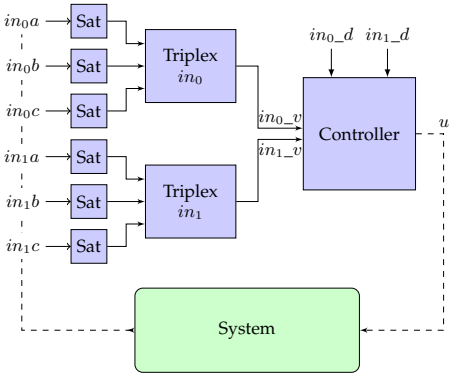


Fig. 3. Full controller

3.3 Need for Formal Specification

The specification of the obtained system should now be precised. We rely on axiomatic semantics with Hoare triples to associate a function contract to each building block of the system. For this running example, we can identify the following contracts:

- the system should ensure that the output on data flow u does not exceed the capability of the moving mass m_1 hardware actuator. For example, if the maximal capability is $200N$, we should ensure that $|u| \leq 200$;
- the controller itself has specific system-level properties as described above. In this paper we restrict on the BIBO property;
- the triplex voter contract, presented in [7], bounds the output of the voter according to bounds on the input. It is also a BIBO property, the formalisation and verification of which is detailed in Section 5.1;
- the *Sat* nodes correspond to the validator nodes evocated above for the replicated sensors, a typical specification would be that $lb \leq o \leq ub$ where o is the output flow, lb and ub are lower and upper saturation bounds.

These requirements should be satisfied by the blocks and their later implementations, and each of them should be formally expressed in a logic language format that can be parsed and processed by available tools.

This running example is simple but representative of control command software, and its analysis is meaningful since most of these properties are not currently analyzable at model or code level with existing formal tools.

4 BIBO (Bounded Input Bounded Output) for the Control Core: The Need for Non-linear Invariants

We focus here on the controller node and its BIBO property as characterized in Section 3.1. Assuming a bound on the inputs in (y_k in the equation), we want to bound x in the system $x_{k+1} = Ax_k + By_k$ and therefore the output u_k.

Abstract interpretation is a good way to compute numerical abstraction of reachable states, therefore computing bounds on reachable values. However the precision of the obtained results highly depends on the abstraction used (intervals, polyhedra, octagons, ...). When we assume a bound of $|y_{0_k}| \leq 3.6$ and $|y_{1_k}| \leq 3.6$, trying to analyze this with intervals gives for the first few iterates (rounding figures to one digit after the point):

x_0	x_1	x_2	x_3
$[0, 0]$	$[0, 0]$	$[0, 0]$	$[0, 0]$
$[-1.8, 1.8]$	$[-2.7, 2.7]$	$[-7.3, 7.3]$	$[-3.0, 3.0]$
$[-4.8, 4.8]$	$[-8.9, 8.9]$	$[-16.1, 16.1]$	$[-14.5, 14.5]$
$[-10.2, 10.2]$	$[-19.9, 19.9]$	$[-29.0, 29.0]$	$[-31.7, 31.7]$
$\vdots$	$\vdots$	$\vdots$	$\vdots$

This does not converge. Such linear system, so called *open-loop stable* (*ie.* admitting this BIBO property), usually do not admit linear inductive invariants. A possible approach to prove such stability is to rely on Lyapunov stability theory, stated as the existence of a positive definite[3] matrix P such that

$$P - A^T P A \succ 0$$

Finding any such P gives an invariant $x^T P x \leq x_0^T P x_0$ for the sequence $x_{k+1} = Ax_k$. When assuming bounds on the inputs y_k, it is possible to compute a scalar λ such that the set of values of $x_{k+1} = Ax_k + By_k$ could be bound by the relation $x^T P x \leq \lambda$.

We developed an approach combining the synthesis of an appropriate quadratic template P [16] for a given linear system with a static analysis engine based on policy iteration to compute the appropriate λ.

Given a finite set $\{ t_1, \ldots, t_n \}$ of expressions on program variables $\mathbb{V}$, the template domain $\mathcal{T}$ is defined as $\overline{\mathbb{R}}^n = (\mathbb{R} \cup \{ -\infty, +\infty \})^n$ and the meaning of an abstract value $(b_1, \ldots, b_n) \in \mathcal{T}$ is the set of environments $\gamma_{\mathcal{T}}(b_1, \ldots, b_n) = \{\rho \in (\mathbb{V} \to \mathbb{R}) \mid [\![t_1]\!](\rho) \leq b_1, \ldots, [\![t_n]\!](\rho) \leq b_n\}$. In other words, the abstract value $(b_1, \ldots, b_n)$ represents all the environments satisfying all the constraints $t_i \leq b_i$. In our case, in addition to templates allowing to bound each variable, a quadratic template will be added for each linear system of the program.

Example 1. We then have five templates on the running example: (1) $t_1 := 0.098\, x_3^2 - 0.224\, x_3\, x_2 + 0.040\, x_3\, x_1 - 0.026\, x_3\, x_0 + 0.141\, x_2^2 - 0.053\, x_2\, x_1 + 0.030\, x_2\, x_0 + 0.024\, x_1^2 - 0.017\, x_1\, x_0 + 0.019\, x_0^2$ (2) $t_2 := x_0^2$ (3) $t_3 := x_1^2$ (4) $t_4 := x_2^3$ (5) $t_5 := x_5^2$.

Policy iterations [1,10,11] give then a direct methods to compute a precise over-approximation of the collecting semantics of a program, given a set of such templates. It intends to achieve better precision than the usual widening/narrowing approach of abstract interpretation Kleene iterations by computing a numerical solution of the problem.

[3] A matrix M is positive definite (noted $M \succ 0$) if $x^T P x > 0$ for all $x \neq 0$.

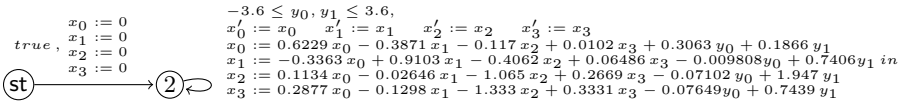


Fig. 4. Control flow graph for our running example

While Kleene iterations iterate locally through each construct of the program, policy iterations require a global view of the analyzed program. For that purpose, the whole program is first translated into a system of equations which is later solved.

A first step in deriving these equations from the program is to build its control flow graph.

Example 2. Figure 4 represents the control flow graph for our running example. Vertex "st" corresponds to the starting point of the program and vertex "2" to the head of the loop. The edge between "st" and "2" reflects the four assignments before the loop and the looping edge on vertex "2" represents the loop body.

It is worth noting that, unlike the usual notion of control flow graph with vertices between each single instruction of the program, whole sequences of instructions are used to label the edges, with graph vertices only for starting point and loop heads of the program. This will both improve the precision of the analysis and decrease its cost by avoiding useless intermediate abstractions.

A system of equations is then defined with a variable $b_{i,j}$ for each vertex i of the graph and each template t_j.

Example 3. Here is the system of equations for our running example:

$$\begin{cases} b_{1,1} = +\infty \quad b_{1,2} = +\infty \quad b_{1,3} = +\infty \quad b_{1,4} = +\infty \quad b_{1,5} = +\infty \\ b_{2,1} = \max\{0 \mid \mathrm{be}(1)\} \vee \max\{a(t_1) \mid -3.6 \le in \le 3.6 \wedge \mathrm{be}(2)\} \\ b_{2,2} = \max\{0 \mid \mathrm{be}(1)\} \vee \max\{a(t_2) \mid -3.6 \le in \le 3.6 \wedge \mathrm{be}(2)\} \\ b_{2,3} = \max\{0 \mid \mathrm{be}(1)\} \vee \max\{a(t_3) \mid -3.6 \le in \le 3.6 \wedge \mathrm{be}(2)\} \\ b_{2,4} = \max\{0 \mid \mathrm{be}(1)\} \vee \max\{a(t_3) \mid -3.6 \le in \le 3.6 \wedge \mathrm{be}(2)\} \\ b_{2,5} = \max\{0 \mid \mathrm{be}(1)\} \vee \max\{a(t_4) \mid -3.6 \le in \le 3.6 \wedge \mathrm{be}(2)\} \end{cases}$$

where $\mathrm{be}(i)$ is a shorthand for $t_1 \le b_{i,1} \wedge t_2 \le b_{i,2} \wedge t_3 \le b_{i,3} \wedge t_4 \le b_{i,4} \wedge t_5 \le b_{i,5}$ and $a(t)$ is the template t in which variable x_0 to x_3 are replaced by their image by the linear combination $Ax_k + By_k$. The usual maximum on $\overline{\mathbb{R}}$ is denoted $\vee$.

Each $b_{i,j}$ bounds the template t_j at program point i and is defined in one equation as a maximum over as many terms as incoming edges in i. More precisely, each edge between two vertices v and v' translates to a term in each equation $b_{v',j}$ on the pattern: $\max\left\{a(t_j) \;\middle|\; c \wedge \bigwedge_j t_j \le b_{v,j}\right\}$ where c and a are respectively the constraints and the assignments associated to this edge. This expresses the maximum value the template t_j can reach in destination vertex v' when applying the assignments a on values satisfying both the constraints c of the edge and the constraints $t_j \le b_{v,j}$ of the initial vertex v.

Our tool computes these steps automatically when provided with the set of variables that participate in a BIBO linear controller. It rebuilds the control flow graph and characterizes linear relations, synthesizes the associated quadratic templates and performs the policy iterations computation to obtain the bounds.

On the running example, we obtain $0.098\,x_3^2 - 0.224\,x_3\,x_2 + 0.040\,x_3\,x_1 - 0.026\,x_3\,x_0 + 0.141\,x_2^2 - 0.053\,x_2\,x_1 + 0.030\,x_2\,x_0 + 0.024\,x_1^2 - 0.017\,x_1\,x_0 + 0.019\,x_0^2 \leq 14.259$ resulting in following bounds on variables

$$|x_0| \leq 25.423 \quad |x_1| \leq 26.844 \quad |x_2| \leq 33.612 \quad |x_3| \leq 37.164 \quad |u| \leq 194.499.$$

Figure 5 depicts a cut of this invariant along plane $x_2 = x_3 = 0$. The ellipsoid shape corresponds to the bound on the quadratic polynomial while vertical cuts on the x_0 axis correspond to the bound on $|x_0|$.

Indeed, the largest value we were able to reach by random simulation for variable u is 190.019. The actual maximum reachable value may be higher but it is worth noting that our bound 194.499 is less than 2.4% larger. Other linear abstractions perform poorly on such reactive systems, or on typical aircraft controllers. Our solution is automatic when provided with the set of variables of the considered linear system.

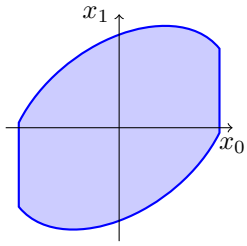


Fig. 5. Resulting quadratic invariant projected on variables x_0 and x_1

5 BIBO Stability for the Triplex Voter: Linear Analysis

5.1 Triplex Voting Verification Challenges

The Triplex node used in the presented controller design implements redundancy management for three sensor input values *InA*, *InB*, *InC*, aiming at providing sensor fault tolerance. This voter does not compute an average value, but uses the $\mathbf{median}(x, y, z)$ function, which returns the median of its arguments(for instance z when $y \leq z \leq x$). The values considered for voting are *equalized* by substracting equalization values from the inputs. The role of the equalization values is to compensate offset errors of the sensors, assuming that the middle value gives the most accurate measurement. The recursive equations in Figure 6 describe the behavior of the voter. The $\mathbf{sat}_c(x)$ function saturates its argument such that $|\mathbf{sat}_c(x)| \leq c$.

Just as with the controller core, we are interested in proving the BIBO stability of the voter, that is in ensuring that the voter output cannot grow indefinitely as long as its inputs remain within a certain range. This property is needed for the overall design validation since the BIBO stability of the controller core makes assumptions on the voter outputs. Here, the difficulty of automating the voter proof lies in inferring the right auxiliary inductive invariant on the *internal*

initialization

$$
\begin{array}{lcl}
EqualizationA_0 & = & 0.0 \\
EqualizationB_0 & = & 0.0 \\
EqualizationC_0 & = & 0.0
\end{array}
$$

transition $\forall k \in \mathbb{N}$

$$
\begin{array}{lcl}
EqualizedA_k & = & InA_k - EqualizationA_k \\
EqualizedB_k & = & InB_k - EqualizationB_k \\
EqualizedC_k & = & InC_k - EqualizationC_k \\
SatA_k & = & \mathbf{sat}_{0.5}(EqualizedA_k - Output_k) \\
SatB_k & = & \mathbf{sat}_{0.5}(EqualizedB_k - Output_k) \\
SatC_k & = & \mathbf{sat}_{0.5}(EqualizedC_k - Output_k) \\
EqualizationA_{k+1} & = & EqualizationA_k + 0.05 * (SatA_k - SatCentering_k)) \\
EqualizationB_{k+1} & = & EqualizationB_k + 0.05 * (SatB_k - SatCentering_k)) \\
EqualizationC_{k+1} & = & EqualizationC_k + 0.05 * (SatC_k - SatCentering_k)) \\
Centering_k & = & \mathbf{median}(EqualizationA_k, EqualizationB_k, EqualizationC_k) \\
SatCentering_k & = & \mathbf{sat}_{0.25}(Centering_k) \\
Output_k & = & \mathbf{median}(EqualizedA_k, EqualizedB_k, EqualizedC_k)
\end{array}
$$

Fig. 6. Triplex voter equations

equalization values *EqualizationA*, *EqualizationB*, *EqualizationC*, whereas the main proof obligation and assumptions are expressed only on the voter's *interface* variables *InA*, *InB*, *InC* and *Output*. Equation (1) shows a refined, yet still not inductive *BIBO* property including the equalization values.

$$
\begin{array}{l}
BIBO(a) \equiv \forall k \in \mathbb{N}, \quad |InA_k| \leq a \wedge |InB_k| \leq a \wedge |InC_k| \leq a \implies |Output_k| \leq 3a \wedge \\
|EqualizationA_k| \leq 2a \wedge |EqualizationB_k| \leq 2a \wedge |EqualizationC_k| \leq 2a
\end{array}
\tag{1}
$$

In [7], the author managed to manually identify octagonal regions enclosing the equalization values (a relational invariant over the sums of equalization values), and prove them using a k-induction tool. The k-induction technique, as it can only prove user-specified properties and not infer new properties by itself, is of no help in this situation. Discovering an appropriate invariant using abstract interpretation is theoretically possible. However, the combination of the non-linear **median** and $\mathbf{sat_c}$ functions with the filter-like calculations over equalization values is such that abstract interpretation cannot infer the desired invariants without specifying numerous domain partitioning directives, which must be guessed by the human user, a task impossible to achieve in practice.

5.2 HullQe: A Technique for Property Directed Invariant Generation

The difficulties posed by systems such as the triplex voter lead us to design a new invariant generation technique named HullQe, based on a backwards state space traversal and polyhedral computations [3].

Given a triplet $\langle\langle I, T\rangle, P\rangle$, where $\langle I, T\rangle$ is transition system specified by an initial state predicate I, and a transition predicate T, and P is a proof objective, HullQe works as follows.

A backwards state space traversal allows to discover successive fringes of *gray states*. Gray states are states which satisfy P but from which a states violating P

can be reached in one or more T-transitions. When only numerical state variables are considered, such fringes of gray states can be thought of as unions of polyhedra over state variables. The quantifier elimination procedure of Algorithm 1 is iterated as follows to discover new gray states:

$$G_1(s) \equiv QE(s', P(s) \wedge T(s, s') \wedge \neg P(s'))$$
$$G_n(s) \equiv QE(s', P(s) \wedge T(s, s') \wedge G_{n-1}(s'))$$

At each iteration n HullQe first checks whether $I(s) \wedge \bigvee_{i \in [1,n]} G_i(s)$ is satisfiable, which would indicate that there exists a path to $\neg P(s)$ from the initial states. HullQe also checks whether $\bigvee_{i \in [1,n-1]} G_i(s) \implies G_n(s)$ is valid, which would indicate a fixed point has been reached in the backwards exploration.

In order to discover new invariants, HullQe performs a thorough exploration of all possible partitionnings of the gray states discovered so far. The following sets of new polyhedra are computed until no new polyhedra can be found or merged. This computation takes place after each pre-image calculation and is initialised with all pre-images found so far ($\bigvee_i G_i$):

$$\begin{array}{llllll}
H_1 = \{ & p & , & p \in \bigvee_i G_i & & \} \\
H_2 = \{ & hull(p, p') , & p \in & H_1 , & p' \in H_1 & \} \\
H_3 = \{ & hull(p, p') , & p \in & H_2 , & p' \in H_1 \cup H_2 & \} \\
\vdots & \vdots & & \vdots & \vdots & \\
H_n = \{ & hull(p, p') , & p \in & H_{n-1} , & p' \in \bigcup_{i \in [1,n-1]} H_i & \}
\end{array}$$

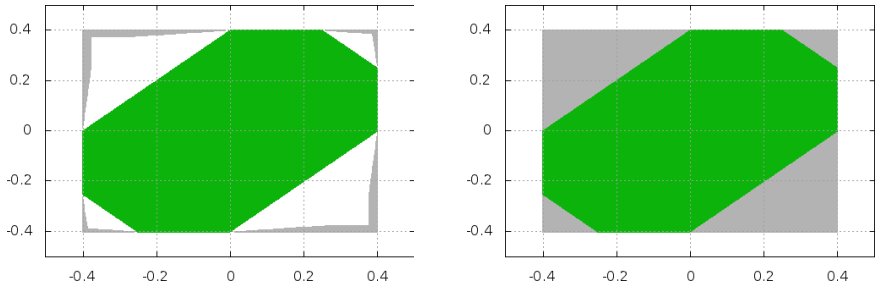
In the above definitions, the *hull* function, which computes a convex hull of two polyhedra, can be used in either *exact mode*, whereby a hull will be returned only if it does not contain more points than the two source polyhedra, or in *inexact mode*, by accepting hulls which contain strictly more points than the two given polyhedra, thus performing abstraction by accepting more states than already present in the root polyhedra. When using the inexact mode, it is possible to accept inexact hulls only for polyhedra with a non-empty intersection. This allows to overapproximate clusters of gray states while avoiding to merge completely disconnected zones of the gray state space. In order to limit the combinatorial explosion, for each hull the algorithm keeps track of the base polyhedra it is derived from, and mathematical properties of convex hulls as well as dynamic programming techniques are used to skip uninteresting combinations of polyhedra, which would necessarily result in already existing hulls, or for which the exact hull computation would fail.

Each polyhedra in $H_1, \ldots, H_n$ is viewed as a conjunction of linear relations over state variables delimiting portions of the gray state space. If P is invariant, none of the gray states should be reachable, so the negations of these relations are then taken as proof obligations and analyzed using k-induction in conjunction with P. Many of the generated relations will be non-invariant or even falsifiable, but sometimes some of them can successfully strengthen the original proof obligation.

Let us illustrate the HullQe technique on a two-input voter, for which the *BIBO* proof objective is that:

$$BIBO(a) \equiv \forall k \in \mathbb{N}, |InA_k| \leq a \wedge |InB_k| \leq a \implies \\ |Output_k| \leq 3a \wedge |EqualizationA_k| \leq 2a \wedge |EqualizationB_k| \leq 2a$$

The Figure 7a shows a plot of the state space for state variables *EqualizationA* and *EqualizationB* of the two-input voter and the proof objective $BIBO(0.2)$. In green, we show the octagonal invariants of [7], in gray, we show the gray states polyhedra obtained after two iterations of the pre-image computation. The Figure 7b shows the result of HullQe's inexact hull computation modulo non-empty intersection on the two-input voter. We can see that the gray regions obtained by HullQe match exactly the octagonal invariant of [7], their negation then exactly delimits the octagonal region enclosing the equalization values.



(a) Octagonal invariant and gray states (b) HullQe inexact hull computation

Fig. 7. Two-input voter

The following results are obtained when analyzing $BIBO(1.2)$ (cf Equation 1) on the triplex voter: the first pre-image of the negated proof objective contain 23 distinct polyhedra, the HullQe lemma generation algorithm creates 41 potential lemmas, out of which 32 are found to be 1-inductive and allow to strengthen the proof objective. Out of all the generated lemmas, the following two suffice to prove the $BIBO(1.2)$ proof objective:

$$-29/10 \leq EqualizationA_k + EqualizationB_k + EqualizationC_k \quad (2)$$

$$EqualizationA_k + EqualizationB_k + EqualizationC_k \leq 29/10 \quad (3)$$

So, we see that, given a non-inductive and non-relational proof objective on the input, output and internal variables of the voter, HullQe is able to discover an inductive polyhedra on internal state variables, which renders the *BIBO* proof objective inductive.

6 Composing Proofs

Using the two previous techniques, implemented in our prototypes, it is now possible to perform an analysis of the complete system presented in Figure 3. When run in parallel, the analyzers communicate their results to each other. We only assume the following knowledge:

- The **Triplex** node description is fitted with the contract presented in Equation 1, page 11;
- The **Controller** node is known to be a linear open stable controller; its internal variables are automatically considered for analysis with our quadratic templates analysis.

We would like to guarantee that the system does not diverge whatever the inputs are. This contract could either be implicit, *ie.* no overflow, or specific, *eg.* the output should satisfy a given constraint $|u| \leq 200(N)$.

We recall that – up to our knowledge – none of these specifications is provable by any academic nor commercial tool available except ours. Let us describe the sequence of analyses needed to achieve the global proof:

1. Using abstract interpretation, **Sat** node outputs are bounded to their saturation value of 1.2;
2. These bounds enable the instanciation of the **Triplex** contracts. As described in Section 5, taking the instanciated contracts as proof objectives, HullQE with k-induction generates the missing lemmas and proves the contracts, effectively bounding the output of both voters by 3.6. The analysis is fully automatic once the proof objective is given.
3. Once the inputs of the controller node are bounded (*ie.* voter outputs), the analysis of Section 4 is enabled: the control flow graph is computed, the quadratic template is synthetized and the policy iteration is performed, bounding the internal variables and the output u of the controller: $|u| \leq 194.499$. The analysis is fully automatic and only requires the list of internal variables of the controller as well as bounds on the inputs.

Finally the global contract can be checked. The quadratic invariant synthesized by abstract interpretation satisfies the required bound of $200N$.

7 Tools

The presented analyses are currently implemented in two different tools, freely available.

Stuff [2] is a Lustre analysis framework that combines a k-induction engine with the HullQE technique presented in Section 5.2. It is about 30k loc of Scala. It uses the Actor-oriented programming approach to use the multiple CPU cores of the machine. It must be provided with the Lustre code as weel as a set of proof objectives to analyse.

SMT-AI [9] is an abstract interpreter that targets Lustre models. It computes the over-approximation of reachable values. It relies on the APRON library for linear domains and also embed the analysis presented in Section 4. It is written mainly in Ocaml for about 20k loc, numerical computation are performed with the CSDP and OCaml-GLPK libraries and Scilab.

8 Concrete Outcomes and Future Works

We proposed a combination of formal methods to achieve the verification of formal functional specifications of control-command systems. This framework of analysis is highly generic and automatic. The two presented analyses are able either to compute precise information about reachable states or to validate functional contracts; in both cases without needing to interact with the user.

The kind of systems targeted by this work – linear controllers stable in open-loop analysis, with a safety architecture based on redundancy and voters – are not analyzable in general at code or model level by any other tool we are aware of.

The objective of the approach is to enable the analysis of realistic aerospace critical software such as guidance and control-command systems, enabling the analysis of system-level specifications (stability, fault tolerance) on the actual implementation.

Future works will focus on a larger set of properties, *eg.* performance properties, and additional fault-tolerance constructs. We also consider studying more complex controllers, for example the combination of linear controllers switched according on the aircraft flight mode.

References

1. Adjé, A., Gaubert, S., Goubault, E.: Coupling policy iteration with semi-definite relaxation to compute accurate numerical invariants in static analysis. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 23–42. Springer, Heidelberg (2010)
2. Champion, A., Delmas, R.: Stuff: Stuff is the ultimate formal framework., `https://cavale.enseeiht.fr/redmine/projects/stuff`
3. Champion, A., Delmas, R., Dierkes, M.: Generating property-directed potential invariants by backward analysis. In: FTSCS, pp. 22–38 (2012)
4. Collins, G.E.: Hauptvortrag: Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In: Brakhage, H. (ed.) GI-Fachtagung 1975. LNCS, vol. 33, pp. 134–183. Springer, Heidelberg (1975)
5. 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)
6. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: POPL, pp. 84–97. ACM Press (1978)
7. Dierkes, M.: Formal analysis of a triplex sensor voter in an industrial context. In: Salaün, G., Schätz, B. (eds.) FMICS 2011. LNCS, vol. 6959, pp. 102–116. Springer, Heidelberg (2011)

8. Feron, E., Brat, G.: Formal methods for areospace applications. In: FMCAD 2012 Tutorial (2012)
9. Garoche, P.-L., Roux, P.: SMT-AI: SMT abstract interpreter, https://cavale.enseeiht.fr/redmine/projects/smt-ai
10. Gawlitza, T.M., Seidl, H.: Computing relaxed abstract semantics w.r.t. Quadratic zones precisely. In: Cousot, R., Martel, M. (eds.) SAS 2010. LNCS, vol. 6337, pp. 271–286. Springer, Heidelberg (2010)
11. Gawlitza, T., Seidl, H., Adjé, A., Gaubert, S., Goubault, E.: Abstract interpretation meets convex optimization. J. Symb. Comput. 47(12) (2012)
12. Kästner, D., Wilhelm, S., Nenova, S., Cousot, P., Cousot, R., Feret, J., Miné, A., Mauborgne, L., Rival, X.: Astrée: Proving the absence of runtime errors. In: ERTSS (2010)
13. Miné, A.: A new numerical abstract domain based on difference-bound matrices. In: Danvy, O., Filinski, A. (eds.) PADO 2001. LNCS, vol. 2053, pp. 155–172. Springer, Heidelberg (2001)
14. Miné, A.: The octagon abstract domain. In: AST (satt. of WCRE), pp. 310–319. IEEE (2001)
15. Monniaux, D.: Quantifier elimination by lazy model enumeration. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 585–599. Springer, Heidelberg (2010)
16. Roux, P., Jobredeaux, R., Garoche, P.L., Féron, E.: A generic ellipsoid abstract domain for linear time invariant systems. In: HSCC. ACM (2012)
17. Rowell, D.: Dicrete time observers and lqg control. MIT, Dpt. of Mechanical Engineering – 2.151 Advanced System Dynamics and Control (2004), http://web.mit.edu/2.151/www/Handouts/Kalman.pdf
18. Souyris, J., Favre-Flix, D.: Proof of properties in avionics. In: Building the Information Society, vol. 156, pp. 527–535. Springer (2004)
19. Tarski, A.: A decision method for elementary algebra and geometry: Prepared for publication with the assistance of j.c.c. mckinsey. Technical report, RAND Corporation (1951)
20. Tinelli, C.: Foundations of satisfiability modulo theories. In: WoLLIC, p. 58 (2010)

HyRev: A Tool for the Automatic Generation of Real-Time Routines for Enabling Fail-Safe Control in a Class of Safety-Critical Embedded Systems Using Backwards Reachability Analysis

Hallstein Asheim Hansen

Buskerud University College, Kongsberg, Norway
Hallstein.Asheim.Hansen@hibu.no

Abstract. A fail-safe embedded system is a system that will transit to a safe state in the event of a system failure. In these situations the system will typically switch from the normal, now faulty, operational mode to an emergency control mode which will ensure the safety of the system. The switch will have a hard real-time constraint if the results of a temporal failure are catastrophic in nature. Many industry-critical systems fall into this category, such as industrial plants and vehicles. We show how hybrid automata can be used to model a failing system and how backwards reachability analysis of this model and a given model of the emergency control can be used to prove the conditions under which safety switching will always succeed in ensuring fail-safe behavior. To show the feasibility of the technique we present the prototype tool HyRev. The tool takes a description of the emergency control system and the catastrophic bad states of the system as input and produces a safety check routine with a well-defined worst-case execution time as output, which can then be run on the embedded system.

1 Introduction

An embedded system is a computer system implementing the functions of control systems [11, 13]. Many embedded systems are subject to safety and hard real-time requirements where the consequences of a failure are potentially catastrophic, such as human injuries or damage to property or to the environment. An embedded system is fail-safe if, upon a system failure, the system will always transit to a safe state by executing an emergency control routine. We say that this system satisfies a specific *safety* property, i.e. that something bad should never happen [4]. Some embedded systems can easily transit to a safe state, e.g., by shutting off power to the system. Others need a more complex emergency control routine. Examples of the latter include chemical plants and automated assembly lines, and the control systems of unmanned vehicles and the safety systems of manned vehicles. Many embedded systems are *periodic* and realized as distributed systems of computational nodes, each node functioning as

C. Pecheur and M. Dierkes (Eds.): FMICS 2013, LNCS 8187, pp. 17–31, 2013.

a fault-containment unit [11]. Different nodes read the state of their environment through sensors, compute outputs, and write those outputs to the system actuators on a periodic basis, with a typical period duration of about $100\mu s$.

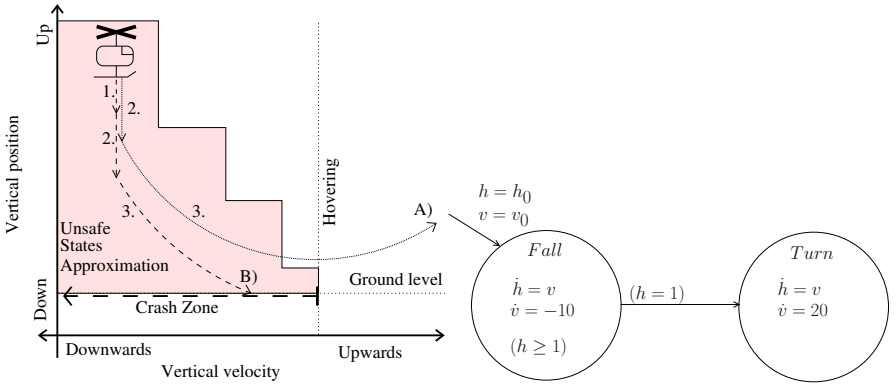
In safety-critical systems it is imperative that the *safety check*, the routine which evaluates the condition for the transition to emergency control, never fails if the condition holds. A failed system can conceivably exhibit any kind of physically allowed, or *free*, behavior so the safety check must run every period and have a well defined worst-case execution time. We say that the safety check should be both *conservative* and *time-bounded*. The safety check itself consists of evaluating a condition on the current state of the system with respect to the future evolution of the system. In many cases it may not be obvious when a system has arrived in a state that may potentially lead to an accident, as this may depend on complex relationships amongst system and environment values, such as position, velocity, or temperature, and on the characteristics of the emergency control. In our solution this condition consists of considering all potential consequences of allowing the (possibly faulty) normal operation to run for one more period, i.e. evaluating all possible future behaviors where:

Step 1. The switch to emergency control is *not* immediately performed, i.e. a further period of free behavior elapses.
Step 2. The switch takes at least two periods to execute, one to run the safety check and one to produce the first actuator values for the emergency control. Due to physical inertia this response time, where free behavior is allowed, is typically larger by orders of magnitude than the minimum two periods.
Step 3. The emergency control is then run, successfully or not

Any possible future, catastrophic evolution is prevented from occurring by immediately switching to emergency control if the safety check detects a potentially bad future state, since the safety check is run every period. The *free time* of the system is the duration of steps 1 and 2 during which free behavior can occur.

As an example we will consider an unmanned aerial vehicle (UAV). Many UAVs are examples of safety-critical, hard real-time embedded systems, typically in the form of small planes or helicopters equipped with a computer system and sensors. In the case of failure the UAV cannot trivially fail safe if it is flying, indeed an uncontrolled landing is a serious hazard. The switching condition is not obvious in this case. In Figure 1-(a) we illustrate a helicopter which in the last period entered a state where possible future evolutions can lead to a catastrophic crash landing unless action is taken immediately. Now the helicopter has a number of possible future evolutions, including the evolutions A) and B) drawn in the figure. In evolution A) the switch to emergency control is initiated immediately, step 2, and the emergency control of step 3 saves the helicopter. In evolution B) the switch is not taken, step 1, and the subsequent switch and emergency control in steps 2 and 3 fail to save the helicopter.

The behavior of embedded systems can be modeled by a combination of continuous evolution over time and near instantaneous discrete state changes. The



(a) Problem illustration (b) Hybrid automaton model

Fig. 1. A UAV

analysis of such a *hybrid system* has traditionally been performed using simulations and visual inspection of phase portraits of the system [13], neither of which allow rigorous proofs of the desirable system properties. A *hybrid automaton* is a mathematical model of a hybrid system [1], allowing formal analysis through model checking techniques [4] as well as simulation of the model. Customarily forward reachability searching is used to analyze properties of hybrid automata [6, 3, 10], but this approach is problematic in our case: We do not know the initial states of the system, indeed the purpose of this work is to identify these initial states from which the safety check will be generated. Our solution is to use backwards reachability techniques [1]: We start with the undesirable states of the system and compute all possible backwards evolutions of the safety check. The resulting *reach set* is an over-approximation of the states that can lead to potentially undesirable future states, unless the switch is made immediately.

HyRev is a prototype implementation of the method introduced above. HyRev is implemented in Python, and also includes an implementation of an approach for over-approximating affine hybrid automata by linear hybrid automata [2, 9]. By showing how linear hybrid automata can be *reversed* we may use standard forward reachability solvers as part of the tool chain, using mature, existing systems [12, 3, 10] to solve a practical problem. Our implementation incorporates the state-of-the-art tool SpaceEx as part of the tool chain [6]. We give experimental evidence that the computational effort of automaton reversal is negligible and that over-approximation errors of the results produced by our method can be reduced.

The rest of the paper is structured as follows: In Section 2 we present the mathematical preliminaries used in the rest of the paper. We develop and prove the results required for modeling and for analysis of safety switching embedded systems in Section 3. The tool HyRev is presented in Section 4, and we discuss related work and conclude in Section 5.

2 Preliminaries

A mathematical formalism for describing embedded control systems is the *hybrid automaton* [1], and in this section we present this and other definitions needed in the rest of the paper.

Definition 1 (Linearity)

- *A linear term over a set of variables* $\mathbf{x}$ *is a linear combination* $\theta = \Sigma_{i=1}^{n} \alpha_i x_i$, $\alpha_i \in \mathbb{R}$, $x_i \in \mathbf{x}$.
- *A linear predicate* φ *is a positive boolean combination of predicates of the form* $\theta \sim c$, *where* θ *is a linear term, and* $\sim$ *is one of the relation symbols* $\{<, \leq, =, \geq, >\}$, *and* c *a constant.*
- *The set* $\{v \mid \varphi\}$ *of all true valuations* $v \in \mathbb{R}^n$ *of* φ *is a linear set if* φ *is a linear predicate.*

Definition 2 (Hybrid automaton). *A* hybrid automaton H *is a tuple* $(Loc, Var, Tra, Act, Inv, Guard, Asg)$ *where*

- $Loc = \{l_1, \ldots, l_m\}$ *is a finite set of locations.*
- $Var = \{x_1, \ldots, x_n\}$ *is a finite set of real-valued variables and* $V \subseteq \mathbb{R}^n$ *the set of their valuations. The* state $(l_i, v_1, \ldots, v_n)$ *of a hybrid automaton is the current location and current valuation of the variables.*
- *Inv, the invariants, are mappings from the locations to predicates on variable valuations, i.e.* $Inv(l) \subseteq V$, *restricting the possible valuations of the variables.*
- $Tra \subseteq Loc \times Loc$ *is a set of transitions.*
- *The function Guard maps transitions to guards, where each guard* $Guard(l, l') \subseteq V$.
- *Asg is a function that maps a set of assignments to transitions, where each assignment* $Asg(l, l') \subseteq Guard(l, l') \times Inv(l')$
- *The function Act, the activities, maps locations to sets of differential equations, one for each variable* $x \in Var$.
- *Init and Final are two sets of states.*

If $x \in Var$ *only occurs in linear atomic predicates of the invariants, guards, and assignments of some hybrid automaton, then* x *is a linear variable. If all variables of a hybrid automaton are linear and the activities are given as constant differential equations we call it a* linear hybrid automaton. *If the activities are given as linear differential equations* $x' = \theta$, *where* θ *is a linear term, and invariants and assignments are given as above, then we call it an* affine *hybrid automaton.*

When necessary we will write $Asg(H; l; x)$ to refer to the assignment of variable x in location l of automaton H, and so on. Reachability is undecidable for linear hybrid automata in general, but there exist efficient methods for solving it and tools have been developed which implement such methods [10, 3, 6], tools which can be used to realize safety check generation.

Example 1. We consider a simplified model of an unmanned aerial vehicle, such as a helicopter or an airplane, where we only treat the vertical position h and vertical velocity v of the system. The purpose of the system is to prevent a free-falling vehicle, i.e. with gravitational acceleration $g = -10m/s^2$, from crashing into the ground, $h = 0$. This is done by accelerating the vehicle upwards with acceleration $a_{uav} = 20m/s^2$, when the height reaches a certain minimal value $h = 1m$. We only consider values in the range $0 - 4m$ for height, and -12 to $12m/s$ for velocity. In Figure 1-(b) we give a graphical illustration of a hybrid automaton H_{uav} modeling the system described above with initial velocity v_0 and position h_0.

The evolutions of a system, see Figure 1-(a), can be formalized as runs:

Definition 3 (Run). *The state of a hybrid automaton H may change in two different ways:*

- *Discrete transitions:* $(l_i, v_i) \rightarrow (l_{i+1}, v_{i+1})$
- *Continuous transitions:* $(l_i, v_i) \rightarrow_{f_i}^{t_i} (l_i, f_i(t_i))$, *where $f_i(t_i)$ represents the value of differential equations $f_i \in Act(l_i)$ with initial values v_i at time t_i, $t_i \in \mathbb{R}_{\geq 0}$,*

A run *ξ is a single state or a finite or infinite alternating sequence of continuous and discrete transitions $s_0 \mapsto_{t_0}^{f_0} s_1 \mapsto_{t_1}^{f_1} \ldots s_N$, subject to*

- $s_0 \in Init(H)$.
- $f_i(0) = v_i$.
- $f_i(t_i) \in Inv(l_i)$, *for all* $0 \leq t \leq t_i$.
- *If $s_i' = (l_i, f(t_i))$ is the continuous transition successor of s_i, then s_{i+1} is the discrete transition successor of s_i'.*

We say that a run ξ_f is a prefix *of a run ξ if ξ_f is of finite length, and is identical to the initial transitions of ξ.*

The set of all runs of a hybrid automaton H is denoted by $[H]$, and provides a way of comparing hybrid automata:

Definition 4 (Over-approximation). *We say that a hybrid automaton H* over-approximates *a hybrid automata H' if both $Var(H) = Var(H')$ and $\xi \in [H']$ implies $\xi \in [H]$.*

If we disregard the location part l of the states (l, v) of a hybrid automaton, we can treat the values $v \in \mathbb{R}^n$ of the reach set as data objects located in n-dimensional space. For a safety check whose run-time is a function of the cardinality of the reach set, this reach set must have a known upper bound to ensure a well-defined worst-case execution time of the check. To reduce the number of objects in the reach set we can represent the reach set as a *spatial* data structure such as an R-tree [8] and use operations on R-trees to produce a set of hyper-rectangular boxes with the desired cardinality, this set being an over-approximation of the reach set.

3 Results

As mentioned in Section 1 we explore the state space of the system model backwards in our analysis. While it is possible to compute backwards reachability directly [1], we want to use one of the many existing high quality forward analysis tools. This requires us to define a linear hybrid automaton that is the *reverse* of the system model, which can then be analyzed forwards.

Definition 5 (Reverse linear hybrid automaton). *We say that a linear hybrid automaton R is a* reverse *of the (forward) linear hybrid automaton L if*

- $Loc(L) = Loc(R)$.
- $Var(L) = Var(R)$.
- $Inv(L) = Inv(R)$.
- *For each edge (l, l') in $Tra(L)$ there is an edge (l', l) in $Tra(R)$ such that for each $x \in Var(L)$:*
  - *If the assignment of x is the 'no-assignment' id_x, then $Guard(R; l', l; x) = Guard(L; l, l'; x) \cap Inv(L; l; x)$ and $Asg(R; l', l; x) = id_x$.*
  - *If the assignment of x is the set $\{v \mid \varphi\}$ different from id_x, then we have $Guard(R; l', l; x) = \{v \mid \varphi\}$ and $Asg(R; l', l; x) = Guard(L; l, l'; x) \cap Inv(L; l; x)$.*
- *Assume $Act(L; l; x)$ is given by $\{v \mid \varphi\}$. We define the corresponding activity of x in l in R as $Act(R; l; x) = \{c \mid -c \in \{v \mid \varphi\}\}$.*

The behavior of a hybrid automaton is given by both its finite and infinite runs and a reverse linear hybrid automaton should contain the runs of the corresponding forward linear hybrid automaton, only in the *opposite* direction time-wise. This is straight-forward definable for finite runs, but more complicated in the case of infinite runs:

Definition 6 (Reverse run). *Given a finite run ξ of length n of a hybrid automaton H, and a finite sequence ξ_r of alternating discrete transitions and time delays of length n, we say that ξ_r is a* reverse *of ξ if for all $i < n$ where $n > 1$:*

- *For the discrete transition $(l, v) \rightarrow (l', v')$ of ξ at step i, there is a discrete transition $(l', v') \rightarrow (l, v)$ of ξ_r at step $n - i$.*
- *For the time delay $(l, v) \rightarrow_f^t (l, f(t))$ of ξ at step i there is a time delay $(l, f(t)) \rightarrow_g^t (l, v)$ of ξ_r at step $n - i$, for some function g.*

If either of ξ or ξ_r are infinite we say that ξ_r is a reverse of ξ if there is a prefix of ξ_r that is the reverse of a prefix of ξ. If $n = 1$ then ξ is a reverse of ξ_r iff $\xi = \xi_r$.

The following result connects reverse linear hybrid automata and reverse runs.

Lemma 1. *Let R, a linear hybrid automaton, be a reverse of a linear hybrid automaton L. Then for each $\xi' \in [L]$ and each finite prefix ξ of ξ' there is a run $\xi_r \in [R]$ that is the reverse of ξ.*

Proof. We prove the lemma by induction on the transitions in ξ.

Base Case: If (l_0, v_0) is the initial state of ξ, then $v_0 \in Inv(L; l_0)$. By definition, $v_0 \in Inv(R; l_0)$. Hence, (l_0, v_0) is its own reverse run.

Induction Step: Let $\xi(n-1)$ be a prefix of ξ of $n-1$ transitions. The induction hypothesis states that R has a reverse run of $\xi(n-1)$, call it $\xi_r(n-1)$. We want to show that the run of length n also has a reverse in $[R]$, and we consider two possible cases:

- ξ ends with a discrete transition: $\cdots (l, v) \to (l', v')$. Definition 5 provides us with the existence of the transition (l', l) of R. If $v = v'$ we know that v' satisfies the guard of (l', l) since $v \in Inv(L; l)$ and $v \in Guard(L; l, l')$. If $v \neq v'$ the assignment of (l, l') is some set $\{v \mid \varphi\}$ different from id. We thus have $v' \in \{v \mid \varphi\}$, which is precisely $Guard(R; l', l)$. The assignment $Asg(R; l', l)$ is $Guard(L; l, l') \cap Inv(L; l, l')$ and we know that $v \in Guard(L; l, l')$ and $v \in Inv(L; l, l')$.
- ξ ends with a time delay: $\cdots \to_f^t (l, f(v))$. We need to show the existence of a function $g \in Act(R; l)$ such that $g(f(v)) = v$. In a linear hybrid automaton the activities are all given by linear sets of *constant* differential equations, hence $f \in Act(L; l)$ is on the form $f(t, v) = c_f t + v$. Definition 5 provides the existence of the function $-c_f t + v \in Act(R; l)$. Composing the functions we get $-c_f t + (c_f t + v) = v$, hence $-c_f t + v$ is the desired function g, i.e. the solution to the differential equation $\dot{v} = -c_f$. □

We now turn to modeling embedded systems where the emergency control takes control over the system in those cases where failure to do so might result in runs with potentially catastrophic future results. The emergency control will be application-dependent, we only assume that we are provided with a linear hybrid automaton model. To be conservative and not miss any potential future evolution we propose to model the failing normal operation under the assumption that any physically allowed behavior is possible.

Definition 7 (Free behavior). *Assume a system with state variables* $\mathbf{x}$ *and free time* t_{free}*. Assume also that the variable* c *is a clock, and that the rate of evolution of each* x_i *is bounded by some interval* $[\epsilon_{x_i}^-, \epsilon_{x_i}^+]$*. Then, in the context of hybrid automata, we define the* free *(linear) behavior of the system to be a location* l_{free} *with invariant* $c \leq t_{free}$ *and activities*

- $\dot{c} = 1$
- $\dot{x_i} \in [\epsilon_{x_i}^-, \epsilon_{x_i}^+]$*, for all* $x_i \in \mathbf{x}$

The product $\prod_{x \in \mathbf{x}} [\epsilon_x^-, \epsilon_x^+]$ forms a *hyper-rectangle*, or box, in the dimension of $\mathbf{x}$. In an emergency control hybrid automaton model the clock variable c will be

redundant: We assume that c holds the value t_{free}, and that $c = t_{free}$ is part of the invariant for all other locations. Using the previous definition we propose that the failing embedded system augmented with emergency control can be modeled using a specialized class of hybrid automaton which models systems that i) start in a free behavior location l_{free} ii) leave l_{free} after time t_{free} iii) never return to l_{free}.

Definition 8. *A* forward *linear emergency control automaton EC is a linear hybrid automaton L with the following augmentations:*

- $Loc(EC) = Loc(L) \cup \{l_{free}\}$, *where the activities and invariant of* l_{free} *are as given in Definition 7.*
- $Var(EC) = Var(L) \cup \{c\}$.
- *For any transition we require the identity assignment for* c *with* $c = t_{free}$ *as part of the guard.*
- *For each* $l \in Loc(EC)$, *with* $l \neq l_{free}$ *there is a transition* (l_{free}, l) *in* $Tra(EC)$ *with* $c = t_{free}$ *as the entire guard.*
- *There are no transitions from* l *to* l_{free} *for any location* l.
- *The initial states Init are all in location* l_{free}, *with* $c = 0$.

Since we are interested in backwards reachability analysis of hybrid automata we will in the following let the term linear emergency control automaton refer to the *reverse* of the automata given by Definition 8. In Figure 2 we give an graphical overview of the characteristics of a general (reverse) linear emergency control automaton, namely that the transition from the reverse linear emergency control to the l_{free} location can be taken at any time but never left, and that the run stops when the clock c, initially t_{free}, reaches 0.

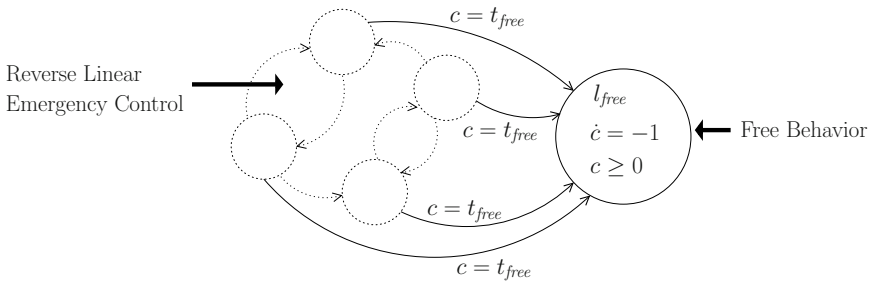


Fig. 2. The characteristics of a linear emergency control automaton

The *reach set* generated from a linear emergency control automaton by a reachability algorithm contains an over-approximation of all the states of the embedded system that may lead to an undesired state unless the mode switch to emergency control is initiated by the safety check within a single period. The safety check considers states of a single location only, namely the l_{free} location, since we assume the system exhibits free behavior when the safety check is run.

Definition 9 (Location-independence). *Reachability in a hybrid automaton H is* location-independent *if there is a location l of H with the property that given any state l' of H and any run $\xi' \in [H]$ such that $(l', v) \in \xi'$ there is a corresponding run $\xi \in [H]$ with the same initial state as ξ', and where $(l, v) \in \xi$.*

Lemma 2. *Reachability is location-independent in linear emergency control automata.*

Proof. We will show that for any run ξ' with state (l, v) where $l \neq l_{free}$ there is a run ξ with state (l_{free}, v). Let ξ'_p be a prefix of ξ' ending in (l, v). We can always apply the transition (l, l_{free}), from Definition 8, which has the guard $c = t_{free}$, see Definition 8, since we know $c = t_{free}$ by Definition 7. Since (l, l_{free}) does not have an assignment, see Definition 8, the run $\xi'_p \rightarrow (l_{free}, v)$ is the desired run ξ. □

Algorithm 1. Safety check

```
procedure IS_SAFE(Set_bounded, Pos)
    while Set_bounded not empty do
        B ← Set_bounded.remove()
        contains := TRUE
        for i in dimension(B) do
            if Pos_i ∉ B_i then
                contains := FALSE
            end if
        end for
        if contains then
            return FALSE
        end if
    end while
    return TRUE
end procedure
```

The location-free reach set of a reachability search in a hybrid automaton contains polytopes of dimension n, and an approximation of the reach set can be produced by replacing each polytope P with a box B of dimension n where $P \subseteq B$. Since a box is a product $\prod_{i=1}^{n} B_i$ of intervals B_i on the real line, checking whether a point Pos is contained in a box B is bounded by the time required to execute n tests of the form $Pos_i \in B_i$. As an optimization we propose only to consider those d dimensions that are of interest for the safety check.

Definition 10 (Safety check). *The* safety check *of an embedded system augmented with emergency control derived from a linear emergency control automaton, is given in Algorithm 1. We define the* reach set bound h_{bound} *to be the largest number of boxes that can be tested within an application-specific time* $t_{deadline} < t_{period}$.

The for-loop of Algorithm 1 is executed d times for each box, so the algorithm executes a total of $d \cdot h_{bound}$ if-tests. If the cardinality of the hyper-rectangular approximation Set_{rect} of the reach set is larger than h_{bound}, then the safety check may miss its hard real-time deadline and we must over-approximate Set_{rect} by a set $Set_{bounded}$ with at most h_{bound} boxes. Operations on the *R-tree* data structure can be used to merge boxes [8]. In Algorithm 2 we give a method for merging the reachable sets of a linear emergency control automaton into h_{bound} boxes.

Algorithm 2. Merging boxes

```
procedure MERGE(Set_rect, h_bound)          ▷ We assume |Set_rect| > h_bound
    Q ← [BoundBox(Set_rect)]                ▷ Initialize queue
    while |Q| < h_bound do
        B ← Q.get()
        if |B| = 1 then
            Q ← B and continue
        end if
        B_1, B_2 ← RTree.split(B)           ▷ Size of B_1 and B_2 minimized [8]
        Q ← B_1, B_2
    end while
    return Q
end procedure
```

Lemma 3. *Algorithm 2 is sound and terminating.*

Soundness means, in this case, that the algorithm produces an over-approximation of its input.

Proof. The algorithm is sound if for all boxes H in Set_{rect}, there is a box B in Q such that $H \subseteq B$. We prove this by induction on the number of iterations of the while loop:

Base Case: All H in Set_{rect} are covered by the bounding box $BoundBox(Set_{rect})$.

Induction Step: We assume that for any H in Set_{rect} there is a bounding box in Q such that H is covered by this bounding box. Let B be the bounding box removed from the queue in the while loop of the algorithm, and let $H_1 \ldots H_n$ be the boxes covered by this bounding box. The correctness of the split algorithm chosen [8], ensures that either $H_i \in B_1$ or $H_i \in B_2$ for all $1 \leq i \leq n$.

The algorithm terminates if $|Q| \geq h_{bound}$. The while loop of the algorithm either keeps the queue of constant size if the bounding box contains a single box, or increases the size by one if the bounding box contains more than one box. Since we assume $|Set_{rect}| > h_{bound}$ the termination condition follows, since Q can otherwise be reduced to a queue of single boxes of length greater than h_{bound}. □

4 The HyRev Tool

To test the feasibility of the approach developed in Section 3 we have implemented a proof-of-concept prototype tool chain called HyRev[1]. HyRev is written in Python, and incorporates the SpaceEx reachability solver [6]. In order to focus on the concepts we have presented, the tool only supports emergency control automata with a single location. However, to accommodate a non-constant behavior specification of this location we have implemented a hybridizing tool for over-approximating affine hybrid automata by linear hybrid automata [2]. Much of the functionality of the system is provided through a software library, but to facilitate new or improved techniques the tool chain itself is modular and split into a set of independent executables, including SpaceEx, where each executable accepts the file output of the previous step. The tool chain should be used as follows:

1. *System Design.*
   The tool user must model the emergency control system as a single location affine hybrid automaton, Definition 2. This means providing the (hyper-rectangular) *invariants*, *activities* and *initial values* of the system *variables*, in addition to the desired *precision* of the hybridization procedure.
2. *Hybridization.*
   A *linear hybrid automaton* over-approximation, Definitions 2 and 4, of the affine hybrid automaton is generated using the hybridization approach [2].
3. *Reversal.*
   The linear hybrid automaton is *reversed* with the undesirable system states as the initial states.
4. *Free Behavior.*
   The linear hybrid automaton is augmented by user-specified reverse initial *free behavior*, Definition 7, to generate a (reverse) *linear emergency control automaton*, Definition 5.
5. *Reach Set Computation.*
   The automaton from the previous step is saved in the XML format accepted by the reachability solver SpaceEx [6]. HyRev configures the solver to only produce reach set output of a list of specified *safety variables*. SpaceEx terminates when all possible reachable states have been generated. The polytope output from SpaceEx is approximated by a set of bounding boxes.
6. *Merging.*
   The reach set of boxes is over-approximated by a set of boxes with an upper, user-specified *bound*, Algorithm 2. The output of the merging step is pseudo-code representing the actual code that will run on the concrete embedded system in question, see Algorithm 1.

An example of a configuration file and the pseudo-code ultimately generated by HyRev is presented in Figure 3. The output produced by HyRev is an over-approximation of the state space of the original affine hybrid automaton containing all potential future runs that lead to bad states if the switch to emergency

[1] `heim.ifi.uio.no/hallstah/hyrev/`

control is not initiated immediately. As seen in Figure 3 the pseudo-code which corresponds to Algorithm 1 consists of two nested and bounded for-loops with a single if-test inside. This makes it easy to compute the worst-case execution time of the safety check. While a conservative result is absolutely required to prove safety of the model we also wish to be as precise as possible. If not, then the number of safe states being interpreted as unsafe may limit the usefulness of the technique. The precision is influenced by the configuration parameters of SpaceEx, but also by two parameters provided for HyRev: The bound on the reach set and the granularity of the hybridization partition. Any increase of the reach set bound is dependent on the concrete embedded system in question and not influenced by the configuration of the formal model. The hybridization granularity, however, can be increased at the cost of increased run times of HyRev and SpaceEx.

```
[[name]]
rise

[[vars]]
h 0.0 4.0
v -12.0 12.0

[[init]]
h 0.0 0.0
v -12.0 0.0

[[flow]]
h' = v
v' = 20.0

[[cuts]]
h 1
v 16

[[free behavior]]
h' -12 12
v' -10 20

[[free time]]
0.02

[[safety test]]
5

[[safety vars]]
h
v
```

```
boolean potentially_dangerous(box[] state_space, point state)

    for box in state_space
        contains = TRUE
        for i in dimension
            if pos[i] not in box[i]
                contains = FALSE
        if contains
            return TRUE
    return FALSE




switch(pos)

    space = [
        box( interval(0.0,0.24), interval(-12.0,0.4)),
        box( interval(0.0,4.0), interval(-12.0,-8.6)),
        box( interval(0.0,2.6025), interval(-9.2,-5.6)),
        box( interval(0.0,1.365), interval(-6.2,-2.6)),
        box( interval(0.0,0.5775), interval(-3.2,0.4))
    ]



    if potentially_dangerous(space,pos)
        switch()
    else
        do_nothing()
```

Fig. 3. HyRev configuration and output

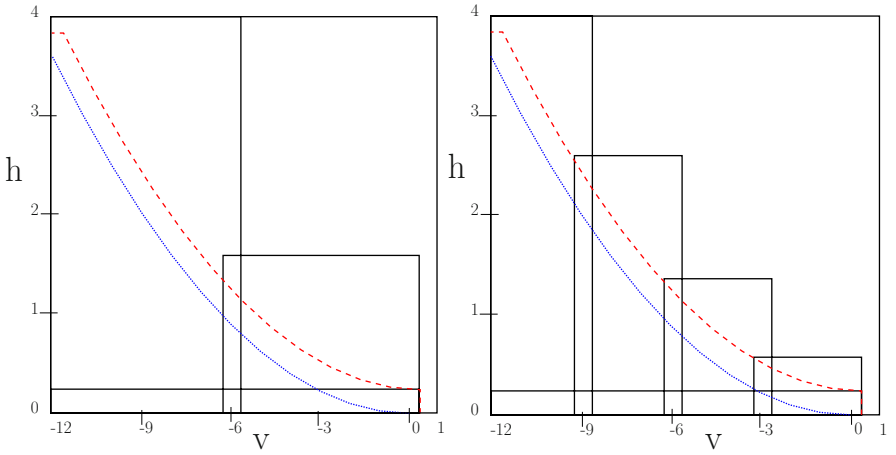
As a demonstration of the capabilities of the HyRev tool we use it on a simplified model of a UAV based on the one presented in Example 1. In Figure 3 we show the configuration file for the case study. The dynamics of the horizontal positon h are affine, $\dot{h} = v$, and so the *cuts* entry specifies that the state space of the v variable will be split into 16 sub-locations as part of the hybridization into linear dynamics. The *free time* and *free behavior* entries specify the duration, t_{free}, and the dynamics of the free behavior. Note that due to the physical inertia in the UAV system we assume a duration of $0.02s$, 200 times longer than a system period of $100\mu s$. The entry *safety vars* lists the variables relevant for the safety check, namely both the positon and velocity of the system. The *safety test* entry supplies the application-specific number of boxes which ensures that the safety check is time-bounded and the hard deadline respected. The output

produced by HyRev is also shown in Figure 3. The tool generates pseudocode for both the safety check in the function *potentially_dangerous()*, as well as the hyper-rectangular, bounded reach set passed to the function. The example configuration partitions and hybridizes the range of the v variable 16 times, and produces safety check sets of cardinality 5. These boxes are shown in Figure 4-(b), where the dotted and dashed curves represent the optimal limits of the reach set without and with the free behavior, respectively. The parts of the safety check sets lying above the dashed curve represents the over-approximation error. This error is contributed to by one or all of the following: The hybridization procedure, the reachability search, the approximation of the reach set by boxes, and the merging of the rectangular reach set into the bounded safety check sets. In Figure 4-(a), -(c), and -(d) are shown other safety check sets, produced by different configuration parameters. Note that only the subset of the state space containing the over-approximation is shown in the figure. In Figure 4-(e) we see the timing results for different levels of hybridization of the v variable, explicitly listing the steps requiring the most run time: SpaceEx reachability search and reach set merging. It is clear that to achieve scalability for an industrial-quality tool, the time complexity of potential merging algorithms must be further evaluated. However, the reversal step takes a negligible amount of time to run.

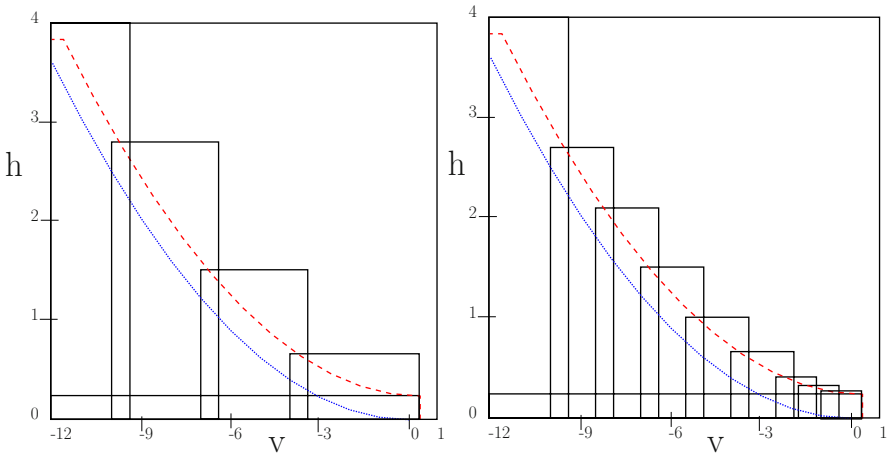
5 Discussion

In this paper we have presented both a theoretical framework for and a prototype implementation of a general technique for generating a time-bounded safety check with a well defined worst-case execution time for safety-switching embedded systems. By modeling the behavior of the emergency control system designed to prevent the failure of the system, and modeling any physically possible system behavior during failure, we apply reverse reachability computation to generate a conservative estimate of the states that could potentially lead to a catastrophe without modeling the system itself nor any failures it may exhibit. The implementation was applied to a case study of an unmanned aerial vehicle. We are aware of no previous work on the particular problem of generating safety checks. The results presented here are complementary to existing techniques for safety controller synthesis [5, 7]. Controller synthesis influences the normal operation of a system, preventing failures from occurring. A safety check only acts when the normal operation may be failing, either through design, software, or hardware faults.

The work represents an initial exploration of the subject, providing a large number of possible directions for future research. A possibility is a more thorough analysis of the performance of the implemented algorithms, both theoretical and practical, through more complex case studies. An extension of the hybridization module to support non-linear dynamics would be an important contribution to this. To reduce the size of the reach set we can investigate the use of more general polytopes in the safety check, a use which must be compared to the increase in the execution time per polytope. From a theoretical point of view it would be



(a) v is partitioned 8 times, bound 3 (b) v is partitioned 16 times, bound 5



(c) v is partitioned 32 times, bound 5 (d) v is partitioned 32 times, bound 10

Hybridization of v	SpaceEx run time	Merging run time	Total run time
32	1.0	0.9	2.2
40	1.6	3.3	5.3
48	2.9	9.9	13.4
56	5.6	24.3	31.0
64	6.5	42.8	50.8

(e) Results of test runs. All timings in seconds

Fig. 4. The bounded reach set and execution times for the UAV linear emergency control automaton

interesting to consider the reversal of more general classes of hybrid automata, i.e. affine or even non-linear automata. The prototype presented in this paper would be an important design artifact in the development of an industry-strength application. Such an application would have to include an arbitrary number of locations, including the support of assignments on transitions.

Acknowledgments. The author wishes to thank Martin Steffen and Goran Frehse for valuable feedback on preliminary versions of this paper.

References

[1] Alur, R., Courcoubetis, C., Halbwachs, N., Henzinger, T.A., Ho, P.-H., Nicollin, X., Olivero, A., Sifakis, J., Yovine, S.: The algorithmic analysis of hybrid systems. Theoretical Computer Science 138, 3–34 (1995)
[2] Asarin, E., Dang, T., Girard, A.: Hybridization methods for the analysis of nonlinear systems. Acta Informatica 43, 451–476 (2007)
[3] Asarin, E., Bournez, O., Dang, T., Maler, O.: Approximate reachability analysis of piecewise-linear dynamical systems. In: Lynch, N.A., Krogh, B.H. (eds.) HSCC 2000. LNCS, vol. 1790, pp. 20–31. Springer, Heidelberg (2000)
[4] Baier, C., Katoen, J.-P.: Principles of Model Checking. The MIT Press (2008)
[5] Camara, J., Girard, A., Gossler, G.: Safety controller synthesis for switched systems using multi-scale symbolic models. In: IEEE Conference on Decision and Control and European Control Conference (CDC-ECC), pp. 520–525 (2011)
[6] Frehse, G., et al.: SpaceEx: Scalable verification of hybrid systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 379–395. Springer, Heidelberg (2011)
[7] Girard, A.: Low-complexity quantized switching controllers using approximate bisimulation. In: Nonlinear Analysis: Hybrid Systems (2013)
[8] Guttman, A.: R-trees: A dynamic index structure for spatial searching. In: International Conference on Management of Data, pp. 47–57. ACM (1984)
[9] Henzinger, T., Ho, P.-H., Wong-Toi, H.: Algorithmic analysis of nonlinear hybrid systems. IEEE Transactions on Automatic Control 43(4), 540–554 (1998)
[10] Henzinger, T.A., Ho, P.-H., Wong-Toi, H.: HyTech: A model checker for hybrid systems. Software Tools for Technology Transfer 1, 110–122 (1997)
[11] Kopetz, H.: Real-Time Systems: Design Principles for Distributed Embedded Applications. In: Realtime Systems. Springer (2011)
[12] Ratschan, S., She, Z.: Safety Verification of Hybrid Systems by Constraint Propagation Based Abstraction Refinement. ACM Transactions in Embedded Computing Systems 6(1), 573–589 (2007)
[13] Skogestad, S., Postlethwaite, I.: Multivariable Feedback Control: Analysis and Design. John Wiley & Sons (1996)

An Outline Workflow for Practical Formal Verification from Software Requirements to Object Code

Darren Sexton

Ricardo UK Ltd., Building 400, Science Park,
Cambridge, CB4 0WH, United Kingdom
darren.sexton@ricardo.com

Abstract. This paper considers current state-of-the-art verification techniques that are based upon, or supported by, formal methods principles to ensure a high degree of assurance. It considers the practical application of such approaches in an industrial context so as to achieve an efficient, coherent and integrated workflow.

The key focus is a clear process that starts from software requirements and works through to the final object code on the target, ensuring key verification aims are fulfilled with a high-degree of confidence at each step. The process combines both analysis and testing to maximise the strengths and to cover the weaknesses of each.

For each step, a high-level description of the approach, potential benefits, prerequisites and limitations is given. The workflow outlined considers tools, methods and the supporting processes.

Keywords: Formal workflow, model checking, ISO 26262, formal requirements.

1 Introduction

As vehicle electronic architectures become more complex and software-intensive systems have increasing authority, then so the need for high assurance of the integrity of the software increases. Functional safety standards such as ISO 26262 prescribe detailed process steps across the engineering life-cycle, including for software. Yet at the same time, the traditional automotive industry pressures of cost and time-to-market remain, making it difficult to achieve the necessary assurance.

Methods and tools that can provide the high-levels of assurance desired, yet deliver it efficiently, are therefore of clear interest to system developers and their end-customers. Formal methods have been claimed as one such silver bullet that can achieve goals such as eliminating effort intensive unit testing. However, although optimistic claims have been made over many years about the benefits of formal methods, many of these techniques have failed to achieve widespread usage across the entire workflow. Many such techniques have also proven not to be scalable to industrial-sized systems or work only with state machine and

C. Pecheur and M. Dierkes (Eds.): FMICS 2013, LNCS 8187, pp. 32–47, 2013.

simple logic based systems. The area where greater market penetration has been achieved has been where the formalisms are hidden from the end user engineers, such as advanced analysis of source code for run-time errors.

Improvements in methods and tools that hide the formalisms from end users and methods that are more scalable are finally starting to make practical usage of these approaches more realistic across a range of activities. The Model Based Analysis and Testing (MBAT) research project brings together almost 40 European companies, tool-vendors and research institutes to evaluate and extend state-of-the-art analysis and testing techniques. In particular, the project aims to deliver near-term deployment of combinations of analysis and test that work effectively together.

Ricardo is an engineering consultancy in which the Controls and Electronics part of the business works with clients to deliver many high-integrity electrical and software systems in a number of industries. Within the MBAT project Ricardo is acting as an industrial partner, focussing on evaluating approaches and developing a process that suits the needs of our organisation. This paper explores a realistic verification workflow from software requirements through to the object code. This combines techniques that are relatively established (such as abstract interpretation) with techniques that are only recently becoming more accessible. It considers also how these activities fit together.

The next section gives an overview of the workflow and is then followed in section 3 by a look at the key steps within each phase. Section 4 discusses how this workflow fits with standards such as ISO 26262, whilst section 5 takes a critical look at the limitations uncovered. Finally, future improvements are discussed, leading onto the overall conclusions.

2 Overall Ricardo Workflow within MBAT

The overall workflow which is being developed and evaluated by Ricardo within the MBAT project is summarised in figure 1. The normal development flow is shown down the centre, with a progression from natural language software requirements to executable designs (in Simulink / Stateflow) to automatically generated code and finally the compiled object code. Activities with dashed lines represent the verification activities. These are either to check the work-products against general quality criteria (e.g. robustness) or that they are a faithful translation of a previous phase.

This workflow is shown in the context of model-based development with automatic code generation. However, aspects of it are also applicable to traditional hand-coded developments without executable designs.

Such a workflow does not exist in a vacuum. At Ricardo the workflow is integrated with our usual practices, such as peer review, tool supported enforcement of modelling guidelines, training etc. It is currently scoped at the level of assurance of software features or units, although its applicability to integration and system testing may be considered in the future.

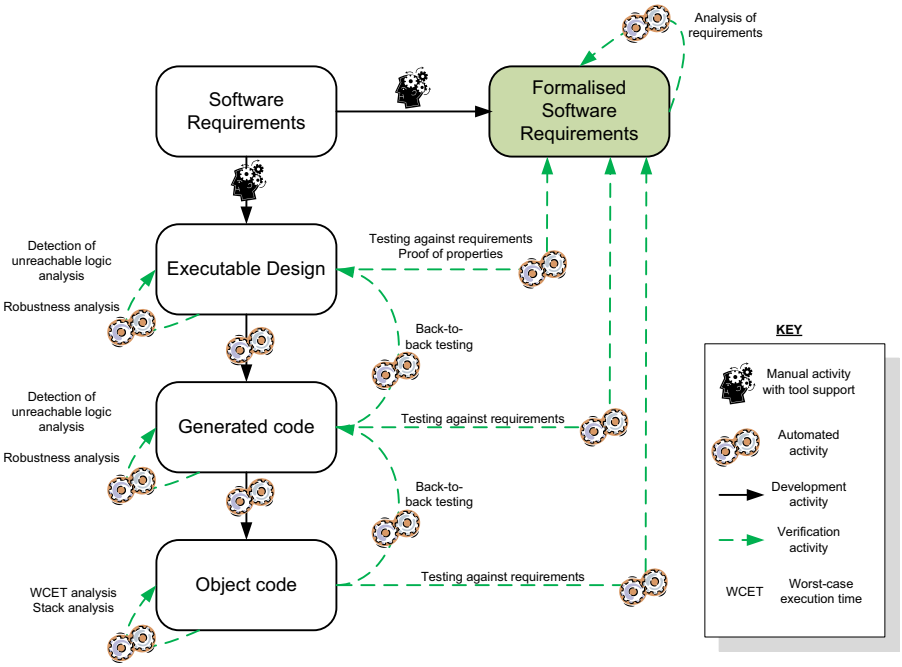


Fig. 1. Overview of Ricardo workflow developed and evaluated within MBAT

The MBAT project is particularly focussed on combining analysis and testing approaches [6]. Within our workflow, analysis often supports testing by (i) detecting defects earlier; (ii) gives a higher degree of confidence than testing alone is able to; (iii) supports generation of test stimuli and oracles; or (iv) potentially reduces the scope of the testing. Conversely, testing supports the analysis in areas where the analysis is unable to cope with scale or underlying complexity, whilst also giving confidence about the final code running on the target. Different levels of coupling of tools and methods are possible, from a loose coupling of work-flow integration through to deeply coupled technical tool integration [2]. With a loose work-flow coupling, the tools might simply share the means of presenting results to the user, whereas for a deeper coupling the tools might make use of feedback loops such as the analysis results being used to refine test vector generation. An example of a workflow integration within our process is the re-use of information from the EmbeddedTester tool to generate harness information for the stack and WCET analysis. A deeper coupling is shown in the derivation of analysis and test objectives from the same formalised requirement source. The MBAT Reference Technology Platform (RTP) aims to act as a bridge enabling tools and methods to be coupled.

The work-flow is being evaluated in the context of case studies. The primary case study is that of the application software for an automatic transmission controller. However, the majority of the experiences described in this paper so far

relate to a smaller case study related to the control of a basic hydraulic system[1]. We have started with the smaller case-study to provide a simpler learning environment as we gain experience in the work-flow.

The next section looks at these verification activities, and discusses the potential benefits, approach adopted, prerequisites and limitations.

3 Key Activities

3.1 Requirements Phase

A key foundation of this workflow is the formalisation of the natural language software requirements. In this activity, an engineer takes the natural language requirements and, with tool support using BTC-EmbeddedSpecifer [21], manually converts them into a notation with formally defined syntax and semantics. In this workflow the underlying formalism uses a pattern based approach, such as P implies globally Q. The semantics of the patterns are defined by Büchi-Automaton charts (an example[2] is shown in figure 2); these provide an expressive power equivalent to Linear Temporal Logic (LTL). Although an understanding of the Bčhi-Automaton notation can be helpful when formalising the requirements or reviewing the formalisation, our the intent is that engineers can generally select the correct pattern through the pattern name and description alone.

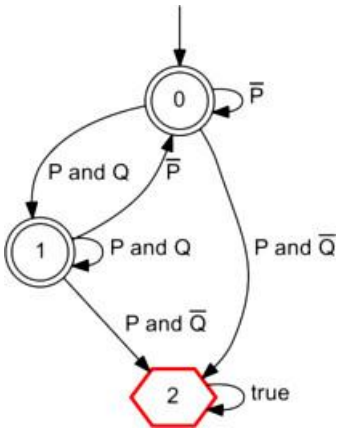


Fig. 2. Büchi-Automaton chart for the pattern "cyclic_Q_while_P_immediate"

This formalism has the advantage of being relatively accessible to engineers whose expertise is in their particular engineering domain rather than formal notations. With the tool support, the natural language requirements are mapped

[1] For confidentiality reasons the application cannot be described.
[2] Diagram provided with kind permission from BTC-EmbeddedSystems.

one-to-one to relatively user-friendly patterns. The use of patterns helps reduce one of the key problems of formal methods notations that are less accessible to engineers. Whilst it is possible to train people in notations such as Z, the lack of wide-spread adoption suggests that this is a significant barrier to the common use of formal requirements. Throughout, the original natural language requirements are maintained with traceability between them and the formalised requirements. The natural language requirements are maintained, since this is the format most suitable for the domain experts to converse in, whilst a one-to-one mapping simplifies the traceability essential to manage the inevitable changes to requirements that will occur.

The potential benefits of formalising requirements are primarily realised later in the life-cycle, where they act as the foundation for further verification and validation tasks, shown in figure 3. However, even at this stage the act of formalisation brings some benefits, such as helping identify ambiguous or incomplete requirements. Additionally, a formal semantics potentially allows some level of analysis to be performed on the requirements themselves [7], although this is an immature part of the process currently and has not yet been evaluated by Ricardo. Such analysis may allow certain types of requirements errors to be detected before software has been implemented, potentially reducing the time and cost to fix them.

Unsurprisingly, experience shows that clearly and consistently written natural requirements are a prerequisite to the formalisation process. In fact our experience so far has been that the requirements must be written with the eventual formalisation in mind. For example, on our initial use-case the requirements were originally written without specific regard to the eventual formalisation process (but with regard to our internal guidance on requirements style). It was possible to formalise 85% of these, whilst 100% of the requirements in a slightly re-written set (taking account of our initial experience of what posed difficulties for the formalisation step) could be formalised. We are currently evaluating use of simple, flexible boilerplates to achieve a consistent style and ease the mapping of the requirements to the formal patterns. However, effective requirements writing remains a challenging area.

The major potential limitations of the formal requirements discussed here are the effort required to formalise and limits on what types of requirements can be formalised. In particular, since the expressiveness is equivalent to LTL notation, certain types of requirements cannot be expressed at all. In particular, requirements related to algorithmic aspects of the functionality (such as control feedback loops) may prove infeasible to formalise. At this stage Ricardo is piloting the workflow and collecting metrics to help us measure these issues.

Additionally, there is a danger that a requirement is incorrectly formalised, thus all down-stream verification activities derived from it test or analyse the wrong thing. In such cases, the analysis and tests may pass nonetheless. For example, with the initial set of requirements previously mentioned, the initial formalisations were (subtly) incorrect for every requirement! This of course partly reflects our lack of experience in a new technique. However, it is worth noting

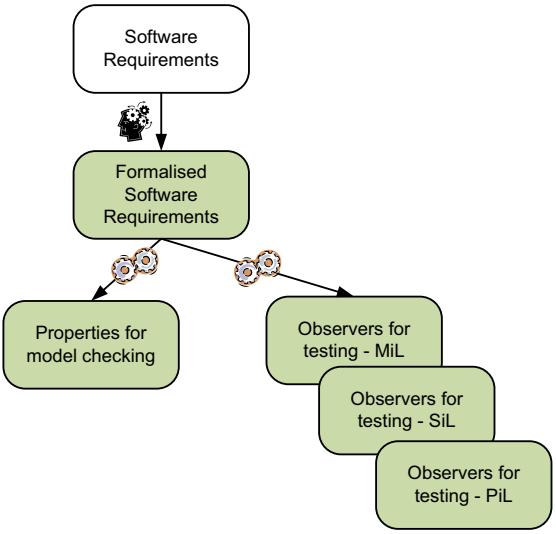


Fig. 3. Use of formalised requirements as basis for later verification

that the alternate set of requirements (written with formalisation more in mind) required only 20% of the formalisations to be re-worked. This difference appears to be down to two factors; firstly the requirements style was more suited to formalisation and, secondly, we developed aids to help guide the requirements engineer to an appropriate pattern. Since this is such a significant issue, we are looking at what process steps can be implemented to (a) prevent incorrect formalisations in the first instance and (b) detect incorrect formalisations.

With respect to prevention, the use of boilerplate text has already been mentioned. The use of aids to guide selection of the appropriate pattern was also mentioned. Based on our early experience, we have developed a set of flow-charts; these do not guide the engineer to the exact pattern, but by asking a number of questions they narrow down the relevant groupings of patterns and provide hints as to what to look for in the pattern name. This is aided by a consistent naming format of the patterns. Anecdotally, these appear to have been very effective; initial results show that using these (in combination with the boilerplate requirements) meant that the initial formalisation effort was reduced by approximately a third. This under-states the reduction in effort since the aids were almost definitely a factor in the reduction of incorrectly formalised requirements. From our initial data it is not possible to separate the influence of the use of the aids from the improved requirements style, but comments from the engineers involved indicate that both provided significant benefits.

3.2 Design

Model Checking. The formal requirements act as the starting point for a number of further verification activities. The first one is attempting to prove,

by model checking formal techniques, that the requirements are met by the implementation. Formal properties to prove this are generated automatically from the formal requirements; the analysis is then run using the tool BTC-EmbeddedValidator, and, if the model if compatible, it will determine that the requirement is always met or cannot be met (and a witness trace for debugging is provided) or is un-decidable.

The potential benefit of this analysis is that it gives a very early indication, during development and prior to any testing, as to whether the implementation meets its requirements. Furthermore, results are complete since the analysis is exhaustive [20]. This is done with minimal human involvement and can be easily re-run following changes.

However, despite the attraction of this step, there are significant limitations [1]. Firstly, there are issues of scalability – in general it is appropriate only with the prerequisite of well-structured, modularised designs. More significantly, the underlying functionality of the model may mean model checking is not feasible. For example the technique works well for state machines and logic-based systems, but poorly once mathematical operations and feedback loops are introduced (features typical of the power-train & complex control applications Ricardo typically develops).

Testing against Requirements. The formal requirements act also as a starting point for testing the executable design against the requirements. In this step, observers are generated automatically from the formal requirements (in the same manner as the model checking) using BTC-EmbeddedSpecifier. These observers will look at signals during simulation and see if they respect or violate the requirement – thus they act as the test oracle [17]. With the observers in place, it is now also possible to automatically generate the test stimuli. Within this workflow, the test stimuli are derived automatically from the design model to achieve very high structural coverage (e.g. MC/DC, boundary values etc.). Furthermore, the test generation can be driven to achieve coverage of the observers themselves. This drives test stimuli to specifically exercise the requirements. Additionally, coverage of the requirements can be measured through coverage of the observers [3]. The steps described are performed within the tool BTC-EmbeddedTester. The observers can also be used when stimulating the model with captured data or simulated drive cycles. Such tests of the implementation against the formalised requirements show an approach where formal methods support another approach (testing). The key potential benefit of this approach is a significant reduction in effort – the process of test case generation and test oracles is effectively automated. As with much of this workflow, Ricardo is collecting metrics to help meas- ure this issue.

A second key benefit is confidence that the tests cover the requirements. Since the test oracles are derived automatically from the requirements, the risk of inadequate manual test case generation is removed. This approach is also very scalable, since it is possible to simulate large models or those with control and mathematical functionality (in contrast with model checking techniques). Thus

we can use a formal testing approach to cover limitations in the model checking analysis.

This approach relies on formalising the requirements; an obvious limitation is that manual testing must be performed to cover any requirements that cannot be formalised. Such tests must then be integrated into the same tool-chain so that test management is simplified with a single view of the results available. Integration of such manual tests is supported by the tools underpinning this workflow.

Checking Robustness of Design. Separate from the requirements-based approach, it is possible to use formal methods to assess the robustness of the design. One such analysis that can be performed prior to testing is to detect unreachable logic in the models [18], [19] – such cases may indicate a flaw in the design or may be the result of calibration or defensive modelling / coding. Therefore, it is generally accepted that such cases should be reviewed to see if they are acceptable or not. The formal methods that allow such analysis are entirely hidden to the end user, thus it acts simply as a tool supported means to highlight attention to areas that should be checked. Within the MBAT project we have used the tool BTC-EmbeddedValidator; however, previously Ricardo has put in place infrastructure that allows an analysis harness to be automatically created for immediate use with Simulink Design Verifier by control engineers.

Similarly, robustness issues such as detection of division by zero can be found through formal methods (e.g. abstract interpretation) hidden from the user, although some interpretation of the results is required.

With both these approaches, a prerequisite once again is small, well-structured design models. In common with the model checking described earlier, a major limitation is that many underlying types of functionality cannot be analysed – generating either false alarms, undecidable results or no results at all.

3.3 Code

The steps on the code closely resemble those performed already on the model. The benefits, prerequisites and limitations are largely the same as at the design stage. For projects without executable models, the techniques can be applied here for the first time.

Firstly, robustness checks for run-time errors or unreachable logic can be applied through use of abstract interpretation and other formal methods. For reasons of existing experience we have used the Mathworks tool Polyspace [5]. However, within the MBAT project a loose coupling between BTC-EmbeddedTester and the abstract interpretation tool Astree [4] has been performed by other partners that reduces some of the set-up effort for this analysis [2].

Secondly, observers can once again be derived from the formalised requirements, and the source code tested against them in a Software-in-the-Loop (SiL) environment using the BTC-EmbeddedTester tool. The test stimuli previously generated are used (potentially with additionally generated ones), and so the

effort to test the code against the requirements is significantly reduced. Furthermore, the results can be compared not only against the observers, but also against the results from model simulation – giving back-to-back testing. This increases confidence that auto-code generation is faithfully representing the model (although only on a host-PC environment at this step) [13], [22].

3.4 Object Code

Ultimately we want confidence that the object code deployed on the target meets its requirements. The previous steps in the formal workflow act only as a means to eliminate defects earlier, typically at lower cost. For example, it is not possible to rely solely on formal proofs at the model level as later phases have the potential to introduce defects that can violate those properties.

Once again when testing the object code, observers can be derived automatically from the formalised requirements. As before, the existing test stimuli are being run with these observers acting as the test oracle, again using BTC-EmbeddedTester. Also, back-to-back testing on the target hardware gives confidence in the translation by the cross-compiler. As with the earlier steps, we are testing each feature individually, within a harness. Outside of this MBAT workflow the integration and system testing must still be performed.

Formal analysis techniques can also be used to help determine the Worst Case Execution Time (WCET) and maximum stack usage of tasks. Problems with timing or stack may be relatively rare compared with other sources of defects but can be difficult to track down and can easily be missed during functional testing. The tools Ricardo is currently evaluating (aiT & StackAnalyzer) rely on static analysis of the object code itself, with guidance from the user. The resulting figures are guaranteed to be sound with respect to providing the worst case timing or stack depth. Of course, overly conservative figures would be of little use and so the tools aim to give a minimally conservative figure. This gives higher confidence than stress testing alone [20], where it is not possible to be sure the worst case was tested or not.

4 Relationship with ISO 26262

4.1 Overview of Standard and Workflow

ISO 26262 is an automotive functional safety standard that was published in November 2011. Adoption of this standard is becoming increasingly common across the supply chain. It places process requirements and recommendation for development processes; these may vary by Automotive Safety Integrity Level (ASIL), which range from A (lowest) to D (highest integrity), with QM representing development processes outside the scope of ISO 26262. Therefore a consideration of how this workflow fits in with the standard (specifically part 6 for the software) is of interest to those in the automotive domains Similar consideration for standards in other domains, such as DO-178C for aerospace,

could be performed. It should be noted that ISO 26262 has no concept of "certification"; thus the discussion of compliance within this section does not imply that such a work-flow is somehow "certified" or suitable in all cases. It is not feasible to present a thorough, systematic comparison of the work-flow against the requirements of ISO 26262 within this paper; the following sections therefore discuss some of the most relevant aspects.

The standard consists of normative textual requirements as well requirements in the forms of tables; these tables consist of techniques with different recommendations based on the ASIL. For example, the use of formal requirements might be recommended or highly recommended or no guidance given, depending on the ASIL. The selection of techniques (or alternative techniques not listed) must always be justified (for example within a Software Development Plan or Safety Management Plan). Where equivalent options are available, then the preference is to select one that is highly recommended over one that is recommended. Beyond this however, the practical impact of highly recommended versus recommended is not stated within the standard; the assumption is therefore made that a stronger justification is required for not using a highly recommended than for a recommended one.

The MBAT workflow Ricardo is evaluating complies with many aspects of ISO 26262s software requirements. In addition, standard Ricardo development activities not discussed in this paper would meet many additional requirements (for example the use of modelling guidelines). I.e. the work-flow described is not, by itself, sufficient to meet the requirements on software development in ISO 26262.

Across all the phases described below, the issue of tool qualification is clearly relevant since any defects in the tools themselves could mean that defects in the software under verification are missed. Qualification kits for all of the commercial tools are available, and this would also need to be present for any in-house infrastructure.

4.2 ISO 26262 Guidance on Requirements Phase

For the requirements phase, ISO 26262 recommends the use of formal requirements, but does not highly recommend then at any ASIL. However, semi-formal notations are highly recommended at ASILs C and D [9], with some specific rules related to decompositions of ASIL D related requirements. We assume that a formal notation will meet the demands of a semi-formal notation; additionally the EmbeddedSpecifier notation supports a semi-formal notation as a step along the road to the fully formal notation. Therefore, the use of formal requirements will clearly satisfy such process requirements of ISO 26262 at all ASILs.

4.3 ISO 26262 Guidance on Design Phase

During the design phase, formal verification is recommended (but not highly recommended) at ASILs C and D, although semi-formal verification, control flow and data-flow analyses are (and these can be met through techniques discussed in

this workflow) [8]. Thus formal techniques can help satisfy the recommendations and the highly recommended clauses.

4.4 ISO 26262 Guidance on Implementation Phase

Control and data-flow analysis are recommend on the architecture design and unit designs / implementations whilst static code analysis and semantic code analysis are recommended for units [8]. These can all be met through the use of the tools used for the abstract interpretation; this technique is specifically mentioned as an example of a semantic analysis, whilst checks such as the use of uninitialized variables, for example, help meet control and data-flow requirements.

4.5 ISO 26262 Guidance on Testing Activities

ISO 26262 assumes that dynamic testing will be performed. For example, a section outlines requirements on the unit testing process and defines the objective as "[...] *demonstrate that the software units fulfil the software unit design specifications and do not contain undesired functionality*" [8]. Although this presumption in favour of testing is made, the standard allows users to tailor the requirements provided they can be justified (and subject to some further requirements). Therefore formal analysis could be used to meet some parts of the objectives. Within this MBAT workflow both testing and analysis are used in combination, meeting the testing requirements of ISO 26262, but their effectiveness is strengthened through use of analysis for those areas where testing is typically less effective, such as checking robustness.

Testing is always required to be driven primarily by the requirements and is then supplemented by other methods such as fault-insertion testing [8]. The use of the observers derived from the formal requirements ensures that the testing covers the requirements. This is emphasised even more by directed generation of test cases to cover the observers.

Back-to-back testing is explicitly called for when executable designs are used (i.e. Model-Based Development) [9]. This workflow efficiently supports highly automated back-to-back testing through the re-use of the automatically generated test vectors and tolerance sensitive comparison of results from the different testing environments.

ISO 26262 places ASIL dependent requirements on the structural coverage achieved by the test cases [9]. Care must be taken here to understand the real objectives of measuring and analysing test coverage, rather than just blindly "ticking a box". Within this workflow, analysis is the primary means of detecting unreachable logic (rather than checking coverage gaps). Ensuring that the test cases cover the requirements is assured through coverage of the observers. This leaves structural coverage mainly as a means to evaluate the thoroughness of the generated test cases to ensure boundary conditions are tested, and to help detect functionality that should not be present.

4.6 ISO 26262 Guidance on Formal Verification

The observer-based approach tests the implementation against test objectives derived from a formally defined input - the formal requirements. It is worth considering whether such testing therefore constitutes formal verification in the context of the ISO 26262 definition. The definition of ISO 26262 defines formal verification as a "*method used to prove the correctness of a system [...] against the specification in formal notation [...] of its required behaviour*" [10]. The observer testing method clearly checks against a "specification in formal notation", but the question remains as to what does ISO 26262 mean by the verb to prove? Since ISO 26262 does not itself define the term to prove, it is not possible to be sure of the authors intent. A dictionary definition gives "*demonstrate the truth or existence of (something) by evidence or argument*" [11] and "*to establish or demonstrate the truth or validity of; verify, esp by using an established sequence of procedures or statements*" [12]. In the context of these definitions, the observer-based testing is generating *evidence*, the method provides an *argument* and is an established *procedure*. The view taken in [3] is that the observer approach is a formal verification.

To some, however, formal tends to imply a purely mathematical argument. The formalised requirements are indeed formal, but the testing itself is not. For example, the aerospace standard DO-178C defines formal methods as "*Descriptive notations and analytical methods used to construct, develop, and reason about mathematical models of system behavior. A formal method is a formal analysis carried out on a formal model.*" [14]. The formal methods based supplement for this standard goes on to describe formal analysis, including the statement "*Proof, or guarantee, implies that all execution cases are taken into account, achieving exhaustive verification*" [15]. Since testing can never realistically be exhaustive, it would appear that this approach is not a formal method within the definition of DO-178C. Whereas, using the observers for *model checking* is unarguably an example of a formal method.

Similarly, the functional safety standard IEC 61508 (of which ISO 26262 is technically a derivative) uses the term formal methods to define notations used to describe a system. It then uses the term "formal proof (verification)" to describe the verification activities. The definition of this term gives the aim as "*To prove the correctness of a program with respect to some abstract model of the program, using theoretical and mathematical models and rules.*" [16]. This appear to suggest a more mathematical based definition, in common with DO-178C.

Of course, ISO 26262 uses the term formal verification rather than DO-178Cs term of formal methods or IEC 61508s formal proof. Therefore, it is quite possible that a technique might satisfy the definition of formal verification within the context of ISO 26262, whilst not satisfying different definitions from other standards from other industries.

Regardless of this debate, exhaustive techniques such as model checking are often not viable for certain types of functionality, as explained earlier. In their absence, a more rigorous testing process, based on formalised requirements, is welcome – however it is labelled.

5 Limitations

Throughout this paper, the main limitations in the methods presented have been outlined. In general, these tend to relate to the type of functionality represented by the system being verified. Systems (or parts of systems) that represent state machines or logic are well suited to this workflow. Those that exhibit control behaviour such as PID controllers, those that are mathematically intense or use data-types such as floating point will find the workflow less applicable, although we are investigating this further. However, this guidance varies according to the specific activity in question; for example, the maximum stack depth analysis of the object code is much less affected by this issue than the model checking.

The workflow deliberately maintains a separation of the natural language requirements and the formalised requirements. That is to say, the latter do not replace the former. Although the formalised requirements are relatively accessible, in Ricardos experience it is still more natural for the domain experts to initially write the requirements in natural language, albeit in a structured manner. This necessarily adds some overhead as there will be inevitable changes to requirements. However, the benefit of keeping requirements in a form that is natural to domain experts is expected to outweigh the additional effort.

Perhaps the key limitation in this workflow that must be recognised is that the workflow is geared towards verification (the requirements were implemented correctly) rather than validation (the requirements were appropriate). The starting point of the workflow is a set of requirements that are intended to reflect the desired behaviour. If the requirements fundamentally do the wrong thing, then of course the later verification activities will be of little use. This is not so much a criticism of the workflow, since it is not intended to address all development challenges, but more a reminder that it must be integrated within an overall development context.

6 Future Work

Currently the translation from natural language requirements to the formal patterns is a manual (but tool supported) activity. Ricardo has found a consistent requirements structure (achieved through boilerplates) is a significant aid to this process and has allowed us to frequently map boilerplates to the most likely pattern. This raises the question as to whether the requirements could be automatically analysed to suggest the most appropriate patterns to the user. Conversely, such an approach could be used instead to help review the manual translation.

Further effort is needed to understand where independence is required in this process. For example, is it acceptable for the original author of requirements to formalise the requirements, or should this be done by an independent person? There are advantages and disadvantages in each case, and we intend to explore this aspect further within the MBAT project. A related area is which work-products should be reviewed, especially those derived automatically. This clearly has a close relationship with the confidence in the tools and tool-qualification.

The process itself still needs optimising to decide which activities should be performed to what degree of rigour. For example, robustness checking on the model and code clearly overlap and so at the model level it might be performed only unofficially by the development engineers themselves as they work, with no need to store results, qualify tools etc. In contrast, the robustness check on the code would be an official verification activity, with processes regarding storage of results, managing deviations, independence etc.

Related to this optimisation is the relationship between the MBAT techniques and the crucial foundation software engineering techniques. For example, when should requirements formalisation be performed with respect to peer review of the requirements? To what extend might the formalisation process cover some aspects of a peer review (if performed with independence)?

Within Ricardo we have not yet evaluated the workflow with respect to product lines or variants. Our main use case is a transmission controller that re-uses common models shared with a number of other projects. At the requirements stage we identify which projects each requirement applies to. Therefore we anticipate significant potential re-use of formalised requirements between the product lines. Since the test stimuli can easily be regenerated for each variant, we expect this also to reduce the overhead of adapting tests for each variant.

The collection of metrics to measure the usability of the work-flow is clearly crucial to any decision on its future adoption. As part of the MBAT project, metrics related to a number of goals (such as effort reduction) are defined and being collected. The metrics for the goals rely on input metrics such as number of requirements in a set and number of those successfully formalised, effort required to formalise a set of requirements, number of defects detected from a set of fault seeded models and so forth. At the time of writing these are still being collected.

Finally, since an important limitation is the ability to formalise requirements for some types of functionality, it would be useful to look at other formalisms that may be able to better represent those areas. Such formalisms must be seamlessly integrated into the tool-chain to make such a mix practical. Alternatively, it may be appropriate only to formalise a subset of requirements, such as those explicitly related to safety that were identified through associated safety analysis.

7 Conclusions

The workflow defined and being evaluated by Ricardo looks promising in delivering a high-degree of confidence that the final object code meets the formalised requirements (derived from the original natural language requirements) in a robust manner. It is expected to do so in both a time and cost efficient manner through combinations of analysis and test, replacement of effort intensive manual tasks with automation, and the ability to catch many issues early in the life-cycle.

A key advantage of the approaches in this workflow is that they are accessible to engineers without them needing to be experts in the underlying formal methods, as these are hidden. This allows them to concentrate on their domain knowledge.

It is important to note that advanced techniques must be built upon strong foundations. Advanced techniques cannot compensate for poor execution of software engineering basics, such as requirements that do not satisfy the real intent, overly complex implementations, poor software architecture, lack of a clear and effective modelling and coding style, and ineffective change control for example. Indeed, such advanced techniques rely on these basics being in place, e.g. overly complex features may be impractical to analyse whilst confidence in any results relies on knowing that what has been deployed is what has actually been verified.

Furthermore, it must be recognised that such a workflow addresses only some issues. The greatest challenge of developing software is ensuring the requirements are appropriate to solve the problem for which the system is being developed. While activities such as requirements formalisation can improve the quality of the requirements with respect to attributes such as ambiguity, they cannot ensure that requirements are appropriate to satisfy the real-world intent. Therefore, the workflow must be considered in a wider context that includes techniques such as rapid proto-typing and similar techniques.

Ricardo, as part of the MBAT project, are currently applying this workflow to software features from a real production project in order to evaluate how it works in practice, to understand better the limitations and if they can be overcome whilst measuring the effect on the overall effort and quality. Additionally, along with our partners, Ricardo aims to address some of the areas of future work identified.

Acknowledgements. The research leading to these results has received funding from the EU ARTEMIS Joint Undertaking under grant agreement n^{o} 269335 and UK Technology Strategy Board. Additionally, the author would like to thank Greg Stuart and Tom Bienmüller for their comments.

References

1. Galloway, A., Iwu, F., McDermid, J.A., Toyn, I.: On the formal development of safety-critical software. In: Meyer, B., Woodcock, J. (eds.) VSTTE 2005. LNCS, vol. 4171, pp. 362–373. Springer, Heidelberg (2008)
2. Kästner, D., Brockmeyer, U., Pister, M., Nenova, S., Bienmüller, T., Dereani, A., Ferdinand, C.: Leveraging from the combination of model-based analysis and testing. In: EmbeddedWorld (2013)
3. ISO 26262 compliant verification of functional requirements in the model-based software development process, BTC Embedded Systems (2011)
4. Astrée, `http://www.absint.com/astree/index.htm`
5. Polyspace, `http://www.mathworks.co.uk/products/polyspaceserverc/`
6. MBAT project, `https://www.mbat-artemis.eu/home/`
7. CESAR deliverable D_SP2_R2.3_M3, On the formal development of safety-critical software
8. ISO 26262, Road vehicles – Functional safety, part 6: Product development at the software level
9. ISO 26262, Road vehicles – Functional safety, part 8: Supporting processes

10. ISO 26262, Road vehicles – Functional safety, part 1: Vocabulary
11. Oxford Dictionaries, http://oxforddictionaries.com
12. Collins Dictionary, http://collinsdictionary.com
13. Model-based software development for safety-critical systems TargetLink reference workflow, dSPACE (2009)
14. Software considerations in airborne, systems and equipment certification, DO-178C / ED-12C
15. Formal methods supplement to ED-12C and ED-109A
16. Functional safety of electrical / electronic/ programmable electronic safety-related systems, IEC 61508, part 7
17. Addendum to TargetLink reference workflow overview and variations, dSPACE and BTC-Embedded Systems (2012)
18. Simulink Design Verifier, http://www.mathworks.co.uk/products/sldesignverifier/
19. Embedded Validator, http://www.btc-es.de/index.php?lang=2andidcatside=5
20. Advanced validation techniques meet complexity, OFFIS and I-Logix
21. EmbeddedSpecifier, http://www.btc-es.de/index.php?lang=2andidcatside=52
22. EmbeddedTester, http://www.btc-es.de/index.php?lang=2andidcatside=2

Boolean Quantifier Elimination for Automotive Configuration – A Case Study

Christoph Zengler and Wolfgang Küchlin

Symbolic Computation Group, Wilhelm-Schickard-Institute for Informatics,
Universität Tübingen, Germany
{zengler,kuechlin}@informatik.uni-tuebingen.de

Abstract. This paper evaluates different algorithms for existential Boolean quantifier elimination in the area of automotive configuration. We compare approaches based on model enumeration, on resolution, on dependency sequents, on substitution, and on knowledge compilation with projection. We describe two real-life applications: model counting on a set of customer-relevant options and projection of BOM (bill of materials) constraints. Our work includes an implementation of the presented techniques on top of state-of-the-art tools. We evaluate the different approaches on real production data from our collaboration with BMW.

1 Introduction

Motivation. For a long time, product configuration systems have been among the most prominent and successful applications of automated reasoning methods in practice [1]. As a result, computer aided configuration systems have been used in managing complex software, hardware, or network settings. Another application area of these configuration systems is the automotive industry. Here they helped to realize the transition from the mass production paradigm to present-day mass customization. Besides CSP encodings [2] also propositional encodings [3] of configuration problems proved to be a viable alternative in the automotive industry. Specific queries to the configuration base can then be answered by a decision procedure for propositional logic, a prominent one being SAT solving. In our cooperation with major German car manufacturers, we encountered a new class of problems which cannot be solved solely by a SAT solver but require the projection of a large formula to a subset of its variables.

Related Work. This projection is equivalent to eliminating quantifiers in a Boolean formula and therefore is a well studied field [4,5,6]. In our case it is sufficient to eliminate existential quantifiers from the formula, which is also well studied in the context of symbolic model checking [7,8]. Approaches based on BDDs are not feasible for our problems because different experiments showed that the formulas we deal with are in general too large to be compiled into BDDs in reasonable time [9,10]. DNNF [11] proved to be a viable alternate knowledge

C. Pecheur and M. Dierkes (Eds.): FMICS 2013, LNCS 8187, pp. 48–62, 2013.

compilation format. Besides DNNF, we tested approaches based on model enumeration [12,13,14], on clause distribution by resolution [15,16], and a very recent approach of Goldberg and Manolios based on dependency sequents [17].

Contribution. This paper presents two new applications for Boolean quantifier elimination which aroused industrial interest. We present an overview of which current techniques and tools in automated reasoning are suitable for this task. We were able to identify quantifier elimination procedures from the above mentioned approaches which could solve all real-life industrial instances in times < 100 ms and therefore are suited for industrial application. We implemented a collection of these algorithms into a prototype for BMW which is planned to be industrialized in 2013. To the best of our knowledge, this is one of the first applications of Boolean quantifier elimination in the area of configuration on this scale.

Outline. In Section 2 we present the theoretical preliminaries for this paper and especially state the quantifier elimination problem for Boolean formulas. The following Section 3 gives a short introduction in the encoding of automotive configuration problems as Boolean formulas. Then it states two industrial problems from our cooperation with BMW, which can be solved by Boolean quantifier elimination. Section 4 presents the different approaches we identified to solve the quantifier elimination problem. We benchmark these different approaches on real-life production data in Section 5. Section 6 concludes the paper and gives an outlook to further research opportunities.

2 Preliminaries

We use the standard notation for propositional logic with propositional variables, Boolean operators $\neg$, $\vee$, $\wedge$, $\longrightarrow$, $\longleftrightarrow$, and Boolean constants $\mathbf{T}$ and $\mathbf{F}$. Variables can be bound by existential quantifiers $\exists x$. A *conjunctive normal form* (CNF) is a conjunction of clauses. A *clause* is a disjunction $(\lambda_1 \vee \cdots \vee \lambda_n)$ of literals. Each *literal* λ_i is either a variable x_i or its negation $\neg x_i$. We denote by $\mathrm{var}(\lambda)$ the variable of a literal. It is convenient to identify a formula φ in CNF with the set $\mathrm{cl}(\varphi)$ of all clauses contained in it and to identify a clause c with the set $\mathrm{lit}(c)$ of all literals contained in it. We write $\mathrm{vars}(\varphi)$ to denote the finite set of variables occurring in a formula φ.

Definition 1 (Existential Sentence). *An existential sentence is a propositional formula $\varphi = \exists x_1 \ldots \exists x_n \psi$, with $\mathrm{vars}(\psi) = \{x_1, \ldots, x_n\}$ and ψ quantifier-free. This means φ is in prenex normal form and all variables are existentially quantified.*

Definition 2 (Existential Formula). *A propositional formula φ is called an existential formula, if it is of the form $\varphi = \exists x_1 \ldots \exists x_n \psi$, with $\{x_1, \ldots, x_n\} \subset \mathrm{vars}(\psi)$ and ψ quantifier-free. This means φ is in prenex normal form and all variables are either existentially quantified or free.*

$\mathrm{fr}(\varphi)$ denotes the free variables of φ, $\mathrm{bd}(\varphi)$ denotes the quantified variables of φ. The quantifier-free part of a prenex formula φ is called its *matrix* $\mathrm{mat}(\varphi)$. An *assignment* for φ is a partial function $\alpha : \mathrm{fr}(\varphi) \to \{\mathbf{true}, \mathbf{false}\}$ mapping free variables to truth values. We follow the convention to write $\alpha \models \varphi$ when some formula φ holds with respect to α and call α a model of φ in this case.

Definition 3 (SAT). *Let φ be an existential sentence.* $\mathrm{SAT}(\varphi) = \mathbf{true}$ *holds iff there exists an assignment α with $\alpha \models \mathrm{mat}(\varphi)$.*

Definition 4 (Existential Boolean Quantifier Elimination (EBQE)). *Let φ be an existential formula. Then* $\mathrm{EBQE}(\varphi) = \varphi'$ *with $\varphi \equiv \varphi'$ and φ' is quantifier-free. This means that φ' establishes necessary and sufficient conditions on* $\mathrm{fr}(\varphi)$ *for the existence of a satisfying assignment.*

See Example 10 as an example for EBQE.

Remark 5. Note that only free variables are assigned. Hence $\varphi \equiv \varphi'$ in this context means that for each assignment α it holds $\alpha \models \varphi$ iff $\alpha \models \varphi'$. Thus φ' is the strongest formula implied by φ in terms of $\mathrm{fr}(\varphi)$ [11].

3 Applications in Automotive Configuration

This section describes two applications of Boolean quantifier elimination in automotive configuration. These applications are real-life requests from our industrial collaborations with major German and international car manufacturers. First we describe how automotive configuration is modeled mathematically in general before we describe the two applications. In this section we will not focus on the configuration method of a single car manufacturer, but sketch the patterns we identified at all major companies.

3.1 Automotive Configuration

In automotive (and many other products) configuration we distinguish between two different configuration levels. The *high level configuration (HLC)* describes all constructable vehicles in an abstract way. Each *option* in the HLC implies many *physical materials* in the actual vehicle. E.g. the option *entertainment system* implies materials such as *head unit*, *cables*, *mounting material*, etc. The second level of configuration is the *low level configuration (LLC)*. At this level the actual physical materials of a vehicle are modeled, meaning which option or combination of options from the HLC implies which physical materials on the low level.

To understand the following sections, it is important to know the usual product hierarchy of a car manufacturer. Most companies have certain *product lines* at the top level of this hierarchy. Within a product line there are different *product series* and within a product series there are finally different *product types*. A product type is characterized by a certain set of fixed options. These options usually include

among others: the motor, left- or right steering, or front- rear- or all-wheel drive. A customer does not choose these options individually but indirectly selects them by fixing a product type. We call these options *type determining options (TDO)* and denote the set of all TDOs with $\mathcal{O}_T$.

Example 6. As an example we look at the current product hierarchy of BMW. An example product line is the *BMW 3 series*. Within this product line we have e.g. product series *BMW 3 Series Sedan, BMW 3 Series Touring*. Within the product series *BMW 3 Series Sedan* we find e.g. the product types *320i Sedan, left steering, rear wheel drive* or *320i xDrive Sedan, left steering, all wheel drive*.

High Level Configuration. The main building blocks of the HLC are abstract options. Each option represents a part of the vehicle or a configuration option of a software component, etc. Typical options are e.g. entertainment system, navigation system, or heated front seats. In general, options are packaged into *option families*. Within one option family the customer can choose at most one option. Some manufacturers force every option to be in one family, some allow options without a family. Also the question whether an option from each family *has* to be selected or *can* be selected, is handled differently in the various companies. For this presentation, we assume that options can be in families, but do not have to, and that you can select at most one option from a family, but do not have to. There are also rules describing constraints between the different options. All together the HLC describes the set of all valid customer orders. Within the options there are often many which are not directly selectable by the customer: *control options*. These control options are used to control certain aspects of the manufacturing process, configurations for digital equipment, and so on. One example for such a control option is e.g. the choice whether the speedometer shows the speed in km/h or miles/h. This is an abstract configuration option of a vehicle, but usually the customer cannot choose between its different values but the car company does, depending on the customer's location.

Formally, we distinguish two sets of high level options: $\mathcal{O}_S$ for customer selectable options and $\mathcal{O}_C$ for control options. The set of all options is denoted by $\mathcal{O} = \mathcal{O}_S \dot{\cup} \mathcal{O}_C$. Each option $o \in \mathcal{O}$ is interpreted as a Boolean variable. Therefore it can have only two different states. If it is assigned to **true**, the option is selected by the customer in her current order of a vehicle; If it is assigned to **false**, it is not selected in the current order. An option family F packages options $o_1^F, \ldots, o_n^F$ with the condition that only one option of $1 \ldots n$ can be selected in a valid order at the same time. The set $\mathcal{R}$ of rules contains all constraints between the options. Each constraint has to evaluate to **true** in a valid order. Rules can be arbitrary Boolean formulas with variables from $\mathcal{O}$ and Boolean connectors $\neg, \vee, \wedge, \longrightarrow$ and $\longleftrightarrow$.

Example 7. We assume two families $G = \{o_1^G, o_2^G, o_3^G\}$ for GPS systems and $E = \{o_4^E, o_5^E\}$ for entertainment systems and three options without a family: o_6 for support for Chinese characters in the headunits, o_7 for special Japan support and o_8 for speech assistance in the vehicle. o_6 and o_7 are not customer-selectable.

Therefore we have: $\mathcal{O}_S = \{o_1^G, o_2^G, o_3^G, o_4^E, o_5^E, o_8\}$ and $\mathcal{O}_C = \{o_6, o_7\}$. As a set of rules we have $\mathcal{R} = \{o_1^G \longrightarrow o_4^E, o_4^E \longrightarrow \neg o_2^G, o_5^E \longrightarrow \neg o_3^G, o_6 \longleftrightarrow \neg o_7, o_8 \longrightarrow (o_1^G \wedge o_4^E)\}$.

Product Description Formula. From the HLC we construct a Boolean formula describing all valid vehicle configurations; this is called the *product description formula (PDF)*. The PDF summarizes the cardinality constraints from the option families, the rules for the options and often some manufacturer-specific specialities in one formula. Each satisfying assignment (model) of the PDF describes one valid customer order of a vehicle. Therefore the model count of the PDF yields the number of different constructible cars. Usually the PDF is built on the product type level, i.e. there is one PDF per product type. Even at this level, the model counts are between 10^{10} and 10^{100} [18].

Example 8. For the HLC in example 7, we construct the following formula

$$\begin{aligned} \mathrm{CC}_G = {} & (o_1^G \longrightarrow \neg o_2^G \wedge \neg o_3^G) \wedge \\ & (o_2^G \longrightarrow \neg o_1^G \wedge \neg o_3^G) \wedge \\ & (o_3^G \longrightarrow \neg o_1^G \wedge \neg o_2^G) && \text{(cardinality constraint for } G\text{)} \\ \mathrm{CC}_E = {} & o_4^E \longleftrightarrow \neg o_5^E && \text{(cardinality constraint for } E\text{)} \end{aligned}$$

$$\begin{aligned} R = {} & (o_1^G \longrightarrow o_4^E) \wedge (o_4^E \longrightarrow \neg o_2^G) \wedge (o_5^E \longrightarrow \neg o_3^G) \wedge \\ & (o_6 \longleftrightarrow \neg o_7) \wedge (o_8 \longrightarrow (o_1^G \wedge o_4^E)) && \text{(rules)} \\ \mathrm{PDF} = {} & \mathrm{CC}_G \wedge \mathrm{CC}_E \wedge R && \text{(PDF)} \end{aligned}$$

Low Level Configuration. The second level of configuration is the *low level configuration (LLC)*. At this level, the actual physical materials of a product series or a product line are modeled. Usually all materials of a vehicle are stored in a *bill of materials (BOM)*. In the BOM each material gets a Boolean *usage constraint* with variables from $\mathcal{O}$ and $\mathcal{O}_T$ and arbitrary Boolean connectors. If this constraint evaluates to **true** for a given order, the material must be part of the ordered vehicle, otherwise it is not used in this vehicle.

Example 9. For each material we store its number, its description and its usage constraint. As an example we show an excerpt from a BOM for head units referring to the HLC in Example 7.

Material Number	Description	Usage Constraint
123451	Head Unit 1	$o_1^G \wedge o_4^E \wedge \neg(o_7 \vee o_8)$
123452	Head Unit 2	$o_2^G \wedge o_5^E$
123453	Head Unit 3	$(o_4^E \wedge o_7 \wedge \neg o_1^G) \vee o_8$
123454	Head Unit 4	$o_3^G \wedge o_4^E \wedge \neg o_7$
123455	Head Unit 5	$o_1^G \wedge o_4^E \wedge o^7$

3.2 Application 1: Model Counting on Customer-Relevant Options

A requirement we heard very often from our industrial partners was to count the number of constructible vehicles of a certain product type, i.e. computing the model count of the respective PDF. This gets especially interesting if we combine this with arbitrary selection criteria. E.g. we can compute how many different cars of a certain product type and a given color can be built for the US market. This information can be combined with certain statistics and pick rates and yield important management insight into the whole product variety. The problem with the first numbers we computed was that they were very large. So the requirement was to count only the different constructible vehicles wrt. the customer-selectable options. Formally we want to compute the model count of the PDF where the options from $\mathcal{O}_C = \{o_1, \ldots, o_n\}$ are eliminated:

$$|\{\alpha \mid \alpha \models \mathrm{EBQE}(\exists o_1 \ldots \exists o_n \, \mathrm{PDF})\}|$$

This reflects the number of different customer orders.

Example 10. We consider the PDF of Example 8. The model count for this PDF is 8: $\{o_1^G, o_4^E, o_6, \neg o_7, \neg o_8\}$, $\{o_1^G, o_4^E, o_6, \neg o_7, o_8\}$, $\{o_1^G, o_4^E, \neg o_6, o_7, \neg o_8\}$, $\{o_1^G, o_4^E, \neg o_6, o_7, o_8\}$, $\{o_2^G, o_5^E, o_6, \neg o_7, \neg o_8\}$, $\{o_2^G, o_5^E, \neg o_6, o_7, \neg o_8\}$, $\{o_3^G, o_4^E, o_6, \neg o_7, \neg o_8\}$, $\{o_3^G, o_4^E, \neg o_6, o_7, \neg o_8\}$. The formula $\mathrm{EBQE}(\exists o_6 \exists o_7 \, \mathrm{PDF})$ has a model count of 4 and hence lists only the different configurations wrt. the customer-selectable options: $\{o_1^G, o_4^E, o_8\}$, $\{o_1^G, o_4^E, \neg o_8\}$, $\{o_2^G, o_5^E, \neg o_8\}$, $\{o_3^G, o_4^E, \neg o_8\}$.

3.3 Application 2: Projection of BOM Constraints

Another common problem is the projection of BOM usage constraints to certain options. A BOM is usually not at the product type level—there would be far too many BOMs to manage. Usually a BOM is at least at the product series level, often even at the product line level, i.e. there is one BOM for the whole product line. In order to distinguish which material is used in different product types, the TDOs have to be used in the usage constraints. This leads to long and complex constraints. In reality we often encounter constraints with 20–30 customer-selectable options, 10–20 type determining options and hundreds of subterms. Often these constraints are not generated by humans but by some computer system. The problem is that once created, humans have to maintain these constraints e. g. when parts or suppliers change. Usually a single specialist is responsible for a certain group of customer-selectable options and is therefore only interested in these options. Looking at a material of a BOM, she needs to know for which combinations of options—relevant to her—this material gets selected. Therefore it can be very helpful to be able to project a usage constraint to a certain set of relevant options and hence to eliminate all other options and especially the TDOs. The result is then a projected constraint, containing only her relevant options, which must be satisfied in order that there exist vehicles which use this material.

Example 11. We have options $\mathcal{O}_C = \{o_1, o_2, o_3, o_4\}$ and TDOs for motors M_1, M_2, M_3 and drives A, F, R. We consider the following usage constraint:

$$[(o_1 \wedge M_1 \wedge (\neg o_2 \vee o_3)) \vee ((M_2 \vee M_3) \wedge \neg o_1 \wedge o_2)] \wedge o_4 \wedge \neg F \wedge \neg G$$

Even for this very short constraint it is not obvious for which combinations of options in $\mathcal{O}_C$ this material can possibly be selected. A restriction of the constraint to $\mathcal{O}_C$ yields

$$o_4 \wedge ((o_1 \wedge \neg o_2) \vee (o_1 \wedge o_3) \vee (\neg o_1 \wedge o_2))$$

To stress the meaning of this result once again: The result is a necessary and sufficient condition for the existence of vehicles which use this part. That is exactly what the maintainer is interested in.

4 Existential Boolean Quantifier Elimination

In this section we present six different approaches for existential Boolean quantifier elimination: (MEP) model enumeration with projection, (MEPI) model enumeration with generation of shortest prime implicants, (CD) variable elimination by clause distribution, (SS) substitute & simplify, (DNNF) DNNF computation with projection, and (DDS) quantifier elimination by dependency sequents. The selection of these approaches was based on literature, availability of implementations, and last but not least our experience with different approaches. We are aware that there are many more approaches in the literature, but many of these are just minor variants of the above mentioned. All six approaches yield a quantifier-free equivalent for a given existential formula as input, but the input and output formats differ. As we saw in Section 3, for different areas of application, different output formats are advantageous. Table 1 summarizes the respective formats of the input formula φ and its quantifier-free equivalent φ' for the different approaches.

Table 1. Comparison of the different approaches for EBQE(φ)

	Approach	**Format of** mat(φ)	**Format of** φ'
(MEP)	model enumeration & projection	CNF	DNF
(MEPI)	model enumeration & prime implicants	CNF	CNF
(CD)	clause distribution	CNF	CNF
(SS)	substitute & simplify	arbitrary	arbitrary
(DNNF)	DNNF compilation & projection	CNF	DNNF
(DS)	dependency sequents	CNF	CNF

Remark 12. Theoretically (ME) and (DNNF) do not require their input formula to be in CNF, but the available algorithms and tools are only available for CNF inputs.

4.1 Model Enumeration with Projection (MEP)

Consider an existential formula $\varphi = \exists x_1, \ldots, x_n \psi$. One way to enumerate all models and to project them to the set of free variables $Y = \mathrm{fr}(\varphi)$ is, to use a SAT solver to compute a cube $c_1 = (\bigwedge Y_1) \wedge \neg(\bigvee Y_2)$, for which $\psi \wedge c_1$ is satisfiable and Y_1 and Y_2 partition Y. Adding the clause $\neg c_1$ to the solver avoids this projected model in the future. This process is repeated until all cubes are enumerated. The disjunction of all cubes is then a quantifier free equivalent to the input formula and is obviously in DNF. Current model enumerators [12,13] are usually implemented on top of CDCL solvers by modifying their backtracking procedures and/or tracking blocking clauses in order to prevent the solver from detecting the same projected solutions again and again.

4.2 Model Enumeration and Generation of Shortest Prime Implicants (MEPI)

A variant of Approach 4.1 was presented in 2011 by Brauer, King, and Kriener [14]. In contrast to adding arbitrary projected cubes found by the solver, their idea is to search for shortest prime implicants first. Therefore they use cardinality constraints based on sorting networks to first find the shortest cubes and increase the length of the implicants successively. They then apply dualisation to construct a CNF formula out of the set of all implicants.

4.3 Variable Elimination by Clause Distribution (CD)

The ideas for this approach go back to Davis and Putnam [15] (Affirmative-Negative Rule and Rule for Eliminating Atomic Formulas) and are used for variable elimination in many QBF solvers, e.g. Quantor [16]. To eliminate an existentially quantified variable x, we (1) compute all resolutions on x and (2) remove all clauses containing x in either phase. In the special case that x occurs only in one phase in the clause set, step (1) is omitted. The output of this procedure is obviously a quantifier-free equivalent.

4.4 Substitute and Simplify (SS)

The principle of *substitute & simplify* goes back to the work of Boole and Shannon. For the *substitute* part, we observe that a formula φ with $x \in \mathrm{vars}(\varphi)$ is equivalent to its Shannon expansion over x: $x \wedge \varphi[x/\mathbf{true}] \vee \neg x \wedge \varphi[x/\mathbf{false}]$. We can eliminate a single existential quantifier with

$$\exists x \varphi \equiv \varphi[x/\mathbf{true}] \vee \varphi[x/\mathbf{false}]. \tag{1}$$

E.g. the QBF solver Qubos [19] uses this approach to eliminate existentially quantified variables. Seidl & Sturm [4] show that for a formula φ and its quantifier-free equivalent φ' it holds that $|\varphi'| = 2^{O(|\varphi|)}$ where $|\cdot|$ denotes the word length. Therefore the *simplify* part is essential after each elimination step.

This approach can be further optimized for our application. In our problem instances, we often encounter pure literals. These pure literals can be eliminated directly without performing a substitution. As we have seen in the last section, we have many cardinality constraints. These cardinality constraints can also be handled separately and therefore superfluous substitution steps can be avoided.

4.5 DNNF Computation and Projection (DNNF)

The knowledge compilation format DNNF (decomposable negation normal form) is considered more succinct than BDDs [11]. It thus might help alleviate the ubiquitous memory explosion problem of BDDs for large formulas. Apart from that, DNNF supports several polynomial time queries among which we will focus our attention to the projection of DNNF formulas to some set of variables A.

A formula in negation normal form is a DNNF if the decompositional property holds, i.e. for each conjunctive subformula $\bigwedge_i \psi_i$ of φ we require $\text{vars}(\psi_i) \cap \text{vars}(\psi_j) = \emptyset$ for all $i \neq j$. Furthermore, a DNNF φ is called *deterministic* (i.e. d-DNNF) if for each disjunctive subformula $\bigvee_i \psi_i$ the conjunction $\psi_i \wedge \psi_j$ is inconsistent for all $i \neq j$.

After transforming the matrix of a formula φ to its d-DNNF representation ψ with the help of a DNNF compiler [20], a formula corresponding to the strongest entailed formula of φ in terms of the set $\text{fr}(\varphi) \subseteq \text{vars}(\varphi)$ can be obtained in polynomial time in $|\psi|$ by replacing each literal whose variable is in $\text{fr}(\varphi)$ with $\mathbf{T}$ [11]. Projecting DNNF formulas maintains decomposability, yet the resulting formula might not be deterministic anymore if it was before.

4.6 Quantifier Elimination by Dependency Sequents (DDS)

A new approach to existential quantifier elimination was presented in 2012 by Goldberg and Manolios [17]. The observation is that quantifier elimination on a formula $\varphi = \exists x_1, \ldots, x_n \psi$ is trivial if ψ does not depend on the quantified variables $x_1, \ldots, x_n$. However, if ψ depends on the quantified variables, the plan is to add additional clauses implied by ψ such that $x_1, \ldots, x_n$ become redundant. Therefore they introduce so called *dependencey sequents*. A dependency sequent is used to record the fact that a set of quantified variables is redundant under a partial assignment. They introduce an algorithm *DDS* (Derivation of Dependency Sequents) which starts by computing simple dependency sequents and computes new dependency sequents by joining old ones. The result is a quantifier free equivalent in CNF.

5 Benchmarks and Evaluation

For benchmarking the different algorithms, we chose a current (2013) vehicle series from BMW. The series contains 30 product types.

5.1 Tools

The following listing summarizes the tools we used for the different approaches:

Model Enumeration with Projection (MEP). We used version 1.3.6 of `clasp`[1] which implements the model enumeration approach as described above.

Model Enumeration with Prime Implicants (MEPI). We used the authors' implementation of their algorithm[2].

Clause Distribution (CD). We extended the simplifying version of MiniSat 2[3] which has the integrated ability to eliminate variables by clause distribution.

Substitute & Simplify (SS). We implemented this approach on top of our own logic library written in Java, running on the JVM 1.7.

DNNF & Projection (DNNF). We used the DNNF Compiler `c2d`[4] which takes Dimacs CNF as an input and has the ability to perform existential QE as described above.

Dependency Sequents (DDS). We used a proof of concept implementation provided to us by the authors.

The substitute & simplify approach could be implemented on top of BDDs which is often the case in symbolic model checking [7]. But our experience is that the PDF is often too complex to be compiled into a BDD in reasonable time [10]. Therefore we chose our own logic library to test this approach.

5.2 Results

For all benchmarks we used a machine with an Intel dual-core i7 2.0 GHz (using only one core), 8 GB of RAM, running Ubuntu 12.04. We chose a timeout of 3,600 seconds. Model enumeration with projection (MEP) and model enumeration with generating shortest prime implicants (MEPI) could not solve a single instance within this time limit. That is why we did not include these in our overview of the results. After 3,600 seconds they enumerated between 500,000,000 and 700,000,000 models, which is obviously only a small fraction of the model counts as stated in Table 2. This does not mean that the approach is not suitable for Boolean QE at all. For other benchmarks [21] this approach performed very well, especially when the projected formulas have a small model count.

Table 2 summarizes the results of the benchmarks. Each line represents one product type. $|\mathcal{O}|$ states the number of HLC options for the respective product type; $|\mathcal{O}_C|$ states the number of control options; $|\mathcal{R}|$ states the number of HLC rules. The number #orig represents the model count (number of constructible cars) for the original PDF; the number #proj represents the model count for the PDF where the options of $\mathcal{O}_C$ were eliminated. The following five columns show

[1] `http://www.cs.uni-potsdam.de/clasp/`
[2] `http://www.cs.kent.ac.uk/people/staff/amk`
[3] `https://github.com/niklasso/minisat`
[4] `http://reasoning.cs.ucla.edu/c2d/`

the computation times of the different approaches for eliminating all options from $\mathcal{O}_C$ from the respective PDF. All times are stated in seconds. For clause distribution, DDS, and substitute & simplify, the stated time covers the time for parsing the input, eliminating the quantifiers, and writing the output. For the DNNF approach it covers also the DNNF compilation time. For DDS we distinguish between the standard algorithm and a version where the resulting CNF formulas are optimized afterwards. Table 3 summarizes the size of the output CNF for the approaches CD and DDS (standard and optimizing version). We will interpret the results for each approach individually.

Table 2. Benchmarks for the BMW vehicle series with 30 product types

	Instance					*QE time in s*				
									DDS	
type	$\lvert\mathcal{O}\rvert$	$\lvert\mathcal{O}_C\rvert$	$\lvert\mathcal{R}\rvert$	**#orig**	**#proj**	**CD**	**DNNF**	**SS**	**stand.**	**opt**
S2T01	423	256	211	$9.95 \cdot 10^{48}$	$2.95 \cdot 10^{16}$	0.07	0.14	0.92	**0.01**	1.08
S2T02	403	237	190	$1.20 \cdot 10^{48}$	$1.18 \cdot 10^{17}$	0.07	0.12	0.60	**0.01**	1.11
S2T03	425	258	208	$9.64 \cdot 10^{48}$	$1.64 \cdot 10^{16}$	0.07	0.13	1.00	**0.02**	1.01
S2T04	408	242	215	$2.52 \cdot 10^{47}$	$1.81 \cdot 10^{16}$	0.06	0.12	0.81	**0.01**	1.07
S2T05	223	85	122	$1.59 \cdot 10^{33}$	$3.84 \cdot 10^{13}$	**0.01**	0.03	0.12	**0.01**	0.48
S2T06	441	272	220	$4.00 \cdot 10^{53}$	$6.35 \cdot 10^{16}$	0.07	0.14	1.14	**0.01**	1.12
S2T07	424	256	229	$5.43 \cdot 10^{52}$	$2.57 \cdot 10^{17}$	0.07	0.12	0.74	**0.02**	1.15
S2T08	220	83	122	$7.78 \cdot 10^{32}$	$1.99 \cdot 10^{13}$	**0.01**	0.02	0.10	**0.01**	0.50
S2T09	433	264	224	$5.75 \cdot 10^{49}$	$3.23 \cdot 10^{16}$	0.07	0.13	1.11	**0.02**	1.13
S2T10	417	249	236	$1.01 \cdot 10^{49}$	$6.54 \cdot 10^{16}$	0.07	0.13	0.98	**0.02**	1.12
S2T11	436	268	221	$6.08 \cdot 10^{52}$	$7.99 \cdot 10^{16}$	0.07	0.15	1.19	**0.01**	1.11
S2T12	420	253	228	$3.09 \cdot 10^{52}$	$3.22 \cdot 10^{17}$	0.07	0.14	0.92	**0.02**	1.14
S2T13	420	254	215	$2.53 \cdot 10^{48}$	$1.02 \cdot 10^{16}$	0.07	0.12	0.81	**0.01**	1.13
S2T14	430	262	223	$1.48 \cdot 10^{49}$	$1.63 \cdot 10^{16}$	**0.01**	0.13	0.84	0.02	1.17
S2T15	421	254	207	$5.85 \cdot 10^{48}$	$3.05 \cdot 10^{16}$	0.07	0.14	0.97	**0.01**	1.08
S2T16	402	234	196	$3.75 \cdot 10^{47}$	$2.50 \cdot 10^{17}$	0.07	0.12	0.59	**0.01**	1.14
S2T17	428	259	228	$3.47 \cdot 10^{49}$	$6.35 \cdot 10^{16}$	0.06	0.14	1.23	**0.02**	1.12
S2T18	405	237	215	$1.57 \cdot 10^{48}$	$2.57 \cdot 10^{17}$	0.07	0.13	0.62	**0.02**	1.12
S2T19	422	254	228	$1.12 \cdot 10^{49}$	$3.99 \cdot 10^{16}$	0.07	0.13	0.93	**0.03**	1.15
S2T20	405	238	191	$1.81 \cdot 10^{49}$	$1.57 \cdot 10^{17}$	0.07	0.12	0.57	**0.01**	1.13
S2T21	418	251	210	$1.21 \cdot 10^{48}$	$1.53 \cdot 10^{16}$	0.06	0.12	0.79	**0.02**	1.06
S2T22	398	232	196	$6.82 \cdot 10^{46}$	$5.64 \cdot 10^{16}$	0.06	0.12	0.55	**0.01**	1.07
S2T23	421	254	202	$2.15 \cdot 10^{48}$	$6.32 \cdot 10^{16}$	0.06	0.12	0.91	**0.01**	1.10
S2T24	400	234	180	$1.32 \cdot 10^{47}$	$2.54 \cdot 10^{17}$	0.06	0.12	0.61	**0.01**	1.11
S2T25	398	253	202	$2.41 \cdot 10^{48}$	$6.31 \cdot 10^{15}$	0.06	0.12	0.56	**0.01**	0.62
S2T26	377	233	181	$8.17 \cdot 10^{46}$	$2.53 \cdot 10^{16}$	0.07	0.11	0.49	**0.01**	0.61
S2T27	417	251	214	$9.61 \cdot 10^{47}$	$1.55 \cdot 10^{16}$	0.06	0.12	0.79	**0.01**	1.04
S2T28	397	232	194	$8.33 \cdot 10^{46}$	$5.68 \cdot 10^{16}$	0.07	0.11	0.70	**0.01**	1.12
S2T29	419	252	206	$1.97 \cdot 10^{48}$	$1.24 \cdot 10^{16}$	0.06	0.12	0.78	**0.01**	1.10
S2T30	399	233	185	$2.33 \cdot 10^{47}$	$4.56 \cdot 10^{16}$	0.07	0.11	0.55	**0.01**	1.00

Model Enumeration with Projection (MEP) and Model Enumeration with Shortest Prime Implicants (MEPI) were—as stated above—not suitable for eliminating the options of $\mathcal{O}_C$ from the PDF (Application 1). Nevertheless they can be very useful for smaller constraints, e.g. the usage constraints of the BOM (Application 2). The big advantage of MEP is its output as DNF. In this context each minterm of the DNF describes one combination of relevant options, such that the material is selected for a vehicle. Therefore a DNF representation of a usage constraint is often human-readable and can be easily converted e.g. into a table where all possible combinations are recorded.

Clause Distribution (CD) performed very well on all benchmarks. It required between 4 ms and 81 ms for the elimination. The approach is well suited for calculating the projected model count (Application 1). The resulting formulas are small enough to be processed by recent model counters [18] or by `c2d` which can also perform model counting. For the projection of BOM constraints (Application 2), this approach is not recommended. In most cases, the resulting formula in CNF has not much resemblance with the original usage constraint found in the BOM. This makes it very hard for a maintainer, to match input and output constraints and find e.g. errors in the constraints.

Substitute and Simplify (SS). SS performed notably worse than clause distribution, DNNF computation, or the unoptimizing DDS. But it is still suited for Application 1. The resulting formulas are small enough to be model counted by current tools. For Application 2—the projection of BOM constraints—this approach is particularly well suited. Since substitute & simplify works on the original rule with no need to convert it to a normalform, the resemblance to the input formula is higher than in the other approaches. This simplifies the task of matching input and output of the elimination process for a human maintainer.

DNNF and Projection (DNNF). As heuristics for the DNNF compilation we chose min-fill [22] which performed notably better than the other alternatives implemented in `c2d`. The times play in the same league as the times of CD or unoptimizing DDS. However, there is one big disadvantage. As stated above, after projecting a DNNF, it is no longer guaranteed to be deterministic. However, model counting on a DNNF works only in linear time on a deterministic DNNF [23]. Therefore, to compute the projected model count, one would first have to compute the DNNF of the PDF, project it, make it deterministic again, and then count it. Unfortunately to the best of our knowledge, there is yet no algorithm and especially no implemented tool, which can convert an arbitrary DNNF into a d-DNNF without the indirection of converting it to a CNF again. Therefore this approach is not suited for Application 1. For Application 2 it is basically suited, but as for the clause distribution approach, the problem is that the resulting formula often has little resemblance with the input formula.

Dependency Sequents (DDS). DDS without optimization of the result formula performs best on all examples. All instances could be projected in < 30 ms. Looking at Table 3 we see that the resulting CNF are slightly larger than the ones constructed by CD. However the version of DDS which optimizes the resulting CNF yields the smallest CNF of all approaches. Therefore DDS is the best approach for Application 1. The resulting formulas are small enough to be counted be recent model counters. For Application 2 the same argument as for CD holds: the resulting CNF is not an ideal format for human maintainers which is why we would not recommend it.

Table 3. Sizes of the resulting CNF formulas

	CD		DDS		DDS (opt)	
type	#vars	#clauses	#vars	#clauses	#vars	#clauses
S2T01	190	1338	221	1978	221	**828**
S2T02	191	1316	226	1964	226	**838**
S2T03	189	1297	218	1849	218	**771**
S2T04	190	1303	221	1844	221	**798**
S2T05	151	812	205	1193	205	**633**
S2T06	192	1345	224	1980	224	**834**
S2T07	193	1342	225	1978	225	**838**
S2T08	150	807	204	1192	204	**629**
S2T09	191	1356	222	1968	222	**863**
S2T10	191	1349	223	1943	223	**828**
S2T11	193	1351	226	2053	226	**838**
S2T12	194	1348	227	2013	227	**815**
S2T13	188	1312	218	1841	218	**797**
S2T14	190	1366	221	1956	221	**876**
S2T15	190	1336	220	1950	220	**819**
S2T16	193	1323	226	1943	226	**837**
S2T17	192	1345	218	1967	218	**826**
S2T18	193	1342	223	1975	223	**838**
S2T19	191	1349	219	2044	219	**836**
S2T20	192	1328	227	2050	227	**826**
S2T21	189	1301	220	1871	220	**777**
S2T22	190	1276	225	1842	225	**794**
S2T23	192	1336	221	1969	221	**824**
S2T24	193	1314	226	1955	226	**834**
S2T25	168	1049	216	1374	216	**661**
S2T26	169	1052	219	1375	219	**663**
S2T27	190	1313	221	1846	221	**770**
S2T28	191	1274	228	1832	228	**855**
S2T29	189	1296	219	1851	219	**770**
S2T30	190	1271	224	1821	224	**787**

5.3 Evaluation in Terms of Our Applications

We want to give a short evaluation in terms of our two applications of Section 3. In order to count the number of constructible vehicles wrt. to the options in $\mathcal{O}_S$ (Application 1), there are three approaches which are suitable: Clause distribution, substitute & simplify, and dependency sequents. All three require an additional model counter. Here DNNF compilation and model counting proved to be a stable solution. We used `c2d` which was able to count the models in < 1 sec for each instance. Model enumeration with projection is not suitable because the model counts of our application instances are too large. DNNF computation with projection is not suitable because after variable elimination the output is no longer necessarily a deterministic DNNF and therefore model counting cannot be performed in linear time.

For the projection of BOM usage constraints (Application 2), we identified two suitable approaches: (1) model enumeration with projection and (2) substitute & simplify. Model enumeration with projection is especially well-fitted because of its output format DNF. Substitute & simplify yields a formula with a high resemblance to the original input formula which can be advantageous for a human maintainer. Clause distribution, DNNF computation and projection,

and DDS are not very well-fitted because their output formats are too distinct from the input formula. Therefore humans would have problems mapping the projected constraints to the original ones.

6 Conclusions and Outlook

We presented six different approaches to perform existential Boolean quantifier elimination. These approaches differ in input and output formats. We presented two real-life applications from the area of automotive configuration: (1) Counting all models of a product description formula restricted to a set of certain relevant options and (2) projecting usage constraints of materials in the BOM. We implemented all approaches on top of current tools and evaluated them on a set of real application instances. We interpreted the benchmark results in terms of our application and identified suitable solutions for our industrial problems.

An interesting question looking at other problem domains and the success of portfolio-based approaches in SAT solving is whether we can identify structural properties of the input formula which gives us information, which approach could perform best. We have done some experiments in this direction and the first results look promising.

Acknowledgements. Thorsten Halbhuber from the BMW Group Munich proposed Application 1 and helped to a great extent to get the necessary data and understand the product structure of BMW.

References

1. Sabin, D., Weigel, R.: Product configuration frameworks-a survey. IEEE Intelligent Systems 13, 42–49 (1998)
2. Astesana, J.M., Bossu, Y., Cosserat, L., Fargier, H.: Constraint-based modeling and exploitation of a vehicle range at Renault's: Requirement analysis and complexity study. In: Proceedings of the ConfWS 2010, pp. 33–39 (2010)
3. Küchlin, W., Sinz, C.: Proving consistency assertions for automotive product data management. Journal of Automated Reasoning 24, 145–163 (2000)
4. Seidl, A.M., Sturm, T.: Boolean quantification in a first-order context. In: Ganzha, V.G., Mayr, E.W., Vorozhtsov, E.V. (eds.) Proceedings of the CASC 2003, pp. 329–345. Institut für Informatik, TU München (2003)
5. Benedetti, M., Mangassarian, H.: QBF-based formal verification: Experience and perspectives. JSAT 5, 133–191 (2008)
6. Sturm, T., Zengler, C.: Parametric quantified SAT solving. In: Proceedings of the ISSAC 2010. ACM, New York (2010)
7. McMillan, K.L.: Symbolic Model Checking. Kluwer Academic Publishers, Norwell (1993)
8. McMillan, K.L.: Applying SAT methods in unbounded symbolic model checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, p. 250. Springer, Heidelberg (2002)

9. Narodytska, N., Walsh, T.: Constraint and variable ordering heuristics for compiling configuration problems. In: Proceedings of the IJCAI 2007, pp. 149–154. Morgan Kaufmann Publishers Inc., San Francisco (2007)
10. Matthes, B., Zengler, C., Küchlin, W.: An improved constraint ordering heuristics for compiling configuration problems. In: Mayer, W., Albert, P. (eds.) Proceedings of the ConfWS, pp. 36–40 (2012)
11. Darwiche, A.: Decomposable negation normal form. Journal of the ACM 48, 608–647 (2001)
12. Grumberg, O., Schuster, A., Yadgar, A.: Memory efficient all-solutions SAT solver and its application for reachability analysis. In: Hu, A.J., Martin, A.K. (eds.) FMCAD 2004. LNCS, vol. 3312, pp. 275–289. Springer, Heidelberg (2004)
13. Gebser, M., Kaufmann, B., Schaub, T.: Solution enumeration for projected boolean search problems. In: van Hoeve, W.-J., Hooker, J.N. (eds.) CPAIOR 2009. LNCS, vol. 5547, pp. 71–86. Springer, Heidelberg (2009)
14. Brauer, J., King, A., Kriener, J.: Existential quantification as incremental sat. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 191–207. Springer, Heidelberg (2011)
15. Davis, M., Putnam, H.: A computing procedure for quantification theory. Journal of the ACM 7, 201–215 (1960)
16. Biere, A.: Resolve and expand. In: Hoos, H.H., Mitchell, D.G. (eds.) SAT 2004. LNCS, vol. 3542, pp. 59–70. Springer, Heidelberg (2005)
17. Goldberg, E., Manolios, P.: Quantifier elimination by dependency sequents. In: Proceedings of the FMCAD 2012, pp. 34–43. IEEE Computer Society (2012)
18. Kübler, A., Zengler, C., Küchlin, W.: Model counting in product configuration. In: Proceedings of the LoCoCo 2010. EPTCS, vol. 29, pp. 44–53 (2010)
19. Ayari, A., Basin, D.: Qubos: Deciding quantified Boolean logic using propositional satisfiability solvers. In: Aagaard, M.D., O'Leary, J.W. (eds.) FMCAD 2002. LNCS, vol. 2517, pp. 187–201. Springer, Heidelberg (2002)
20. Darwiche, A.: New advances in compiling CNF to decomposable negational normal form. In: Proceedings of the ECAI 2004. IOS Press (2004)
21. Zengler, C., Kübler, A., Küchlin, W.: New approaches to boolean quantifier elimination. ACM Communications in Computer Algebra 45, 139–140 (2011)
22. Darwiche, A., Hopkins, M.: Using recursive decomposition to construct elimination orders, jointrees, and dtrees. In: Benferhat, S., Besnard, P. (eds.) ECSQARU 2001. LNCS (LNAI), vol. 2143, pp. 180–191. Springer, Heidelberg (2001)
23. Pipatsrisawat, K., Darwiche, A.: New compilation languages based on structured decomposability. In: Proceedings of the AAAI 2008, pp. 517–522. AAAI Press (2008)

Study on the Barriers to the Industrial Adoption of Formal Methods*

Jennifer A. Davis[1], Matthew Clark[2], Darren Cofer[1], Aaron Fifarek[3], Jacob Hinchman[2], Jonathan Hoffman[2], Brian Hulbert[3], Steven P. Miller[1], and Lucas Wagner[1]

[1] Rockwell Collins, Cedar Rapids, IA 52402, USA
{jadavis4,ddcofer,spmiller,lgwagner}@rockwellcollins.com
[2] Air Force Research Laboratory, WPAFB, OH 45433, USA
{Matthew.Clark3,Jacob.Hinchman,Jonathan.Hoffman}@wpafb.af.mil
[3] LinQuest Corporation, an AFRL subcontractor, Beavercreek, OH 45431, USA
{Aaron.Fifarek.ctr,Brian.Hulbert.ctr}@wpafb.af.mil

Abstract. The authors conducted an informal survey of contractors, customers, and certification authorities in the United States aerospace domain to identify barriers to the adoption of formal methods and suggested mitigations for those barriers. We surveyed 31 individuals from the following nine organizations: United States Army, Boeing, FAA, Galois, Honeywell, Lockheed Martin, NASA, Rockwell Collins, and Wind River. The top three barrier categories were education, tools, and the industrial environment (i.e., non-technical barriers with respect to personnel changes, contracts, and schedules). The top three mitigation categories were education, improving tool integration, and creating and disseminating evidence of the benefits of formal analysis. Strategies to accelerate adoption of formal methods include making formal methods a part of the undergraduate software engineering curriculum, hosting courses in formal methods for working engineers, funding the integration of tools, funding improvements to tool interfaces, and promoting/requiring the use of formal methods on future contracts.

Keywords: formal methods, survey, barriers, education, certification, industrial, tools, benefits.

1 Introduction

Aerospace systems, such as Unmanned Aerial Vehicles (UAVs), are becoming increasingly complex and non-deterministic by design. Advances in software autonomy are helping drive the development of future systems that will require greater levels of on-board, autonomous decision making as well as cooperative behaviors to achieve greater performance in their operational environment. With the complexity of these autonomous systems rising at an exponential rate, system integrators are beginning to reach the limit of their ability to exhaustively test the system. To make matters worse,

* Distribution Statement A. Approved for public release; distribution is unlimited. Case 88ABW-2012-6299.

C. Pecheur and M. Dierkes (Eds.): FMICS 2013, LNCS 8187, pp. 63–77, 2013.

many non-deterministic algorithms like adaptive control and neural networks will be impossible to fully test with traditional methods due to the enormous number of configurations the system may adopt. Whether or not these future system capabilities can be fielded safely and securely is dependent on the ability of developers to verify and validate the performance of highly complex systems. To do so requires a paradigm shift in the way system test and certification are conducted. Part of this shift, and a promising approach to mitigating this explosion in Verification and Validation (V&V) costs, is the use of advanced analysis techniques such as Formal Methods (FM). For the purposes of this paper, we include in the definition of formal methods all forms of formal analysis including static code analysis, abstract interpretation, model-checking, and theorem proving.

Formal methods have had V&V successes previously in communities such as computer hardware and software security [1] [2]. However, these techniques have made few inroads into the safety-critical software arena. The purpose of this study is to investigate why formal methods have been slow to be adopted in the aerospace domain. By identifying the largest barriers to the adoption of formal methods in the development of aerospace systems, as reported by respected domain leaders, it is easier to see which strategies would yield the greatest return on investment and maximize the adoption of these analysis techniques.

Several formal methods surveys have been conducted in the past. See, for example, [3], [4], and [5], which were published in the 1990s. Much has changed since then, especially with respect to tool performance. A fairly recent (2008) survey by Woodcock et al. [6] includes 62 applications of formal methods over 25 years in a wide range of application domains. The results were published in 2009 in both overview [7] and full report [8] forms. This and previous studies found the barriers to industrial adoption to be tool usability, lack of "ruggedized" tools, integration into the development processes, lack of evidence to support adoption decisions and appropriate cost models, perceived high entry cost of doing formal methods, lack of evidence of reduced cost for the second use of formal methods, psychological barriers, and skills barriers. Finally, The Formal Methods Manifesto 2010 [9] reports that there are still barriers (namely, the need for automation and scalability) preventing widespread use of formal methods in developing new software, despite 30 years of progress in methods and tools.

The contributions of this paper are:

1. Make current the knowledge about barriers to widespread adoption.
2. Identify barriers specific to the US aerospace domain.
3. Provide the perspective of individuals who are familiar with formal methods but have not used them.

2 Interview Process and Questions

Organizations and individuals were selected for the survey based on prior known interest or experience with the use of formal methods in the United States aerospace industry. An effort was made to identify individuals from a variety of roles in their organizations with diverse perspectives on formal methods. We sent email requests

for participation to 37 individuals representing ten organizations. Of these requests, 31 individuals agreed to participate. These individuals are employed by the following nine organizations: United States Army (3 individuals), Boeing (2), FAA (1), Galois (2), Honeywell (4), Lockheed Martin (2), NASA (5), Rockwell Collins (11), and Wind River (1). Our respondents included 14 experts, 5 users, 9 individuals familiar with formal methods but not using them, and 3 managers of users.

Listed below are the first four questions that were asked of the interviewees. These questions were intended to be open-ended and avoid biasing the respondent toward any particular kind of barrier or mitigation. After question #4, we asked respondents to rate the barriers found in the 2008 formal methods survey by Bicarregui et al. [7]

1. Please describe (at as high a level as you like) the use of formal methods within your organization, if any.
2. Has the use of formal methods in your organization increased, decreased, or stayed the same in the last 5 years?
3. What do you see as the current barriers to further adoption of formal methods (especially in your organization)?
4. Do you have any suggestions for removing these barriers?

All interviews were conducted in person or by phone. Survey responses are anonymous in this paper. Furthermore, any conclusions drawn should not be attributed to any particular individual or organization.

3 Results

The following sections summarize the results of the survey, including the change in the amount of use of formal methods, the barriers to the adoption of formal methods, and the suggested mitigations to alleviate those barriers.

3.1 Use of Formal Methods

The Use of Formal Methods Is Increasing. Eighteen out of 31 interviewees reported that the use of formal methods has increased within their organizations in the last 5 years. Five of these 18 individuals specified that the use of FM has increased slightly and one said that its use has increased dramatically. Another 8 respondents said the amount of use of formal methods has stayed the same (i.e., no noticeable change in the amount of use). Three of these 8 respondents said that their teams are using formal methods very little or not at all, so "stayed the same" means a continued lack of use. Two respondents reported that formal methods use has decreased. The remaining three did not respond or did not know the answer. These results are depicted in Fig. 1. Note that 84% of survey respondents said the use of formal methods has increased or stayed the same. If we look at the relative majority of responses for each organization, six organizations have seen a growth in the use of formal methods, and the use of formal methods has stayed the same in the remaining three organizations.

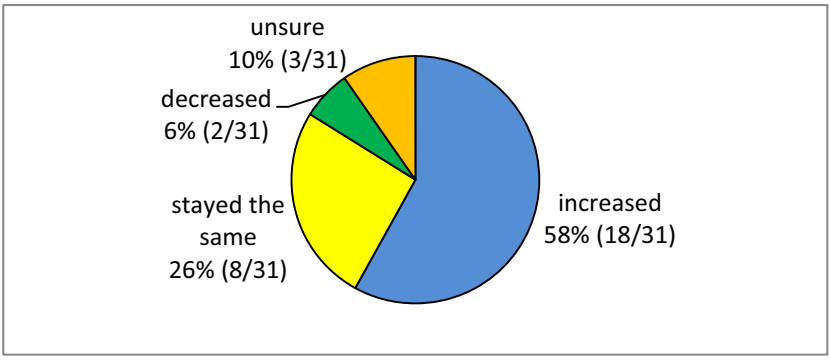


Fig. 1. Change in the Use of Formal Methods in the Last 5 Years

3.2 Barriers

Survey participants listed 120 items in response to the question "*What do you see as the current barriers to the industrial adoption of formal methods (especially in your organization)?*" The authors grouped these 120 responses into like statements (the barriers listed in this section) and then identified broad categories encompassing all of the barriers listed. These categories are: education, tools, industrial environment, engineering, certification, misconceptions, scalability, evidence of benefits, and cost. The Industrial Environment Category includes non-technical barriers with respect to personnel changes, contracts, and project schedules. The Engineering Category includes technical barriers to the use of formal methods that result from the manner in which projects are executed and how industrial problems are solved. Fig. 2 shows the number of responses for each category. Most interviewees listed more than one barrier, and many listed more than one barrier for some categories.

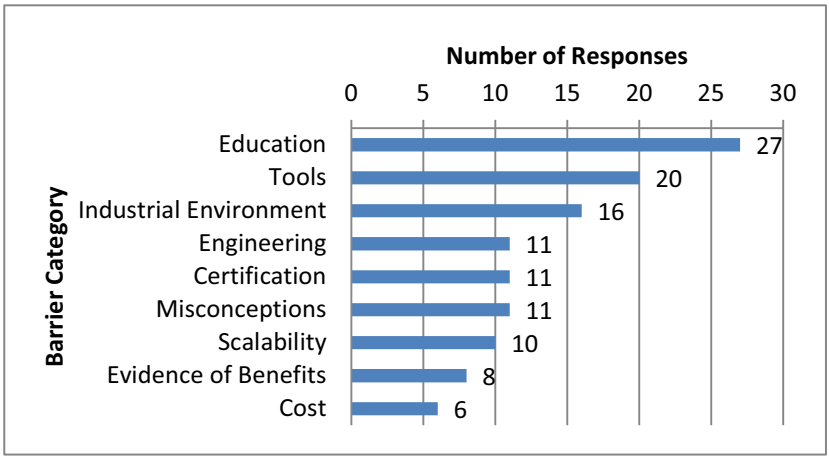


Fig. 2. Number of Responses for each Barrier Category

Next we will look at the specific barriers mentioned within each category. The barriers listed in the subsections that follow are the authors' restatements/groupings of the survey responses to the barriers question. While individuals may have multiple responses for a given category, care was taken so that each individual is counted at most once for a given barrier restatement.

Education Barriers. Education barriers include all barriers regarding education of the individuals involved in the design, development, management, or certification of products. A major theme is the need to train the current workforce. Specific issues mentioned are that users do not know how to properly apply formal methods or how to properly interpret the results. Respondents also expressed the need for formal methods experts. At least three of the organizations we spoke with have an expert group of formal methods practitioners that are responsible for developing and maintaining the formal analysis tools used within the organization. The education barriers are listed next with the number of respondents indicating each barrier in parentheses.

- General education on formal methods techniques and tools is needed, especially for the working engineer. (7)
- Highly trained formal methods experts are needed. (6)
- Users do not know how to properly apply formal analysis. (6)
- Certification authorities need to be educated on how to evaluate FM artifacts. (3)
- Certification authorities are not familiar with FM techniques or their benefits. (2)
- Formal methods advocates do not have sufficient appreciation for the practical issues that the product engineers face. (1)
- Lack of awareness of resources. (1)
- Users do not know how to properly interpret the results of formal analysis. (1)

Tools Barriers. Tools barriers include all barriers with respect to tools including, but not limited to, usability, capabilities, and integration. The majority of responses in this category are on the need for improved usability and integration of tools, rather than improved capabilities.

- Tools are not user-friendly. (5)
- Tools are distributed and not integrated. (4)
- Formal methods tools are not compatible with development tools. (3)
- Tools are not sufficiently automated. (3)
- Lack of support for real-time embedded systems. (2)
- Uncertainty if or how long it will take for formal analysis to complete. (2)
- Inconsistencies between the mathematics behind the model and the mathematics of the real world. (1)

Industrial Environment Barriers. Industrial environment barriers include non-technical barriers with respect to personnel changes, contracts, and schedules. While there is not one major theme in this category, we note that both the third barrier about export restrictions and the sixth barrier about people changing positions frequently would be alleviated somewhat by educating more of the workforce on formal methods.

- Some projects operate on too short of a timeline for formal analysis. (3)
- Contractually requiring the use of formal methods can be difficult. (2)
- US export control laws on technical data make it difficult to hire foreign nationals and/or collaborate internationally. (2)
- Psychological: some engineers do not like to formalize things. (2)
- The benefits of formal analysis are not reaped by those who do the analysis but rather by those downstream in the development process. (2)
- Transitory nature of people (people change positions frequently). (2)
- Proprietary methods and tools that cannot be shared. (1)
- Uncertainty regarding how FM will affect system modification/maintenance. (1)
- Uncertainty with respect to how to adapt formal methods to legacy systems. (1)

Engineering Barriers. Engineering barriers are technical barriers to the use of formal methods that result from the manner in which projects are executed and how industrial problems are solved. The most common problem cited in this category is that requirements are informal, changing, and sometimes wrong. On the one hand, formal methods can help formalize and catch inconsistencies with requirements. On the other hand, it may not be cost effective to formalize the requirements until the informal requirements have stabilized.

- Uncertain requirements (i.e., requirements that are informal, incomplete, changing, or simply wrong). (4)
- Formal analysis is not integrated into the existing development process. (2)
- Validating a model in a new domain is difficult. (2)
- Designs are not organized with formal methods in mind. (1)
- FM often require a high level of expertise with the system being analyzed. (1)
- Sometimes unclear how to show that assumptions of analysis are being met. (1)

Certification Barriers. Certification barriers include barriers specific to airworthiness or airborne systems certification authorities. The most common barrier mentioned in this category is the need for certification credit for formal methods. This will be an option under DO-178C [10]. Two other issues mentioned include a reluctance to change on the part of the certification authorities and the uncertainty of how to qualify formal methods tools. One respondent listed "It is unknown whether certification based on formal analysis will stand up in court" as a barrier. We note that it is also unknown whether certification *without* formal methods will stand up in court. Michael Holloway's fictional court case presents a tongue-in-cheek examination of the possibilities [11].

- No certification credit for formal methods. (4)
- Certification authorities are reluctant to change. (3)
- Tool qualification of formal methods tools is uncertain. (2)
- International certification authorities must agree on certification credit for FM. (1)
- Unknown whether certification based on formal analysis will stand up in court. (1)

Misconception Barriers. Misconception barriers include barriers that result from poor publicity, misunderstandings about formal analysis, and questions about the trustworthiness of formal analysis tools. The first two barriers in this category show a need for evidence of successful applications of formal methods and dissemination of that information. Recall that six respondents said there is a need for formal methods experts (listed under the Education Category), and here two respondents said there is a false perception that a formal methods expert is needed to do the work. The primary difference is that the barrier listed here refers to the *user* of formal methods tools rather than the developers and maintainers of such tools.

- There is skepticism about formal methods, sometimes due to past failures. (3)
- Too much emphasis on the theory rather than the application. (3)
- Concern that a given FM tool might have a bug in it, in which case we would be placing our trust in something unreliable. (2)
- False perception that FM expert is needed to do the work. (2)
- Misconception that formal methods will replace all testing. (1)

Scalability Barriers. Scalability barriers are barriers regarding the limits on size and/or types of problems that formal analysis techniques and tools can handle. About half of those that said we need a means to scale the approach referred to the need for composability so that one can work at the system level, employing different kinds of analysis and testing (*not* all formal) to handle the different parts of the system.

- Need a means to scale the approach. (7)
- Formal methods research challenges remain. (3)

Evidence-of-Benefits Barriers. Evidence-of-benefits barriers relate to a lack of evidence (or perhaps lack of awareness of the evidence) for the benefits of formal methods. Benefits can be with respect to the business case (especially savings in schedule or cost) or the improved quality of systems (i.e., fewer defects) when formal analysis is used. The number one barrier in this category is that decision makers do not see the advantage of formal analysis over testing.

- Decision makers do not see the advantage over testing. (7)
- Lack of evidence to support adoption decisions. (1)

Cost Barriers. Cost barriers include barriers with respect to the cost of doing formal analysis. Several projects report a savings in cost when using formal methods [8]. However, some types of analysis (e.g., theorem proving) are more expensive than others (e.g., static code analysis).

- Formal analysis can be expensive in actual cost or measured financial risk. (5)
- It is time-consuming to write robust properties. (1)

Non-barriers. In addition to the responses we collected on barriers to the adoption of formal methods, some individuals wanted to highlight items they did **not** see as barriers. These responses are listed next.

1. More than one person said that evidence on savings during the second and subsequent use of formal methods is not that important. The reason is that program managers are thinking about their current program and how decisions will impact cost and schedule for that program.
2. A former formal methods group manager emphasized that the mathematical sophistication of product engineers is **not** a barrier. This manager pointed out that the complexity of building safety-critical systems exceeds the complexity of using formal methods.
3. A department head said that skills/training is **not** a barrier. He said that if there is a good business case for using formal methods, then his department would hire and/or train people as needed. He is not concerned with the skill set required.

3.3 Mitigations

We received 76 responses to the question "*Do you have any suggestions for removing [the barriers you mentioned]*?" The authors grouped these responses into like statements and then identified broad categories encompassing all of the mitigations listed. These categories are: education, tool integration, evidence of benefits, tool capabilities, tool usability, requiring formal methods, and certification concerns. Fig. 3 shows the number of responses for each category. Note that most interviewees listed more than one mitigation, and many listed more than one mitigation for some categories.

Next we will look at the specific mitigations mentioned within each category. The mitigations listed in the subsections that follow are the authors' restatements/groupings of the survey responses to the mitigations question. While individuals may have multiple responses for a given category, care was taken so that each individual is counted at most once for a given mitigation restatement.

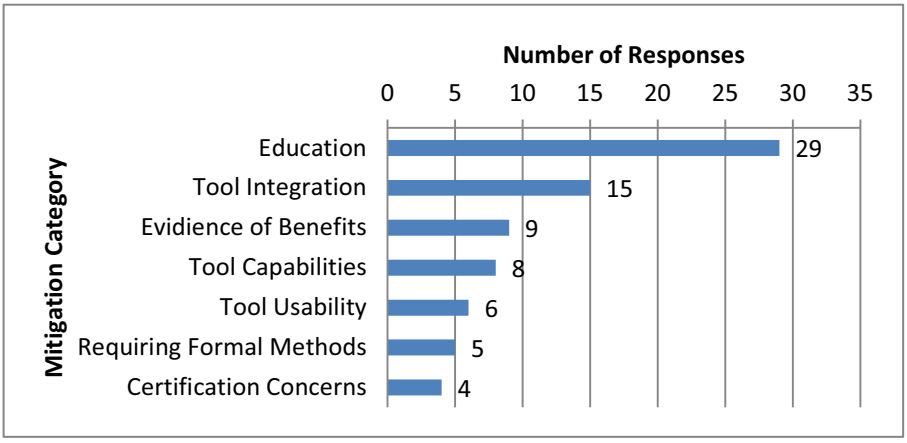


Fig. 3. Number of Responses for each Mitigation Category

Educational Mitigations. Educational mitigations are recommendations on how to improve education on formal methods. Two themes in this list of mitigations are that we need to include formal methods in undergraduate education and that we need training courses available for working engineers who will be using formal methods tools. The educational mitigations are listed next with the number of respondents indicating each mitigation in parentheses.

- Include formal methods in undergraduate education. (4)
- Caution that FM cannot solve everything; there is still a need for testing. (2)
- Get word out about the tools, expertise, and training available. (2)
- Attack the perception that a formal methods expert is needed to do the analysis. (1)
- Document how to demonstrate the analysis and how the assumptions are met. (1)
- Do not use the term "formal methods" in training classes for working engineers. Instead, show the engineers what the tools can do. (1)
- Education—need to convince people that doing this is a better way because it will help them downstream. (1)
- FAA-specific training on formal methods: one course at the familiarity level to provide top-level education on formal methods and another course at the expertise level to provide people the skill set necessary to evaluate a vendor proposing formal methods on a project. (1)
- Find more people interested in doing theorem-proving work. (1)
- Formal methods training at multiple levels: new users, users with some formal methods experience, and formal methods experts. (1)
- General education on FM for working engineers (e.g., via a university course). (1)
- Guidance on which pieces need formal analysis, which need semi-formal analysis, which do not need formal or semi-formal analysis, etc. (1)
- Include formal methods in an organization's systems engineering curriculum. Also, include training on using formal methods to gain certification credit under DO-333 in DO-178C training. (1)
- Need to get young people (starting in high school) interested in the notion of engineering safe systems. Make it attractive from an educational perspective. Make “engineering safe/secure systems” an engineering specialty one can choose in college. FM would come with the package and be part of the core curriculum. (1)
- Perhaps a professional society such as AIAA or IEEE could provide professional training in the use and application of one or more formal analysis techniques. (1)
- Required training for senior managers and developers of mission-critical SW. (1)
- Talk about the problems formal methods can and has solved in engineering newsletters (not just a sales pitch). (1)
- There is no way to remove the expertise barrier other than through education and getting people to use the tools (rewards in finding bugs). (1)
- Usability workshop. FM succeeding in Integrated Circuit community. (1)
- User education about when and where to apply formal methods. (1)
- We have to persuade people to use a different approach, especially those that have been working in their field for decades. Technical arguments are not sufficient to convince them. We need to show teams what the tools can do. (1)
- We need a way to explain FM that a jury can wrap their minds around. (1)

- We need FM related to domains and the interfaces that people are used to. (1)
- Work with teams on tool reviews to encourage the use of formal methods. (1)

Tool-Integration Mitigations. The mitigations listed in this section are recommendations for how to better integrate formal analysis tools, either with each other or with other development tools. A common theme among the mitigations listed is that we need formal methods tools to be integrated with existing tools. This includes system-level design tools, model-based design tools, and compilers.

- Automated process or guidance to constrain the way systems and software engineers do things, to enable the use of formal methods. (2)
- Coding standard with compliance checkers. (2)
- Bridge the gap to the dominant notation (UML, AADL, etc.). (1)
- Build static analysis into the development tool. (1)
- Develop open-source formal verification tools. (1)
- Emphasis on translation between tools. Then one can choose the right tool with the right strengths for the job. (1)
- Get FM early in chain, at the requirements level. (1)
- Integrate tools (e.g., compilers and formal methods tools). (1)
- Model checking the language people like to use in an automated fashion. (1)
- Model-Based Design (MBD) and simulation to address "getting the requirements right." (1)
- Need formal methods tools in sync with our development tools. (1)
- Strategy for what to do when FM tools fail. Combine analysis and testing into a global solution. (1)
- The opportunity for the most impact is for modeling tool vendors to embed FM. (1)

Evidence-of-Benefits Mitigations. Evidence-of-benefits mitigations include recommendations for the type of evidence that needs to be created and with whom it should be shared. Respondents cited a need for evidence of cost savings, time savings, and defects found on industrial-sized problems. The authors note that several published examples of formal methods applications and benefits do exist [8] [12].

- Insist that formal methods tools be applied to industrial-sized examples and disseminate those examples, including the cost and benefits data. (2)
- Demonstrating the value of FM and expanding the area where it can provide value. Show that higher quality SW can be produced in a more automated fashion. (1)
- Good, meaty, fully and publicly worked and documented formal methods examples beyond toy problems. (1)
- Highlight products that were fielded with defects that could have been caught with formal methods. (1)
- Need a direct measure / direct evidence of success when FM are used. (1)
- Need to convince program managers in DoD of savings of cost and time and that the quality is improved. (1)
- Solve practical problems with FM. (1)
- Use FM to debug and get systems to market faster rather than to get a proof. (1)

Tool-Capabilities Mitigations. The mitigations listed in this section are recommendations for what tool capabilities are needed. Responses vary from increased automation, composability to analyze system architectures, better support for continuous systems, and the ability to predict how much time it will take to do the analysis.

- Better tool support for continuous systems (e.g., more data types). (1)
- Continue to invest in research in these areas (FM tools and techniques). (1)
- Develop tools for composability to model and analyze system architectures. (1)
- We must provide sound abstractions automatically for data types we cannot handle (floating-point, etc.). There are function-based approximations (approximating the function) and data-based approximations (approximating the data types). (1)
- More automation, including automated test vector generation. (1)
- More robust tool (so analysis completes and/or makes it easier to make modifications to help analysis complete). (1)
- Predict how much time it will take to do the analysis. (1)
- Research on technical barriers such as floating point numbers and nonlinear mathematics. (1)

Tool-Usability Mitigations. The mitigations listed in this section are recommendations for how to improve the user-friendliness of formal methods tools.

- Better tool support for properties with respect to time. (1)
- Develop better tools to help people write their requirements more formally. Use a template-based approach with a built-in dictionary so that the requirements can be turned into logical-based expressions. (1)
- FM community needs to simplify tools and abstraction tools. FM theory (i.e., what the tools are doing) is very abstract. (1)
- Make theorem proving simpler or more amenable to complex systems. (1)
- Make tools easier to use. This is a difficult problem that will not be solved for a good while. (1)
- System-level tools and frameworks to help guide engineers on what needs to be done where. (1)

Requiring-Formal-Methods Mitigations. The mitigation mentioned here is simple: require the use of formal methods on new contracts. This implies a customer-driven decision to use formal analysis.

- Require the use of formal methods on new contracts. (4)
- Require and incentivize the use of FM. (1)

Certification-Concern Mitigations. The mitigations suggested here are giving credit toward certification for formal methods, and working an example to demonstrate how to certify a system with formal analysis.

- Certification authorities giving credit toward certification for the use of formal methods, or even requiring its use. (3)
- Need a small example (at the LRU level) to go through airworthiness certification to demonstrate how to certify a system with formal analysis. (1)

4 Discussion

4.1 Cross-Correlations

In this section we investigate results for particular subgroups. One of the most interesting subgroups consists of what we will here call novices, i.e., those who are familiar with formal methods but have not yet used them. Our survey included 9 novices, which is 29% of the surveyed population. They provided 42 responses to the barrier question. Novices tended to be more concerned about misconceptions and less about tools compared to the rest of the survey respondents. They accounted for 7 of the 11 misconception barrier responses. Novices were also concerned about education (10 responses) and the industrial environment (7 responses). The top three specific barriers listed by novices were:

- General education on formal methods techniques and tools is needed, especially for the working engineer.
- Need a means to scale the approach.
- Formal analysis can be expensive, either in actual cost or measured financial risk.

Our survey included 14 individuals who are considered formal methods experts. Experts almost unanimously (13 out of 14) reported a growth in the use of formal methods. Much of this growth is the result of increased internal and external funding of formal methods research and development. The percentages of barriers listed in each category by experts were about the same as the percentages for the entire group. In other words, no particular kind of barrier stood out more for this subgroup.

Another subgroup of interest consists of the 18 individuals who reported seeing an increase in the amount of use of formal methods. The percentages of listed barriers in each category for these individuals were about the same as the percentages for the entire group.

Five individuals reported "Tools are not user-friendly" as a barrier. Three of these individuals listed no other barriers, from which we can deduce that they consider this to be the most important barrier to widespread adoption of formal methods. Mitigations suggested by these five individuals included the following:

- Work with teams on tool reviews to encourage the use of formal methods.
- Coding standard with compliance checkers.
- Model checking the language people like to use in an automated fashion.
- Make tools easier to use. This is a difficult problem.

4.2 Comparison with Prior Work and New Insights

Our survey confirmed that several previously known barriers are still issues: tools are not user-friendly, the need for automation and scalability of tools, a lack of evidence to support adoption decisions, and skills deficiencies. Our respondents did not, however, consider the lack of evidence on the reduced cost for second and subsequent use of formal methods to be a significant barrier. The need for education on formal methods was the most frequently cited barrier by our participants, and this was not

emphasized in prior surveys. Our survey also found non-technical barriers regarding project timelines and personnel changes to be significant. Finally, we identified several barriers unique to the US aerospace domain, such as:

- No certification credit for formal methods. (4)
- Certification authorities are reluctant to change. (3)
- Certification authorities need to be educated on how to evaluate FM artifacts. (3)
- Certification authorities are not familiar with FM techniques or their benefits. (2)
- Tool qualification of formal methods tools is uncertain. (2)
- International certification authorities must agree on certification credit for FM. (1)
- Unknown whether certification based on formal analysis will stand up in court. (1)
- US export control laws on technical data make it difficult to hire foreign nationals and/or collaborate internationally. (2)

At the outset of our survey, we expected to see similar awareness of formal methods to what has been observed in the past. We expected to see modest use of formal methods by a few researchers in companies and government labs on most systems, and an increased use of formal methods on security applications. We were surprised to see increased awareness and use of formal methods in companies and government labs overall (even though it is far from mainstream). Regarding barriers, we expected commonality between the barriers in our domain and those of industry at large. We were surprised by the importance of buy-in from an organization's management chain. This drives the need for management to be somewhat knowledgeable of formal methods so that they will help push the adoption and use of this technology.

5 Summary

Education. Based on the large number (27) of barrier responses in the Education Category, it is clear that the need for education is a significant barrier to further adoption of formal methods. A major theme among survey responses is the need to train the current workforce. Also, decision makers need to know what formal analysis is and its benefits. Three levels of education need to be addressed: general awareness, users, and experts. Suggested strategies for addressing Education Barriers are:

1. Make formal methods a part of the standard software engineering curriculum, possibly within a course on "designing safety- and security-critical systems."
2. Develop and host courses on formal methods, designed for the working engineer at the user (i.e., non-expert) level.

Tools. Formal methods tools have come a long way in the last 5-10 years in terms of their performance and the complexity they can handle. Most research dollars continue to be invested in improving the scalability and the types of problems the tools can handle. However, significant issues remain that are not being funded: outdated user interfaces, lack of integration between formal methods tools, and lack of integration with other tools in the development process (e.g., compilers and tools for requirements capture). Improving tool integration and user interfaces is not that difficult

from a research point of view, but it is time-consuming and requires both formal methods and engineering expertise. Suggested strategies to impact Tool Barriers are:

1. Fund improvements to FM tool interfaces.
2. Fund the integration of FM tools with each and other development tools.

Customer/Executive Support. Many barriers remain with respect to the industrial environment, the way projects are currently executed, certification concerns, and the cost of formal methods. Most of these barriers can be overcome by a top-level decision to use formal methods. Strategies for encouraging the use of formal methods on future contracts include:

1. Customer requirements. If customers and/or certification authorities require the use of formal methods on programs, then formal methods will be used.
2. Credit toward certification. If conducting formal analysis yields credit toward certification and saves some measurable (i.e., in both schedule and cost) effort down the road, then contractor program managers are equipped to make that trade decision. Certification credit for formal analysis will be an option under DO-178C [10].
3. Creating and disseminating evidence of benefits. For years, formal methods advocates have shared the benefits of formal analysis to the quality of a system or product. There are even several published examples of its application to industrial-strength problems [8] [12]. However, it is very difficult to convince contractor program managers to use formal methods based on this evidence alone.

In summary, strategies to accelerate adoption of formal methods include making formal methods a part of the undergraduate software engineering curriculum, hosting courses in formal methods for working engineers, funding the integration of tools, funding improvements to tool interfaces, and promoting or requiring the use of formal methods on future contracts.

References

1. Hardin, D.S.: Design and Verification of Microprocessor Systems for High- Assurance Applications. Springer (2010)
2. Harrison, J.: Floating-Point Verification Using Theorem Proving. In: Bernardo, M., Cimatti, A. (eds.) SFM 2006. LNCS, vol. 3965, pp. 211–242. Springer, Heidelberg (2006)
3. Austin, S., Parkin, G.: Formal Methods: A survey, National Physical Laboratory, Teddington, Middlesex, UK (1993)
4. Craigen, D., Gerhart, S., Ralston, T.: An International Survey of Industrial Applications of Formal Methods (2 volumes), U.S. National Institute of Standards and Technology, Computer Systems Laboratory (1993)
5. Clarke, E.M., Wing, J.M.: Formal Methods: State of the Art and Future Directions. ACM Computing Surveys 28, 626–643 (1996)
6. Woodcock, J., Larsen, P.G., Bicarregui, J., Fitzgerald, J.: The Industrial Application of Formal Methods: an International Survey, `http://fmsurvey.org/` (accessed June 2012)

7. Bicarregui, J.C., Fitzgerald, J.S., Larsen, P.G., Woodcock, J.C.P.: Industrial Practice in Formal Methods: A Review. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 810–813. Springer, Heidelberg (2009)
8. Woodcock, J., Larsen, P.G., Bicarregui, J., Fitzgerald, J.: Formal Methods: Practice and experience. ACM Computing Surveys 41(4), 1–40 (2009)
9. Krieker, J., Tarlecki, A., Vardi, M.Y., Wilhelm, R.: Modeling, Analysis, and Verification - The Formal Methods Manifesto 2010. In: Dagstuhl Manifestos 1. Schloss Dagstuhl, Germany (2011)
10. 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)
11. Holloway, C.M.: Issues in Software Safety: Polly Ann Smith Co. v. Ned I. Ludd. In: Proceedings of the 20th International System Safety Conference, August 5-9. Denver, Colorado (2002)
12. Miller, S.P.: Lessons from Twenty Years of Industrial Formal Methods. In: Proceedings of HCSS (2012), http://cps-vo.org/node/3434

On the Effectiveness of Assertion-Based Verification in an Industrial Context*

Laurence Pierre[1], Fabrice Pancher[1],
Rodolphe Suescun[2], and Jérôme Quévremont[3]

[1] TIMA Lab. (CNRS-INPG-UJF), 46 Av. Félix Viallet, 38031 Grenoble, France
[2] Dolphin Integration, 39 Av. du Granier, 38242 Meylan, France
[3] Thales Communications & Security, 4 Av. des Louvresses, 92622 Gennevilliers Cedex, France

Abstract. Assertion-Based Verification is widely gaining acceptance. It makes use of assertions, which are formal expressions of the expected specification or requirements. Writing assertions concurrently with the design can bring significant benefits to both the design and verification processes for digital circuits. From the concrete perspective of an industrial development flow, inserting *synthesized assertion monitors* and associated debug infrastructures in an FPGA-based environment can improve the debugging phases in many application domains. This paper advocates this approach, through the presentation of the validation of an industrial HDLC controller IP using synthesizable property monitors, and draws conclusions from these experiments.

1 Introduction

Circuit design has become significantly more complex in deep submicron technologies with multiple processor cores, several types of memories, silicon IP blocks, synthesized logic, etc. being integrated onto the same chip with both in-SoC and system level communication protocols. The number of chips that are right on first pass is less than 40% and over 50% of the total development cycle of a new product is spent on validating the system's behavior after the first silicon becomes available [1]. Writing *assertions* concurrently with the design can bring significant benefits to both the design and verification processes for digital circuits. Assertions provide a way to drive the formal analysis of the design model. They help detect more functional bugs, earlier and closer to their original cause which leads to fewer bugs remaining undetected after production, shorter verification times and faster debugging [2,3,4,5].

Assertions can be evaluated by one or more techniques among simulation, emulation or formal verification. However formal verification techniques like model-checking [6] suffer from the well-known state explosion problem for circuits of the complexity of the one used as case study here, which should be decomposed into small pieces to be workable. As an alternative, though non-exhaustive, compiling the assertions as *synthesizable modules* can help to automate the debugging using both simulation and emulation, while taking benefit from the formal

* This work was partly supported by the French project SFINCS (ANR).

C. Pecheur and M. Dierkes (Eds.): FMICS 2013, LNCS 8187, pp. 78–93, 2013.

context provided by Assertion-Based Verification (ABV). Whether intended for design validation or for online embedded testing, a common technique can be applied: synthesize from the assertion a *property monitor* under the form of a RTL (Register-Transfer Level) sequential circuit, and interconnect the design and the monitor via the monitored variables. From the concrete perspective of an industrial development flow, integrated circuits are most often verified with a prior prototyping on an FPGA-based environment that can run test suites way faster than simulation. However this kind of prototyping offers far less internal observability and debug capabilities than simulation. Inserting *synthesized properties* and associated debug infrastructures can improve the debugging phases in many application domains.

But two crucial questions arise in that context. First, specifications are usually expressed in natural language and we may wonder if they can easily be disambiguated to be transformed into assertions in a formal language. Second, it is important to have an idea of the area and power consumption of the synthesized monitors. To the best of our knowledge, this is the first time such results are reported and detailed for a real-size industrial case study, which is here an HDLC controller IP proposed by Thales Communications & Security. We describe this return of experience, that points out the advantages of Assertion-Based Verification compared to classical simulation-based validation. We thus demonstrate the application of an original technique for the automatic generation of synthesizable monitors from PSL verification units [7,8] that has been implemented in industrial EDA (Electronic Design Automation) tools [9], thus providing a novel solution for ABV during simulation at RT Level and also after synthesis.

2 Effective ABV for PSL Assertions

2.1 Brief Overview of PSL

In the Assertion-Based Verification approach [10], a variety of logics and languages can be used. This study focuses on the PSL language [11]. Formulas of the FL (Foundation Language) class of the PSL Temporal layer essentially represent linear temporal logic (LTL, [6]). This class contains Boolean expressions, Sequential Extended Regular Expressions (SEREs) and temporal operators. The semantics of these formulas is defined with respect to execution traces.

SEREs represent subordinate behaviours, occurring in successive cycles in the trace. The concatenation operator (;) is used to build the concatenation $s_1; s_2$ of two SEREs. The consecutive repetition operator enables the construction of $s[*]$ or $s[+]$, the repeated consecutive concatenation of a sequence s (0, 1 or more times for $s[*]$, and 1 or more times for $s[+]$). The overlapping and non-overlapping *suffix implication* operators $\mapsto$ and $\Mapsto$ are used to specify that a sequence holds if some pre-requisite sequence holds.

One key basic temporal operator of the FL class is *strong until*, denoted until!. Roughly speaking, φ until! ψ holds iff there exists an evaluation point in the trace from which ψ holds, and φ holds until that point.

$$v \models [\varphi \text{ until! } \psi] \iff \exists k < |v| \; s.t. \; v^{k\cdot\cdot} \models \psi \; and \; \forall j < k, v^{j\cdot\cdot} \models \varphi$$

The next! operator means that the sub-property denoted by its operand should be verified from the next evaluation point of the execution trace.

$$v \models \text{next! } \varphi \iff |v| > 1 \; and \; v^{1\cdot\cdot} \models \varphi$$

The formula always φ means that φ must be verified on each evaluation point of the trace. The next_event$(b)(\varphi)$ operator requires the satisfaction of φ the next time a Boolean expression b is verified.

2.2 Hardware Monitors for PSL Assertions

Several tools can produce checkers from temporal assertions for runtime verification (e.g., MBAC [12], FoCs [13]). Here we use the compositional, formally proven, Horus technique for automatically building synthesizable, hardware assertion checkers from PSL properties [7,8].

The technique is based on a VHDL [14] library of primitive PSL operators and on an interconnection method for those operators: checkers are *built compositionally from elementary components* that correspond to the primitive PSL operators. A monitor for a property $\wp$ is built as a device that takes as inputs the reset, the synchronization clock, a signal *Start* that triggers the evaluation, and the signals of the *Design Under Verification* (DUV) that are operands of the PSL operators in $\wp$. Two output signals $Valid$ and *Pending* (this latter output is for strong operators) provide information about the levels of satisfaction of the property, according to the PSL definition (holds, holds strongly, pending, fails) [11]. The checkers (produced as VHDL RTL descriptions), once connected to the DUV, thus inform about the status of the assertions at runtime. Both the library and the interconnection method have been formally proven correct using the PVS theorem prover [15].

2.3 Integration in EDA Tools

This technology has been implemented in Dolphin's EDA tools [9] (SMASH simulator and SLED schematic editor), with FSM-based generation of sequences. Tools like *ModelSim* (Mentor Graphics) [16], *Incisive* platform (Cadence) [17], and *VCS* (Synopsys) [18] support SVA [19] and/or PSL assertions during simulation. Dolphin's solution enables the verification of assertions during simulation as well as the construction of hardware verification components.

Construction of Monitors. SLED SDG (Synthesizable Detector Generator) enables generating synthesizable verification units that can be embedded in a prototype or in the circuit itself. It accepts PSL files as inputs and produces VHDL or Verilog descriptions. To make integration easier, it makes it possible to generate monitors with a customized interface.

Simulation. Verification units defined in PSL files can also automatically be included in the environment such that the properties are verified during simulation. When simulation ends, the user gets a report containing the status of all properties (number of violations and unsatisfied requirements). When the debug tools are active, breakpoints can be associated with the PSL properties to pause simulation whenever a violation occurs. Properties can also be verified after simulation using generated waveform files; this feature is provided by ICD (the waveform viewer distributed with SMASH and SLED). The simulator makes it possible to change the way boolean expressions are sampled when evaluating PSL properties. The following choices are available:

- immediate sampling: when the property's clock toggles, PSL evaluation uses the current values of the signals used in the expression
- preponed sampling: when the property's clock toggle, PSL evaluation uses the values that the signals had at the beginning of the cycle (i.e., before all delta-cycles at the current time). This sampling scheme is similar to the one used by SVA [19].

In simple cases, both sampling schemes will generate the same results. However preponed sampling ensures that the results obtained during simulation do not depend on the evaluation order of delta cycles in the same cycle, thus providing analogous results before and after DUV synthesis.

3 Case Study: HDLC Controller

The HDLC (High-level Data Link Control) controller IP (Intellectual Property component) includes independent transmitter and receiver modules that are able to handle transmission and reception with a specific frame format, a synchronous serial protocol called HDLC (ISO/IEC 13239:2002). The IP performs the physical level of the HDLC protocol (serialization/de-serialization, CRC generation, transparency and abort generation/detection). In addition to the VHDL source code and testbenches, the IP has been provided with a "requirements" document written prior to its design. The requirements specifications of section 4 are extracted from this latter document, and the description in the current section comes from the user guide.

3.1 Structure of the IP

The overall system is composed of two parts, see Figure 1: the Transmitter (TX) and the Receiver (RX). The transmitter receives its input data from an external device. TxClk is the actual clock of the transmitter, however TxDataEn and TxDataEnOut (kind of derived clocks) give the input and output data rates. Similarly, RxClk is the actual clock of the receiver but RxDataEn and RxDataEnOut give the input and output data rates.

Fig. 1. HDLC controller external interfaces

3.2 Behavioural Description

Each frame is made of an Open Flag (the flag value is 0x7E), the Information (payload), the CRC, and the Closing Flag, that may be shared with open flag of the next frame. The CRC size (no checksum, 8, 16 or 32 bits) can be set using the signal TxCRCSel, only when the transmitter is disabled.

Activation, Deactivation. The transmitter is enabled or disabled through its external input TxEnable. When TxEnable is low, the transmitter is disabled: flags or partial flags are continuously sent out to the transmitter TxDout output. Once the receiver is enabled, it seeks a flag sequence. Once a flag is detected, it starts seeking data. The receiver is disabled when RxEnable is low. This action is immediate: when re-enabled the receiver will look for a flag again.

Abort Command. The input signal SendAbort is a command telling the transmitter to send an abort sequence (7 consecutive ones) to terminate the transmission of the current frame. The effect of this command is immediate.

When the receiver detects this abort sequence, it activates the AbortFound signal that stays activated until a flag is received from the transmitter. If the receiver gets consecutive ones just after being enabled (after a rising edge of RxEnable), this will not be considered as an abort sequence.

Nominal Behavior. When operating normally, the external processor puts valid data on the transmitter input bus TxData and signals data availability on the TxDataWr input. If enabled, the transmitter starts the transmission of a frame on TxDout as soon as valid data are present. The receiver transmits the data it receives to its external processor by its RxDout output bus, while signaling data availability by the RxDataAvail signal. It uses the signals StartOfFrame and EndOfFrame to indicate the beginning and the end of a frame respectively. StartOfFrame is a one RxDataEnOut period wide pulse indicating the value on the RxDout bus is the first data of the frame (it is only valid when RxDataAvail is high). EndOfFrame is a one RxDataEnOut period wide pulse indicating the value on the RxDout bus is the last data of the frame (also only valid when RxDataAvail is high). The concurrent value of the CRCError signal indicates if the frame was error-free.

Transparency. Transparency is the process of preventing the flag pattern from occurring in the payload or CRC fields. A '0' bit is inserted, by the transmitter, any time a sequence of five consecutive ones has occurred. The receiver must remove this bit to recover the originally sent data.

4 HDLC Requirements in PSL

We now present a selection of our experiments with the *functional* requirements of the HDLC controller (some basic data-oriented requirements have been ignored: frame format, data rate, etc.). The requirement document has been written prior to this study, and has not been designed to ease the formalization of the specification. Nevertheless, even if some of those requirements had to be clarified during the experiments reported here, all of them have been translatable into PSL formulas. Simulations have been performed using the original testbenches, with some minor improvements to achieve sufficient assertions coverage. Unless otherwise stated, no property violations have been found.

Since most requirements express constraints on the input/output behaviour, we have chosen to sample them (PSL statement *default clock*) using the data enable signals (TxDataEn, TxDataEnOut, RxDataEn, RxDataEnOut). However, an implicit hypothesis is that the data outputs will be buffered, such that they can be considered stable one TxClk or RxClk clock tick after being available. We will see that it may be necessary to consider this hypothesis when formalizing the requirements.

4.1 HDLC200. Transmitter: Sending Aborts

Requirement: *When SendAbort gets active, the transmitter shall abort the frame by transmitting at least seven consecutive '1' bits without inserted zeros. This action shall be immediate (i.e. it shall not wait for the current character to be completely shifted out). It shall be followed by flags before the transmission of new words.*

The PSL translation of this property is straightforward, at least seven ones followed by one or several flags should be sent on TxDout after a rising edge of SendAbort:

```
default clock: posedge(TXDATAENOUT);
property HDLC_200 :
   always({!SENDABORT && TXENABLE; SENDABORT && TXENABLE}
          |-> {TXDOUT[*7]; TXDOUT[*];
                {!TXDOUT; TXDOUT[*6]; !TXDOUT}[+]});
```

The simulations revealed violations of this assertion. This is due to a delta delay on the assignment of TxDout (sampled on TxDataEnOut rising edge). The presence of an intermediate signal adds this delay in the generation of TxDout, and causes the violation (TxDout receives the first '1' too late, one delta after TxDataEnOut).

However, since TxDout should actually been available one TxClk tick later (see the remark in preamble, which comes from a clarification by the designer), we have adapted the assertion accordingly and there is no more violation.

4.2 HDLC210. Receiver: Receiving Aborts

Requirement: *The reception of an abort sequence shall cause the receiver to ignore the rest of the frame and resume the reception after a flag is detected. The signal AbortFound shall be set high when the condition is detected.*

The right hand side of the implication is triggered when the abort sequence is recognized (seven consecutive ones). When this condition is met, the AbortFound signal must be set. However, the requirement lacks information, it does not mention explicitly when AbortFound should be set (e.g., before a limited number of cycles?). Since this signal should be followed by flags, and then by a new frame (in case of abort, the frame must be sent again), it was suggested to verify that AbortFound arrives before StartOfFrame. The requirement was rewritten as *the reception of an abort sequence shall cause the receiver to ignore the rest of the frame and resume the reception after a flag is detected. The signal AbortFound shall be set high before the arrival of a new frame*, and expressed as follows:

```
default clock: posedge(RXDATAEN);
property HDLC_210 :
   always( {RXENABLE; !RXDATA && RXENABLE; (RXDATA && RXENABLE)[*7]}
           |-> (ABORTFOUND == 1'b1 before! STARTOFFRAME));
```

4.3 HDLC260. Receiving All Ones after Start

Requirement: *The receiver shall be able to handle the mark-hold condition (contiguous '1' bits) just after having been enabled (i.e. after the receiver is enabled, if all it sees is one, it shall not detect an abort sequence).*

The left side of the property recognizes an activation of the receiver followed by the reception of at least seven ones. If this condition is met, the AbortFound signal must not be set. In other words, if AbortFound is set in the future, this should be the consequence of the next send abort command coming from the transmitter. Thus, AbortFound must stay equal to '0' until the presence of a new send abort

sequence “01111111” on the receiver input data line, which gives the following assertion:

```
default clock: posedge(RXDATAENOUT);
property HDLC_260:
   always
     ({!RXENABLE; (RXDATA && RXENABLE)[*7] } |->
       ((ABORTFOUND == 1'b0) until
          ((RXDATA == 1'b1) && (prev(RXDATA) == 1'b1)
            && (prev(RXDATA, 2) == 1'b1)
            && (prev(RxData, 3) == 1'b1)
            && (prev(RxData, 4) == 1'b1)
            && (prev(RxData, 5) == 1'b1)
            && (prev(RxData, 6) == 1'b1)
            && (prev(RxData, 7) == 1'b0))));
```

4.4 HDLC240. Transmitter Idle Mode

Requirement: *While disabled (idle mode), the transmitter shall send flags or partial flags (i.e. no more than six contiguous ones allowed).*

The transmitter is in idle mode can be interpreted as: the last bit of the last flag of the transmission has been sent out (TxLastBit set to ’1’) and no valid data is present on TxData (TxDataWr set to ’0’). When the transmitter is disabled, full flags “01111110” will be sent, optionally followed by a partial flag if TxEnable suddenly goes high while sending a flag, see Figure 2. Therefore, only flags must be read on the transmitter output data line. As soon as the transmitter becomes enabled, the generation of flags is immediately stopped: a partial flag is a flag interrupted by reactivation of the transmitter. We thus have the following formalization:

```
default clock: posedge(TXDATAENOUT);
property HDLC_240:
   always( {!TXLASTBIT && !TXDATAWR; TXLASTBIT && !TXDATAWR}
           |=> {{!TXDOUT && !TXENABLE; (TXDOUT && !TXENABLE)[*6];
                 !TXDOUT && !TXENABLE}[*];
                 { {TXENABLE} |
                   {!TXDOUT; (TXDOUT && !TXENABLE)[*0..6]; TXENABLE}}});
```

Fig. 2. Flags and partial flags

4.5 HDLC250. Transmitter Disabled before the End of a Frame

Requirement: *If disabled while sending a character, the transmitter shall finish sending that character, followed by CRC and close flag before coming back to idle mode.*

Four properties have been written here, since the CRC can have different sizes, according to the value of TxCrcSel. Thus, after the end of the current character transmission (between 0 and 8 cycles), we should find the CRC transmission: this is the difference between the four properties, as the number of clock cycles to elapse varies from 0 to 32. Finally, a flag must follow. We just give here the assertions for a CRC of size 8 (TxCrcSel is 1) and 16 (TxCrcSel is 2):

```
default clock: posedge(TXDATAENOUT);
property HDLC_250_1:
   always
      ( {TXENABLE; TXENABLE == 1'b0 &&
         TXCRCSEL == 2'b01 && ZEROINSAVAIL == 1'b1 }
      |-> next {[*0..8]; [*8]; !TXDOUT; TXDOUT[*6]; !TXDOUT});

property HDLC_250_2:
   always
      ( {TXENABLE; TXENABLE == 1'b0 &&
         TXCRCSEL == 2'b10 && ZEROINSAVAIL == 1'b1 }
      |-> next {[*0..8]; [*16]; !TXDOUT; TXDOUT[*6]; !TXDOUT});
```

The simulations revealed violations of this requirement, transmission does not always stop when disabling. This is explained as follows. In theory, the behaviours of signals TxEnable and TxData should be independent: TxData is connected to the IP that sends messages, and TxEnable could have any connection that may allow to disable the transmitter if needed. It means that TxEnable may be disabled while data still arrive on TxData, yet the transmitter should stop the transmission, which is not the case. In fact, this component has an implicit hypothesis (not mentioned in the documents): if TxEnable is disabled, no more input data should be sent to the IP. The behaviour is then as expected.

4.6 HDLC300. Transmitter: Ending a Frame

Requirement: *The transmitter shall activate the UnderRun signal and end the frame automatically if no new valid byte has been written in the input buffer on the seventh rising edge of the transmitter clock following the emission of the buffer empty signal.*

It means that, after the rising edge of BuffEmpty, if the signal TxDataWr is '0' for seven clock cycles, then the UnderRun signal shall be activated, see Figure 3.

```
default clock: posedge(TXDATAEN);
property HDLC_300 :
```

```
always({!BUFFEMPTY && TXENABLE;
        (BUFFEMPTY && !TXDATAWR && TXENABLE);
        (!TXDATAWR && TXENABLE)[*7]}
            |->  next(UNDERRUN) );
```

Fig. 3. UnderRun signal generation

In most cases the UnderRun signal appears as expected. Nevertheless, there are few cases where the rising edge of UnderRun is delayed by one clock cycle, thus leading to a violation of this assertion. According to the designers, the reason is that the specification is unclear: a latency for the arrival of the UnderRun signal can be allowed, but this feature was not explicitly stated in the specification.

4.7 HDLC310. Receiver: Start of Frame Detection

Requirement: *The receiver shall activate the signal StartOfFrame when it is outputting the first byte of a frame.*

This assertion uses two auxiliary sequences: FLAG allows the detection of a flag on RxData, and NOT_FLAG detects that this flag is followed by a non flag (on 8 bits):

```
default clock: posedge(RXDATAENOUT);
sequence FLAG (boolean x, y) :
    { !x && y; (x && y)[*6] ; !x && y};
sequence NOT_FLAG (boolean x, y):
    {{{ !x && y; (x && y)[*0..5];
           {!x && y; (x && y)[*0..5]}[+]} |
      { (x && y)[*1..5];
           {!x && y; (x && y)[*0..5]}[+]}} && [*8] };
property HDLC_310:
   always( {FLAG(RXDATA, RXENABLE);
            NOT_FLAG(RXDATA, RXENABLE)}
         |-> next_event(RXDATAAVAIL)(STARTOFFRAME));
```

4.8 HDLC320. Receiver: End of Frame Detection

Requirement: *The receiver shall activate the signal EndOfFrame when it is outputting the last byte of a frame.*

Since it is not easy to design SEREs that detect a non flag followed by a flag, the assertion detects the whole frame (flag, series of non flag, and the closing flag):

```
default clock: posedge(RXDATAENOUT);
sequence FLAG (boolean x, y) :
    { !x && y; (x && y)[*6] ; !x && y};
sequence NOT_FLAG (boolean x, y) :
    {{{ !x && y; (x && y)[*0..5];
        {!x && y; (x && y)[*0..5]}[+]}
      | { (x && y)[*1..5]; {!x && y;
          (x && y)[*0..5]}[+] }}  && {[*8] ; [*]} };
property HDLC_320:
    always
       ( { FLAG(RXDATA, RXENABLE);
           NOT_FLAG(RXDATA, RXENABLE) ;
           FLAG(RXDATA, RXENABLE)}
         |-> next_event(RXDATAAVAIL)(ENDOFFRAME));
```

This assertion has some violations when RxDataAvail lasts two clock cycles (instead of one) and is raised one cycle before EndOfFrame (whereas the specification says that EndOfFrame should last only one RxDataEnOut cycle). Simulations also revealed an incorrect synchronization between signals CRCError and EndOfFrame/RxDataAvail. No actual bug have been reported by the teams who have integrated the HDLC controller, as the unexpected behaviour is worked around by the including component.

4.9 HDLC160. Transmitter: Transparency

Requirement: *The transmitter shall examine the frame content between the open and close flags including the payload and CRC sequences and shall insert a '0' bit after all sequences of five consecutive '1' bits (up to, and including, the last five bits of the CRC).*

Two properties have been written for the subcomponent ZeroInsertion:

The first property, $p160\_a$ checks that when no '0' has to be inserted, the transmitter output data are equal to its input data. This should stay true from the moment valid data start arriving until the moment a '0' has to be inserted or the input data are no more valid.

The second property, $p160\_b$ states that, when five consecutive valid ones arrive on TxData, a '0' should be inserted on TxDout and ZeroStopTx should be set. Property $p160\_b\_1$ considers only the case of five ones, and property $p160\_b\_2$

corresponds to multiples of five (in that case, each time five ones have been recognized, an additional cycle is needed to take into account the fact that the CRC generator is stopped). It must be noted that the ones may not be really consecutive: we must only consider the ones that are valid (TxDataValid), provided that the component is not stopped by the next one (TxStop), and the end of the frame is not reached (EndOfFcs is false). To formalize $p160\_b\_2$, we make use of four auxiliary sequences : $START\_160$ starts the matching of the SERE (a last valid '0' is on TxData or a frame is ending), HOP_160 is used to ignore the data that do not have to be taken into account, REC_1_160 is used to recognize a valid one (we also have to consider that the ZeroInsertion component may have stored a '1' in BitReg to take this value into account if it arrived concurrently with TxStop), and $REC\_1\_END\_160$ recognizes the last valid '1', possibly on the end of the frame.

```
default clock : posedge(TXDATAEN);
property p160_a :
   always ({ !TXDATAVALID; TXDATAVALID } |=>
               ((TXDOUT == prev(TXDATA))
                   until ((ZEROSTOPTX) || !prev(TXDATAVALID))));
sequence START_160 :
   {(! TXDATA && ! TXSTOP && TXDATAVALID) || ENDOFFCS};
sequence HOP_160 :
   { ((!ENDOFFCS) && (BITSTORED == prev(BITSTORED))
        && (TXSTOP || (!TXDATAVALID)))[*] };
sequence REC_1_160 :
   { (! ENDOFFCS && ((TXDATA && ! TXSTOP && TXDATAVALID)
                || (BITSTORED && ! prev(BITSTORED) && BITREG)))};
sequence REC_1_END_160 :
   { TXDATA && ! TXSTOP && TXDATAVALID };
property p160_b_1 :
   always({START_160; {HOP_160; REC_1_160}[*4];
           HOP_160; REC_1_END_160 }
          |=> {ZEROSTOPTX;!TXDOUT} );
property p160_b_2 :
   always({START_160; {HOP_160; REC_1_160}[*4];
           HOP_160; REC_1_END_160;
           { HOP_160; REC_1_160; [*1];
             {HOP_160; REC_1_160}[*3];
             HOP_160; REC_1_END_160 }[+]
          } |=> {ZEROSTOPTX;!TXDOUT} );
```

Using a similar approach, we have also processed the corresponding requirement for the receiver: *The receiver shall examine the frame content between the open and close flags and shall discard any '0' bit which directly follows five consecutive '1' bits.*

5 Characteristics of the Monitors

5.1 Synthesis

RTL synthesis of the HDLC design and the RTL monitors generated from PSL was performed using Synopsys Design Compiler. The target cell library is Dolphin's SESAMEeHSvHD_TSMC_0.18um that is optimized for density and speed. A total of 16 properties have been synthesized. In order to make sure that all sampling signals of PSL properties are processed as proper clock signals, the HDLC and generated monitors have been synthesized as a single design.

Table 1. Results - Synthesis and practical benefit

Properties	Area overhead	Leakage power overhead	Detection of violations
Transmitter			
HDLC200	3.81 %	3.61 %	✘
HDLC240	3.40 %	3.27 %	
HDLC250(1)	14.70 %	13.55 %	✘
HDLC250(2)	16.35 %	14.96 %	✘
HDLC300	2.19 %	1.92 %	✘
HDLC160	7.21 %	6.30 %	
Receiver			
HDLC210	3.41 %	2.98 %	
HDLC260	3.58 %	3.02 %	
HDLC310	7.48 %	6.70 %	
HDLC320	20.80 %	19.28 %	✘

Table 1 gives details about area and leakage power overheads for the properties presented here. The synthesizer generated information about the area of the synthesized design. Due to the complexity of the selected properties (e.g., the ones of sections 4.7, 4.8 and 4.5), the total area overhead introduced by the monitors is approximately 80%. Static information about power consumption is also generated by Design Compiler. The leakage power overhead is approximately 72%, whereas dynamic power overhead should be less than 10% (note that dynamic power estimation is calculated from average activity of all signals in the circuit, and does correspond to a realistic use of the design)[1]. This table also summarizes the practical benefit of formalizing those requirements and generating the corresponding monitors: several assertion violations have been detected, mainly in the transmitter part.

A frequency of about 2GHz is obtainable for each monitor. The critical path of the HDLC design may be slightly impacted when connected to the monitors (due to their Boolean parts), but the frequency remains roughly the same.

[1] Other results are under NDA.

5.2 Gate Level Simulation and Power Consumption Analysis

In order to verify that the automatically generated monitors still behave correctly after synthesis, a purely functional simulation has been performed. This gate level simulation showed that the synthesized monitors have the same behavior as the RTL monitors, they still report the same violations using the same testbenches. To obtain more precise measures of power consumption, the gate-level design has been simulated with Dolphin's simulation-based power consumption estimation software. The results showed that the average power consumption of the monitors is 45 μW (as a comparison, the average power consumption of a microcontroller is about 10 mW).

6 Conclusions

From the users' viewpoint, two main conclusions can be drawn. Simulations using PSL assertions allow to *detect more bugs* than simulations without assertions. When all requirements of the HDLC module are synthesized as PSL modules, the silicon area and the power approximately double. In most cases, this might not be compatible with industrial requirements. However, to reduce the resources used by the synthesised monitors, one might reduce the set of properties to target the ones that are considered more critical. The *tradeoff between criticality and resource overhead* might be different for each application and for each design need: the monitors could be used in emulation or FPGA-based fast prototyping of ASIC and could even be embedded in critical circuits such as avionic or information security products, where the detection of abnormal conditions might be a concern.

This study has shown that when requirements are expressed in natural language, the corresponding PSL property might exhibit some inconsistencies. Two reasons can explain these inconsistencies:

- The requirement expressed in natural language is not precise enough. For instance, in HDLC250, an implicit hypothesis has not been properly expressed in the natural language requirement
- The requirement can leave some flexibility to the designer. For instance, in HDLC300, a latency is allowed on the UnderRun signal but it is not specified (it is neither specified that a latency is not allowed).

In an industrial context, we do not see as practical to specify all requirements straightforwardly in PSL as not all stakeholders will "speak" PSL. However, because of its benefits, PSL can be considered as a way to make *safer designs*. To bridge the gap between the "natural language" requirements and their translations into PSL, future work includes the development of an interactive environment to ease the requirements formalization. The design organisation might also need some adaptations. At this point, we can see several possibilities:

- Describe more precise requirements and/or express the flexibility level
- Make a distinction between "hard" requirements which do not leave any flexibility and "soft" requirements
- Target critical requirements only.

In the context of industrial applications, PSL and efficient synthesizable property checkers can improve the design quality:

- At the simulation-level, emulation-level and in FPGA-based prototypes, to track and detect more bugs more quickly
- As monitors embedded in the design, to detect abnormal environmental conditions that influence over the device expected behaviour, such as a tampering attempt on an information security device or the occurrence of a single event upset (SEU) on an avionic device exposed to higher altitude radiations. To that goal, we still have to work on optimizing the overall checkers area. Beyond selecting the relevant properties to be embedded and the logical and temporal relations between them, it will be necessary to develop solutions to "mutualize" possible common expressions. This optimisation approach is likely to take advantage of the semantics of the PSL operators.

References

1. Geuzebroek, J., Vermeulen, B.: Integration of Hardware Assertions in Systems-on-Chip. In: Proc. IEEE International Test Conference, ITC 2008 (2008)
2. Gupta, A.: Assertion-based Verification Turns the Corner. IEEE Design & Test of Computers 19 (2002)
3. Maliniak, D.: Assertion-Based Verification Smoothes The Road To IP Reuse (2002), http://www.elecdesign.com/Articles/ArticleID/2748/2748.html
4. Michelson, J., Haque, F.: Assertions Improve Productivity for All Development Phases. EETimes (2007), http://www.eetimes.com/design/eda-design/4018491/Assertions-Improve-Productivity-for-All-Development-Phases
5. Steffora Mutschler, A.: Avoiding Chip Melt. Chip Design (2012), http://chipdesignmag.com/lpd/blog/2012/03/08/avoiding-chip-melt/
6. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press (2000)
7. Morin-Allory, K., Borrione, D.: Proven correct monitors from PSL specifications. In: Proc. DATE 2006 (2006)
8. Borrione, D., Morin-Allory, K., Oddos, Y.: Property-Based Dynamic Verification and Test. In: Design Technology for Heterogeneous Embedded Systems. Springer (2012)
9. Dolphin: Web page, http://www.dolphin.fr/medal/applications/applications_Assertion_based_verification.php
10. Foster, H.: Applied Assertion-Based Verification: An Industry Perspective. Foundations and Trends in Electronic Design Automation 3 (2009)
11. IEEE Std 1850-2005, IEEE Standard for Property Specification Language (PSL). IEEE (2005)
12. Boulé, M., Zilic, Z.: Generating Hardware Assertion Checkers: For Hardware Verification, Emulation, Post-Fabrication Debugging and On-Line Monitoring. Springer (2008)
13. http://www.haifa.il.ibm.com/projects/verification/focs
14. IEEE Std 1076-2002, IEEE Standard VHDL Language Reference Manual (2002)

15. Shankar, N., Owre, S., Rushby, J., Stringer-Calvert, D.: PVS Prover Guide. Computer Science Laboratory, SRI International (2001)
16. http://www.model.com/
17. http://www.cadence.com/products/ld/formal_verifier/pages/default.aspx
18. http://www.synopsys.com/tools/verification/functionalverification/pages/vcs.aspx
19. IEEE Std 1800-2005, IEEE Standard for System Verilog: Unified Hardware Design, Specification and Verification Language. IEEE (2005)

Complex Digital System Design: A Methodology and Its Application to Medical Implants

Helene Leroux, Karen Godary-Dejean, and David Andreu

INRIA DEMAR Project
LIRMM Montpellier, France 34193
{leroux,godary,andreu}@lirmm.fr

Abstract. In the context of specification of complex digital systems and their implementation on FPGA, a tool-based methodology is developed using a component-based approach. The component's behavior is described by means of Interpreted Prioritized Time Petri nets which are formalized in this article. Formal analysis is used to validate the model's properties and to optimize its implementation. Our approach is illustrated on the micro machine of a distributed stimulation unit.

Keywords: Interpreted Prioritized Time Petri nets, implementation, formal analysis, FPGA.

1 Introduction

Functional electrical stimulation (FES) has been successfully used in a growing set of medical applications (pacemakers, pain management, movement rehabilitation)[1]. Implanted FES relies on implantable active medical devices (IAMD) that stimulate nerves or muscles. We focus on the design and prototyping of the digital part of an IAMD which is implemented on a programmable logical device: a Field-Programmable Gate Array (FPGA). FPGA is a technology of increasing interest, on which complex digital systems are implemented. They are known for their flexibility and their simplified design while offering important dynamical partial reconfigurability perspectives. Recent improvements already allow FPGA to meet power, area, cost, and time to market requirements of embedded digital electronics. Low operating voltage, logic density increase, nonvolatile flash technology, etc. made FPGA a competitive solution for embedded devices, even though it still suffers from limitations compared to ASIC (lower unit costs, power, higher clock speeds). Often used as an intermediary step to prototype ASIC designs, they are now the targets on which control applications are built since they favor designing powerful dedicated architectures, for example in aerospace applications where their radiation-tolerance and their reconfigurability are of significant interest [2]. The programming of FPGA is usually specified using VHDL[3].

As IAMD are critical and complex systems, it is necessary to offer to designers, mainly electronics engineers, an efficient tool-based methodology. Hence the HILECOP (HIgh LEvel hardware COmponent Programming) methodology

C. Pecheur and M. Dierkes (Eds.): FMICS 2013, LNCS 8187, pp. 94–107, 2013.

was developed [4]. It is based on a component approach in which components' behavior and their composition are specified by Interpreted Time Petri nets [5]. It provides designers with a way to decompose the complex digital architecture thanks to components but also a way to formally specify the behavior of the system in order to be able to analyze it. Indeed, the formal analysis provides guarantees on the behavior in an exhaustive way that is not possible in test nor in simulation where sets of input values and scenarios are necessarily restricted. This tool-based methodology also integrates an automatic VHDL code generation to create, by model transformation, a code that is equivalent to the model[5].The behavioral equivalence between the analyzed HILECOP-Component based model and its implementation on FPGA must be ensured.

As IAMD are implanted in a human body, we have to be efficient when implementing the model on a FPGA in terms of number of cells (size) and power consumption. Indeed, to be implantable a device must have a limited size and use as less energy as possible. This article aims to improve the HILECOP methodology using a formal analysis; not only to validate its behavior but also to increase reliability and efficiency. The goal is to minimize the possibility of human errors and to have an implementation as compact as possible. Reaching an optimal circuit's size for a FPGA is not easy; many parameters have to be taken into account. Circuit size depends also on how the VHDL code is compiled, placed, and routed by the software used to implement the code on the FPGA. This concern is taken into account within our methodology at the code generation phase, allowing us to define different ways to carry the model-to-text transformation. For example: different ways to describe the VHDL model nodes and their interconnections are envisaged. Also different working modes of the resulting digital architecture (from basic synchronous to clock gating approaches) are allowed. This won't be detailed; rather we focus on exploiting formalism and existing analysis approaches to size the implementation from the abstract model.

We will first present basis of the HILECOP methodology where the formalism of the Interpreted Prioritized Timed Petri nets is introduced. Then section 2 explains how the analysis of the model is used to adequatly size the implementation. Finally, the modeling of an industrial case: the digital architecture of an IAMD will be exposed and discussed.

2 The HILECOP Methodology

Our methodology relies on a component-based approach, allowing us to take advantage of such approaches like modularity, reusability, and upgradability. These components are referred to as HILECOP-Component (HC). Their behavior is described thanks to a Petri net (PN)[6]. PN are used because of their easily understood graphical representation - which allows parallelism, synchronization, and resource sharing to be easily expressed - in addition to their well-formed mathematical formalism from which structural and behavioral analysis can be performed [7]. Both explicit representation of parallelism and formal analysis are important for electronic engineers creating such strongly parallel digital

architecture involving multiple components. The class of PN used to specify digital architectures, is Interpreted Prioritized Time Petri nets (IPrTPN).

2.1 Formal Definition of IPrTPN

The formalism used to describe the behavior of an HC is a generalized IPrTPN which includes weighted test and inhibitor arcs. IPrTPN extends Prioritized Time Petri Nets with interpretation, which enables the model to interact with inputs and outputs of the system. Our formalism is largely based on that of [8] and [10], the latter adding test and inhibitor arcs to PrTPN. Our resulting formalism of IPrTPN is then (notations not detailed here can be found in [8]):

Definition 1. *Let I^+ be the set of non empty real intervals with non negative rational endpoints. For $i \in I^+$, $\uparrow i$ is its right end-point or ∞ if i is unbounded. Let $\mathbb{B}^n$ be the set of boolean expressions where $\mathbb{B} = \{0, 1\}$. Let $\mathcal{F}$ be the set of functions described in VHDL. Let $\mathcal{A}$ be the set of actions described in VHDL. $\mathcal{A}$ and $\mathcal{F}$ interacts with external signals (inputs and outputs of the system) or manipulates internal variables of the system. An IPrTPN is a tuple $< P, T, Pre, Pre_t, Pre_i, Post, m_0, I_s, \succ, C, F, A >$, in which :*

- $< P, T, Pre, Pre_t, Pre_i, Post, m_0 >$ *is a Petri net. P is the set of places, T is the set of transitions, m_0 is the initial marking. $Pre, Pre_t, Pre_i, Post : T \to P \to \mathbb{N}$ are the precondition function, the test function, the inhibition function and the postcondition function, respectively.*
- $I_s : T \to I^+$ *is the static interval function.*
- $C : T \to \mathbb{B}^n$ *is the condition function.*
- $F : T \to \mathcal{F}$ *is the impulsive action function.*
- $A : P \to \mathcal{A}$ *is the continuous action function.*
- $\succ$ *is the priority relation, assumed irreflexive, asymmetric and transitive.*

A marking is a function $m : P \to \mathbb{N}$. A transition $t \in T$ is said to be sensibilized by m iff $(m \geq Pre(t) + Pre_t(t)) \wedge (m < Pre_i(t))$. It is noted $t \in sensibilized(m)$. A transition $k \in T$ is said to be newly sensibilized by the firing of the transition t from the marking m, and it is noted by $\uparrow sensibilized(k, m, t)$, iff $k \neq t$ is sensibilized by the new marking $m - Pre(t) + Post(t)$ but was not by $m - Pre(t)$ or $k = t$ and t is still sensibilized by the new marking. By extension, we will denote by $\uparrow sensibilized(m, t)$ the set of transitions newly sensibilized firing the transition t from the marking m. The marking $m - Pre(t)$ is considered because the tokens in Pre(t) are consumed by the firing of t. Formally:

$$\begin{aligned}\uparrow sensibilized(k, m, t) = &\big(m - Pre(t) + Post(t)\big) \geq Pre(k) + Pre_t(k) \\ &\wedge \big(m - Pre(t) + Post(t) < Pre_i(k)\big) \wedge \big[\big(Pre_i(k) \leq m - Pre(t)\big) \\ &\vee (k = t) \vee \big(Pre(k) + Pre_t(k) > m - Pre(t)\big)\big]\end{aligned}$$

A state of an IPrTPN is a pair $s = (m, I)$ in which m is a marking and I is a function called the time-interval function. Function $I : T \to I^+$ associates a time-interval with every transition enabled at m.

2.2 Semantics of IPrTPN

In IPrTPN, a transition $t \in T$ is fireable from (m, I) if : $t \in sensibilized(m) \wedge C(t) \equiv 1 \wedge 0 \in I(t) \wedge ((\forall k \in T)(k \succ t) \Rightarrow (k \notin sensibilized(m) \vee C(k) = 0 \vee 0 \notin I(t)))$. It is denoted $fireable(m)$.

Two semantics are usually used as far as transition firing is concerned in time PN. The strong semantics defines that, if a transition is fireable, it has to be fired before the upper bound of its time interval is reached. The weak semantics defines that a fireable transition does not have to be fired. If the limit of the time interval is overtaken, the transition cannot be fired anymore unless it is unsensibilized and sensibilized again. The strong one is the most used for the description of real-time systems as it can model the emergency of some events. It is also the one used in analysis approaches and tools. However our model carries time and interpretation. It is then impossible to guaranty a strong semantics. Indeed, it is not possible to ensure that the condition is true when the maximum of the time interval is reached. Hence we define that if it is possible to fire a given transition in its associated time interval, the transition has to be fired; if its not, the transition cannot be fired until it is sensibilized again. So we define a new semantics, taking into account [8] and [10] contributions and hypothesis.

Definition 2. *The semantics of an IPrTPN* $< P, T, Pre, Pre_t, Pre_i, Post, m_0, I_s, \succ, C, F, A >$ *is the timed transition system* $< S, s_0, \rightsquigarrow >$ *where:*

- S *is the set of states* (m, I) *of the IPrTPN.*
- $s_0 = (m_0, I_0)$ *is the initial state, where* m_0 *is the initial marking and* I_0 *is the static interval function* I_s *restricted to the transitions enabled at* m_0*.*
- $\rightsquigarrow \subseteq S \times (T \cup \mathbb{R}^+) \times S$ *is the state transition relation, defined as follows* ($(s, a, s') \in \rightsquigarrow$ *is written* $s \overset{a}{\rightsquigarrow} s'$)*:*
  - *Discrete transitions: we have* $(m, I) \overset{t}{\rightsquigarrow} (m', I')$ *iff* $t \in T$ *and:*
    1. $t \in fireable(m)$
    2. $m' = m - Pre(t) + Post(t)$
    3. $(\forall t' \in T)\ t' \in sensibilized(m) \wedge C(t') = 1 \wedge (t' \succ t) \Rightarrow 0 \notin I(t')$
    4. $(\forall k \in T)$,*if* $\uparrow sensibilized(m), I'(k) = I_s(k)$, *otherwise* $I'(k) = I(k)$.
  - *Continuous transitions: we have* $(m, I) \overset{\theta}{\rightsquigarrow} (m, I')$ *iff* $\theta \in \mathbb{R}^+$ *and:*
    1. $(\forall t \in T)\ I(t) \neq \varnothing \wedge t \in sensibilized(m) \Rightarrow \theta \leq \uparrow I(t)$
    2. $(\forall t \in T)\ I(t) \neq \varnothing \wedge t \in sensibilized(m) \Rightarrow I'(t) = I(t) \dot{-} \theta$
    3. $(\forall t \in T)\ I(t) = \varnothing \Rightarrow I'(t) = I(t)$
  - *Blocking transitions: we have* $(m, I) \overset{t}{\rightsquigarrow} (m, I')$ *iff* $(t \in T)$,
    1. $t \in sensibilized(m) \wedge (C(t) = 0) \wedge (\uparrow I(t) = 0) \Rightarrow I'(t) = \varnothing$
    2. $(\forall k \in T \backslash t)\ I'(k) = I(k)$

This formalism being defined, the component model will be briefly described.

2.3 HILECOP-Components

A component is an instance of a component descriptor which is executed on an execution target (FPGA here). To simplify the description, the word 'component' is used for both the descriptor and its instance. A component (fig 1) is described by its behavior and its interface allowing a structural view, within the meaning of digital architecture. The behavior is defined by IPrTPN; it represents the state evolution of the component and the manipulation of the variables and signals linked to the interpretation (carried by the IPrTPN). The component interface is a set of ports allowing to connect components to each other. Two port types are considered: nodes of the behavior and digital signals. Then the digital architecture is build connecting components through the ports of their interfaces.

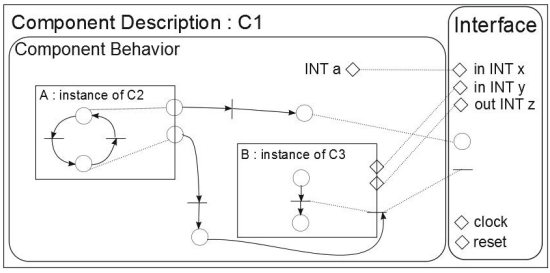


Fig. 1. A HILECOP component

The aggregation of components consists of encapsulating them inside a higher level component: this creates composites (components containing components) which reveal a hierarchical structure. In the case of a composite, the ports of a component can also be connected to the elements of the higher level component. Similarly, a composite can export in its interface ports coming from the interface of components it includes. Since interfaces of components can export nodes of their behavior, interactions between them can be specified using the same formalism (i.e. IPrTPN), using merging operators if needed. The complete architecture is thus defined by the assembly of a set of given components. Components and architectures can be seen and manipulated at different level of detail. The designer can describe his architecture either by interconnecting existing components, or by creating new components, or by mixing both approaches.

2.4 Automatic Generation of the VHDL Code

Implementing such complex models on an FPGA requires automatic translation of the Hilecop model into its equivalent VHDL code. This model-to-code transformation is not explained here in detail [5] but key points are given. In order to let the designer as much model-to-code translation choices as possible to obtain the 'best' implementation of his model, different options are possible for the automatic generation of the VHDL code. In every case, the HC-based model is first

transformed in order to solve the components' assembly, in particular to obtain the resulting global IPrTPN model.

By definition, a PN is an asynchronous model [15]. But, as an IPrTPN can contain functions, it is very difficult to implement it on an FPGA asynchronously. Indeed there is no way to know when the execution of a function is finished (i.e. when signals are stable) as a VHDL code is not executed sequentially but in a combinatorial way. Therefore an asynchronous implementation is possible but very intricate. That's why we choose to implement IPrTPN synchronously: the synchronous implementation ensures signal stability when manipulated, assuming of course an adequate clock . Yet, it can introduce some problems in case of effective conflicts in the PN (see §3.2).

The PN is composed of three main elements: place, transition and arc. Two VHDL components, referred to as basic-components, are used for the model-to-code transformation: the place-component and the transition-component. Arcs are described in one of these components depending on the designer choices. In case of 'Place-pivot' (respectively 'Transition-pivot') arcs are embedded in the place (transition) basic-component. According to this choice, the VHDL codes and the behavior of the basic-components are slightly different; . In both cases, the place-component has to store its marking and to update it on the rising edge of the clock: $\uparrow$ *clock*(phase 4) and the transition-component has to decide if it can be fired or not on the falling edge of the clock:$\downarrow$ *clock* (phase 2) (fig. 2).

In the case of 'Place-pivot', every place $p \in Pre(t) \cup Pre_t(t) \cup Pre_i(t)$ sends to the transition t a signal which indicates if $m(p)$ sensibilizes t or not (phase 1), taking into account type and weight of the given arcs. If t is fireable(phase 2), it sends to every place $p \in Pre(t) \cup Pre_t(t) \cup Pre_i(t) \cup Post(t)$ a signal which indicates that it fires (phase 3). Then each place updates its marking (phase 4) and can now send updated signals of sensibilization to transitions. In the case of 'Transition-pivot', every place $p \in Pre(t) \cup Pre_t(t) \cup Pre_i(t)$ sends a signal giving its marking to t (phase 1). t determines if it is sensibilized by this marking and if it is fireable. If it can fire, it fires (phase 2) and hence t sends to every place $p \in Pre(t) \cup Pre_t(t) \cup Pre_i(t) \cup Post(t)$ the number of tokens they have to remove or add to their markings (phase 3). Then the places update their

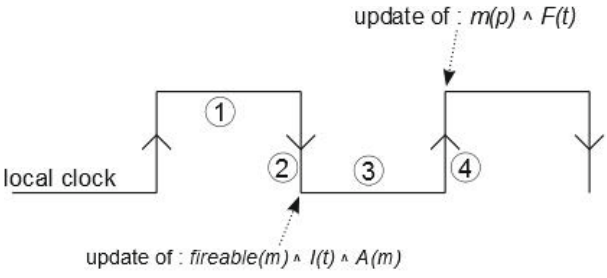


Fig. 2. Synchronous execution of an IPrTPN

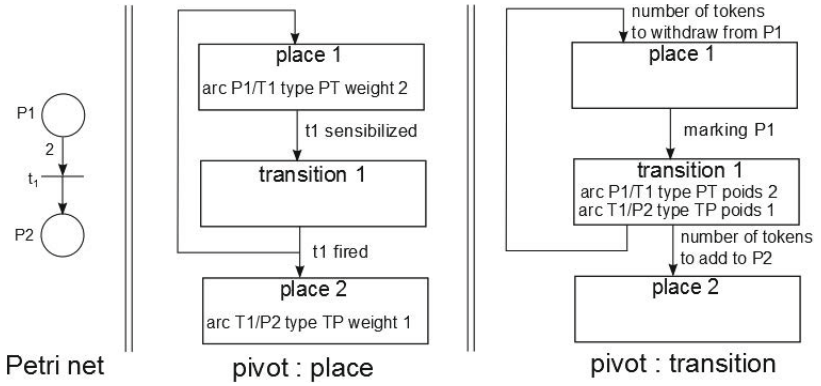


Fig. 3. Place pivot vs Transition pivot implementation

markings (phase 4) and can send an updated marking to t and so on. Hence the firing of a transition takes exactly one clock period.

The basic-components are instantiated and linked to each other in order to create the VHDL code corresponding to one or several HC, depending on the component hierarchy; this VHDL code is referred to as a macro-component. Our approach preserves a component-based approach from high to low level, which has several advantages. First, there is no constraint on the PN structure. Also, the model transformation (model-to-code transformation) can be optimized according to the types of places and transitions of the given PN. For instance, if a PN has time transitions and transitions without time, two different transition basic-components can be used: one dealing with time constraints, the other not. This prevents using a counter for a transition without time and hence allows reducing the circuit's size. The same can be done for binary vs. generalized places. Moreover, changes can be made on the behavior of the basic-component without any impact on the macro-component, to deal with power consumption and surface (number of cells). Indeed, it is possible for instance to define different kinds of operation (working modes); from simple synchronous operation to control of basic-component activity (i.e. enabling the activity of instances) according to tokens propagation within the PN[5]. Finally, the designer can choose to keep the hierarchy of HC assembling or to "flatten" the model. Keeping the hierarchy is interesting, for example, to specify GALS (globally asynchronous locally synchronous) complex digital architectures [11].

3 Use of Analysis in HILECOP Methodology

The figure 4 explicits how analysis is used to validate and optimize the VHDL code generation. Models are represented by ellipses and 'tools' by rectangles. 'Tools' concerning analysis will be further explained in detail in this section.

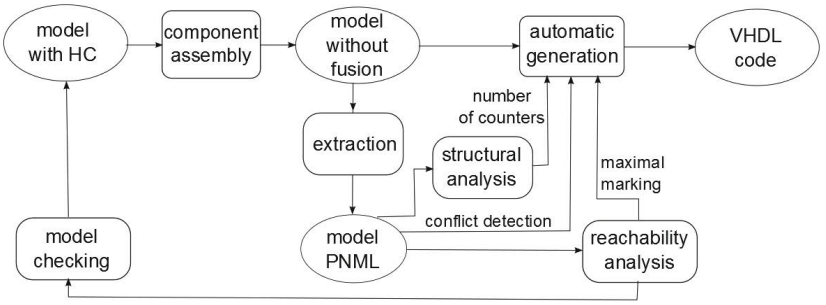


Fig. 4. Use of analysis to obtain an optimized VHDL code

3.1 Formal Verification of System's Properties

We developed a specific model-to-text transformation to get the description of the flattened TPN written in PNML [12] from the initial HILECOP model. This output description is used as an input of existing PN analysis tools. It contains all time information, i.e. time-interval associated to transitions and time impact of the synchronous implementation. This TPN description only differs from the initial IPrTPN regarding: associated interpretation since analysis tools do not handle it and associated priorities. On the IPrTPN model priorities specifies how to deterministically deal with effective conflicts. When conditions are not accounted for, priorities can cause sections of the PN to be considered dead. This because, when considering all conflicts the transitions with the highest priority will always be fired. Consequently, the PNML description we generate aims at guarantying that the corresponding TPN is as behaviorally equivalent as possible to the implemented model and that the possible behaviors of the implemented IPrTPN are all included in those of this TPN. For example, for transitions without time-interval in the IPrTPN model we distinguish two cases: 1/ if there is no condition associated to the given transition, this transition will always be fired in one time unit in the implemented model (fig. 2), hence a time interval [1,1] is associated to this transition in the equivalent TPN, and 2/ if there is a condition associated to the given transition, this transition cannot fire in less than one unit of time but could also never be fired, and so a time interval $[1, \infty[$ is associated to this transition in the equivalent TPN.

One of the existing analysis tool we are using is TINA [9]. It allows verifying properties like semi-flows from structural analysis, bounded, live, and reversible usual properties but also LTL properties from reachability analysis. Hence it is possible to verify that the behavior will be executed correctly or to verify that an unwanted situation will never occur. Model-checking has also been successfully used to verify the behavior of the protocol interpreter of a DSU [13].

3.2 Handling Effective Conflicts

Definition 3. *Let N be an IPrTPN. Two transitions* $(t, k) \in T$ *are in a structural conflict iff* $(Pre(t) \cup Pre_t(t)) \cap (Pre(k) \cup Pre_t(k)) \neq \varnothing$.

Definition 4. *Let N be an IPrTPN. Two transitions* $(t, k) \in T$ *are in an effective conflict for a marking m iff t and k are in a structural conflict and* $(t, k) \in fireable(m)^2$ *and* $m < Pre(t) + Pre_t(t) + Pre(k) + Pre_t(k)$.

In the case of such an effective conflict, there can be problems in a synchronous implementation. Indeed, only one of the conflicting transitions must be fired at a time in order to be consistent with the PN behavior. This problem can be solved by asking the designer to ensure conflict resolution using mutually exclusive conditions (as done in Sequential Function Chart), formally saying: $\forall(t, k)$ in a structural conflict, $C(t) \wedge C(k) = 0$. However this does not prevent human error. Our solution is to automatically detect every possible conflict and to impose a priority between conflicting transitions for every possible choice. Thus priorities are automatically set if not defined by the designer.

Automatically detect effective conflicts is difficult since there is no existing way to analyze IPrTPN (due to the interpretation). Hence all structural conflicts are searched since the set of effective conflicts is necessarily included into the set of structural conflicts. The PNML description is parsed to detect them, allowing isolating the conflicting transitions. Several transitions belonging to a set of structural conflicts can be simultaneously fired in the implemented model as soon as they are not in effective conflict, i.e. dealing with sets of structural conflicts rather than sets of effective conflicts does not avoid allowed state evolutions.

Implementation of priority-based conflict solving must be efficient, both in terms of FPGA cells and execution time (i.e. without requiring an higher clock frequency to execute the model). Two solutions have been considered in the HILECOP methodology: the first one is to create a new type of component which is a group of transitions in conflicts, the second one is to create a VHDL process which deals with conflicts between possible transition firings and decides the effective firings. The best solution considering both mentioned criteria is the latter, since it only requires adding a combinatorial part. Only the 'Place-pivot' case is explained here. The transition component is modified: it now indicates if the transition is fireable but does not directly fire if the transition is implied in a conflict (fig. 5). In this case, a combinatorial treatment of these $fireable$ signals allows to determine which transitions must be fired depending on defined priorities. Outputs of this treatment are $fire$ signals, one for each transition in conflict. The signals $fire$ are given to places (P1,P2,P3) to update their markings but also to time transitions (T2) which have to reset their counter when fired. These signals are also used to start functions associated with transitions.

The main benefit of this approach is that there is no more risk of human error (on this point) as every possible conflict is automatically detected and handled. Besides, no significant delay is introduced in the execution of the model: one step of evolution on the model still requires only one clock period. Yet, if firing two transitions in conflicts is forbidden only for effective conflicts, the combinatorial

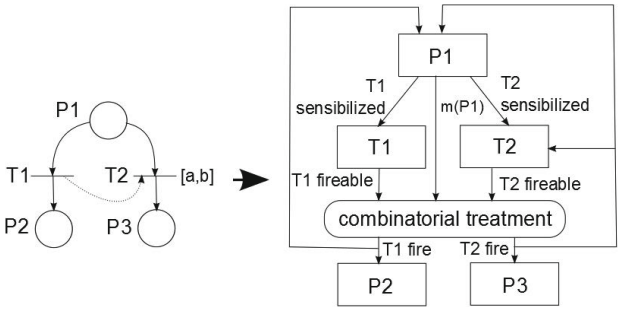


Fig. 5. Illustration of how a conflict is handled

treatment is implemented for every structural conflict, thus inducing an often small overhead in terms of circuit's size. However, depending on the model size, the overhead can be significant if the considered PN contains a high number of conflicts. Note that, as interpreted PN cannot be analyzed, this cannot be prevented without a safety loss. Indeed, there is no way to guarantee, without human intervention, that a structural conflict cannot be an effective conflict. As our field of application is IAMD, safety remains our main priority and so the drawback in terms of circuit's size is accepted.

3.3 Finding Maximal Marking of Each Place

In order to implement the model, the number of bits required to store the marking of places must be defined. Of course only bounded PN can be implemented. The designer can indicate a maximal marking from which the number of bits can be set for every place. But this solution is not optimal as far as circuit size is concerned as the maximal marking of each place can be very different. But more importantly, in a medical context, it is necessary to have a safe behavior. Yet if the maximal marking given by the designer is wrong, the behavior of the implemented system will not be the expected one. For example, if the marking of a place is stored in 2 bits and the marking becomes equal to 4, the marking will in fact be equal to 0 which is of course a big issue. Hence a way to automatically determine the maximal marking of each place in a safe way has to be defined.

Existing analysis algorithms like the reachability graph computation, given for example by TINA, can be used. By going through the set of all possible states and by recording the worst possible marking of each place, the maximal marking for each is determined. Then, in HILECOP, the maximal marking of each place is automatically used in the corresponding instance of the place-component allowing to precisely size it and hence to adjust the circuit's size.

If the states graph cannot be used due to combinatory explosion, several partial reduction approaches can be used, for example Boussin's rules. It is still also possible to have some interesting results thanks to the structural analysis

[7] which is usually less complex. This analysis indicates the places that are in a P-invariant (P-semi-flow). For each place in one invariant, the maximal marking is equal to the constant of this invariant (determined thanks to initial marking) divided by the coefficient associated to this place in the P-invariant equation. The limitation being that a place is not necessarily covered by a P-invariant.

3.4 Determining the Number of Necessary Counters

For time transitions, it is necessary to have a counter to determine for how much time the transition has been sensibilized. This has a cost as far as circuit size is concerned. The goal is hence to have as few counters as possible. Indeed two transitions can share the same time counter iff they are never sensibilized at the same time [7]. To determine which transitions are never sensibilized at the same time, the structural analysis of PN can be used. All transitions that are in a T-invariant (T-semi flow) are necessarily not sensibilized at the same time. This allows us to exactly determine exactly the number of necessary counters without taking any risk of modifying the behavior of the model. A more thorough study is currently done to prove the efficiency and reliability of this method.

4 Application on an Industrial System

When developing neuroprostheses as palliative solutions for motor control disabilities, one of the challenges is to have small, reliable, and easy-to-implant devices. Hence a specific architecture has been defined, based on communicating distributed stimulation units that embed the electronics needed to generate stimulation profiles [14]. The system relies on a two-wire network that interconnects these distributed units and a controller, providing them both data and power. In this article, we are interested in the distributed units.

4.1 Description of a Distributed Stimulation Unit (DSU)

DSU are described in details in [14], only a short presentation is done here. A DSU is a device that generates an electrical stimulus by executing programmable stimulation profiles. It has communication facilities so that it can be remotely managed and synchronized with other DSUs. It includes an analog part (fig. 6)- the stimulation pulses generator that generates stimuli and provides high voltage for the output stage- being connected to the multipolar electrode placed on the nerve (or on the muscle). It also includes a digital part (fig. 6) which embeds: a micro-machine in charge of the execution of the stimulation sequence (stored in the micro-program), reference models in charge of monitoring the stimulation sequence and a protocol interpreter in charge of all the communication aspects.

The micro-machine can be seen as a specific small processor, similar to an application-specific instruction-set processor that runs micro-programs written in FES-dedicated, reduced 32-bit instruction set. It configures the analog subsystem and drives it by calibrating the current pulse (waveform, amplitude, duration) to be applied to the multipolar electrode.

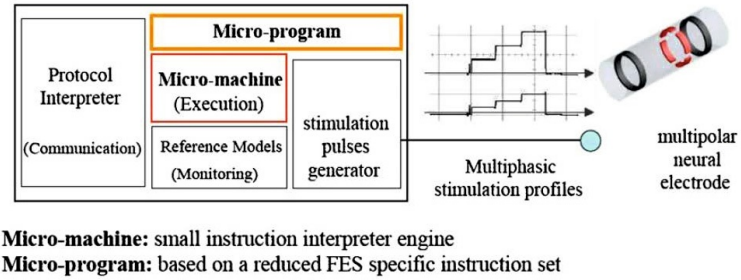


Fig. 6. The DSU embeds five functional blocks

The protocol interpreter manages exchanges between the DSU and the remote controller via the communication bus. The exchanges are of different types: micro-program downloads, start requests, error notification, and messages dedicated to the DSU configuration. The communication module is a three-layer protocol stack, according to the reference given by OSI model. These layers are the application layer, the MAC layer [13] and the physical layer.

The reference models are in charge of monitoring the respect of neurophysiological constraints. They are formulated in terms of temporal constraints and are monitored independently on each cathode. The monitoring of these constraints is based on simultaneous application of the stimulation to the nerve and to the reference models. If any constraint is violated, the reference model forces the micro-machine to immediately stop the stimulation and commute to a safe state. The micro-machine then must be externally rearmed by the global controller.

4.2 DSU Model

The digital part of the DSU has been designed thanks to the HILECOP methodology. The global HILECOP model contains 650 places and 770 transitions. This model is too large to be detailed here. As an example, the behavioral model of the micro-machine is shown in figure 7. It contains 61 places, 98 transitions and 4 HC; indeed it contains one reference model for each cathode which contains each 10 places and 15 transitions. The global model has been implemented on a FPGA (Actel Igloo) and its behavior formally and experimentally validated, before being the core of a real stimulator. [14].

4.3 Results Obtained

We now present analysis results of the methods given in section 3 on the micro machine model 7. The industrial software Libero used for implementation on Actel FPGA indicates that 6973 logic blocks are necessary to implement this model without any priorities. Then the model was treated in order to determine all structural conflicts as explained in §3.2. 130 out of the 158 transitions were

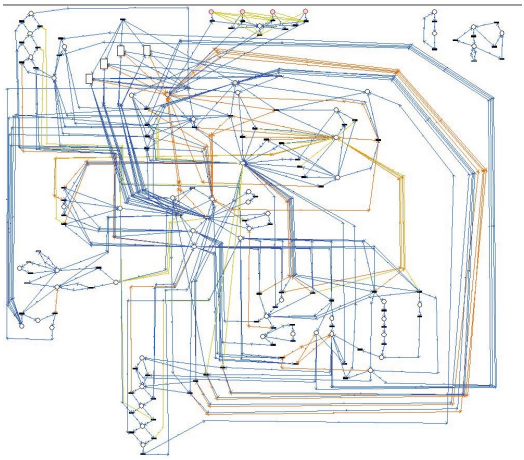


Fig. 7. Model of the micro-machine of an IAMD

in at least one conflict. The model was implemented again this time with added priorities. As the model is a binary PN, the case where there are enough tokens to fire two transitions in conflict cannot occur and so was not tested. With this implementation, 7035 logic blocks were necessary, thus an increase of 62 logic blocks (0,9 %) compared with the implementation without priorities. Hence, even if handling priorities increase the circuit's size, this increase is sufficiently small to be acceptable even when most of transitions are implied in a conflict.

As mentioned section 3.3, the maximal marking is extracted from the states graph of the model. We first generated this graph using the TINA toolbox. But the inherent complexity of the micro-machine model led to the well-known combinatory explosion. Thus, we decided to independently validate the normal behavior of the micro-machine, i.e. removing the error management. Indeed, the reference model monitoring and the treatment of errors increased the model size: only 36 of the 101 places and 50 of the 158 transitions are related to the normal behavior. The reachability graph of the micro-machine model without error management has 155812 classes and 837169 transitions. The analysis of the maximal marking of all the places shows that the TPN is binary.

Error management complicates the model which leads to large models, even with the use of components, as in figure 7. The error management is what makes the model complicated as mentioned. The problem is, for now, if a designer have to model exception handling, i.e. the fact that part of the system must be stopped no matter its state, he must necessarily make the exhaustive list of every possible situation in which the given subsystem could be. For complex systems, this leads to: an increased risk in design mistakes or omissions, an increase in circuit size, a less reactive system (as the firing of each transition takes one time unit), and a model too complex to be analyzed. The idea is then to modify the formalism used for the description of the HC behavior in order to ease the work of the designer and to optimize the model's implementation. For that goal, the concept of macroplace for IPrTPN has been defined and is under validation.

5 Conclusion

The tool-based methodology HILECOP, allowing the design of complex digital systems and their implementation on a FPGA, has been presented. This component-based methodology favors modularity and reusability. The use of PN also allows us to benefit from existing analysis tools or algorithms. This allows not only to verify the model but also to optimize its implementation in terms of circuit size. Different methods have been explained in order to do this optimization. They have been applied in the context of IAMD. This application allows us to confirm the feasibility of the methodology. They will now be integrated in the tool to be more transparently and more effectively used in HILECOP. Evolutions of the IPrTPN formalism are also studied in order to have an easy and efficient way to deal with exception handling. Once stable, the HILECOP tool will be freely available for the academic community.

References

1. Kralj, A., Bajd, T.: Functional Electrical Stimulation: Standing and Walking After Spinal Cord Injury. Chemical Rubber Company (CRC Press), Boca Raton (1989)
2. Rodriguez-Andina, J.J., Moure, M.J., Valdes, M.D.: Features, Design Tools, and Application Domains of FPGAs. IEEE Transactions on Industrial Electronics 54(4) (August 2007)
3. Navabi, Z.: VHDL: Analysis and Modeling of Digital Systems. McGraw-Hill (1997)
4. Souquet, G., Andreu, D., Guiraud, D.: Petri nets based methodology for communicating neuroprothesis design and prototyping. In: ISABEL, Aalborg, Denmark (2008)
5. Andreu, D., Souquet, G., Gil, T.: Petri Net Based Rapid Prototyping of Digital Complex System. In: ISVLSI, Montpellier, France (2008)
6. David, R., Alla, H.: Discrete, Continuous, and Hybrid Petri nets. Springer (2005)
7. Murata, T.: Petri nets: Properties, analysis and applications. Proceedings of the IEEE 77(4), 541–580 (1989)
8. Berthomieu, B., Peres, F., Vernadat, F.: Model Checking Bounded Prioritized Time Petri Nets. In: Namjoshi, K.S., Yoneda, T., Higashino, T., Okamura, Y. (eds.) ATVA 2007. LNCS, vol. 4762, pp. 523–532. Springer, Heidelberg (2007)
9. 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)
10. Traonouez, L.M., Lime, D., Roux, O.: Parametric Model-Checking of Stopwatch Petri Nets. Journal of Universal Computer Science 15(17), 3273–3304 (2009)
11. Royal, A., Cheung, P.Y.K.: Globally asynchronous locally synchronous FPGA architectures. In: Cheung, P.Y.K., Constantinides, G.A. (eds.) FPL 2003. LNCS, vol. 2778, pp. 355–364. Springer, Heidelberg (2003)
12. Billington, J., et al.: The Petri Net Markup Language: Concepts, Technology, and Tools. In: van der Aalst, W.M.P., Best, E. (eds.) ICATPN 2003. LNCS, vol. 2679, pp. 483–505. Springer, Heidelberg (2003)
13. Godary-Dejean, K., Andreu, D.: Formal Validation of a Deterministic MAC Protocol. ACM Trans. Embed. Comput. Syst., 6:1–6:23 (2013)
14. Andreu, D., Guiraud, D., Souquet, G.: A Distributed Architecture for Activating the Peripheral Nervous System. Jounal of Neural Engineering 6, 1–18 (2009)
15. Petri, C.A.: Kommunikation mit Automaten. Dissertation der Fakultät für Mathematik und Physik, Technische Hochschule Darmstadt, Bonn (1962)

Formal Analysis of the ACE Specification for Cache Coherent Systems-on-Chip

Abderahman Kriouile[1,2] and Wendelin Serwe[2]

[1] STMicroelectronics, 12, rue Jules Horowitz, BP 217, 38019 Grenoble, France
[2] Inria/LIG, 655, av. de l'Europe, Montbonnot, 38334 Saint Ismier, France

Abstract. System-on-Chip (SoC) architectures integrate now many different components, such as processors, accelerators, memory, and I/O blocks, some but not all of which may have caches. Because the validation effort with simulation-based validation techniques, as currently used in industry, grows exponentially with the complexity of the SoC, we investigate in this paper the use of formal verification techniques. More precisely, we use the CADP toolbox to develop and validate a generic formal model of an SoC compliant with the recent ACE specification proposed by ARM to implement system-level coherency.

1 Introduction

The integration of ever more functionalities in set-top boxes or mobile appliances such as smartphones increases the complexity of both the embedded software and the hardware architecture. The latter is usually a complex System-on-Chip (SoC), featuring a significant number of heterogeneous components. Indeed, a typical SoC includes nowadays not only processors and memory, but also dedicated hardware accelerators and (analog) I/O blocks. Integrating caches into some of these components (in particular, into processors and hardware accelerators) can increase performance and reduce power consumption, for instance by avoiding accesses to (possibly off-chip) memory.

In the past, prevalence of fast processors encouraged designers to manage cache coherency in software, taking advantage of the flexibility of software solutions. However, due to increased software complexity, a recent trend [14,23] is to introduce hardware support for cache coherency to improve performance and to lower power consumption by lightening the load on the processors. Hence, ARM proposed ACE (AXI Coherency Extensions) [1], which is becoming a de facto industrial standard for system-level cache coherence in heterogeneous SoCs (ACE explicitly includes operations, called ACE-Lite operations, for components without cache). ACE is used in ARM's *big.LITTLE* framework, which takes advantage of two processors (i.e., a "big" and a "LITTLE" one) for low-power SoCs. Also, STMicroelectronics is about to integrate system level coherency (based on ACE) in its upcoming SoCs.

As cache coherence protocols are known to be complex and difficult to validate, assuring system-level cache coherency is one of the major challenges faced

C. Pecheur and M. Dierkes (Eds.): FMICS 2013, LNCS 8187, pp. 108–122, 2013.

by architects of current SoC and NoC (Network-on-Chip) designs. Current industrial validation flows are based on simulation techniques. Because the related validation effort grows exponentially with the complexity of hardware architectures, we study the application of formal verification techniques, where the human modeling effort increases linearly with the complexity of architectures (each component is modeled by a process). Thus, the exponential complexity is supported by automated verification tools.

Concretely, we use the CADP toolbox [10] and its modeling language LNT [3] for the analysis of system-level cache coherency in a heterogeneous SoC. We focus on enumerative model checking methods to prove their feasibility on an industrial case study.

As a first step, we develop a generic formal LNT model of an SoC, including an ACE-compliant cache coherent interconnect and abstractions of master and slave components (e.g., processors and shared memory). The model is parametric and can be instantiated with different configurations (number of masters, number of cache lines, and number of memory lines) and different sets of supported ACE transactions. We use a constraint-oriented specification style to model the global requirements of the ACE specification, which must be guaranteed by any implementation. The LNT model enables STMicroelectronics architects to interactively simulate a coherent SoC at system level. We also express several correctness properties in the MCL language [15] and check them on the LNT model using the EVALUATOR 4.0 model checker. From the counterexamples generated by EVALUATOR 4.0, we extract interesting scenarios to be tested on any implementation of an ACE-compliant interconnect.

The rest of this paper is organized as follows. Section 2 presents the ACE specification. Section 3 describes our LNT model of an ACE-compliant SoC. Section 4 discusses the validation of correctness properties. Section 5 surveys related work. Section 6 gives concluding remarks and directions of future work.

2 System Level Cache Coherency with ACE

In general, a System-on-Chip (SoC) is composed of different hardware blocks like generic or specialized processors, memories, interconnects, dedicated Intellectual Properties (IPs), or input/output components. These heterogeneous components usually access a *shared memory* consisting of several *memory lines*. To increase data access performance, some components may use a *cache*, containing local copies of memory lines. An SoC is called *cache coherent* if write operations to the same memory line by two components are observable in the same order by all components of the system. One may distinguish *sharable* and *non-sharable* memory lines. For example, the graphics memory of an SoC might be dedicated to image processing and exclusively used by the Graphics Processing Unit (GPU), whereas the remaining memory might be used by either the generic processors (Central Processing Units, CPUs) and the GPU: in this case, the graphics memory is non-sharable, and the remaining memory is sharable.

The components of an SoC can be grouped into *master* components (such as CPUs) and *slave* components (such as memories). Components communicate

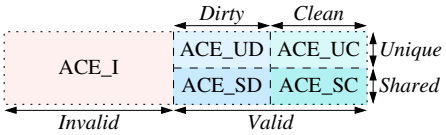


Fig. 1. ACE states of a cache line

via an interconnection medium, called the *interconnect*. In the case of a cache coherent system, the interconnect is also called a Cache Coherent Interconnect (CCI). Each component communicates with the interconnect by a communication port, each of which may consist of several channels. Operations performed on ports are called *transactions*.

2.1 ACE

To support system-level coherency, ARM has recently proposed the ACE (AXI Coherency Extensions) protocol specification [1,22], which extends the AMBA (Advanced Microcontroller Bus Architecture)/AXI (Advanced eXtensible Interface) specification. ACE is designed to maintain coherence when sharing data across caches of an SoC, to enable interaction between heterogeneous components, and to ensure maximal reuse of cached data. ACE also supports a flexible framework for system level coherency: the system designer can determine the ranges of memory lines that are coherent, the system components that implement the coherency extensions, and the communication policies.

The ACE specification defines the hardware interface protocol (between components and the interconnect), the expected behavior of the components, and the responsibilities of the interconnect. ACE admits different cache coherence policies, known as directory based, snoop filter, or no snoop filter models.

2.2 ACE States

ACE distinguishes five states (shown in Figure 1) of a cache line.

A cache line is *invalid* if it does not contain a copy of any memory line. A cache line is *unique* if all other copies of the same memory line are invalid. A cache line is *shared* if all other copies of the same memory line are shared or invalid. A cache line is *dirty* (respectively *clean*) if the master is responsible (respectively not responsible) of writing the data back to the shared memory.

2.3 ACE Ports and Channels

The ACE specification distinguishes three kinds of ports to connect a component to an interconnect. An *ACE port* is used for components having a cache memory. An *ACE-Lite port* is used for components without a cache. An *AXI port* is used for components that do not use coherency.

Each port consists of several channels. ACE distinguishes three types of channels: *read channels*, *write channels*, and *snoop channels*. Read (respectively, write) channels are used to read (respectively, write) data; these channels extend AMBA AXI channels with coherency related parameters. Read channels are the *address read* channel (AR, to send read requests) and the *data read* channel (R, to send the data back). Write channels are the *address write* channel (AW, to send write requests), the *data write* channel (W, to send the data to be written), and the *write response* channel (B, to signal completion of a write).

Snoop channels are used for snoop requests issued by the interconnect to masters with a cache. Snoop channels are the *address coherency* channel (AC, to send snoop requests), the *coherency response* channel (CR, to answer snoop requests, indicating whether a data transfer will follow), and the *coherency data* channel (CD, to send data to the interconnect).

2.4 ACE Transactions

The ACE specification defines several types of transactions. In the sequel, we focus on a significant subset of the transactions related to cache coherency. For each transaction, we present the expected order of operations on the channels. A master initiating a transaction is called *initiator*. A master with a cache receiving a snoop from the CCI is called a *snooped master*.

Snoop transactions are initiated by the interconnect while handling coherent transactions and cache maintenance transactions (see below). The interconnect initiates a snoop request on the AC channel. The snooped master responds on the CR channel indicating if a data transfer is needed. If so, the data is transferred on the CD channel indicating also whether the data is shared and whether the snooped master keeps the responsibility to write the data to memory.

Coherent transactions are used to access sharable memory lines, which might be in the caches of other components. We focus on four coherent transactions, all of which are initiated by a master through a request on the AR channel. The interconnect initiates corresponding snoop transactions to all other masters with a cache and, if necessary, reads the data from the sharable memory. Finally, the interconnect sends a reply transaction to the initiator on the R channel, indicating whether the data is shared and whether the responsibility to write the data to memory is passed to the initiator.

- A *ReadShared* transaction obtains a copy of the memory line without any constraint on the resulting state of the cache line.
- A *ReadUnique* transaction obtains a copy of the memory line and ensures that the copy is unique (i.e., no other copies exist).
- A *MakeUnique* transaction invalidates all other copies of the memory line.
- A *ReadOnce* transaction obtains the current contents of a memory line, which may not be copied into the cache.

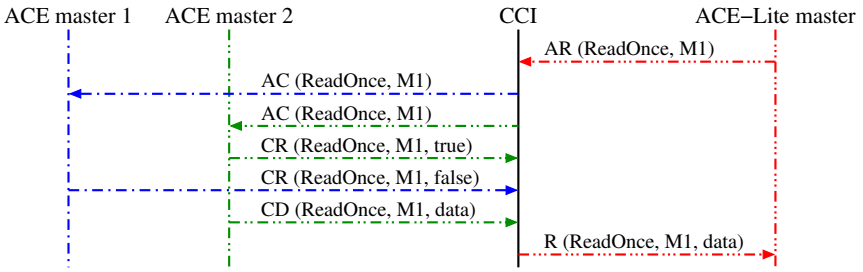


Fig. 2. Execution scenario of a *ReadOnce* transaction

Example 1. Consider an SoC with two ACE masters, a CCI, and an ACE-Lite master. Figure 2 shows the execution of a *ReadOnce* transaction (for memory line M1) initiated by the ACE-Lite master. The CCI snoops both ACE masters, which answer with a Boolean indicating whether the data is in their cache. The cache of ACE master 2 contains the data, hence this master also sends the data, which the CCI forwards to the ACE-Lite master to complete the transaction.

Non-snooping transactions are used to access non-shareable memory lines which must not be in the caches of other master components. We consider two non-snooping transactions: *ReadNoSnoop* and *WriteNoSnoop*.[1]

Memory update transactions are used to update shared memory. These transactions (e.g., *WriteBack*) are initiated by a master on the AW channel; the data to write is sent by the master on the W channel. The interconnect writes the data to the memory and returns an acknowledgement on the B channel.

Cache maintenance transactions are used by master components to access and impact the caches of other components. In particular, cache maintenance transactions enable a master to observe the effect of load and store operations on system caches (which cannot otherwise be accessed). The ACE specification distinguishes three cache maintenance transactions: *CleanShared*, *CleanInvalid*, and *MakeInvalid*. These transactions are initiated by sending a request on the AR channel. The interconnect initiates corresponding snoop transactions to all other masters with a cache. For a *CleanShared* transaction, a snooped master may retain its local copy of the memory line, but for a *CleanInvalid* or *MakeInvalid* transaction, a snooped master must invalidate its local copy. For a *CleanShared* or *CleanInvalid* transaction, a snooped master must also provide the data if the corresponding cache line is dirty. After all snooped masters have answered, the interconnect returns an acknowledgement to the initiator, on the R channel.

ACE-Lite transactions are a subset of ACE transactions, namely: *ReadNoSnoop*, *ReadOnce*, *CleanShared*, *CleanInvalid*, and *MakeInvalid*.

[1] Those transactions are equivalent to the AXI Read and AXI Write transactions.

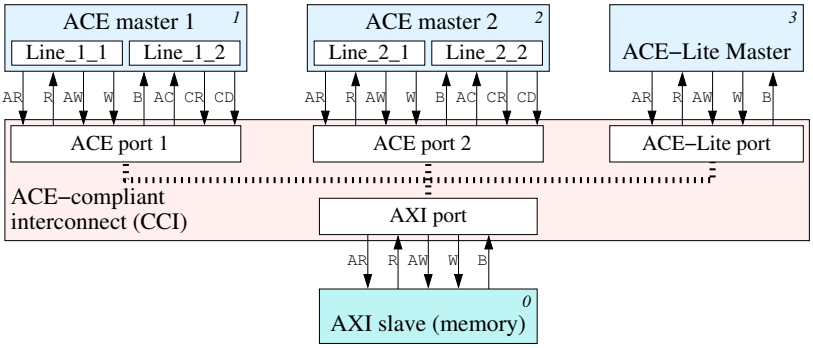


Fig. 3. Model architecture

3 Formally Modeling an ACE-Compliant SoC in CADP

We developed a formal model of an ACE-compliant SoC, consisting of a CCI, masters, and slaves, using the LNT (also called LOTOS NT) language [3], supported by the CADP toolbox [10]. LNT combines the best features of process-algebraic and imperative programming languages. The semantics of LNT model is defined as a *Labeled Transition System* (LTS), following a black box view of the system. The LNT.OPEN tool translates an LNT model into an LTS suitable for (on-the-fly) verification.

Our formal model (about 3200 lines of LNT code) represents the behavior of the system focusing on the interactions between components. It is parametric and can be instantiated with different configurations (number of masters, number of cache lines for each master, number of memory lines per slave, etc.). The model is generic in the sense that it includes all the behaviors permitted by the ACE specification for any correct implementation. The masters are non-deterministic agents, which may initiate all the transactions described in Section 2.

We opted for modeling a *fully connected snoop* topology, i.e., all coherent transactions lead to snoop transactions for all masters with cache. Note that the first industrial implementation [2] of the ACE protocol also has a fully connected snoop topology.

Each operation on a channel is modeled by an LNT rendezvous[2] on a gate of the same name as the channel.

Example 2. Figure 3 shows the model of an SoC consisting of a CCI, two ACE masters, an ACE-Lite master, and a shared memory (consisting of three memory lines). Each ACE master contains two cache lines. The component index is 0 for

[2] The semantics of an LNT rendezvous avoids the need to model the acknowledgement *signals* at the level of channel transmission. However, the acknowledgement *operation* for a non-atomic transaction (e.g., the operation on the B channel for Write transactions) is represented by an independent LNT rendezvous (on gate `B`).

the shared memory, 1 (respectively, 2) for the ACE masters, and 3 for the ACE-Lite master. Notice that this configuration shares most characteristics with the big.LITTLE architecture.

3.1 Types and Data Structures

Each memory line is characterized by two parameters: an index (of range type `Index_Mem`, where `N` is the number of memory lines) and a data (of type `Nat`). Hence the shared memory can be represented by an array of values of type `Nat`, indexed by the range of `Index_Mem`. ACE states are represented by an enumerated type called `ACE_state`.

```
type Index_Mem is range 1 .. N of Nat end type
type Cache_Line is
   LINE_C (indC: Index_Cache, S: ACE_state, indM: Index_Mem, data: Nat)
end type
type Mem_Lines is array [1 .. N] of Nat end type
type ACE_state is ACE_I, ACE_UC, ACE_UD, ACE_SC, ACE_SD end type
```

Similarly, we define an index for system components (`Index_Component`) and an index for cache lines of a master with cache (`Index_Cache`). ACE transactions are modeled by an enumerated type `ACE_Trans`.

We introduce an abstract transaction $\mathcal{A}$ that simulates any ACE transaction by executing all the phases of an ACE transaction without changing the ACE state of cache lines.

3.2 Channels

Each ACE channel is modeled by a typed LNT gate. The types of LNT gates (called LNT channels) specify the number and types of the parameters (called *offers*), i.e., the values exchanged during a rendezvous. All gates have an offer to represent the ongoing ACE transaction, an offer to represent the initiator of the current transaction, and an offer to designate the concerned memory line. Snooping gates (`AC` and `CR`) have also an offer to represent the snooped master. Gates which transfer data (`R`, `W`, and `CD`) have also an offer for the data. The gates `R` (read data channel) and `CD` (snoop data channel) have also three Boolean offers. *DataStatus* indicates whether the data is valid, *PassDirty* indicates whether the responsibility of writing data to memory is passed, and *IsShared* indicates whether the data is shared. The gate `CR` has a Boolean offer *DataTransfer* to indicate if a data transfer will be follow on the `CD` gate. The gate `B` has a Boolean offer indicating if the write has completed correctly.

For verification purposes, we add an offer representing the ACE state of the cache line to all gates going out from an ACE master (i.e., `AR`, `AW`, and `CR`). Similarly, the gates between the CCI and a slave have an additional offer corresponding to the initiator.

```
process memory [AR: CHANNEL_AXI_AR, R: CHANNEL_AXI_R,
                AW: CHANNEL_AXI_AW, W: CHANNEL_AXI_W, B: CHANNEL_AXI_B]
               (idMEM: Index_Component)
is
 var LINES: Mem_Lines, pending_read: Bool, transR, transW: ACE_Trans,
     ind_R, ind_W: Index_Mem, CPU_R, CPU_W: Index_Component, data: Nat
 in
   -- initializations (not included)
   loop select
      when pending_read == false then
         AR (?transR, idMEM, ?indM_R, ?CPU_R);
         pending_read := true
      end when
   []
      when pending_read == true then
         R (transR, idMEM, LINES[Nat(indM_R)], CPU_R);
         pending_read := false
      end when
   []
      AW (?transW, idMEM, ?ind_W, ?CPU_W);
      W (transW, idMEM, ind_W, ?data, CPU_W);
      LINES[Nat(ind_W)] := data;
      B (transW, idMEM, indM_W, true, CPU_W)
   end select end loop
end var end process
```

Fig. 4. LNT process representing the shared memory

3.3 ACE Slave: Shared Memory

The shared memory is modeled by the LNT process shown in Figure 4. The five gates `AR`, `R`, `AW`, `W`, and `B` correspond to the AXI channels. We use the Boolean `pending_read` to indicate if a read operation is in progress. The behavior of the memory process is a non-terminating `loop`, the body of which is a non-deterministic choice (`select`[3]) between three possibilities:[4]

- Receiving a read request on the `AR` gate, which is only possible if no read operation is in progress (`pending_read == false`),
- Sending back a read data on the `R` gate, which is only possible if a previous read request was received (`pending_read == true`),
- Receiving a write request.

In our model, a write operation cannot be interrupted by a read operation.

[3] The LNT construction "`select` *A* `[]` *B* `[]` *C* `end select`" presents a non-deterministic choice between *A*, *B*, and *C*.

[4] In an LNT rendezvous, an offer "`?x`" accepts any value of the same type as variable `x`, and the received value is stored in variable `x`.

3.4 ACE Masters

The cache lines of a master are essentially independent from each other, i.e., transactions on different cache lines can freely interleave.[5] Hence we choose to model each master by a parallel composition of cache lines. Each cache line is modeled by five mutually recursive LNT processes: (1) process *cpu* initializes the cache line; (2) process *cpu_ready* represents a cache line that is ready to initiate an ACE transaction or to receive a snoop request from the CCI; (3) process *cpu_reply* represents a cache line that has previously initiated an ACE transaction and waits for the reply from the CCI (the cache line is also ready to receive any snoop request from the CCI); (4) process *cpu_snoop* represents a cache line that has previously received a snoop request from the CCI and can either reply to this request or initiate a new ACE transaction; (5) process *cpu_reply_snoop* represents a cache line that has previously initiated an ACE transaction and has also received a snoop request from CCI: thus, it is both waiting for the reply of the ACE transaction and ready to reply to the snoop request. Each of these processes behaves as a large non-deterministic choice between all possible rendezvous. Each branch consists of a guard, a rendezvous with parameters to handle the ongoing transaction, and a recursive call corresponding to the new state of the cache line.

In order to generate a smaller LTS in the debug phase of the model, we can deactivate the non-deterministic choice of cache lines relative to the state changes permitted by the ACE specification.

3.5 ACE-Lite Masters

The LNT model of an ACE-Lite master is obtained from the model of an ACE master by removing the handling of snoop requests. Thus, an ACE-Lite master is modeled by three mutually recursive LNT processes: a process to initialize the ACE-Lite master, a process *lite_ready*, which can initiate any of the ACE-Lite transactions presented in Section 2.4, and a process *lite_reply*, which waits for a reply from the CCI.

3.6 Cache Coherent Interconnect (CCI)

To ease the modeling of all interleavings between the ports of the CCI, we employ two techniques. First, we model the CCI by a parallel composition of as many processes as there are ports; each port is always ready to receive a request from both the corresponding component and other ports. Second, all received requests are stored in a set and are handled in any order.[6]

The ports of the CCI communicate internally via dedicated gates, which are not part of the ACE specification and can be hidden (in the LTS and in counterexamples), but are useful in the debug phase of the model.

[5] Actually, the only constraint is to store the same memory line in at most one cache line of a same master.

[6] Because the numbers of CPUs and cache lines are fixed, the number of requests in a set is bounded by construction.

Example 3. The CCI of Figure 3 contains four ports: two ACE ports, each communicating with an ACE master, one ACE-Lite port communicating with an ACE-Lite master, and one AXI port communicating with the shared memory.

3.7 Requirements on the Global Ordering of Transaction

The ACE specification includes some global requirements concerning system-level coherency for the implementation of any ACE-compliant interconnect. Following a constraint-oriented specification style, our LNT model integrates these global requirements as dedicated processes (one process per requirement and memory line), composed in parallel with the remainder of the model. Hence, those processes monitor the system and have a global view of all transactions. There are two kinds of global requirements:

- Coherency between caches (called *horizontal coherency*) [1, section C4.10]: When two masters attempt to write to the same memory line simultaneously (i.e., the second transaction begins before the end of the first transaction), then the interconnect must ensure a strict order of the transactions. Concretely, while handling a snoop transaction, the constraint process ensures that a subset of snoop transactions (relative to the same memory line) are not handled before the end of the first transaction.
- Coherency between the memory and caches (called *vertical coherency*) [1, section C6.5.3]: Data received from caches must be written to the memory in the correct order. The constraint process monitors write transactions, prohibiting that an old data overwrites a more recent one.

3.8 State Space Generation

For our analysis, we consider several SoC configurations, each consisting of a shared memory, one ACE-Lite master, and two ACE masters, with two cache lines each. To focus on coherency issues, the first cache lines of each ACE master execute transactions concerning the same memory line (this is suitable according to [8]). Each master initiates at most one transaction (chosen from a set of allowed transactions); thus, the second cache lines of each ACE master never initiate a transaction (but answer snoop requests). We selected subsets of transaction that could create problems for properties to verify.

For each considered configuration, Table 1 gives the size of the corresponding LTS. Columns one, two, and three give the set of transactions that master 1 (respectively, master 2, or the ACE-Lite master) are allowed to initiate. Column four tells whether the model includes the processes enforcing the global ordering requirements; we generate LTSs for models without the corresponding constraint processes to study their impact on the properties of the system. An LTS of the model with global constraints is included in the one without global constraints with respect to strong bisimulation (i.e., the constraints only removed behaviors), but the state space may be larger because a state now also integrates the current state of the control process.

Table 1. Experimental results: state space generation and verification

allowed transactions			global	LTS size		properties				
m1	m2	lite	constraints	states	transitions	φ_1	φ_2	φ_3	φ_4	φ_5
S_0	$\{\mathcal{A}\}$	S_0	yes	93,481,270	308,087,560	√	√	√	√	√
S_0	$\{\mathcal{A}\}$	S_0	no	105,376,971	351,344,207	√	√	√	√	×
S_0	$\emptyset$	S_0	yes	7,518,552	21,227,610	√	√	√	√	√
S_1	$\emptyset$	S_1	yes	3,685,311	10,649,422	√	√	√	√	√
S_1	$\emptyset$	S_1	no	3,127,707	9,121,134	√	√	×	×	×
S_2	S_2	$\emptyset$	yes	3,545,801	11,122,536	√	√	√	√	√
S_2	S_2	$\emptyset$	no	2,819,505	9,095,620	√	√	×	×	√
S_3	$\emptyset$	S'_3	yes	1,834,195	5,170,829	√	√	√	√	√
S_3	$\emptyset$	S'_3	no	1,437,412	4,547,398	√	√	√	√	×
S_4	S_4	$\emptyset$	yes	560,299	1,669,886	√	√	√	√	√
S_4	S_4	$\emptyset$	no	599,971	1,780,634	√	√	×	×	×
S_5	S_5	$\emptyset$	yes	40,983	63,922	√	√	√	√	√
S_5	S_5	$\emptyset$	no	55,439	98,688	√	√	√	√	√

In the table above, we use those sets of allowed transactions:
S_0 = set of all ACE (respectively ACE-Lite) transactions
$S_1 = \{MakeUnique, ReadOnce, ReadUnique, WriteBack\}$
$S_2 = \{MakeInvalid, MakeUnique, ReadShared, ReadUnique, WriteBack\}$
$S_3 = \{MakeUnique, WriteBack\}$, $S'_3 = \{ReadOnce\}$
$S_4 = \{CleanInvalid, CleanShared, ReadUnique, WriteBack\}$
$S_5 = \{MakeInvalid, MakeUnique, WriteBack\}$

4 Validation

We verify several system-level properties on our model of the ACE-compliant SoC presented in Example 2. We start by validating the complete and correct execution of separate transactions, then we verify the coherency of the cache states, and finally we check data integrity in the system.

We express all these properties in *Model Checking Language* (MCL) [15], an extension of the modal μ-calculus with high-level operators aimed at improving expressiveness and conciseness of formulæ. The main ingredients of MCL are parametrized fixed points, action patterns enabling to extract data values from LTS transition labels, modalities on transition sequences described using extended regular expressions and programming language constructs, and an infinite looping operator specifying fairness. The EVALUATOR 4.0 model checker of CADP can verify MCL properties on the fly, based on the local resolution of Boolean equation systems, which has a linear-time complexity for (data-less) alternation-free and fairness formulæ. We wrote also several macros to simplify writing properties for industrial users.

4.1 Complete Execution of Transactions

To verify that every transaction inevitably finishes, we use the following two liveness formulæ (φ_1 and φ_2):

```
[ true * . { AR ?op:String ?n:Nat ?l:Nat ... } ] inev ( { R !op !n !l } )
[ true * . { AW ?op:String ?n:Nat ?l:Nat ... } ] inev ( { B !op !n !l } )
```

These formulæ use the macro `inev (L)`, which expresses that a transition labeled with `L` will eventually occur. This macro can be defined as follows:

```
macro inev (L) = mu X . ( < true > true and [ not L ] X ) end_macro
```

The first (respectively second) formula requires that each action `AR` (respectively `AW`) is eventually followed by an action `R` (respectively `B`). Note the capture of the exchanged values into the variables `op`, `n`, and `l` in the first action predicate (using the LNT-like syntax "?*variable*:*Type*", where *Type* is one of the predefined types of MCL) and the use of the captured values in the second action predicate.

4.2 Cache Coherency

To verify the coherency of the ACE states of all the caches of the system, we have to translate the state-based properties to action-based properties, using the ACE state offer added to transactions issued by cache lines (see Section 3.2). To simplify the formulæ and to reduce verification complexity, we rename these transitions using a unique gate `G` keeping only useful offers.

Two safety formulæ express coherency. The first one (φ_3), requires that if a cache line is in the state ACE_UD then all other cache lines containing the same memory line must be in the state ACE_I:

```
[ true * .
   {G ?m1:Nat ?indM:Nat "ACE_UD"} .
   ( not ({G !m1 !indM ?s1:String where s1<>"ACE_UD"}) ) * .
   {G ?m2:Nat !indM ?s2:String where (m2<>m1) and (s2<>"ACE_I")}
] false
```

The second formulæ (φ_4) is similar to φ_3 and requires that if a cache line is in the state ACE_SD then all other cache lines containing the same memory line must be either in the state ACE_SC or the state ACE_I.

```
[ true * .
   {G ?m1:Nat ?indM:Nat "ACE_SD"} .
   ( not ({G !m1 !indM ?s1:String where s1<>"ACE_SD"}) ) * .
   {G ?m2:Nat !indM ?s2:String
      where (m2<>m1) and (s2<>"ACE_I") and (s2<>"ACE_SC")}
] false
```

4.3 Data Integrity

To verify the data integrity of the system, we use a safety property (φ_5), which enforces a correct order of write operations to the shared memory:

```
[ true * .
   { W !"WRITEBACK" ?c:Nat ?l:Nat ?d:Nat } .
   ( not { W !"WRITEBACK" !"0" !l !d !c } ) * .
   { W !"WRITEBACK" !"0" !l !d !c } .
   (
      ( not { AC ?any of String ?any of Nat !c ?any of Nat !l } ) and
      ( not { W ?any of String !"0" !l ?any of Nat ?any of Nat } )
   ) * .
   { W ?any of String !"0" !l ?h:Nat ?any of Nat where h<>d }
] false
```

Once a master `c` initiates a *WriteBack* transaction to a memory line `l` of a data `d`, and effectively written to memory (which has port number `0`), the property forbids a data `h` different from `d` to be written to the same memory line `l` without previously receiving a snoop request concerning line `l`.[7]

4.4 Model-Checking Results

The verification results of the properties presented in Sections 4.1 to 4.3 on the LTSs of Section 3.8 are given in columns seven to eleven of Table 1. All LTSs including the global ordering requirements satisfy ($\surd$) all five properties.

For LTSs without global ordering constraints, coherency (φ_3 and φ_4) and data integrity (φ_5) properties may not be satisfied ($\times$). In this case, EVALUATOR 4.0 generates minimal counterexample sequences, which correspond to scenarios to be tested (using an industrial testbench) on any implementation of an ACE-compliant interconnect. This interests STMicroelectronics, because these intricate test cases challenge the (complex) implementation of the coherency constraints in an interconnect.

5 Related Work

Formal verification techniques, e.g., (symbolic) model checking and theorem proving, has been often applied to the verification of hardware designs of cache coherence protocols, using various modeling languages, temporal logics, and verification tools [17,13]. Most works [4,6,7,9,12,16,18,20,21] concern elaborated protocols using more complex topologies than the fully connected snoop topology of our LNT model. The principal differences to our work are that we focus on a generic interconnect that includes the behavior of all correct implementations and that we study a heterogeneous SoC, rather than verifying a particular coherency protocol for a homogeneous system. Notice that the notion of a component without cache (ACE-Lite) snooping components with caches was introduced by the ACE specification.

[7] The number of parameters differs for the rendezvous on gate `W` between the CCI and the memory and those between a master and the CCI: for the former, the fifth parameter corresponds to index of the initiator.

The only paper dedicated to the formal verification of the ACE specification is a methodological guide [19], which shows the benefits of high-level modeling of system-level cache coherency using Jasper's formal verification tools. Our approach differs from [19] by addressing heterogeneous systems (in particular ACE-Lite masters) and by presenting validation results on an example.

6 Conclusion

We developed a generic formal LNT model of the recent ACE specification [1]. The constraint-oriented specification style proved helpful in the modeling of general requirements expressed in natural language. Our model has been found valuable by STMicroelectronics architects, because it enables interactive and backtrackable step-by-step system-level simulation (using the OCIS tool) of all ACE-compliant behaviors. We expressed correctness properties as temporal logic formulæ (in MCL) and verified them automatically (using the EVALUATOR 4.0 tool). Hence, we found that formal verification techniques can be used for the analysis of heterogeneous coherent SoCs.

This work can be pursued along several directions. First, the generic interconnect model can be used to analyze the impact of a coherent interconnect in a model of a concrete SoC. This requires to refine the models of masters and slaves to match those used in the SoC. Second, the formal model can be used to guide test and validation, as a reference model for co-simulation or by (automatically) extracting interesting test scenarios [11,5]. For instance, the counterexamples of Section 4 seem interesting test cases for any ACE-compliant interconnect. STMicroelectronics has expressed interest in both directions, as they address issues faced in the development of future products.

Acknowledgements. We are grateful to R. Mateescu (Inria) and M. Zendri (STMicroelectronics) for their contribution and their valuable remarks. We would also like to thank H. Garaval (Inria), G. Barthes, C. Chevallaz, G. Faux, O. Haller, and M. Soulie (STMicroelectronics) for helpful discussions.

References

1. ARM. AMBA AXI and ACE Protocol Specification, version ARM IHI 0022E (February 2013), `http://infocenter.arm.com/help/topic/com.arm.doc.ihi0022e`
2. ARM. CoreLink CCI-400 Cache Coherent Interconnect: Technical Reference Manual, revision r1p1 (November 2012), `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0470g/DDI0470G_cci400_r1p1_trm.pdf`
3. Champelovier, D., Clerc, X., Garavel, H., Guerte, Y., McKinty, C., Powazny, V., Lang, F., Serwe, W., Smeding, G.: Reference manual of the LOTOS NT to LOTOS translator (version 5.8). INRIA/VASY, 155 pages (March 2013)
4. Chehaibar, G.: Integrating formal verification with Murϕ of distributed cache coherence protocols in FAME multiprocessor system design. In: de Frutos-Escrig, D., Núñez, M. (eds.) FORTE 2004. LNCS, vol. 3235, pp. 243–258. Springer, Heidelberg (2004)

5. Chen, M., Qin, X., Koo, H.-M., Mishra, P.: System-Level Validation: High-Level Modeling and Directed Test Generation Techniques. Springer (2013)
6. Chen, X., Yang, Y., Gopalakrishnan, G., Chou, C.-T.: Efficient methods for formally verifying safety properties of hierarchical cache coherence protocols. Formal Methods in System Design 36(1), 37–64 (2010)
7. Clarke, E.M., Grumberg, O., Hiraishi, H., Jha, S., Long, D.E., McMillan, K.L., Ness, L.A.: Verification of the Futurebus+ cache coherence protocol. Formal Methods in System Design 6(2), 217–232 (1995)
8. Dill, D.L., Drexler, A.J., Hu, A.J., Yang, C.H.: Protocol verification as a hardware design aid. In: Proc. of the Int. Conf. on Computer Design: VLSI in Computers and Processors, pp. 522–525. IEEE (October 1992)
9. Eiríksson, A.T., McMillan, K.L.: Using Formal Verification/Analysis Methods on the Critical Path in System Design: A Case Study. In: Wolper, P. (ed.) CAV 1995. LNCS, vol. 939, pp. 367–380. Springer, Heidelberg (1995)
10. Garavel, H., Lang, F., Mateescu, R., Serwe, W.: Cadp 2011: A toolbox for the construction and analysis of distributed processes. Software Tools for Technology Transfer 15(2), 89–107 (2013)
11. Kahlouche, H., Viho, C., Zendri, M.: An industrial experiment in automatic generation of executable test suites for a cache coherency protocol. In: Proc. of the Int. Workshop on Testing of Communicating Systems. Chapman&Hall (1998)
12. Kapoor, H.K., Kanakala, P., Verma, M., Das, S.: Design and formal verification of a hierarchical cache coherence protocol for NoC based multiprocessors. The Journal of Supercomputing (2013)
13. Kern, C., Greenstreet, M.R.: Formal Verification in Hardware Design: A Survey. ACM Trans. on Design Automation of Electronic Systems 4(2), 123–193 (1999)
14. Martin, M.M.K., Hill, M.D., Sorin, D.J.: Why On-Chip Cache Coherence Is Here to Stay. Communications of the ACM 55(7), 78–89 (2012)
15. Mateescu, R., Thivolle, D.: A model checking language for concurrent value-passing systems. In: Cuellar, J., Sere, K. (eds.) FM 2008. LNCS, vol. 5014, pp. 148–164. Springer, Heidelberg (2008)
16. McMillan, K.L., Schwalbe: Formal Verification of the Encore Gigamax cache consistency protocol. In: Proc. of the Int. Symposium on Shared Memory Multiprocessors, pp. 242–251 (1991)
17. Pong, F., Dubois, M.: Verification Techniques for Cache Coherence Protocols. ACM Computing Surveys 29(1), 82–126 (1997)
18. Pong, F., Nowatzyk, A., Aybay, G., Dubois, M.: Verifying Distributed Directory-based Cache Coherence Protocols: S3.mp, a Case Study. In: Haridi, S., Ali, K., Magnusson, P. (eds.) Euro-Par 1995. LNCS, vol. 966, pp. 287–300. Springer, Heidelberg (1995)
19. Ranjan, R.: Formal Techniques for Protocol Verification: A Case Study On Verifying the ARM ACE Protocol. In: Electronic Design (January 2012)
20. Slobodová, A., Davis, J., Swords, S., Hunt Jr., W.: A Flexible Formal Verification Framework for Industrial Scale Verification. In: Proc. of the Int. Conf. on Formal Methods and Models for Codesign, pp. 89–97 (July 2011)
21. Stern, U., Dill, D.L.: Automatic Verification of the SCI Cache Coherence Protocol. In: Camurati, P.E., Eveking, H. (eds.) CHARME 1995. LNCS, vol. 987, pp. 21–34. Springer, Heidelberg (1995)
22. Stevens, A.: Introduction to AMBA 4 ACE. ARM whitepaper (June 2011)
23. Thompson, C.: Verifying Cache Coherency Protocols with Verification IP. Synopsis (October 2012)

Predicate Abstraction for Programmable Logic Controllers

Sebastian Biallas, Mirco Giacobbe, and Stefan Kowalewski

Embedded Software Laboratory, RWTH Aachen University, Germany

Abstract. In this paper, we present a predicate abstraction for programs for programmable logic controllers (PLCs) so as to allow for model checking safety related properties. Our contribution is twofold: First, we give a formalization of PLC programs in first order logic, which is then used to automatically derive a predicate abstraction using SMT solving. Second, we employ an abstraction called predicate scoping which reduces the evaluation of predicates to certain program locations and thus can be used to exploit the cyclic scanning mode of PLC programs. We show the effectiveness of this approach in a small case study using programs from industry and academia.

1 Introduction

Programmable Logic Controllers (PLCs) [13] are control devices used in industry to monitor, operate and control various machines, robots, assembly lines, chemical plants, power plants and oil rigs. PLCs comprise input connectors (typically connected to sensors), output connectors (typically connected to actuators) and a program which controls the operation. In the most common mode of operation, the so-called *cyclic scanning mode*, three phases are executed atomically: (a) the current values at the input connectors are stored in input variables, (b) the program is called to determine new values for the output variables and (c) the values of the output variables are copied to the output connectors. By executing these phases repeatedly at a high frequency the connected machinery can be controlled in a closed loop. Additionally to the input and output variables, the program can access non-temporal variables whose values are retained for the next cycle. The program itself is usually composed of functional components called *function blocks* (FBs), which can be written in various languages [13, Part 3].

Since PLCs operate in highly critical environments, it is recommended to verify their programs using formal methods [14]. In this paper we advocate model checking to verify functional properties.

1.1 Model Checking PLC Programs

Typical properties to be checked relate input values to output values. One might want to check, e. g., an invariant such as: If an emergency stop is signaled (input

C. Pecheur and M. Dierkes (Eds.): FMICS 2013, LNCS 8187, pp. 123–138, 2013.

```
1  PROGRAM Example
2  VAR_INPUT
3    in0, in1, in2: USINT;
4    flag : BOOL;
5  END_VAR
6  VAR_OUTPUT
7    out : USINT;
8  END_VAR
9  VAR
10   var : USINT;
11 END_VAR
12 IF flag THEN
13   IF in0+in1+in2 < 100 THEN
14     var := in0;
15   ELSE
16     var := 0;
17   END_IF;
18 ELSE
19   out := var;
20 END_IF;
21
22 END_PROGRAM
```

Fig. 1. Example PLC program used throughout the paper

is set), the motor is stopped (output is not set). Other formulae might refer to the order of certain events, e. g., after a failure, the motor does not start until the failure is acknowledged. For such properties ∀CTL is a suitable formalism. This approach is already implemented in the ARCADE.PLC framework [5] on which our work builds. Yet, the implementation is limited due to the availability of non-relational domains only and the restrictions of a hand written constraint solver, which then results in a state explosion for more complicated programs.

1.2 Contribution and Outline

To overcome these obstacles we propose a fully automatic predicate abstraction. In the underlying principles of this approach, we first encode the semantics of a given PLC program as a first order logic formula (Sect. 3). In Sect. 4, we then automatically derive the transition relation of the abstracted state space and introduce the predicate scoping to further reduce state spaces. The feasibility of our approach is demonstrated in Sect. 5 on various PLC programs. The paper ends with a discussion of related work in Sect. 6 and a conclusion in Sect. 7. We start by introducing our worked example.

2 Worked Example

We motivate our approach with the small example program shown in Fig. 1. This program is written in *structured text*, one of the five standard PLC programming languages [13]. Each cycle, i. e., each time the program is called, the following operation is performed: The input `flag` is tested (line 12). If the flag is not set then the output variable `out` is set to the value of `var` (line 19). Otherwise, the program tests whether the sum of the three inputs `in0`, `in1`, `in2` is less than 100 (line 13). If so, `in0` is assigned to `var`, otherwise `var` is set to 0. Note that `var` is a non-temporary variable, which retains its value for the next cycle.

Suppose we want to verify the program invariant `out` < 100. Careful inspection of the program reveals that this invariant is true: the variable `out` is only

set to the value of `var`, which in turn is either set to 0 or `in0`. The first case is trivial, the second case is only executed if $\texttt{in0} + \texttt{in1} + \texttt{in2} < 100$ which implies that $\texttt{in0} < 100$ (overflow can not occur here, since arithmetic is implicitly cast to a bigger accumulator data type here). Note that it is not obvious how to prove this invariant automatically.

Our approach works by model checking the state space of a PLC program, which allows to not only check simple invariants but also the correct ordering of certain events. The state space itself comprises of states that are tuples of the current line number and the variable values (we will defer a formal introduction to Sect. 3). It is important to observe that during the execution of the program the PLC does not communicate with input/output connectors. All communication is performed atomically at the beginning/end of the cycle so as to allow preliminary values during the computation. If, e. g., the program sequentially sets a number of outputs, all intermediate states are not observable until the end of the program and thus should not be subject to the verification. Therefore, all properties should only be checked in the very last program line. The invariant we want to verify will hence be encoded in ∀CTL as

$$AG\,(\mathsf{exitpoint} \implies \texttt{out} < 100), \tag{1}$$

where exitpoint evaluates to true only at exit. Such a property could naïvely be checked by enumerating the complete concrete state space. Yet, even for the small example program, this state space would have an excess amount of states making this approach unfeasible in practice. We therefore turn to a predicate abstraction where sets of concrete states are abstractly represented by a predicate (or conjunctions of predicates).

3 Encoding of PLC Semantics in FOL

In this section, we describe the transformation of PLC programs into first order logic (FOL) formulae. These formulae are written in a way that each model of these formulae represents a possible state change by statement encoded in the formula. Syntactically, this is indicated by using unprimed variables as pre- and primed variables as post-variables for encoding the semantics. This allows to derive the transition relation between (possible abstract) states with decision procedures. Later, we will then combine formulae describing a single step of the program into formulae describing the application of a basic block or even a whole cycle. We start by encoding the different types of variables.

3.1 Encoding of Variables and the Program

Let Var be the set of variables of the PLC program after flattening all structures and arrays. Depending on their lifetime and their semantics, we partition Var into three distinct sets:

- Var_M contains variables that retain their value between cycles. Such variables are defined, e. g., using the keywords `VAR` or `VAR_GLOBAL`.

- The set VAR_I holds the input variables. These are the variables declared as `VAR_INPUT` or `VAR_INOUT` in the outmost function block or in the program (input variables of inner function block instances are determined at the call site and thus not non-deterministic).
- Finally, all temporary variables are contained in VAR_T. These variables are reinitialized to their default value each cycle and thus do not retain their value for the next cycle.

Note that recursion is not possible in PLC programs. We can thus determine the number of variables used in total and do not need special handling of local variables.

Definition 1 (Memory). *The PLC memory state is defined as a tuple* $\langle \text{VAR}, \mathcal{D} \rangle$ *where* $\mathcal{D}$ *is the domain of discourse.*

The domain of discourse can be selected as the union of the data types of the variables in VAR, which after flattening are all elementary (i. e., non-aggregate) data types.

Definition 2 (Memory State). *A program state is given by a tuple* $\langle \ell, \nu \rangle$ *where* $\ell \in L$ *is a program location and* $\nu \colon \text{VAR} \to \mathcal{D}$ *is a variable assignment. Here,* ℓ *contains a symbolic address (e. g., a line number) of the next statement to be executed.*

We define two special program locations ℓ_a and ℓ_z which denote the first and last statement, respectively (the program can always be transformed to have only one exit point). A memory state uniquely determines which operation(s) are performed next and their result. We thus can define all possible executions of the PLC program in the following form:

Definition 3 (Program Model). *A program model is a state transition system* $\langle S, I, R \rangle$ *where* S *is the set of memory states,* $I \subseteq S$ *is the set of initial memory states and* $R \subseteq S \times S$ *is a transition relation.*

Example 1. Consider the program `Example` of Fig. 1. Its set of variables is encoded as $\text{VAR} = \{\texttt{in0}, \texttt{in1}, \texttt{in2}, \texttt{flag}, \texttt{out}, \texttt{var}\}$, which is subdivided into $\text{VAR}_I = \{\texttt{in0}, \texttt{in1}, \texttt{in2}, \texttt{flag}\}$, $\text{VAR}_M = \{\texttt{var}, \texttt{out}\}$ and $\text{VAR}_T = \{\}$ and the domain of discourse is `BOOL` $\cup$ `USINT` $\cup$ `UDINT`. The program model of the program would contain, e. g., the transition $(19, \langle out = 1, var = 0, \dots \rangle) \sim (20, \langle out = 0, var = 0, \dots \rangle)$.

Since the state of a program is given by an assignment to all variables, the number of possible states explodes exponentially in the number of variables. A symbolic encoding of transition systems allows us to represent huge — potentially infinite — sets of states by describing them using some language. In this work, we use the first-order logics to represent sets of states, where each variable represents a declared variable of the program.

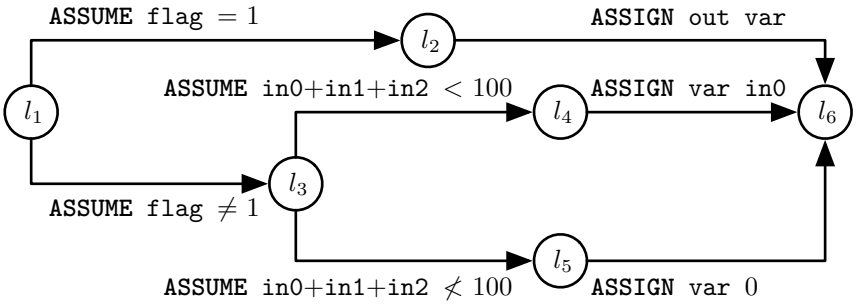


Fig. 2. Control-flow automaton of program `Example`

3.2 From PLC Programs to FOL Formulae

Our transformation is performed in two steps. First, we translate the program statements into a control-flow automaton [3] which itself uses an intermediate representation of the program statements. Then, we will transform the automaton into formulae.

Definition 4 (Control-flow automaton). *A control-flow automaton (CFA) is a labeled state transition system* $\langle L, \textsc{Ops}, G\rangle$ *where* L *is a set of program locations,* Ops *is a set of operations over the memory and* $G \subseteq L \times \textsc{Ops} \times L$ *is a set of control-flow edges.*

Each $\ell \in L$ represents a location in the program. The semantics of a control-flow edge $\langle \ell, \cdot, \ell' \rangle$ is that if the current state is in location ℓ, then the next state will be in location ℓ' after the execution of the operation. An operation is given as one of the following instructions:

- `ASSUME` π
- `ASSIGN` $x\ t$

The operation *assume* is followed by a predicate application π with positive $p(t_1, \ldots, t_n)$ or negative $\bar{p}(t_1, \ldots, t_n)$ polarity and it means that the transition is executed if and only if the given predicate application is valid or not valid, respectively. The operation *assign* is followed by a variable x and a term t of the same type and it means that in the next state the variable x will be assigned with the evaluation of the expression t in the current state. A term t can be a variable or a function application:

Definition 5 (Term). *The rules for forming a term* t *are defined by the following abstract grammar:*

$$t ::= \quad x \quad | \quad f(t_1, \ldots, t_n)$$

where $x \in \textsc{Var}$ *is a variable and* f *is a function symbol.*

Note that constants are applications of functions with arity 0. We allow all standard PLC operators as functions, which also includes casts to different scalar data types.

Example 2. Consider the program `Example` of Fig. 1 and the memory defined as in Ex. 1. The labeled state transition system of Fig. 2 represents the translation into a control-flow automaton as in Def. 4. For simplicity we represent binary predicate and function applications using infix notation, which is interpreted according to the standard operator precedence, e. g., $\texttt{in0} + \texttt{in1} + \texttt{in2} < 100$ stands for $<(+(+(\texttt{in0}, \texttt{in1}), \texttt{in2}), 100)$.

In order to be manipulated by SMT solvers, memory operations must be translated into quantifier-free first-order logic formulae that encode in some way assumptions and transformations over the memory. Quantifier-free first-order logics formulae can be expressed in terms of predicate applications, negations and conjunctions.

Definition 6 (Quantifier-free Formula). *The formation rules of a formula φ are defined by the following abstract grammar:*

$$\varphi ::= \quad p(t_1, \ldots, t_n) \quad | \quad \neg\varphi \quad | \quad \varphi_1 \wedge \varphi_2$$

where p is a predicate symbol and $t_1, \ldots, t_n$ are terms (cp. Def. 5).

Operations are encoded as cubes over equalities between *post-variables* and terms over *pre-variables*, representing valuations of pre- and post-states of the transition. Pre- and post-variables are denoted by the set of unprimed and primed symbols respectively. Let $\mathcal{L}$ be the language of first-order logic. We define an operation encoder as a function $\epsilon\colon \textsc{Ops} \to \mathcal{L}$. According to its semantics, the encoding of an *assume* operation is given by the assertion of the predicate on the pre-variables conjoined with the equality between all pre- and post-variables, which remain unchanged:

$$\epsilon(\texttt{ASSUME}\ p(t_1, \ldots, t_n)) \quad \triangleq \quad p(t_1, \ldots, t_n) \wedge \textstyle\bigwedge_{y \in \textsc{Var}} y' = y \ ,$$
$$\epsilon(\texttt{ASSUME}\ \bar{p}(t_1, \ldots, t_n)) \quad \triangleq \quad \neg p(t_1, \ldots, t_n) \wedge \textstyle\bigwedge_{y \in \textsc{Var}} y' = y \ .$$

An assignment is encoded by asserting the equality between the post-variable that has to be assigned with the term over the pre-variables, conjuncted with equalities between pre- and post- versions of the variables that have to remain unchanged, which are all except for the assigned one:

$$\epsilon(\texttt{ASSIGN}\ x\ t) \quad \triangleq \quad x' = t \wedge \textstyle\bigwedge_{y \in \textsc{Var} \setminus \{x\}} y' = y \ .$$

A first-order logic control-flow-based symbolic encoding of a PLC program is then given by the following:

- $\langle \textsc{Var}, \mathcal{D} \rangle$ the memory and $\textsc{Var}_I$, $\textsc{Var}_M$, $\textsc{Var}_T$ the variable classes,
- $\langle L, \mathcal{L}, G \rangle$ the CFA where all operations in Ops are encoded into the first-order logic language $\mathcal{L}$ through the encoder ϵ,
- $\mathit{Start} \in L \times \mathcal{L}$ the start-up phase and
- $\mathit{Scan} \in L \times \mathcal{L}$ the scanning phase.

The *start-up phase* defines the initialization in the initial location ℓ_a:

$$\mathit{Start} \triangleq \langle \ell_a, \textstyle\bigwedge_{x \in \textsc{Var} \setminus \textsc{Var}_I} x = \mathsf{Init}_x() \rangle$$

where $\mathsf{Init}_x()$ is the constant x is initialized to. The *scanning phase* defines the behavior of the controller after the execution of the program body after it reaches the last program location $\ell_z \in L$. During this phase the values of the variables in $\textsc{Var}_M$ are retained while input variables are read from the environment, hence their value is non-deterministic:

$$\mathit{Scan} \triangleq \langle l_z, \textstyle\bigwedge_{x \in \textsc{Var} \setminus \textsc{Var}_I} x' = x \rangle$$

We obtained a symbolic encoding of a program $\langle S, I, R \rangle$ as in Def. 3 that can be handled with SMT solving techniques [2]. Given a theory $\mathcal{T}$ chosen for the interpretation of variables and predicates, the set of states is defined as the set of all locations and all possible assignments consistent under $\mathcal{T}$:

$$S \triangleq \{ \langle \ell, \nu \rangle \mid \ell \in L \text{ and } \nu \in \textsc{Var} \to \mathcal{D} \}$$

The set of initial states is defined as all those states in the start-up location and variables are initialized:

$$I \triangleq \{ \langle \ell_a, \nu \rangle \mid \nu \models_{\mathcal{T}} \varphi \text{ and } \mathit{Start} = \langle \ell_a, \varphi \rangle \}$$

The transition relation is given by pairs of states over consecutive locations such that the assignment of the first state over unprimed variables ν_1 and the assignment of the second state over primed variables ν_2' together satisfy the formula. Moreover, all transitions are those pairs of states from the last to the first location that satisfy the scanning phase:

$$\begin{aligned} R \triangleq {} & \{ \langle \langle \ell, \nu_1 \rangle, \langle \ell', \nu_2 \rangle \rangle \mid \nu_1, \nu_2' \models_{\mathcal{T}} \varphi \text{ and } \langle \ell, \varphi, \ell' \rangle \in G \} \cup \\ & \{ \langle \langle \ell_z, \nu_1 \rangle, \langle \ell_a, \nu_2 \rangle \rangle \mid \nu_1, \nu_2' \models_{\mathcal{T}} \varphi \text{ and } \mathit{Scan} = \langle \ell_z, \varphi \rangle \text{ and } \mathit{Start} = \langle \ell_a, \cdot \rangle \}. \end{aligned}$$

3.3 Encoding of Timers

PLC programs are allowed to use the timer FBs `TP`, `TON` and `TOF`, which by definition have continuous behavior, which cannot be encoded by the default operation set. For this reason, we augment Ops with these operations to represent timer calls:

– $\mathtt{TP}\ n\ t_{\mathtt{IN}}\ t_{\mathtt{PT}}$, – $\mathtt{TON}\ n\ t_{\mathtt{IN}}\ t_{\mathtt{PT}}$, – $\mathtt{TOF}\ n\ t_{\mathtt{IN}}\ t_{\mathtt{PT}}$,

where $n \in$ TIMER is the name of the timer and $t_{\mathtt{IN}}$ (timer input) and $t_{\mathtt{PT}}$ (programmed time) are terms. For each timer $n \in$ TIMER the variables $n.\mathtt{IN}$ (timer input), $n.\mathtt{Q}$ (timer output) are added to VAR. For timers of type TON and TOF the propositional variable $n.r$ is added to VAR, which keeps track whether the timer is running. None of those variables is added to any variables class.

The rising and falling edges on the input and the relative instant behavior are encoded at the call site. Timer TP starts (setting Q to 1) if Q is 0 and there is a rising edge on input IN; nothing is changed otherwise:

$$\begin{aligned}\epsilon(\mathtt{TP}\ n\ t_{\mathtt{IN}}\ t_{\mathtt{PT}}) \quad \triangleq \quad & n.\mathtt{IN}' = t_{\mathtt{IN}} \wedge \textstyle\bigwedge_{x \in \mathrm{VAR} \setminus \{n.\mathtt{IN}, n.\mathtt{Q}\}} x' = x \quad \wedge \\ & ((n.\mathtt{Q} = 0 \wedge n.\mathtt{IN}' > n.\mathtt{IN}) \rightarrow n.\mathtt{Q}' = 1) \quad \wedge \\ & (\neg(n.\mathtt{Q} = 0 \wedge n.\mathtt{IN}' > n.\mathtt{IN}) \rightarrow n.\mathtt{Q}' = n.\mathtt{Q})\end{aligned}$$

Timer TON starts on rising edges of IN and stops (setting Q to 0) on falling edges:

$$\begin{aligned}\epsilon(\mathtt{TON}\ n\ t_{\mathtt{IN}}\ t_{\mathtt{PT}}) \quad \triangleq \quad & n.\mathtt{IN}' = t_{\mathtt{IN}} \wedge \textstyle\bigwedge_{x \in \mathrm{VAR} \setminus \{n.\mathtt{IN}, n.\mathtt{Q}, n.r\}} x' = x \quad \wedge \\ & (n.\mathtt{IN}' > n.\mathtt{IN} \rightarrow n.r' \wedge n.\mathtt{Q}' = n.\mathtt{Q}) \quad \wedge \\ & (n.\mathtt{IN}' < n.\mathtt{IN} \rightarrow \neg n.r' \wedge n.\mathtt{Q}' = 0) \quad \wedge \\ & (n.\mathtt{IN}' = n.\mathtt{IN} \rightarrow n.r' \leftrightarrow n.r \wedge n.\mathtt{Q}' = n.\mathtt{Q})\end{aligned}$$

Timer TOF starts on falling edges and stops setting Q to 1 on rising edges and is defined similar to TON. At the start-up phase and the scanning phase are augmented to encoded initialization and time elapsing, respectively. Timers are initially disabled and with all values set to 0. Timer TP *may* have a falling edge on Q if running, it remains disabled with Q set to 0 otherwise:

$$\mathcal{TE}_{\mathtt{TP}}\ (n) \triangleq n.\mathtt{IN}' = n.\mathtt{IN} \wedge n.\mathtt{Q} = 0 \rightarrow n.\mathtt{Q}' = 0$$

Timer TON (TOF) either remains unchanged or *may* have a rising (falling) edge on Q if running:

$$\mathcal{TE}_{\mathtt{TON}}(n) \triangleq n.\mathtt{IN}' = n.\mathtt{IN} \wedge n.r' \leftrightarrow n.r \wedge (n.\mathtt{Q}' = n.\mathtt{Q} \vee n.r \wedge n.\mathtt{Q} = 0 \wedge n.\mathtt{Q}' = 1)$$
$$\mathcal{TE}_{\mathtt{TOF}}(n) \triangleq n.\mathtt{IN}' = n.\mathtt{IN} \wedge n.r' \leftrightarrow n.r \wedge (n.\mathtt{Q}' = n.\mathtt{Q} \vee n.r \wedge n.\mathtt{Q} = 1 \wedge n.\mathtt{Q}' = 0)$$

3.4 Summarization of Control-Flow Automata

The obtained control-flow automaton contains one transition for every instruction of the intermediate language. This kind of encoding is called *single-block encoding* (SBE) and it has two drawbacks: First, one has to compute the evaluation of all formulae at each intermediate step, even if these evaluations bear no further meaning. Secondly, the conjunction of intermediate steps (formulae) might be easier to evaluate using automated decision procedures than each step

on its own—it might even be more precise. Such a conjunction of simple intermediate steps is called *basic-block encoding* (BBE). This idea can further be improved to control flow trees, which then is called *extended-basic-block encoding* (EBBE), and even loop-free fragments, called *large-block encoding* (LBE) [3]. We use a BBE in our approach.

4 Predicate Abstraction

Let $P = \{\pi_1, \ldots, \pi_n\}$ a set of predicates over the set of concrete variables VAR, which we call the *abstraction precision*. The Boolean predicate abstraction of a system constitutes computing an over-approximation which keeps track where each of the predicates in P is valid or not. An abstract state is defined as a pair $\langle \ell, c \rangle$ of location ℓ and a minterm c over a set $B = \{b_1, \ldots, b_n\}$ of only propositional variables. A minterm over B is a conjunction of all variables $b_i \in B$, each of them occurring either with positive b_i or with negative $\neg b_i$ polarity. We define the *abstraction function* α of a concrete state $\langle \ell, \nu \rangle$ as the abstract state $\langle \ell, c \rangle$ in which the polarity of each variable in c states the validity of the respective predicate in ν:

$$\alpha(\langle \ell, \nu \rangle) \triangleq \langle \ell, c \rangle \text{ s.t. } c \models b_i \text{ iff } \nu \models_{\mathcal{T}} \pi_i \text{ for each } 1 \leq i \leq n$$

The abstraction function creates an over-approximation, meaning that the abstraction of a state s represents a region of states in which s is contained, i. e., $s \in (\alpha \circ \alpha^{-1})(s)$. Note that α is a surjection, hence we are abusing the notation defining its inverse as $\alpha^{-1}(\hat{s}) \triangleq \{s \mid \hat{s} = \alpha(s)\}$.

We are interested in verifying universal properties and we want the abstraction to be conservative for such properties. This is guaranteed by assuring all predicates of the property to be contained in the precision. In this way, the abstraction of the region of states S_ϕ in which the property ϕ is satisfied is represented with the highest precision possible, i. e., $S_\phi = \alpha \circ \alpha^{-1}(S_\phi)$. Since we are over-approximating the system, i. e., $S \subseteq \alpha \circ \alpha^{-1}(S)$, we have that if we prove the set of reachable states $S^{\rightarrow}$ to satisfy the property in the abstract system, then it is valid in the concrete system as well: $\alpha(S^{\rightarrow}) \subseteq \alpha(S_\phi) \implies S^{\rightarrow} \subseteq S_\phi$. In general, the vice-versa does not hold, which gives rise to counterexample guided abstraction refinement (CEGAR) techniques [6].

4.1 Predicate Abstraction with Arcade.PLC

We extended the tool ARCADE.PLC. Its architecture consists of two main components: *model checker* and *simulator*. The model checker has the task of verifying ∀CTL properties on explicit Kripke models. The predicate abstraction allows us to represent a program $\langle S, I, R \rangle$ as a Boolean over-approximation in terms of a Kripke Model $\langle \hat{S}, \hat{I}, \hat{R}, \hat{AP}, \hat{L} \rangle$ where

- $\hat{S}$ is the set of states s. t. $\hat{S} \triangleq \{\alpha(s) \mid s \in S\}$,
- $\hat{I} \subseteq \hat{S}$ is the set of initial states s. t. $\hat{I} \triangleq \{\alpha(s) \mid s \in I\}$,

- $\hat{R} \subseteq \hat{S} \times \hat{S}$ is the transition relation s. t. $\hat{R} \triangleq \{\langle \alpha(s), \alpha(s') \rangle \mid \langle s, s' \rangle \in R\}$,
- $\hat{AP}$ is the set of atomic propositions s. t. $\hat{AP} \subseteq B$ and
- $\hat{L} \colon \hat{S} \to 2^{\hat{AP}}$ is the labeling function s. t. $\hat{L}(\langle \ell, c \rangle) \triangleq \{b \in \hat{AP} \mid c \models b\}$.

The simulator has the task of generating such Kripke model on-the-fly by providing two main primitives: *precondition* and *strongest postcondition*. The precondition $p \subseteq \hat{S}$ corresponds to the set of abstract initial states $\hat{I}$ while the strongest postcondition $sp \colon \hat{S} \to 2^{\hat{S}}$ corresponds to the set of successors of an abstract state under the abstract transition relation $\hat{R}$. Both $\hat{I}$ and $\hat{R}$ are defined as application of the abstraction function over sets. Given a fixed location, this can be characterized as the enumeration of all minterms over B that are $\mathcal{T}$-satisfiable when conjoined with the set and the *abstraction constraint* $\bigwedge_{i=1}^{n}(b_i \leftrightarrow \pi_i)$.

The precondition is defined as the set of abstract states $\langle \ell_a, c \rangle$ at start location which minterms abstract the start condition:

$$p \triangleq \{\langle \ell_a, c \rangle \mid \langle \ell_a, \varphi \rangle = \mathit{Start} \text{ and } c \wedge \varphi \wedge \textstyle\bigwedge_{i=1}^{n}(b_i \leftrightarrow \pi_i) \text{ is } \mathcal{T}\text{-SAT}\}.$$

The strongest postcondition of an abstract state $\langle \ell, c_1 \rangle$ is defined as the set of abstract states $\langle \ell', c_2 \rangle$ at successor locations such that the minterms c_1 and c_2 abstract the transition formula on the pre- and post-variables respectively:

$$sp(\langle \ell, c_1 \rangle) \triangleq \\ \{\langle \ell', c_2 \rangle \mid \langle \ell, \varphi, \ell' \rangle \in G \text{ and } c_1 \wedge c_2' \wedge \varphi \wedge \textstyle\bigwedge_{i=1}^{n}(b_i \leftrightarrow \pi_i \wedge b_i' \leftrightarrow \pi_i') \text{ is } \mathcal{T}\text{-SAT}\}.$$

Computing them reduces to an *AllSAT* problem over a set of important variables [15] that are B for the precondition and B' for the strongest postcondition. Considering the second, in our implementation we iterate among outgoing transitions $\langle \ell, \varphi, \ell' \rangle \in G$ explicitly. For each location ℓ' we query the Z3 SMT solver [8] for models of the formula. From the model we extract a minterm c_2' with which we instantiate an abstract state $\langle \ell', c_2 \rangle$. After that we conjoin the blocking clause $\neg c_2'$ to the formula and repeat the process until unsatisfiable. When iterating among different outgoing transitions directed to the same location ℓ' blocking clauses are maintained in order to avoid double occurrences. The result is memoized.

Example 3. To verify (1) from Sect. 2 we start with $P = \{\pi_1 = (\texttt{out} < 100)\}$ and the property we want to verify thus becomes $AG\, b_1$ (ignoring the exit point for the presentation). Since we do not have any restriction on `var`, we get a counterexample where at the end $\texttt{var} \geq 100$ and is assigned to `out`. This gives rise to the predicate $\pi_2 = (\texttt{var} < 100)$ which we add to P and rerun the process. In the next refinement step we similarly detect: $\texttt{in}_0$ is assigned `var`, hence we deduce $\pi_{\text{skip}} = (\texttt{in}_0 < 100)$ (we skip this predicate to make our presentation more accessible). Finally, we discover that the previous statement can only be executed if $\pi_3 = (\texttt{in}_0 + \texttt{in}_1 + \texttt{in}_2 < 100)$ is satisfied (from ℓ_3 to ℓ_4), so π_3 is added to P. The abstracted state space is shown in Fig. 3 (where each b_i represents the validity of π_i) and it allows us to verify (1).

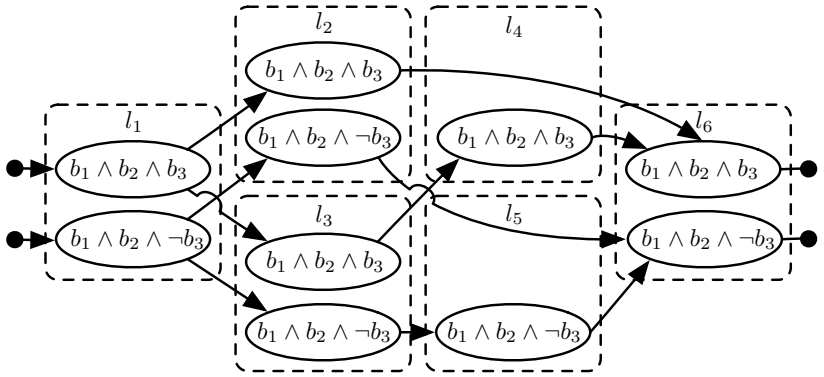


Fig. 3. Predicate abstraction of `Example`. Solid circles on the left and right indicate the scanning phase and are connected by transitions.

4.2 Scoping of Predicates

Thus far, we evaluated all predicates in every location, which is potentially wasteful: Consider the example again with notation as in Ex. 3 and Fig. 3. For the initial location ℓ_1, we have to consider the two states $(b_1 \wedge b_2 \wedge b_3)$ and $(b_1 \wedge b_2 \wedge \neg b_3)$. Note that the predicate π_3 (and thus the evaluation b_3) is of no use in the initial state but only in ℓ_3. In particular it also pollutes the path through ℓ_2. In this section we will therefore reduce the scope of certain predicates and first define:

Definition 7 (Weak Reachability). *Let $\langle L, \cdot, G\rangle$ a control-flow automaton. The weak reachability relation $\preceq \subseteq L \times L$ is defined as follows:*

$$\ell \preceq \ell'' \text{ iff } \ell = \ell'' \text{ or exists } \langle \ell, \cdot, \ell'\rangle \in G \text{ s. t. } \ell' \preceq \ell''.$$

Two locations are weakly reachable if there is a path of locations between them. This path does not consider the transition over data variables, hence two weakly reachable locations could be not actually reachable in a real execution. We associate to each predicate π_i a scope $\langle \check{\ell}_i, \hat{\ell}_i\rangle \in L \times L$ and we redefine the abstraction function in such a way that predicates are used only if they are in the given scope:

$$\alpha(\langle \ell, \nu\rangle) \triangleq \langle \ell, c\rangle \text{ s.t. } c \models b_i \text{ iff } (\check{\ell}_i \preceq \ell \preceq \hat{\ell}_i \implies \nu \models_{\mathcal{T}} \pi_i) \text{ for each } 1 \leq i \leq n.$$

We use the weakest preconditions to automatically limit the scope for new predicates. If we have a sequence of consecutive preimages with common predicate $\langle \ell_1, \varphi_1\rangle, \ldots, \langle \ell_m, \varphi_m\rangle$, those predicates will use the scope $\langle \ell_1, \ell_m\rangle$. If this sequence passes through the scanning phase, we break it up into two different predicates with scope $\langle \ell_1, \ell_z\rangle$ and $\langle \ell_a, \ell_m\rangle$, respectively. If the sequence passes through the scanning phase more than once, we do not scope.

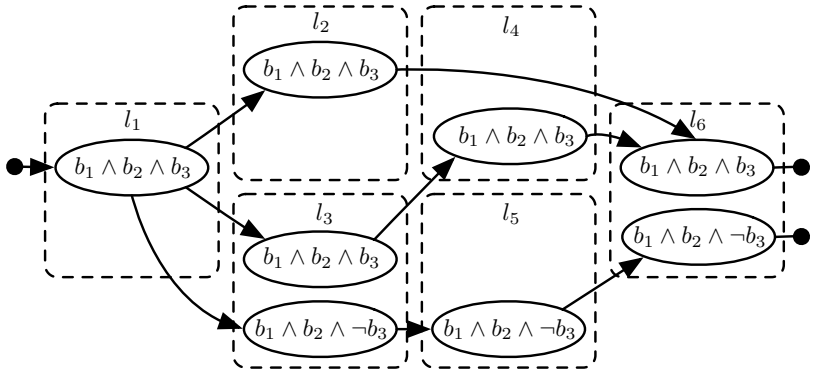


Fig. 4. Predicate abstraction with b_3 scoped to $\langle \ell_3, \ell_6 \rangle$. Solid circles on the left and right indicate the scanning phase and are connected by transitions.

Example 4. In our example we would associate the predicate scope $\langle \ell_3, \ell_6 \rangle$ to π_3. This means that, e. g., in state ℓ_1 the variable b_3 is fixed to 1, hence only state $b_1 \wedge b_2 \wedge b_3$ appears there, further reducing the number of states and transitions. The complete state space is depicted in Fig. 4.

5 Case Study

We implemented our approach in the ARCADE.PLC framework[1] which already offered the necessary parsers and the translation into intermediate code for PLC programs written in *structured text*. To show the effectiveness of our approach beyond the running example, we applied it to various FBs from industrial and academic background and compared it to the existing approach. All experiments were performed on a MacBook Pro equipped with an Intel Core i5 processor with 2.53 GHz and 8 GB of main memory.

Programs. For the first experiments, we selected two complex safety-critical FBs proposed by the PLCopen [18] consortium. Since these FBs are defined in terms of state machines without providing an actual implementation we used our own implementation. The first FB `SF_ModeSelector` has 14 inputs, 12 outputs and 5 internal variables and is implemented in 175 lines of structured text. It controls that (up to eight) different modes of operation of a machinery are selected in a consistent way, i. e., that at most one mode is active at a time and only for a short period while switching no mode is selected. It additionally allows the locking of modes. We verified that (1) at most one mode is selected at a time and that (2) exactly one mode is selected if it is locked. Furthermore, we verified the `SF_MutingPar` FB which allows for muting a safety function while monitoring

[1] Aachen Rigorous Code Analysis and Debugging Environment for PLCs: `http://arcade.embedded.rwth-aachen.de`

Table 1. Evaluation with ARCADE.PLC

Program	Formula	Abs.	#loc	#states	#trans.	#P	t_{abs}	t_{total}
`Example`	`out` < 100	—	22	>4k	>40M	n/a	OOM	OOM
`Example`	`out` < 100	PA	22	40	19	4	1 s	1 s
`Example`	`out` < 100	PA+PS	22	10	13	5	1 s	1 s
`SF_ModeSelector`	(1)	—	190	18,759	6.3M	n/a	370 s	370 s
`SF_ModeSelector`	(1)	PA+PS	190	95	142	1	1 s	1 s
`SF_ModeSelector`	(2)	PA	190	>27k	>28k	>40	OOM	OOM
`SF_ModeSelector`	(2)	PA+PS	190	214	291	30	2 s	72 s
`SF_MutingPar`	(3)	PA+PS	377	442	727	1	2 s	2 s
`SF_MutingPar`	(4)	PA+PS	377	>10k	>14k	>100	OOM	OOM
`SafetyFunction`	(5)	PA+PS	442	1,481	2,210	4	4 s	4 s

that certain safety sensors are operated in the correct order. It has 13 inputs and 12 internal variables. We first verified that the FB only signals *Ready* when it is activated (3). Afterwards we tried to verify that a certain safety output (AOPD) is only set when the muting lamp is switched on (4). Finally, we verified a safety applications implemented using PLCopen safety function blocks. The safety application was taken from Soliman and Frey [20] and comprises four FB instances. We were able to verify that the safe stop output is only assumed if it is requested and acknowledged (5).

Evaluation. The results are shown in Tab. 1; the columns indicate: the program, the formula checked, the abstraction used ("—" = plain ARCADE.PLC with interval and bit set abstraction, PA = predicate abstraction, PS = predicate scoping), the number of states in the model, the number of transitions, the number of predicates used, the time for generating the abstract state space and model checking (where OOM means out of memory) and the total runtime (including predicate discovery). Our approach is clearly faster than the old approach and sometimes — as for the example program — even necessary to get a result. The predicate scoping reduces the state space further: We were not able to verify formula (2) without predicate scoping. This example also shows the force of this abstraction: Although 30 predicates were in use, the final state space comprised only 214 states. The muting FB shows that sometimes simple invariants can be proven using a single predicate as in (3). Yet, our approach still not scales well enough to prove (4). Finally, formula (5) shows how our approach can in principle be used to verify complex safety functions consisting of multiple blocks.

Regarding the runtime, we can observe that the actual model-checking process is performed in seconds even for the most complex programs. If the initial abstraction is not sufficient, refinement steps are necessary, which can be quite costly as shown with formula (2) where this takes 70 s of the total runtime. This predicate discovery seems to be the limiting factor of our current approach.

6 Related Work

Model Checking of PLC Programs. To the best of our knowledge, Moon [16] was the first to work on the formal verification of PLC programs. He translates PLC programs written in a limited subset of *ladder diagram* into the input language of the model checker SVM. This general approach, i. e., rewriting of PLC semantics into the input language of existing model checkers, have been used in numerous paper later on. In 2007, e. g., Pavlovic et al. [17] used NUSMV to verify PLC programs written in a vendor-specific dialect of *instruction list*. This approach, however, required manual insertions of invariants to the model to become feasible. NUSMV was also used by Gourcuff et al. [9] to verify *structured text*, although their approach only supports Boolean variables and a limited subset of control structures. In a second work [10], they made use of abstractions by (a) introducing a notation of observable and unobservable states (induced by the cyclic scanning mode) and (b) abstracting away variables that are overwritten before being used again. Our predicate scoping can be seen as a similar approach although implemented in a much more general way. Schlich et al. [19] then applied direct model checking to PLC programs. This approach, which creates the model using abstract simulation, was later augmented with a CEGAR refinement loop by Biallas et al. [4]. Our work can be seen as a continuation of this line of research.

Predicate Abstraction. Graf and Saïdi [11] showed how to derive abstract state spaces using decision procedures. Their approach works by adding all derived transitions as blocking clauses until a formula becomes unsatisfiable. Numerous works refined this approach in different directions. Ball et al. [1], e. g., make use of abstraction interpretation [7] for C code verification so as to derive the successors of multiple (unrelated) predicates in one decision procedure call. This also allows the representation of *don't cares* for the predicate evaluation. Henzinger et al. [12], on the other hand, introduce a lazy abstraction scheme, which works by using a different precision for different parts of the program which is deeply ingrained in their refinement loop. Our predicate scoping technique can be seen as a special case of these approaches, tailored for the cyclic scanning mode of PLCs.

7 Conclusion

We advocate a predicate abstraction for PLC programs, which is used to verify $\forall$CTL formulae. We demonstrated that the technique is effective for proving properties in safety-critical programs and operates usually in the order of seconds or minutes. Yet, more powerful techniques seems to be required for the most complicated function blocks we looked at, especially for deriving new predicates.

Acknowledgement. Sebastian Biallas is supported by the DFG. Further, this work is supported by the DFG Research Training Group 1298 *Algorithmic Synthesis of Reactive and Discrete-Continuous Systems* (AlgoSyn) and by the DFG Cluster of Excellence on Ultra-high Speed Information and Communication (UMIC), German Research Foundation grant DFG EXC 89.

References

1. Ball, T., Podelski, A., Rajamani, S.K.: Boolean and cartesian abstraction for model checking C programs. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 268–283. Springer, Heidelberg (2001)
2. Barrett, C., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. Handbook of Satisfiability 185, 825–885 (2009)
3. Beyer, D., Cimatti, A., Griggio, A., Keremoglu, M., Sebastiani, R.: Software model checking via large-block encoding. In: Formal Methods in Computer-Aided Design, FMCAD 2009, pp. 25–32. IEEE (2009)
4. Biallas, S., Brauer, J., Kowalewski, S.: Counterexample-guided abstraction refinement for PLCs. In: Proceedings of SSV, pp. 2–9. USENIX Association, Berkeley (2010)
5. Biallas, S., Brauer, J., Kowalewski, S.: Arcade.PLC: A verification platform for programmable logic controllers. In: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, pp. 338–341. ACM (2012)
6. Clarke, E.M., 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)
7. 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)
8. de Moura, L., Bjørner, N.S.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
9. Gourcuff, V., De Smet, O., Faure, J.M.: Efficient representation for formal verification of PLC programs. In: Proceedings WODES, pp. 182–187 (2006)
10. Gourcuff, V., De Smet, O., Faure, J.M.: Improving large-sized PLC programs verification using abstractions. In: Proceedings of the 17th IFAC World Congress, pp. 5101–5106 (2008)
11. Graf, S., Saïdi, H.: Construction of Abstract State Graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)
12. Henzinger, T., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: POPL, pp. 58–70. ACM Press (2002)
13. International Electrotechnical Commission: IEC 61131: Programmable Controllers. International Electrotechnical Commission, Geneva, Switzerland (1993)
14. International Electrotechnical Commission: IEC 61508: Functional Safety of Electrical, Electronic and Programmable Electronic Safety-Related Systems. International Electrotechnical Commission, Geneva, Switzerland (1998)
15. Lahiri, S.K., Nieuwenhuis, R., Oliveras, A.: SMT techniques for fast predicate abstraction. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 424–437. Springer, Heidelberg (2006)
16. Moon, I.: Modeling programmable logic controllers for logic verification. IEEE Control Systems Magazine 14(2), 53–59 (1994)
17. Pavlovic, O., Pinger, R., Kollmann, M.: Automated formal verification of PLC programms written in IL. In: VERIFY. Workshop Proce, vol. 259, pp. 152–163. CEUR-WS.org (2007)

18. PLCopen TC5: Safety Software Technical Specification, Version 1.0, Part 1: Concepts and Function Blocks. PLCopen, Germany (2006)
19. Schlich, B., Brauer, J., Wernerus, J., Kowalewski, S.: Direct model checking of PLC programs in IL. In: Proceedings of DCDS, pp. 28–33 (2009)
20. Soliman, D., Frey, G.: Verification and validation of safety applications based on PLCopen safety function blocks. Control Engineering Practice 19(9), 929–946 (2011); special Section: DCDS 2009 — The 2nd IFAC Workshop on Dependable Control of Discrete Systems

High-Level Guidance for Managers Deploying Formal Methods in Their Organisation

Christophe Ponsard, Jean-Christophe Deprez, and Renaud De Landtsheer

CETIC, rue des frères Wright 29/3, Charleroi, Belgium
{cp,jcd,rdl}@cetic.be

Abstract. Developing complex critical software should require proper validation with regards to requirements as well as showing a high level of certainty on correctness of the resulting system. While formal methods (FM) have a large potential to address these two challenges, their current Industry adoption is still hampered by a number of hurdles of technical and organizational natures. Furthermore, many misconceptions (myths) about FM remain deeply anchored in Industry. To help to bring down these hurdles and myths, this paper presents evidence that FM can be successfully used in Industry. The evidence repository follows two strategy to present its content. First, a company-specific approach is used where success stories describe how a given company deployed FM in one or several of its development projects. Second, a more general approach identifies general questions of interest (FAQ) to many companies in various Industry sectors. Success stories and FAQs are made available using a public collaborative wiki-based website open to external contributions (http://www.fm4industry.org).

1 Introduction

Formal methods (FM) have been successfully applied in a number of industrial cases and are even broadly adopted in specific industrial fields. However, they are still not widely used in commercial software development, even in safety/security/business-critical sectors such as automotive, mass transport, aerospace or business information systems. The reasons for this situation have been studied at different points in time. A number of obstacles and often misconceptions ("myths") were identified such as the difficulty of using the underlying mathematics and logic framework, customers reluctance, adverse effect on project costs and duration, incompatibility with traditional development methods, lack of support and tools [5,14]. General guidelines (stated as "commandments") have also been suggested by [5]. Surveys such as the one in [24] highlight the need for collecting pieces of evidence to convince companies in adopting Formal Methods (FM) and to help them integrate FM in their development process.

The DEPLOY FP7 project (2008-2012) was initiated with the goal to improve the maturity of formal engineering methods to Industry expectation and to provide the necessary tools to increase Industry interest and convert them to using FM [10]. Many near-real world deployments of FM took place, initially with

C. Pecheur and M. Dierkes (Eds.): FMICS 2013, LNCS 8187, pp. 139–153, 2013.

the four industrial partners involved in DEPLOY (SAP, Robert Bosch Gmbh, Space System Finland and Siemens SAS IMO) and then with associate partners (XMOS, Critical Software Technologies Ltd and AeS Brasil). Other successful applications also took place among the industry group (STMicroelectronics) and FM-service partners (Systerel, ClearSy, CETIC, Formal Minds). In total seven major industrial domains where covered: Aeronautics, Automotive, Business Information Systems, Chip Design/Smartcards, Operating Systems, Space Systems and Mass Transport. As part of the project, a significant effort was devoted to collecting and structuring pieces of evidence identified in all those deployments with the goal to later help other companies in their decision in adopting FM and in deploying them in their development process.

The goal of this paper is to explain how the collected material is structured in a helpful format for Industry readers. Clearly, this should transitively help Academics to understand how to structure their research results to become more convincing to Industry. Prior to collecting data, the root of our effort noted that Industry readers are particularly receptive to hearing about others' cases, in other words, evidence that a considered formal method has been used successfully by others. However, efficient ways to organize and format pieces of evidence needed to be further explored in particular to determine if and how the FM targeted by DEPLOY were improved. Although pieces of evidence were also collected with the short term objective to help Industry partners of the DEPLOY project in their adoption process, a longer term target was to provide the collected material as evidence on FM to other companies not involved or connected to DEPLOY. To achieve this long term objective, it is important to pay attention not only to the content but also on how to deliver this content in an appropriate form for an Industry audience. A very important decision regarding the structuring of pieces of evidence observed that FM adoption takes place at different levels of an organisation thus, to gain acceptance by different Industry roles, it is important to ease (hence partition) the reading according to a role-based approach. A role not well addressed by many articles is the one of managers. Whether project managers, QA managers, and high-level managers who often make strategic decisions, their concerns are often not well understood. To address this issue, the evidence repository captured in DEPLOY is presented based on two general approaches: a company specific one based on success stories and a general approach based on frequently asked questions by various typical roles in Industry.

The project ended with a wiki-based website (`www.fm4industry.org`) combining a FAQ and inter-related collection of illustrating cases in specific domains with different ways to access to information (role-based, process-based, domain-specific,...). The global approach was published as part of the DEPLOY book [21]. In this paper, we recall the main points behind this evidence repository but the essential contribution is to provide a high-level guidance to managers in order to help them recognise key milestones in their endeavor to deploy FM in their organization and to show how they can utilize the evidence repository to answers their questions to reach each milestone.

This paper is organized as follows. First, the general method for collecting and presenting evidence material is presented. Second, different usage scenarios for management roles are proposed. Then, the three-phase management decision approach is presented. Related works are then discussed. Finally, methods and tools to set up evidence collection and presentation within an organization are proposed including how the organization can share and enrich its experience with others by contribution to the fm4industry evidence repository.

2 A Method for Collecting and Presenting Evidence

Success stories have shown to be a simple way of collecting information and also an efficient way to show evidence of successful Industry adoption. They often take the form of a white paper. In some cases, they are published in Industry tracks of conferences. The nature of such publication rarely allows for sufficient space. Consequently, success stories often lack details on the very specific context in which they took place, for instance, little information is usually provided about the overall innovation cycle of the enterprise involved in the success story, or on how researchers and production engineers collaborated during a transfer project.

In conclusion, traditional **success stories** provide an interesting base for building an evidence repository of Industry adoption but they must be augmented to provide readers with additional contextual information as well as links to other related effort. Consequently, a specific "success story" template was designed to enhance and structure the presentation of success stories emerging from conducted deployments. This template describes the case but also asks information on the context in which a FM success story developed. In particular, the Source section asks that an organization describes its structure and its research and innovation process. Usually, this context information can be lengthy. Hence it is recommended for the Source field to only provide the organization name and a link to another document that describes the full company context. In addition, the template also requires information about the application domain, the impact in terms of benefits/limitations, status (e.g. on-going work still being refined or whether it is finished work), and keywords for classification. Importantly, a field to capture cross-references to related FAQs and to external published information is available. This mechanism allows for the reader to navigate between general FAQ and company-specific success stories.

Success stories provide evidence of formal method transfer and usage in Industry from a very specific viewpoint. By their very specific nature, success stories do not however address high-level cross concerns of various Industry sectors or in many cases, they do not present information from which a general argument on applicability to other contexts can be easily deduced. This means that success stories must be complemented by another technique for presenting topics of general concerns to many readers from various Industries. During the DEPLOY project, researchers and Industry partners proposed to structure this cross-cutting information in a **Frequently Asked Question format (FAQ)**. Industry partners are accustomed to this type of format where generic questions

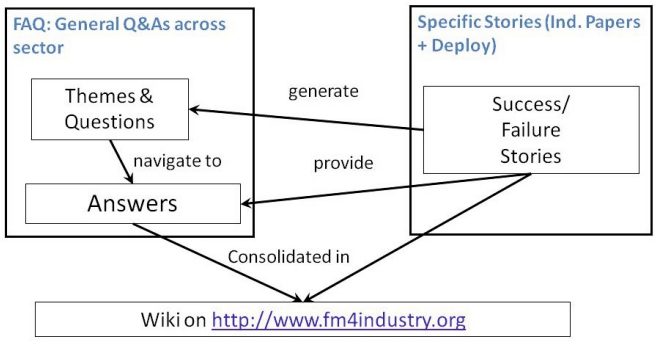


Fig. 1. Organisation of the Evidence repository

are answered in a fairly short space. The average length targeted for answers was about one page as illustrated in section 4. The resulting structure of our evidence repository is shown in Figure 1.

At this point, an approach is needed to identify and structure a list of common concerns from different Industry sectors. In order to identify common high-level concerns across domains, many documents provided by the Industry as well as by the academic researchers and the tool providers were analysed. The high-level topics identified are:

- **G** - General topics of concerns related to formal methods in Industry
- **TSP** - Impact on an organisation w.r.t Training Scope and Resourcing
- Understanding the impact on the Software/System Development Process
  - **QI** - Impact on quality of product (or work products) developed using formal methods
  - **EM** - Capability to Exploit formal Models at various stages of the development process
  - **CIF** - Capability to control the impact of introducing a formal method , for instance, by limiting the scope of who must understand and become an expert in a FM in a company
  - **R** - Capability to reuse across development projects when formal methods are used, including reuse of formal and proven artefacts
  - **MF** - Capability to phase the Migration to using a Formal method incrementally (given the existence of products not initially developed using formal methods)
- **TOOL** - Known strengths and weaknesses of tools associated to a formal method as well as the quality of support by tool providers
- **ExFac** - External factors advocating take-up (from competition, standard bodies, laws) of formal methods

Pieces of evidence are labelled with the notion of **Industry Roles** to specifically target the relevant types of audience, and increase the impact of answers. The organisations involved in deployments were quite different, ranging from

the technical SME with highly adaptable roles to large companies with a specific R&D unit and very well defined roles. Based on a careful study of the innovation cycle of the companies involved in DEPLOY and on the authors' experience in technology transfer with many Walloon companies, a set of key roles common in the conducted deployments were identified, namely, high-level managers, project and QA managers, engineers and technical analysts, and quality assurance staff. For a transfer project to be successful, many of these roles must be involved and supportive of the proposed transfer of research results. Thus, to obtain clear and concise answers, each FAQ always explicitly targets a single role. This does not necessarily mean that other roles will not be interested in the question but rather that the named role is the primary target hence the answer is provided to match with the expectation from the stated role. For instance, the wording of an answer targeted to engineers is different from answers to questions whose primary audience is high-level management. Thus, if a high-level manager is interested to read a FAQ answer primarily targeting engineers, he will understand that the answer is presented from the engineer's viewpoint. Making the targeted audience explicit, by stating their roles, definitely helps the reader.

In the context of transferring formal method engineering techniques to Industry, the following roles are important:

Table 1. Roles

Id	Name	Role description
HM	High-Level Managers	taking enterprise's strategic decisions and their financial impact
PQAM	Project and QA Managers	supervising people who actively use FM (in production or R&D), planning projects and performing safety analysis and more traditional QA activities
EA	Engineers and Analysts	People actively using formal methods and tools
QAP	QA Practitioners	people who must understand documents involving FM notations but don't need to develop the capabilities to produce them

At the presentation level, the most adequate and commonly used media is an Internet wiki which gives wide access to the material. Wiki technology enables feedback from the people accessing the material, from comments to more active contributions, with a degree of control/moderation that can be tuned. Many Open Source wiki implementations exist and we finally opted for mediawiki [19] which is very powerful through many extension plugins. The homepage is accessible at `http://www.fm4industry.org`.

3 Manager Scenarios Exploiting Shared Evidence Material

A few customary situations faced by professionals with manager roles in software-intensive system development in Industry are presented, along with the kind of questions about formal methods usually asked in each of these situations. Subsequently, a brief explanation shows how managers can browse the evidence repository to find elements of answers to their questions.

Scenario I - I am a High Level Manager Who Has Heard That Others in the Sector Have Successfully Used Formal Methods. My staff has no prior experience on formal method. My question is now "How should I proceed in the evidence repository to better understand if FM could also be used in my company?"

Scenario II - I am a Project Manager in an SME, Our Development Team Has Successfully Used Advanced Static Analysis Tools during the Debugging and Testing Phased of Product Development. However, in a recent project, important requirements remained implicit until the customer validation phase. Subsequently, a significant overhaul of the architecture was needed to handle these requirements. Static analysis was clearly of no help. My question is now "How should I proceed in the evidence repository to better understand how our development team could start using one or several formal methods earlier in the development lifecycle to increase chances to identify all important requirements?"

Scenario III - I am a product (line) manager. I have heard of several success stories on the use of formal methods but in my company's case, we already have well-tested components that fulfil their quality requirements. Our project mostly consists in configuring these existing components and integrating them. My question is "Does your evidence repository elaborate on the use of formal method to ease the integrating of components, even if these components were not developed using formal methods?".

These few situations traditionally faced by Industry helps to illustrate how to navigate in the evidence repository to find FAQ and success stories with information relevant to one's context. For each of the scenarios above, two navigation approaches are possible:

- **Top-Down Approach:** for general question about formal methods and how it could be answered for their specific organisational context. The way to search information is to identify the relevant themes in the FAQ and focus on the specific role in the answer, then possibly browse to a related success story as illustration or use pointers to the literature.
- **Bottom-Up Approach:** for domain specific concerns, one can start by browsing success stories and from there identify key issues and then navigate to relevant FAQ giving details on how to handle them, possibly for a specific role.

For example, considering scenario 1, a high level manager starts with a very broad enquiry. Thus, the manager should use a top-down approach for navigating in the evidence repository starting with the selection of themes. His initial concern relates the lack of current expertise on formal methods. Consequently, the whole theme on training would provide relevant information (TSP). Furthermore, the high level manager should be informed on the different ways to include formal methods in a development lifecycle such as CIF-HM-2 ("How do organisational procedures used in various system development lifecycle processes need to be adapted when formal methods are introduced?). From there he consider phasing issue such as MF-HM-1 ("Is it possible to migrate an existing

system iteratively to using a formal method?"). In addition, early in an enquiry, a high level manager will likely be interested in the general topic theme (G). Explanation for the other scenarios may be found in the DEPLOY book [21].

4 High-Level Guidance to Manager for FM Adoption

4.1 Overview

When considering the adoption of formal methods, managers have to take a number of key strategic decisions. In this section, we highlight a typical decision path as observed in most of the deployment experiments conducted during DEPLOY quite independently from the specific context of each company. The guidance focuses on the following three key milestones when attempting to introduce a new development method in an organisation.

- **Milestone 1 - Deciding Whether Going for FM.** This is a first go/no-go decision based on an analysis of the weaknesses of the current development process, identifying where and what FM could help (scope) and being able to have an idea of return on investment and impact on the existing processes.
- **Milestone 2 - Conducting a First Deployment.** In this phase, inefficiency hence significant overheads are expected due to the effort to train people, decide about which formal method and the best way to use it, the tool support, etc. In this step, heavy support by FM expert is required.
- **Milestone 3 and following - Next Deployments.** The second deployment is a key step as it will reveal the level of remaining overhead. The goal is to become more and more autonomous from external support, more efficient but also more predictable, especially considering the different effort distribution along the development process.

To reach each of these milestones, managers will raise a fairly similar set of questions. Below, a guidance is presented to help managers to exploit information from the evidence repository to answer their common questions to reach a milestone. Due to space limitation only three questions related to the Milestone 1 are further described below. Furthermore, to provide compact answers additional FAQs of interest are sometimes merely referenced using the fm4industry wiki reference style, that is, a category acronym followed by a role acronym, for example, [TSP-HM-1] refers to question 1 of the category "Training Scope and Pace" (TSP) for the role "High Level Managers" (HM). For the other two milestones, more details are available from our online wiki [12]. A document snapshot from April 2012 is also available as DEPLOY Deliverable [20].

4.2 What Is the Impact on the Process and People Involved?

A. High-Level Managers. The use of FM dramatically shifts the workload of a project towards the analysis phase: much more work is required in the analysis phase to develop formal models. Subsequently, much less effort is required during the testing phase. As such, managers need to:

- Adapt their personal workload metrics to estimate appropriately the costs and deviation of projects using formal-method engineering
- Adapt their personal progress monitoring metrics to monitor the proper progress of projects using formal-method engineering
- Adapt their go/no-go procedure to decide on the use (or not) of formal method in a project.

Estimating the cost factors in the context of formal method engineering requires some experience as described in [7] and factors involved in such go/no-go procedure are discussed in [22] and cover:

- **An obligation to use formal method**, for instance, because it is requested by the customer (see FAQ category ExFac about "External Factors Advocating Formal Method Adoption")
- **A Development Project Is Internal, or on a Shared-Risk Contract.** This is mainly to compensate for the possible low accuracy of cost and delay estimates in particular the first time (or even first few times) any new method is being used in an organisation.
- **The management is ready to face the cost shift** from late phase of the project to early phase. This can lead to an early red flagging of the project as being over schedule, and indirectly make it augment this deviation due to increased reporting requests from managers.

B. Project and QA Managers need to properly design the development process and select the most adequate formal method to ensure that:

- **Claims Proven Using a Formal Method Are Relevant:** One can prove many things about a model; one should ensure that what is proven is useful to the project, and that all what should be proven is actually proven, or discharged to another validation method
- **The Artefact on Which a Claim Is Proven Is the Most Adequate One:** One can prove different things on different artefacts. For instance, one can prove some behavioural properties on abstract state machines or on the programmatic source code. It is much easier to prove such claims on abstract state machines. This is a trade-off between cost (defects detected early are cheaper to correct and checking more abstract artefacts is simpler than checking concrete ones) and level of assurance (the later the verification is performed, the fewer opportunities there are to introduce defects in the process)
- **Abstractions Made to Produce a Model Are Verified and Valid:** Models introduce abstraction; one should ensure that the established proofs remain valid in the real world although they might be done on an abstract model. This is further discussed in the FAQ [G-EA-1].
- **The Level of Assurance That Is Delivered by the Provers Is Adequate:** Not all formal method deliver the same level of assurance. One should select the most appropriate formal method according to its specific context. This is discussed in details in [EM-PQAM-3]

– **The Formal Method Is Compliant with the Targeted Standard.** Some sectors require very high assurance, and might require proofs to be cross checked different redundant provers, or to rely only on certified provers. Some provers of the DEPLOY project are certified for some CENELEC level of assurance. This is discussed in [ExFac-HM-1].

C. Engineers and Analysts need to:

– **Develop Formal Artefacts:** starting from informal requirements, different formalisation strategies can be used, e.g. using semi-formal notations as intermediate step. The FM notation(s) should also be adapted to the domain and should be selected based on some pilot study. It is also advised to grow the model iteratively using decomposition, refinements strategies allowing the start verifying properties on less complex models.
– **Formally Define All the Claims to Be Proven:** These can typically be derived from requirements documents. Some formal methods natively include these claims, for instance if they are related to the proper use of programming language constructs such as in software code verification tools like Polyspace.
– **Prove the Claims on the Artefact:** the effort to prove can depend on many things: the chosen FM, the level of guidance that is required by the tooling, the run-time of the tooling, and, possibly, the intertwined development process where the model is gradually proven as it is developed. It
– **Exploit the Model in the Next Steps of the Development Process:** When a formal model has been developed, and some claims proven on it, this artefact should be exploited in the development process. For example, one may use an approach based on model refinement until automatic code generation becomes possible. Alternatively, a model formally proven can be used to generate tests inputs and expected results to use later to verify the executable system.

D. QA and Safety Engineers need to assess that:

– **Proven Claims:** the proven claims should be inspected for relevancy as well as the introduced abstractions (as agreed as process level) and the artefacts on which claims have been proven should be properly exploited in the downwards development process.
– **The Formal Methods Have Been Used in an Adequate Way.** For instance, an absurdity included in a model may automatically induce correctness. QA and safety engineers must therefore verify that no such absurdity have been injected in the model.
– **The Model is Structured in a Way to Ease Proofs.** Some proof technology reach higher percentages of proof automation if the model is developed according to some rules (avoid some type of constructs, avoid symmetries, etc.). Such verification can be performed by QA before the model is actually proven, to spare the time of engineers.

QA then amounts to checking that a formal model complies with the established guidelines. At Siemens transportation, the above tasks are under the responsibility of the safety engineers. QA is in charge of verifying that the developed models match some reference good practices that have been synthesized into a set of guidelines. This verification is performed before proofs are made. Some of these good practices are related to the three aforementioned bullets while others aims at improving the percentage of automated proving. This is how Siemens reported to be internally organized.

4.3 Can the Use of a Formal Method Be Hidden From Most of Development and Management Teams?

Before deciding of specific notations, it is important to decide about the visibility of the formal notation as it will impact the requirements on the training and tooling. In certain case, formal methods can be hidden from most of development and management teams except a few selected experts who will use them. If notations should be explicitly handled it is also possible to restrict the scope across the lifecycle to the production of specific artefacts. Let's elaborate those two possibilities.

A. Hiding Formal Methods. The adoption of formal methods is in some case easier if hidden behind some domain specific notations. For instance, in SAP they used an existing domain specific language as front-end language on the top of a formal tool. Developers continue working with their familiar notations and some formal verification can take place in the background. Such an approach as been successfully deployed by SAP as described in the related success story in the business information domain [12].

Certain formal methods are easier to hide in particular, methods where no user guidance is required. This scenario work when first, a fairly accurate formal model can be inferred from the input semi-formal models and second, the associate tooling works without requiring additional user input to perform a selected set of verifications. For instance, a success story in the transportation domain at Siemens showed dramatic productivity improvement of data consistency of transportation Model for verifying geographical and circuit data for underground systems [17].

However not all formal methods can be successfully hidden notably because they require user interaction and human intelligence to transform an input into a model associated to a given formal method. Risks and mitigation actions associated with the approach that hides formal methods are further discussed on a related FAQ [CIF-PQAM-1].

B. Keeping Formal Methods Confined. The idea is to select a specific development step or a specific artefact, and to deploy formal methods on this step or artefact exclusively. The goal is to be able to accomplish the work with a reduced team of selected experts.

A typical example of such setting would be a project where one develops embedded communicating devices. The communication protocol can be developed and verified e.g. for deadlocks through formal methods, then implemented by another team who must only understand the protocol as described by the formal models and not the properties it is supposed to enforce neither the proofs that have been built on these models.

Another example is the use of the Polyspace code analysis tool [18]. Polyspace is able to spot bugs in source code through static analysis. It however requires some expertise to properly understand and interpret the results of an analysis. Some companies employ a dedicated team of workers to run Polyspace on code developed by another team. This is a form of confinement.

It has also been assessed that the math behind formal methods can be hidden from engineers during their training, emphasizing the essence of formal methods instead of notations [15].

4.4 What Strategy Increases the Chance of FM Adoption?

Some approaches observed during DEPLOY are presented. Although different strategies are applied, the key is to keep the operational level in the loop.

Building an Early Partnership between R&D and Operations. This strategy is more adapted to larger organisations with a dedicated R&D department (e.g. it was followed by SAP). In order to build bridges on which transfers can take place at some point, partnership should be built early with development units and ideally should be motivated by needs from the development units. Interesting collaboration points are:

- collecting requirements
- getting early feedback
- conducting validation experiments

The organisation should explicitly support the process by providing required resources and collaboration instruments. The prioritization of the research effort should then be driven by the development team's priorities consequently, leading to R&D effort not focusing on researchers agenda but the one decided by users. For example, improving usability of a prototype becomes more important than investigating advanced concepts or delivering missing functionality. However this is the price to pay to convince development units that the resulting tools have reached the necessary technical maturity for a productive use.

Forcing Employee Turn-Over on R&D Projects Investigating Formal Methods. Another strategy more adapted for companies of smaller size and greater flexibility is to give a chance to a maximum number of people to gain familiarity with the various formal methods explored. To achieve this, important actions are:

- partly renew the team at an appropriate timing of the project, for example for developing a second pilot, for a new lifecycle phase requiring a specific profile (e.g. code generation, test generation...)
- have the initial set of people becoming trainers/advisors for a second wave of engineers

This strategy was followed by Space System Finland. In the end, about one third of all engineers in that company worked on DEPLOY at some point. Interesting FAQs in the category TPS relate information regarding training people with no FM background [TSP-HM-1].

5 Related Work

As discussed in section 2, success stories have proved efficient to report evidence of successful Industry adoption. Major success stories strike the attention and help in the adoption in specific domains, for example the METEOR case in transportation [3], Estelle for communication protocols [11], the Mars rover in space [6], OS driver verification [2] or real time OS design[23]. In addition books also compile many success stories [13,21]. However success stories can only give a partial view, specific to a given context and targeting only a subset of the whole adoption process. In contrast, our work proposes a more systematic approach, proposing a common template for all the success stories and also providing connection to the specific aspects (FAQ) that are being explicitly covered by the success story. For example, the level of proof automation in METEOR is a key aspect to ensure the industrial applicability of the method.

Papers analysing lessons learned [16], identifying common "myths" [14] or issuing good practices (or "commandments") [5] have also been published in the literature. However given the space constraints, they can only give essential information and cannot dig into details and provide illustrations in connection with success stories as proposed in our wiki.

There also exists another initiatives to build an online repository by Bowen [4]. Its scope is however larger than the industrial adoption, it is rather a "Virtual Library" which maintains up-to-date classification of important types of ressources such as formal notations, tools, projects, who's who. This work is a useful complement to it which could eventually be more closely integrated or even made available under the same platform.

International surveys are also carried out regularly in the 1990's[9], 2000's [24]. Conducting surveys is a huge work requiring to pay a lot of attention in the design of questions, data collection and result interpretation process. They result in a good snapshot at a given time of what kind of FM are being adopted, what are the major barriers/drivers, what kind of domains are better integrated such methods etc. Our work relies on such surveys to identify what kind of issues should be better explained for example the training phase, or to take into account all the impacted aspects of a development, e.g the position of certification w.r.t the use of FM.

Domain specific studies have also been published about the adoption in specific sectors, for example in transportation [1] and the aerospace industry [8]. Those source of information are directly relevant for populating our work and relating them with impacted aspects (e.g. lifecycle activity supported, certification issues) and providing cross-sectorial consolidation.

Managerial issues have been analysed by [22]. This paper takes the manager point of view and provides guidance in deciding if a particular project is a good candidate for the use of formal methods. It covers steps such as go/no-go conditions, leveraging existing processes, scoping the deployment and some considerations on the choice of methods/tools and cost estimation. The perspective we propose in this paper encompasses this view with the extra dimension of finer grainer roles (high-level vs project vs QA manager). It also covers extra dimensions such as external factors and certification.

6 Conclusion and Perspectives

In this paper, we described the structure and content of an evidence repository gathering useful information for helping organisation considering the adoption of formal methods. The paper focused on the key management roles in particular on the High Level manager taking strategic decisions. We highlighted the approach on a few important FAQ extracted from the evidence repository collected during the DEPLOY project.

This work remains partial, although it involved more than 10 organisations of various size, culture and background wrt to formal methods, spread across 7 application domains. The material produced resulted from a collaborative validation process and can vary in maturity. Conducting such a work is not easy and is also to some extend hindered by the natural tendency to protect internal process information. However for an organisation to make sound decision on its innovation cycle, it is also important to spend time sorting and normalizing information collected from trials. This entails having to identify appropriate themes and questions on the targeted topic. So we believe our approach can also be adopted internally by organisation by running a private instance of the repository from which selected pieces could be shared in a subsequent step. The extended process that we used including content quality control is the following:

- Have a duplicate evidence repository, keep one of them private to store work in progress and another one to publish to the targeted audience
- Organise a trusted editorial board to vouch for the independence and the overall quality of the information provided.
- Control the editing of published evidence material
- Facilitate commenting and obtaining feedback on work in progress as well as on the published evidence material

Clearly, organisations will initially prefer to keep most information private at the organisation level or even a more restricted set of people. However, once the information has matured, it is usual when well crafted pieces of evidence

can be used as advertising. For example, publishing information about training can indirectly show the competition that an organisation is promoting the adoption of formal methods. Publishing data on exploiting formal models will also definitely display a growing expertise. This is exactly the type of information sought for publication on the `www.fm4industry.org` site. However, the caveat is that the information must be validated and approved by the editorial board of fm4industry, which incidentally may evolve over time. A basic validation principle followed by the board is that any peer reviewed article at workshop or conference provides sufficient guarantees for acceptation. It is nonetheless possible that other information may meet a sufficient standard to be accepted, for example, a white paper published by an organisation in the context of an EU research project supported by an academic partner may be accepted for publication.

As a final word, the main goal of fm4industry.org is to promote the use of formal methods in Industry and become an almost self-sustainable site where contributions from external parties will be proposed spontaneously. We strongly encourage Industry transfer pilots to contribute to the fm4industry repository. Accounts can be requested using a standard wiki registration procedure enriched by some controls to exclude spamming robots.

Acknowledgment. This work is was partly funded by the European Commission under the FP7 project DEPLOY (nr 214158) and the Wallon region under the LOCOTRAC project (nr 6577). We warming thanks the all deployment partners for their engagement and contribution: Siemens SAS I-MO, Space Systems, Robert Bosch Gmbh, SAP, ClearSy, Systerel, XMOS, AeS Brasil, Critical Software Technologies Ltd and STMicroelectronics.

References

1. Bacherini, S., Fantechi, A., Tempestini, M., Zingoni, N.: A story about formal methods adoption by a railway signaling manufacturer. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 179–189. Springer, Heidelberg (2006)
2. Ball, T., Cook, B., Levin, V., Rajamani, S.K.: SLAM and static driver verifier: Technology transfer of formal methods inside microsoft. In: Boiten, E.A., Derrick, J., Smith, G.P. (eds.) IFM 2004. LNCS, vol. 2999, pp. 1–20. Springer, Heidelberg (2004)
3. Behm, P., Benoit, P., Faivre, A., Meynadier, J.M.: METEOR: A successful application of B in a large project. In: Wing, et al (eds.), pp. 369–387 (1999)
4. Bowen, J.: Formal Methods Wiki, `http://formalmethods.wikia.com`
5. Bowen, J.P., Hinchey, M.G.: Ten commandments of formal methods. IEEE Computer 28(4), 56–63 (1995)
6. Brat, G., Drusinsky, D., Giannakopoulou, D., Goldberg, A., Havelund, K., Lowry, M., Pasareanu, C., Venet, A., Visser, W., Washington, R.: Experimental evaluation of verification and validation tools on martian rover software. Form. Methods Syst. Des. 25(2-3), 167–198 (2004)

7. Clabaut, M.: Challenges in Applying Formal Methods - An SME View. In: Dagstuhl Seminar on Refinement Based Methods for the Construction of Dependable Systems (September 2009)
8. Cofer, D.D.: Formal methods in the aerospace industry: Follow the money. In: Aoki, T., Taguchi, K. (eds.) ICFEM 2012. LNCS, vol. 7635, pp. 2–3. Springer, Heidelberg (2012)
9. Craigen, D., Gerhart, S.L., Ralston, T.: An international survey of industrial applications of formal methods. In: Z User Workshop, pp. 1–5 (1992)
10. DEPLOY, Industrial deployment of system engineering methods providing high dependability and productivity, http://www.deploy-project.eu
11. Fecko, M.A., Amer, P.D., Sethi, A.S., Umit Uyar, M., Dzik, T., Menell, R., Mcmahon, M.: A success story of formal description techniques: Estelle specification and test generation for mil-std 188-220. In: FDTs in Practice, pp. 1196–1213 (2000)
12. FM4Industry, Evidence on Formal Methods Uses and Impact on Industry, http://www.fm4industry.org
13. Gnesi, S., Margaria, T.: Formal methods for industrial critical systems: A survey of applications, 1st edn. John Wiley & Sons, Inc. (2012)
14. Hall, A.: Seven myths of formal methods. IEEE Softw. 7(5), 11–19 (1990)
15. Kuliamin, V.V., Omelchenko, V.A., Petrenko, O.L.: Formal methods: for all or for chosen? In: Cordeiro, J.A.M., Shishkov, B., Verbraeck, A., Helfert, M. (eds.) CSEDU (2). INSTICC Press (2009)
16. Larsen, P.G., Odense, M., Fitzgerald, J.S., Brookes, T.: Lessons learned from applying formal specification in industry. IEEE Software (1995)
17. Leuschel, M., Falampin, J., Fritz, F., Plagge, D.: Automated Property Verification for Large Scale B Models. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 708–723. Springer, Heidelberg (2009)
18. Mathworks, Polyspace, http://www.mathworks.com/products/polyspace
19. Mediawiki, Collaborative online tools for knowledge sharing, http://www.mediawiki.org
20. DEPLOY Project, D47 Deliverable HOWTO Guide for Managers V2.0 (April 2012), http://www.deploy-project.eu/pdf/D47.pdf
21. Romanovsky, A., Thomas, M.: Industrial deployment of system engineering methods. Springer-Verlag New York Incorporated (June 2013)
22. Stidolph, D.C., Whitehead, J.: Managerial issues for the consideration and use of formal methods. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 170–186. Springer, Heidelberg (2003)
23. Verhulst, E., de Jong, G., Mezhuyev, V.: An industrial case: Pitfalls and benefits of applying formal methods to the development of a network-centric RTOS. In: Cuellar, J., Sere, K. (eds.) FM 2008. LNCS, vol. 5014, pp. 411–418. Springer, Heidelberg (2008)
24. Woodcock, J., Larsen, P.G., Bicarregui, J., Fitzgerald, J.S.: Formal methods: Practice and experience. ACM Comput. Surv. 41(4) (2009)

Auditing User-Provided Axioms in Software Verification Conditions

Paul Jackson[1], Florian Schanda[2], and Angela Wallenburg[2]

[1] School of Informatics, University of Edinburgh, UK
pbj@inf.ed.ac.uk
[2] Altran, Bath, UK
{Florian.Schanda,Angela.Wallenburg}@altran.com

Abstract. A common approach to formally checking assertions inserted into a program is to first generate verification conditions, logical sentences that, if then proven, ensure the assertions are correct. Sometimes users provide axioms that get incorporated into verification conditions. Such axioms can capture aspects of the program's specification or can be hints to help automatic provers. There is always the danger of mistakes in these axioms. In the worst case these mistakes introduce inconsistencies and verification conditions become erroneously provable.

We discuss here our use of an SMT solver to investigate the quality of user-provided axioms, to check for inconsistencies in axioms and to verify expected relationships between axioms, for example.

1 Introduction

1.1 Use of Verification Conditions

One common approach to the formal verification of software involves the generation of verification conditions (VCs). VCs are typed first-order sentences that, if proven, assure the correctness of assertions that annotate programs. Often assertions check for the absence of exceptions, that arithmetic operations do not overflow or array indices are not out of bounds, for example. Sometimes assertions capture information from program specifications about intended behaviour. Typically one tries to discharge VCs using automatic theorem provers. SMT solvers such as Z3 [16], CVC4 [2] and Alt-Ergo [1] are popular choices. More rarely, particularly when the assertions are more complex, interactive semi-automated theorem provers such as PVS, Coq or Isabelle [17] are employed. Current examples of VC-based software verification tools and frameworks include Why3 [9], Boogie [6], Perfect Developer [13] and the SPARK tool-set [5]. These support verification of programs in (subsets of) such languages as C, C#, Java and Ada.

1.2 The Need for Axioms

It is often necessary for users of VC-based tools to write axioms which are added as assumptions to VCs. We describe below two classes of axioms: specification axioms and prover-hint axioms.

C. Pecheur and M. Dierkes (Eds.): FMICS 2013, LNCS 8187, pp. 154–168, 2013.

Specification axioms provide essential specification-related information. For example, such axioms can capture information about the environment a program is to operate in, information that might be difficult to capture in the preconditions of individual functions and procedures. Such axioms can also describe properties of constants, functions and relations that are introduced to help with program specification. For example, if verifying a sorting program, relations are needed to describe how the output is sorted and how the output is a permutation of the input.

Prover-hint axioms address incompletenesses in automatic provers, their failure to prove VCs that are logically valid. VCs frequently include quantified assumptions and can involve non-linear integer or real arithmetic. In general such VCs are intractable or undecidable. While automatic provers are continually improving, it is usually unrealistic to expect them to prove all VCs that might be generated for a program.

Sometimes VCs not proved automatically are verified by human inspection. Unfortunately this can be an exceptionally tedious and error prone process. An alternative to human inspection is the use of an interactive theorem prover such as PVS, Coq or Isabelle. Such systems rarely have soundness issues. However they have steep learning curves and place huge demands on the patience and mathematical sophistication of users.

Another approach is to use a combination of automatic proving, human inspection and axioms. The idea is to figure out why an automatic prover is failing to prove some given VC, and then to add one or more axioms that summarise logically-valid facts that the prover is missing and that are sufficient to enable the prover to complete the VC proof. Often these missing facts concern just a small part of the reasoning needed to justify the VC, so it is significantly simpler to manually check their correctness than to manually check the whole VC.

Sometimes, the automatic prover can be used to check the missing facts, even though the prover is not able to derive them or their equivalent when attempting the proof of the whole VC. Other times, it may be practical to use an interactive prover to check just these facts added as hints to the automatic prover.

In practice, this process of analysing failed automatic proofs is useful not only for identifying logically-valid prover-hint axioms, but also for discovering overlooked specification axioms.

Sometimes, because of the general nature of an axiom or because VCs are often similar, an axiom can turn out to be useful for multiple VCs.

One benefit of the user axiom approach to addressing VCs that fail to be automatically proved is that it potentially reduces the maintenance work needed when programs or program annotations change. If such VCs are manually checked, it is likely that they will need rechecking on every program or assertion change. If such VCs are discharged using an interactive theorem prover, the proofs might break and need fixing. However, commonly prover-hint axiom states a general truth that is independent of the particular context of the VCs it is needed in, so it remains valid as the program and program annotations evolve.

1.3 Problems with Using Axioms

A major issue with adding axioms is the possibility of introducing an inconsistency. Sometimes software verification engineers will not be experienced in the use of formal logic and in how to phrase axioms appropriately. Also, needed axioms often have much detail that is tedious and awkward to work through, and mistakes are consequently easy.

Introducing an inconsistency into a VC is obviously a bad thing: it makes the VC trivially true. If the automatic prover detects the inconsistency, the VC and corresponding program assertion will be claimed true by the prover, irrespective of whether they are actually true or not.

Diagnosing why a VC is not proven and crafting appropriate axioms takes time. In our experience, while this process is sometimes fast, taking just a fraction of an hour, other times it can be 1–2 hours or sometimes even days. On a large verification project, there could be several 100s of unproven VCs and perhaps 100s of axioms needed. In such cases, the extra time and cost can be significant, and support for reducing this time could be very useful.

As remarked above, sometimes user-provided axioms remain valid as programs and annotations change, but other times they need reworking and re-reviewing. For example, axioms might refer to specification constants or functions whose definitions change. So user-provided axioms can still present a significant maintenance burden to any SPARK development project.

1.4 Formal Support for Auditing Axioms

The core issue we explore in this paper is the use of an automatic theorem prover to investigate the quality of axiom sets. Most critically we are interested in identifying inconsistencies, but also we investigate relationships between axioms. If these relationships are not as expected, then there might be mistakes in the phrasing of axioms. For example, we might discover that some axiom is derivable from two others, when we previously thought it was independent of those axioms. This study of relationships could help axiom writers find errors in their axioms more quickly. It could both boost confidence in the correctness of axiom sets and reduce their development time. Additional aspects we discuss include the determination of minimal sets of axioms needed for proving VCs and the resolution of ambiguities in axiom definitions.

Our specific context for the work reported is the tool-set from Altran UK (formerly Praxis) and AdaCore for the formal verification of programs in the SPARK[1] subset of Ada [5]. This tool-set allows the user to associate a set of axioms with each section of code being verified. These axioms are then added as extra assumptions to each VC associated with the relevant section.

We have extended this tool-set with features for investigating the quality of user-provided axioms. We describe in this paper our experiences of using these

[1] The SPARK Programming Language is not sponsored by or affiliated with SPARC International Inc and is not based on the SPARC® architecture.

features to audit user-provided axioms employed in several case-study SPARK programs.

2 Related Work

Systems for verifying programs by proving VCs have been around since the 1960s. For example, King's PhD thesis [15] is the first description of such a system. In these early systems the theorem provers were poor at discharging VCs. Boyer and Moore, in the preface to their 1979 book on their NQTHM inductive theorem prover [10], take aim the dangers in the practice of assuming as axioms any simplified VCs that could not be automatically proved. They remark that often such axioms are seen as obvious facts when they are not obvious to many, and sometimes are even false.

Aspects of the approaches we investigate here have been implemented in both the Boogie-based VCC tool for verifying concurrent C [12] and the Why3 software verification framework [8,9]. VCC has an option for enabling the generation of *smoke test* VCs. These VCs are phrased as the unreachability of control points in code. However, if proven for obviously reachable control points, they indicate inconsistencies in specifications and axioms. VCC's documentation[2] remarks that these tests are useful, especially early in the verification/development cycle. Why3 supports a *bisect* feature for finding minimal subsets of declarations that can prove a given VC. *Declarations* in Why3 not only include declarations of constants, functions and relations, but also include user-asserted axioms. The Why3 documentation does not comment on the provision of the feature. However, its very existence suggests that someone thought it might be useful.

Beckert, Bormer and Klebanov [7] review the uses of in-program annotations in VC-based program verification systems. Some of these annotations have a function similar to the user axioms we consider. They make a distinction between *requirement annotations* that capture specification information and *auxiliary annotations* that are needed to guide proofs. They further sub-divide the auxiliary annotations into those needed only for efficiency reasons, e.g. hints for quantifier instantiation and intermediate lemmas, and those that are logically essential, e.g. loop invariants. The prover-hint axioms we identify serve the same purpose as their first class of auxiliary annotations.

Ahn and Denney [3] use both an SMT solver and random testing to analyse axioms in a program verification system for verifying aerospace flight code. The approach tries to find false instances of axioms of form $\forall \boldsymbol{x}.\ A \Rightarrow B$. While A typically involves theories within the scope of an SMT solver (in their case Yices [14]), B often uses richer theories beyond the solver's capabilities, so they cannot use the SMT solver to check satisfiability of $\neg(A \Rightarrow B)$. They find that they can compute the truth value of ground instances of the axioms, so they exploit the Haskell-based QuickCheck library [11] to try a random search for falsifying instances. However, they observe this does not work well when random instances of A are rarely true. In this case they first use the SMT solver to find

[2] `http://vcc.codeplex.com`

satisfying instances of A and then use a random search of instantiations of the remaining variables in $\boldsymbol{x}$ to try to find an instantiation that makes B false and hence $A \Rightarrow B$ false.

A QuickCheck-based approach is also used in the Isabelle interactive theorem prover [17] for attempting to show conjectures false before users spend effort trying to prove the conjectures.

We have not yet ourselves investigated a testing approach for axiom auditing, but are considering it in future work.

Consistency checking is of importance in many other formal approaches to software and systems verification. For example, in model checking, it can be worth checking that at least some run satisfies the model and any environment assumptions, before going on to check temporal logic properties of runs: if there are no such runs, there is a problem with the combination of the model and the assumed environment.

3 Framework for Investigations

3.1 The SPARK Language

SPARK is an Ada subset extended with an annotation language for expressing program specification information. It is designed for use in high-integrity applications. The Ada subset is tailored to make verification more straightforward. For example, it does not allow recursion, nor does it support heap-based data-structures. Several of the constraints of the SPARK language also correspond to common language constraints employed for embedded software in critical systems. For example, the no recursion and no heap restrictions ensure that a program's memory requirements are statically known.

SPARK has been used for high-integrity applications in sectors including aerospace, railways, automotive and nuclear. For example, it is currently being used in the development of the iFACTS support system for UK air traffic control, over 200K SLOC.

SPARK functions and procedures are collectively referred to as *subprograms*. Subprograms are grouped together into *packages*, and together subprograms and packages are examples of *program units*.

3.2 The SPARK Formal Verification Tool-Set

Virtually all developments in SPARK make use a formal verification tool-set developed by Altran UK. Traditionally the most-used tools are the *Examiner* which generates VCs from SPARK programs and the *Simplifier* automatic prover for discharging VCs. These VCs typically include *system axioms* which give definitions to standard constants and functions introduced by the VC generation process.

In addition, the user can provide *user axioms* which typically contain prover hints and information related to the specification of the particular program being

verified. The SPARK language has been recently extended to allow axiomatic information to be included in the bodies of program definitions and as annotations for specification functions declared in the SPARK program files. This improves the visibility of the axioms and helps software developers keep them synchronised with program changes. However, all user axioms considered in this paper were supplied in separate user-axiom files associated with each program unit.

3.3 The Victor VC Translator and Prover Driver

Recently the SPARK tool-set incorporated a program *Victor*[3] developed by the first author which enables SMT solvers to be used as automatic provers for discharging VCs. Victor translates VCs in the FDL language output by the Examiner into API calls or standard languages accepted by SMT solvers. It also manages running the solvers and collecting results. Victor is most commonly used with the solvers Z3, CVC4 and Alt-Ergo. Of these, Z3 gives the best performance, and we report here only on experiments with Z3. Generally SMT solvers perform significantly better than the Simplifier. However Victor has only become a fully supported component of the tool-set in the past year, and many ongoing industrial users of the tool-set are still using the Simplifier as the primary tool for VC discharge.

Victor's translation typically involves introducing axiomatisations for various types such as the array, record and ordered enumeration types of the FDL language that do not have standard correspondences in the SMT solver input languages we use (SMTLIB 1.2 and 2.0). In this paper, we consider the axioms introduced by Victor as additional system axioms.

The prover driver component of Victor has been adapted to enable the exploration of the axiom auditing features introduced in Section 4 of this paper.

4 Axiom Analysis

4.1 Exploring Properties of Axioms

We describe here how we approach examining the consistency and interdependency of user-provided axioms.

A VC sentence generated by the SPARK tool-set has the general structure

$$S \wedge U \wedge H \Rightarrow C$$

where S, U, H and C are each an implicitly-conjoined set of closed formulas expressed in a typed first-order logic. More specifically:

- S is a set of system axioms,
- U is a set of user axioms,
- H is a set of hypotheses,
- C is a set of conclusions.

[3] `http://code.google.com/p/vct/`

A VC is considered to be valid when it is satisfied by all possible interpretations for uninterpreted functions, constants and types, and by standard interpretations for interpreted functions, constants and types. A standard example of interpreted constants, functions, and types is natural number literals, integer arithmetic operations and the integer type.

The SPARK tool-set groups VCs according to the program unit they are associated with. Across a set of VCs for given program unit, the Hs and Cs vary, but the Ss and Us are the same.

To explore issues of consistency and inter-relatedness between user-provided axioms, Victor generates special kinds of goals, and attempts proof of each using a designated automatic theorem prover. The kinds of goals are shown in Table 1. In the table, we assume that the set of user axioms for some given program unit

Table 1. Kinds of Axiom Audit Goals

Kind	Goal shape	Description
S-incon	$S \Rightarrow \bot$	Are system axioms inconsistent?
U-incon	$S \wedge U \Rightarrow \bot$	Are user axioms inconsistent?
u-incon	$S \wedge u_i \Rightarrow \bot$	Is user axiom u_i inconsistent?
u-taut	$S \Rightarrow u_i$	Is user axiom u_i always true?
u-deriv	$S \wedge (U \setminus \{u_i\}) \Rightarrow u_i$	Does user axiom u_i follow from other user axioms?

is $U = \{u_1, \ldots, u_n\}$, so, for that program unit, n goals of each of kinds u-incon, u-taut and u-deriv are generated, but only 1 goal of each of kinds S-incon and U-incon is generated. The symbol $\bot$ is for falsity. Note that no use is made of the hypotheses H and conclusions C of each original VC.

Validity of all goals is considered with the system axioms S assumed, i.e. validity is considered in the combination of theories described both by these axioms and built-in to the prover used. Examples of built-in theories are theories of integer and real arithmetic.

The S-incon goals are baseline checks. We expect inconsistencies in system-provided axioms very rarely and the main focus of our work has been to examine the user axioms. In other VC-based verification environments, the system axioms can be much richer than the system axioms we encounter, and in such cases it would definitely make sense to also audit the system axioms more thoroughly.

The U-incon goals directly look for some inconsistency involving one or more goals, whereas u-incon goals consider the consistency of each axiom on its own. The u-taut goals check whether axioms are tautologies. We hope that most axioms added as prover hints are tautologies. The u-deriv goals look at relationships between axioms, whether each axiom depends on the others. Generally u-deriv goals are only relevant if the corresponding u-incon and u-taut goals are unproven and there are no inconsistencies in the user-provided axioms.

It is useful to know not just whether an auditing goal is provable but also, if it is provable, which formulas are needed for the proof. Automatic provers can

provide this information in various ways. Some, like the Simplifier prover provided with the SPARK tool-set, output a trace of their deductions, and this trace includes information on the formulas used. Several SMT solvers have facilities for outputing proof certificates. These certificates are independently checkable and provide evidence for the correctness of their deductions. The formulas used can be read off these deductions.

Another approach is to make use of a facility some SMT solvers have for generating *unsat cores*. When an SMT solver is used as a prover, the negation of the sentence to be proved is passed to the solver. This negation usually has the form of a conjunction. For example, for a VC sentence of the form shown above, it has form

$$S \wedge U \wedge H \wedge \neg C \quad .$$

Here, as before, each set of formulas is implicitly conjoined. If the solver finds this negated sentence to be unsatisfiable, i.e. it deduces there are no models of the negated sentence, then the sentence itself is true in all models, i.e. it is valid. An unsat core is a (usually) minimal subset of the conjuncts of the conjunction that itself is unsatisfiable. From the point of view of provability of the original sentence, this subset is those formulas whose consideration is sufficient to show the sentence's validity. In our work we have experimented with generating and examining these unsat cores for provable auditing goals.

Interpretation of the results of these tests has to bear in mind the usual incompleteness of the used prover. Proved goals indicate validity of the goals, but if a goal is unproven, the goal might or might not be valid - we don't know. Failure for a goal to be proven is just a suggestion that it might not be valid.

4.2 Finding Minimal Sets of Axioms

It is useful to be able to identify minimal sets of user axioms needed for the proof of individual VCs or collections of VCs. Issues with axioms can be diagnosed if these sets are not as expected. And, once minimal sets are identified, unrequired axioms can be deleted in order to reduce the future axiom maintenance burden. Also, the fewer axioms there are, the more likely it is that axioms will be well written, compact and general, and the less likely it is that anything will need changing.

The minimal set for a VC can be smaller than the set that might be identified from a proof certificate or unsat core. The reason has to do with the fact that a user axiom is sometimes added to make proofs easier, but the VC is still valid without the axiom. In these cases a prover might be able to prove the VC without the axiom, but, if the axiom is present, it might use it because it provides a short-cut.

In our experiments, we worked with SPARK case studies where the user axioms had originally been added when the Simplifier SPARK tool-set prover had been used. However, we checked the auditing goals using stronger SMT-solver-based provers, principally Z3. This made it all the more likely that we would encounter user-provided axioms that were unnecessary.

As each user axiom file is associated with a whole program unit, not an individual VC, each user axiom potentially could be used in the proof of multiple VCs. In order to identify unused axioms, we identify user axioms that do not feature in the minimal sets for any of the VCs for a program unit.

We adopt a simple approach to computing a minimal set of user axioms needed for proving a given VC. First we establish whether the original VC of form

$$S \wedge U \wedge H \Rightarrow C$$

is provable. If it is, we then in turn try removing each user axiom from U and see whether the resulting goal is provable: when it is, we leave the axiom out, when it is not, we add the axiom back in.

In general of course there might be multiple minimal sets, and the minimal set we find might not be a set of smallest size. However, we decided to start with just one minimal set to explore its potential usefulness.

4.3 Resolving Ambiguities in Axioms

A particular issue we ran into with user-provided axioms was that they were not always unambiguous. Typically user-provided axioms are of form

$$\forall x_1 : T_1, \ldots, x_n : T_n.\ P \quad ,$$

where $T_1, \ldots, T_n$ are the types of variables $x_1, \ldots, x_n$. However the concrete syntax for axioms permits these outermost universal typed quantifiers to be implicit, and the common practice to date by user-rule writers has been almost always to leave these quantifiers implicit. In some scenarios this is not a problem: one can always deduce the types of the quantified variables from the immediate context of their occurrences. However, with the concrete syntax for VCs, many operators are overloaded. For example, the same successor function is used for integer, real and enumeration types. With this overloading, variable types sometimes cannot be deduced from their immediate context. And further, sometimes the overloading cannot be resolved until types of free variables have been deduced.

We implemented an algorithm that combines operator overload resolution and variable type inference. This tries to make as much progress on both fronts, and warns the user when it does not complete. The expectation is then that the user goes in and adds sufficient explicit quantifiers to the user axioms that both overload resolution and type inference can complete.

All the experiments we report on here were run on axiom sets where we first had gone through the process of making sure all the axioms were unambiguous.

Resolving this ambiguity carefully is critical for soundness reasons. It is easy to construct examples where there are multiple ways of resolving operator overloading and variable typing, and some ways yield a sound axiom, other ways an unsound axiom.

The general lesson here for designers of languages for expressing logical formulas is that they should be very wary of adopting convenient notational ambiguities.

5 Experimental Results

5.1 General Setup

The evaluation we present here is based on our examination of recent SPARK program developments that we have access to and that make use of user axioms. We select two developments that provide interesting illustrations of the potential of rule auditing. For each development, we first resolved variable type and operator overloading ambiguities in the axioms as described in Section 4.3, before running the two kinds of analysis described in Sections 4.1 and 4.2.

We have found the Z3 SMT solver currently the best to use for proving VCs. It is generally faster than the competition and can almost always prove more VCs than the others. We ran Z3 both its default mode for checking satisfiability and in an unsat core generation mode. We used a development version of release 4.3.2 from March 2013 that included patches to fix a bug we encountered in the unsat core functionality of the 4.3.1 release.

5.2 Case Study 1: Tokeneer ID Station

Overview. The Tokeneer ID Station (TIS) [4] was a research project commissioned by the US NSA (National Security Agency) to develop part of an existing secure system in accordance with a high-integrity development process advocated by Altran UK. One phase of the project involved developing and verifying SPARK code. This comprised about 10k lines of declarations and executable code, and 2k lines of SPARK proof annotations. All materials from this project are now publically available.

In the formal verification of the absence of runtime errors, 7001 verification conditions were generated and 107 user axioms were written. 40 of these user axioms were hints to the Simplifier prover of the SPARK tool-set. All these prover-hint axioms had outermost universal quantifiers surrounding quantifier-free formulas involving propositional variables, equalities, function symbols and integer arithmetic operators and constants. Some of the arithmetic involved integer division operators and non-linear multiplication. The truth of the axioms did not rely on the interpretation of any of the function symbols. It was clear from their form that all these prover-hint axioms were expected to be tautologies. The other 67 axioms concerned properties of SPARK subprograms and of specification functions and relations.

Inconsistency Checking. Checks of u-incon goals with Z3 directly identified two inconsistent prover-hint user axioms. One of the inconsistent axioms, in the form in which it was written, is:

```
  B1 and Op = Op_1 -> B2
  may_be_deduced_from
 [ St = St_1 or (St = St_2 or St = St_3),
   St_1 <> St_2,
   St_1 <> St_3,
   St_2 <> St_3,
   St = St_1 or St = St_2 -> B1 and (B3 and Op = Op_2),
   Op_1 <> Op_2,
   St = S_3 -> not B1 ].
```

In this SPARK syntax for axioms, `->` is implication, `may_be_deduced_from` is reverse implication, `<>` is disequality, formulas within the `[]` list are implicitly conjoined, and all the variables are implicitly universally quantified. The reason this axiom is inconsistent is that it contains a typo: in the last line the `S_3` should be `St_3`. As written, the types of the equalities and inequalities are ambiguous. Before we carried out the rule audit, we added extra type constraints to this axiom to ensure the ambiguities were resolved.

This inconsistency was not detected with the VC of kind U-incon where all 6 user axioms for the relevant program unit were included as hypotheses. We often observe that Z3's performance with problems involving quantified hypotheses is sensitive to the number and ordering of these hypotheses.

This inconsistency was only detected when we ran Z3 in its unsat-core generation mode. When we ran it in its normal mode this inconsistency was missed. When producing unsat cores, Z3 is inhibited from making certain preprocessing simplifications to input problems. In our case, the change happened to make a difference to the way Z3 explored the problem search space and helped it find the inconsistency.

The other inconsistency picked up by Z3 concerned an axiom expressing a property of the integer division operator. Again, in SPARK axiom syntax, we have:

```
X - (Y - 1) * 100 <= 200 -> Y + 1 = (X - 1) div 100 + 1
                  may_be_deduced_from
                [ 100 < X - (Y - 1) * 100,
                  goal(checktype(X, integer)),
                  goal(checktype(Y, integer)) ] .
```

Here, we see how one expresses constraints on the type of variables `X` and `Y`. Without the constraint on `Y`, it is unclear if `Y` is integer or real. To see the falsity, consider when `X = 0`, `Y = -1`, and note that integer division in the FDL language is real division rounded towards zero.

With the VCs of kind u-taut, Z3 demonstrated that 37 of the 40 prover-hint axioms were indeed tautologies. Two of the 3 remaining are those shown above. The other remaining was a false statement of non-linear arithmetic:

```
(B - 1) * X + A < Z may_be_deduced_from
[ A < X or B < Y,
  A <= X,
```

```
B <= Y,
X >= 0,
X * Y = Z ] .
```

To see the falsity of this, consider the axiom when `X = Z = A = B = 0`, `Y = 1`. While Z3 could not prove the u-incon check for this axiom, it could prove a variation of this check where all the system axioms were deleted. It is easy to see that these axioms were a significant distraction: there were over 300 system axioms, including over 100 with universal quantifiers.

These latter two axioms shown above are incorrect over-generalisations of two VC subgoals the Simplifier prover could not prove. For example, the axiom concerning division should have had a pre-condition that `X >= 1`.

The VCs the above axioms were intended as hints for are all valid. Indeed, Z3 is able to prove them all (without these inconsistent axioms) in 10s of milliseconds.

Axiom Inter-relationships. The u-deriv checks along with unsat cores revealed a number of relationships between the specification axioms. For example, among one set of axioms $A_1, \ldots A_9$ for a package program unit, we had

$$
\begin{aligned}
&A_2 \Rightarrow A_1,\\
&A_1 \wedge A_7 \Rightarrow A_2,\\
&A_4 \Rightarrow A_3,\\
&A_3 \wedge A_8 \Rightarrow A_4,\\
&A_6 \Rightarrow A_5,\\
&A_5 \wedge A_9 \Rightarrow A_6,\\
&A_2 \Rightarrow A_7,\\
&A_4 \Rightarrow A_8,\\
&A_6 \Rightarrow A_9.
\end{aligned}
$$

Here the relationships are in the context of the system axioms. These relationships can be more succinctly expressed as:

$$
\begin{aligned}
&A_2 \Leftrightarrow A_1 \wedge A_7,\\
&A_4 \Leftrightarrow A_3 \wedge A_8,\\
&A_6 \Leftrightarrow A_5 \wedge A_9.
\end{aligned}
$$

While one might think that axioms A_2, A_4 and A_6 would be sufficient hints, it appears that the Simplifier prover needed more of the axioms. For example, its proofs made use of both A_2 and A_1, though A_7 was unused.

Redundant Axioms. The redundant axiom analysis identified 50 redundant axioms, including the 40 prover-hint axioms. The u-deriv checks indicated that 7 of the remaining 10 specification axioms were subsumed by other specification axioms.

5.3 Case Study 2: Mixed Floating-Point and Integer Arithmetic

Overview. This case study considered SPARK code of industrial origin that comprised a collection of around 30 functions and procedures for carrying out computations in a mix of floating point and integer arithmetic. Assertions and user axioms were added by the company developing the SPARK code as part of the company's evaluation of Altran's SPARK tool-set. Of particular interest to us was a collection of 25 or so specification axioms that concerned conversions from floating point numbers to integers.

In the SPARK tool-set, computations with floating-point numbers are reasoned about approximately by using the mathematical reals in VCs.

Inconsistency Checking. The floating-point to integer conversions were characterised by floor and ceiling functions which rounded towards $-\infty$ or $+\infty$ respectively. For example, the ceiling axioms had form

$$c0 : \forall x : R.\ x \leq k - 1 \Rightarrow \text{ceil}(x) \leq x + 1$$
$$c1 : \forall x : R.\ x \leq k - 1 \Rightarrow \text{ceil}(x) \leq k$$
$$c2 : \forall x : R.\ x \leq k - 1 \Rightarrow x \leq \text{ceil}(x)$$
$$c3 : \forall x : R.\ x \leq k - 1 \Rightarrow -k \leq \text{ceil}(x)$$

Here k was an integer constant for the largest representable floating point number.

The U-incon check identified that axioms $c0$ and $c3$ were mutually contradictory: consider instantiating both with $-k - 2$. $c0$ then says that $\text{ceil}(-k-2) \leq -k-1$, but $c3$ says that $\text{ceil}(-k-2) \geq -k$. However Z3 missed a very similar U-incon check for inconsistency between axioms for a floor function. Interestingly, a u-deriv check succeeded involving the 2 mutually-inconsistent floor axioms as hypotheses and an axiom involving a round to nearest function as conclusion, though the unsat core showed the conclusion playing no formal part in the truth of the check. Nevertheless, we suspect that the conclusion provided a hint to Z3 as to how to instantiate the hypotheses to obtain the contradiction.

Inter-relationship Checking. This identified for example that $c0 \Rightarrow c1$, i.e. that $c1$ is redundant if $c0$ is retained in sorting out the inconsistency.

6 Discussion

There are different scenarios in which one could envisage undertaking axiom auditing. Firstly, after a SPARK development has reached some milestone, as part of a review process. Here the identification of inconsistencies would be of definite interest, as would unused axioms. However, it's not clear how much use inter-relationship information would be. Secondly, during a SPARK development, as user axioms are being written. Here it seems that the the axiom creator could

perhaps benefit more from the feedback provided by the inter-relationship analysis, as the feedback would help confirm and correct expectations about the axioms.

There are questions about how representative the examples are of issues from the previous section. The examples partly relied on the fact that the user axioms had originally been developed with provers with significantly weaker arithmetic capabilities than Z3. Still, non-linear arithmetic reasoning is a real challenge area for SMT solvers, and one could expect users in the near future will still need to provide some help with more complex non-linear VCs.

As noted, Z3 sometimes struggles and has unpredictable behaviour with handling the quantifiers in axioms. This is not just an issue with Z3, all SMT solvers have similar issues.

7 Conclusions and Future Work

We have discussed here several approaches we have investigated for checking the quality of user-provided axioms employed in the formal verification of SPARK Ada programs. The main approaches are to check for inconsistencies, tautologies and dependencies in axiom sets, and to derive minimal axioms sets. We also needed to resolve ambiguities in the axioms that were a consequence of the particular language they were expressed in.

We have shown the value of these approaches on two case study SPARK programs, most significantly finding inconsistent axioms in both cases.

We hope shortly to experiment with axiom auditing on much larger SPARK examples than those case studies we have considered to date. We are also looking for an opportunity to engage with SPARK verification engineers on a live project, to have them explore how axiom auditing might improve their confidence in the correctness of the axioms.

The results obtained with main approaches are dependent on the theorem proving power of the selected prover. We have had good results with the Z3 SMT solver, but also have seen its behaviour is not always consistent.

It is clear that large sets of system axioms can be a significant distraction to Z3, and we need to look at pruning these axiom sets to those of obvious relevance to the user axioms being investigated.

It would be interesting to investigate the use of other provers such as resolution-based automated theorem provers which have more mature technology for handling quantifiers, though their arithmetic support is not as strong as that of SMT solvers. We also will consider experimenting with a testing-based approach for attempting to falsify axioms, as described in Section 2. A relatively lightweight way of doing this would be to exploit a feature of Victor for generating VCs in the theory language of the Isabelle/HOL theorem prover [17], and using Isabelle's built-in QuickCheck procedures.

References

1. Alt-Ergo: an OCAML SMT solver for software verification, homepage at http://alt-ergo.lri.fr/
2. CVC3: an automatic theorem prover for Satisfiability Modulo Theories (SMT), homepage at http://cvc4.cs.nyu.edu
3. Ahn, K.Y., Denney, E.: A framework for testing first-order logic axioms in program verification. Software Quality Journal 21, 159–200 (2013)
4. Barnes, J., Chapman, R., Johnson, R., Widmaier, J., Cooper, D., Everett, B.: Engineering the Tokeneer enclave protection software. In: 1st International Symposium Secure Software Engineering (ISSSE). IEEE (2006), http://www.adacore.com/sparkpro/tokeneer
5. Barnes, J., with Altran Praxis: SPARK: the proven approach to high Integrity Software. Altran Praxis (2012)
6. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006), http://research.microsoft.com/en-us/projects/boogie/ for current information on Boogie
7. Beckert, B., Bormer, T., Klebanov, V.: On essential program annotations and completeness of verifying compilers. In: Filliâtre, J.C., Freitas, L. (eds.) Proceedings, Workshop on Verified Software: Theory, Tools, and Experiments, VSTTE (2009)
8. Bobot, F., Filliâtre, J.C., Marché, C., , Melquiond, G., Paskevich, A.: The why3 platform, version 0.80. Tech. rep., University Paris-Sud, CNRS, Inria (October 2012), http://why3.lri.fr
9. Bobot, F., Filliâtre, J.C., Marché, C., Paskevich, A.: Why3: Shepherd your herd of provers. In: Leino, K.R.M., Moskal, M. (eds.) Boogie 2011: First International Workshop on Intermediate Verification Languages, pp. 53–64 (August 2011), http://proval.lri.fr/publications/boogie11final.pdf
10. Boyer, R.S., Moore, J.S.: A Computational Logic. Academic Press (1979), http://www.cs.utexas.edu/users/boyer/acl.pdf
11. Claessen, K., Hughes, J.: Quickcheck: A lightweight tool for random testing of haskell programs. In: Proceedings of the ACM SIGPLAN International Conference on Functional Programming, pp. 268–279 (2000)
12. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A practical system for verifying concurrent C. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 23–42. Springer, Heidelberg (2009)
13. Crocker, D., Carlton, J.: Verification of c programs using automated reasoning. In: Fifth IEEE International Conference on Software Engineering and Formal Methods (SEFM), pp. 7–14. IEEE Computer Society (2007)
14. Dutertre, B., de Moura, L.: The Yices SMT solver (August 2006), tool paper at http://yices.csl.sri.com/tool-paper.pdf
15. King, J.C.: A Program Verifier. Ph.D. thesis, Carnegie-Mellon University (1969)
16. de Moura, L., Bjørner, N.S.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
17. Nipkow, T., Paulson, L.C., Wenzel, M.T.: Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002), http://www.cl.cam.ac.uk/research/hvg/Isabelle/ for current information

Formal Reliability Analysis of Protective Relays in Power Distribution Systems

Adil Khurram, Haider Ali, Arham Tariq, and Osman Hasan

School of Electrical Engineering and Computer Sciences (SEECS),
National University of Sciences and Technology (NUST), Islamabad, Pakistan
{09beeakhurram,09beehali,09beeatariq,osman.hasan}@seecs.nust.edu.pk

Abstract. Relays are widely used in power distribution systems to isolate their faulty components and thus avoid disruption of power and damaging expensive equipment. The reliability of relay-based protection of power distribution systems is of utmost importance and is judged by first constructing Markovian models of individual modules and then analyzing these models analytically or using simulation. However, due to their inherent limitations, simulations and analytical methods cannot ascertain accurate results and are not scalable, respectively. To overcome these limitations, we propose a modular approach for developing Markovian models of relay-based protected components and then analyzing the reliability of the overall power distribution system by executing its individual modules in parallel using the PRISM probabilistic model checker. The paper presents a foundational model for a relay-based protected component that can be incrementally updated to represent more advanced behaviors, such as self-checking, routine test and continuous monitoring. Moreover, the paper provides a set of reliability assessment properties of power distribution systems that can be formally verified by PRISM. For illustration purposes, we present the analysis of a typical power distribution substation.

Keywords: Probabilistic Model Checking, PRISM, Markov Chains, Relay-based Protection Systems.

1 Introduction

Power distribution systems [6], depicted in Figure 1, are used to ensure reliable distribution of electricity, generated by power stations, to end users. The main idea behind this enormously used distribution network is to first route power from the main power station to various substations via transmission lines. The substations then in turn distribute the power to their respective consumers. Besides the transmission lines, the other integral component of power distribution networks is a power transformer. It is installed at the main power station to step up the generation level voltages (11kV or 33kV) to transmission levels (230kV or 69kV) or at the substations to step down the transmission level voltages to distribution levels (34.5kV or 24.kV) [24].

C. Pecheur and M. Dierkes (Eds.): FMICS 2013, LNCS 8187, pp. 169–183, 2013.

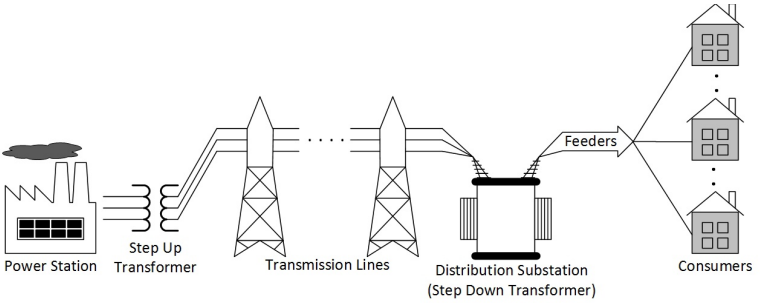


Fig. 1. A Typical Power Distribution System

The power distribution system is a highly sensitive and safety-critical domain. Faults like switching surges and short circuits can not only damage the sophisticated and expensive components, like transmission lines or transformers, of a power distribution system but could also lead to catastrophic consequences like the major power failure in the UCTE grid [13] or the 2005 Moscow power blackout [11]. In order to protect the power distribution systems and their components, it has now become customary to associate a protective relay [21] with every safety-critical component of a power distribution system. These protective relays are capable of sensing the fault and then isolating the faulty component from the power distribution system by tripping the circuit breaker. The protective relays have undergone a transformation from electromechanical components to presently deployed multi-functional digital relays.

The reliability of the relay-based protection system of power distribution is of utmost importance. The main idea behind the reliability analysis of these protection systems is to model the behavior of the given system, along with its unpredictable elements, like fault occurrence, as a Markov chain and then analyze this model to find probabilities associated with various parameters of interest, like failures, testing intervals, fault clearing times etc. Traditionally, this reliability analysis of power distribution systems is done either analytically or using computer-based simulation. In the analytical approach, a Markovian model of the given relay-based protection system is developed on paper and the desired properties in this model are either judged analytically using paper-and-pencil proofs or computer based numerical methods. Simulation technique, on the other hand, considers the stochastic behavior of the given system while treating it as a series of real experiments [5]. Both of the above mentioned techniques have their shortcomings. Analytical techniques are more accurate compared to the simulation techniques, which compromise the completeness of the analysis by considering only a subset of all possible scenarios. On the other hand, the analytical approaches are not scalable and simulation allows us to analyze more complex and larger systems. Moreover, the absolute accuracy of analytical analysis cannot be guaranteed as it is prone to human errors and the rounding errors of numerical methods. Another major drawback of the existing

reliability analysis methods for the relay-based protection systems is that the Markovian model of the given system is usually developed on paper and in an ad-hoc manner. This kind of an informal modeling approach has limited scalability and is also quite prone to human errors. Due to the above mentioned scalability issues, all the existing reliability analyses of relay-based protection systems focus on the reliability assessment of single components rather than the complete power distribution system. Whereas, the inaccuracy limitations have been reported as the main causes behind the 2003 Northeast blackout in the United States and Canada [12] and the 2012 blackout in India [4].

Formal methods [2] are capable of overcoming the above mentioned inaccuracy limitation and have been successfully used to guarantee correctness of many real-world software, hardware and physical systems. However, to the best of our knowledge, no prior work regarding the formal reliability analysis of power distribution systems exists so far. In order to fill this gap, we propose to use probabilistic model checking [18], which is a widely used formal method for analyzing Markovian models. In particular, the paper presents a generic and formal approach to analyze the reliability of power distribution systems that use digital relay based protection. We provide a foundational Markovian model for a power distribution component that is protected by a digital relay. This proposed model can be used to represent the behavior of any relay protected component, such as transmission line or transformer, of a power distribution system. The main distinguishing feature of this model is that it allows the incorporation of additional states and transitions, to build more complex and advanced relay models, in a methodological way. The Markovian models of different relay-based protected components can then be combined to model and analyze the reliability of the complete power distribution system using a probabilistic model checker.

The proposed approach provides more accurate results than the traditional counterparts due to the exploration of an exhaustive state-based model of the given power distribution system. We have chosen the PRISM model checker [20] for the proposed work as it supports CSL and can analyze larger models in a very efficient manner. It also supports the evaluation of expected values, specified in terms of reward functions, which allows us to verify some interesting properties in the context of relay-based protection for power distribution systems. In order to illustrate the usefulness of the proposed approach, we utilize it to analyze the reliability of a typical power distribution subsystem [24]. To the best of our knowledge, this is the first reliability analysis of a complete power distribution system that caters for simultaneous failures of multiple components.

The paper is organized as follows. A brief review of the available research regarding reliability analysis of digital relays and probabilistic model checking is presented in Section 2. Section 3 describes our foundational protective relay model and the proposed approach for its incremental refinements. Section 4 provides an overview of the PRISM model checker along with the proposed properties of interest in the context of reliability analysis of power distribution systems. Section 5 presents the case study along with the verification results and some discussions. Finally, Section 6 concludes the paper.

2 Related Work

Grimes [14] laid the foundations for the reliability assessment of electric power systems by presenting mathematical equations for the failure probabilities of protective relays. A simple protection system reliability model is developed in [23] that establishes a relationship between the un-readiness probability and the undetected failure rate. This model is then further refined in [3] by including back-up protection, common cause failure and fault clearing phenomena. In [22], a Markovian model for the transmission line is proposed for determining the optimum routine test and self-checking intervals to maximize the reliability. This model is extended in [16] by including the parameters of inrush currents and over excitation to analyze the protective system of a power transformer. Both of these models considered the backup relay to be fully reliable. This limitation is overcome in [7] by considering the failure probability associated with the backup relay. Another model presented in [10] takes into consideration the causes of relay failures including software, hardware, auxiliary equipment and human errors. This model also investigates routine test effectiveness, level of reliance on self-checking, stuck breakers and human errors during tests and repair actions. In [1], the routine test approach is further improved by considering the impacts of individual components of the protective relaying system, such as current transformer (CT), voltage transformer (VT), Power Supply Unit (PSU), Circuit Breaker (C.B) and the trip coil. An optimum routine test inspection interval is then determined for each component individually and its comparison with the conventional method is given. All of the above mentioned models have been developed in an ad-hoc way, i.e., without following any particular principles or rules, and they are analyzed using numerical methods and simulations, which compromise the accuracy of reliability assessment. Moreover, due to the non-scalable modeling approach and the high computation requirements of numerical methods, only the models of single components have been analyzed. In this paper, we alleviate these problems by proposing a scalable modeling approach and probabilistic model checking based verification.

Many probabilistic model checkers, including PRISM [20], YMER [25], MRMC [17], VESTA [19] and ETMCC [15] are available. The main objective of the proposed work is to find steady state probabilities of power distribution systems, which are usually modeled as continuous time Markov chains (CTMC). However, YMER and VESTA do not support steady state probabilities and thus cannot be used for our purpose. The PRISM model checker has been reported as the most efficient one in terms of memory consumption [9] compared to MRMC and ETMCC. Thus, we have used the PRISM model checker for this work as it also supports analyzing CTMCs and offers a very user friendly graphical interface.

3 Modeling Relay-Based Protected Components

The proposed modeling approach is primarily inspired from Endrenyi's three-state Markovian model [9] used to analyze the behavior of a component under

fault. According to this model, the component can be in one of the three possible states; normal operating state, failed state or isolated state. When the fault occurs, the model moves from the normal operating state to the failed state. Relay, protecting the component, isolates it from the system thereby undergoing a transition to the isolated state. The isolated component is then restored to the normal operating state after repair.

The proposed foundational model of a relay-based protected component (Figure 2) caters for temporary and permanent component faults and the faults that may appear in the relay themselves. Temporary faults exist only for a short period of time and the circuit can be re-energized without repairing the component. Permanent faults, on the other hand, damage the component permanently and component repair action is necessary for their clearance. The above mentioned three-state model can be used to model both temporary and permanent faults. The difference being the transition rates from the faulty to normal state, i.e, the model returns to the normal state with reclosing switching rate (λ_{rct}) in the case of a temporary fault and the component repair rate (μ_c) in the case of a permanent fault. Two types of relay failures, i.e., internal and external are

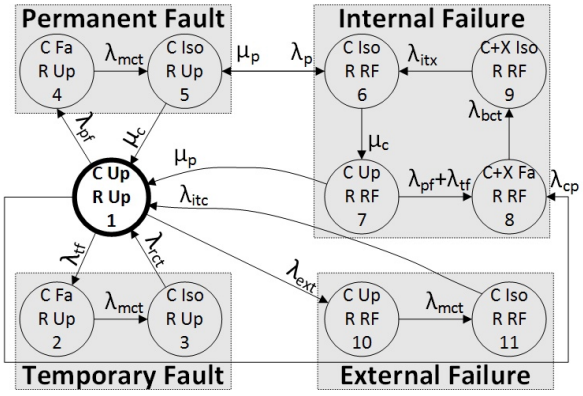


Fig. 2. Foundational Model of a Relay-based Protected Component[1]

considered in the proposed model. Relay failures due to external faults cause the model to transition to state 10 with rate λ_{ext}. The component is then isolated in state 11, with the main switching rate of the component (λ_{mct}). The component is restored with the switching rate λ_{itc} when the external fault in the relay is rectified. Relay failures due to internal relay faults can also occur with a failure rate λ_p, when the component has been isolated with an identified permanent fault, i.e., state 5, thereby causing a transition to state 6. In this

[1] C UP = component is energized, C Fa = component is faulty, C Iso = component is isolated, R UP = relay is operational, R RF = revealed failure, X = additional equipment.

state, the relay has been identified as faulty and the component is isolated due to a permanent fault, and therefore, either of them can be repaired first. If the relay is repaired first, the model moves back to state 5 with the relay repair rate (μ_p). The component is then repaired, with component repair rate (μ_c), and restored to the normal operating conditions (state 1). On the other hand, if the component is repaired first (μ_c), then the model transitions to state 7, after which, relay repairing action (μ_p) moves the model back to state 1. While in state 7, if either type of component fault occurs, the model transitions to state 8 in which the component, the relay and the additional equipment (X) connected to the component are all considered faulty. The common cause failure rate of the component and the relay is λ_{cp} and the model transitions from its normal state (state 1) to state 8 with this rate. It is assumed that the back-up relay is fully reliable and its switching operation with rate λ_{bct} isolates both the equipment (X) and the component (state 9). The additional equipment (X) is first restored with switching rate λ_{itx} in state 6, followed by the component repair action and restoration of the component as mentioned above.

The proposed model of Figure 2 is very generic and can be extended to represent more advanced behaviors of relay-based protected components. For example, the testing procedures for self-checking, routine test and continuous monitoring, can be added to the basic model mentioned above. During the self-checking test, the relay checks for the correct operation of its core components and is unavailable for normal operation. Similarly, the relay also becomes unavailable for operation in the event of routine test inspection during which a detailed analysis of operation of the digital relay is performed. The relay is available for service only during continuous monitoring in which only its critical components are checked for correctness. Routine tests and self-checking can detect relay failures and mal-trips and take appropriate actions to remove these faults. Mal-trips can be either instantaneous, i.e., they manifest immediately, or potential, which require certain conditions to manifest.

In order to illustrate the generic nature of the proposed foundational model of Figure 2, we form the model of a relay-based protected component that supports the routine test capability in Figure 3. This can be mainly done by including some more states and transitions in the model of Figure 2. The foundational model of Figure 2 is represented by the grey shaded region in Figure 3 and for the sake of simplicity, only the states that interact with the new states are mentioned. The model moves to state 12 when routine tests are being carried out and relay is unavailable for operation with rate λ_{rt}, which represents the frequency with which the routine test procedures are carried out. At this state, permanent faults (state 15) in the component and instantaneous mal-trips (state 8) can occur with relay failure rate (λ_p) and instantaneous mal-trip rate (λ_{rt-op}), respectively. When in state 15, the model transitions to state 16 with the manual switching rate (λ_{mct}) where the component is isolated. The component is then re-energized manually with rate λ_{itc} in state 9 after which the system returns to its normal operating conditions once the repairing of the relay is complete

(μ_p). States 13 and 14 represent the portion of relay failures due to internal faults (λ_{prt}) and potential mal-trip (λ_{pt-rt}), respectively, that are detectable by routine tests. These faults, when detected by routine tests, transfer the model to state 9 with rate λ_{rt}. Consequently, the relay is repaired as mentioned in the basic model description. It is also possible that before the detection of relay failure, due to potential mal-trip or internal relay failure by routine test, the component gets a permanent fault (λ_{pf}), which will lead to the isolation of the corresponding component and additional equipment (X).

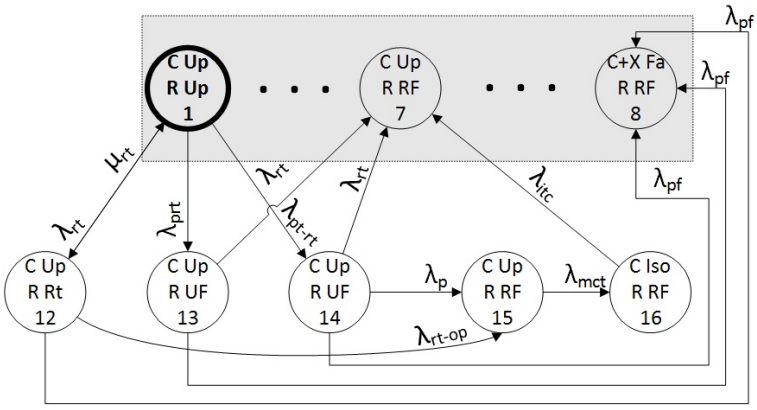


Fig. 3. Basic Model with Routine Test[2]

Similarly, the proposed generic model, given in Figure 2, can be used to construct reliability models with the self checking capability [22,16]. In this case, the instantaneous mal-trips cannot occur while the relay is under self-checking. This can be done by removing the transition from state 12 to 15 and replacing the transition rates with the corresponding ones for self-checking in the model of Figure 3. Similarly, continuous monitoring [22] can be included in the model by adding a transition from normal operating state to state 7 with rate λ_{pmn}[3].

The existing models (e.g. [22,16,1]) for the reliability analysis of protective relays are specific for one particular scenario and, to the best of our knowledge, no fixed set of rules is available to develop or incrementally update these models. Thus, the modularity and the ability for incremental updates in the behavior are the main distinguishing features of the proposed modeling approach. This feature facilitates the construction of system-level models of power distribution systems by allowing reusability of components and also is less error prone due to the modular and regular development process.

[2] R UF = unrevealed failure, R Rt = relay under routine test inspection.
[3] λ_{pmn} = portion of relay failure detected by continuous monitoring.

4 Verification of Reliability Properties

We propose to analyze the reliability of the relay-based protective system of a complete power distribution substation by executing the Markovian models of its individual components in parallel. The PRISM model checker supports for parallel execution of different processes through parallel composition of its modules and is thus used for modeling and analyzing complex power distribution systems in this paper. The main idea of the proposed approach is to first describe the Markovian models of individual modules, such as transmission line or transformer, of the power distribution system, based on the foundational model of Figure 2, as a PRISM module. These individual modules are then combined in a single PRISM model for their parallel execution. This way, we analyze the reliability of the real-world scenario, where multiple faults can simultaneously occur in various components of a power distribution system. To the best of our knowledge, this kind of an overall power distribution system analysis has not been reported in the open literature so far.

In this section, we first provide a quick overview of the PRISM model checker. This is followed by the description of the properties that we propose to verify using PRISM for the reliability assessment of relay-based protection systems.

4.1 The PRISM Model Checker

PRISM is a probabilistic model checker for the construction and analysis of Markov Chains, Markov Decision Processes (MDPs) and Probabilistic Timed Automatas (PTAs). The models are described using a state-based language called the PRISM language. Modules and variables are basic components of the modeling language. A model can consist of a number of modules whose state at a given time is represented by the values of local variables defined in those modules. The values of local variables of all the modules define the overall state of the system. A set of guarded commands describe the behavior of each module:

$$[\,]\ \ guard \rightarrow prob_1 : update_1 + prob_2 : update_2 ... + prob_n : update_n;$$

The *guard* is a predicate over all the variables and a command is enabled when its guard becomes true. Each $update_i$ defines a possible transition with probability $prob_i$. PRISM provides support for a variety of property specifications such as PCTL, CSL, LTL and PCTL*. For example:

$S_{\geq 0.99}[\text{“}normal\text{”}]$ – “is the steady state probability of *normal* state ≥ 0.99”

PRISM supports verification and analysis of time based properties which we use for the time based analysis of Markovian models. These properties are analyzed by associating a certain reward with each state of the model through a reward structure. PRISM also allows the use of customized properties using the *filter* operator: $filter(op, prop, states)$, where *op* represents the filter operator (min, avg, max), *prop* represents the PRISM property and *states* (optional field) represents the set of states over which to apply the filter.

4.2 Reliability Properties

The purpose of reliability analysis of protective relays deployed in power distribution systems is to minimize the steady-state probabilities of undesirable states and maximize the ones of desirable states. This is achieved by optimizing the values of different testing intervals. The classification between desirable and undesirable states is based on labeling the model states as *normal*, *dependability*, *security* and *unavailability* states where *normal* and *dependability* are the desirable states and *security* and *unavailability* are the undesirable states [8].

In the *normal* state of the model, both the component and the relay are operational with no faults. Thus, State 1 of Figure 3, can conveniently be labeled as the *normal* state. *Dependability* is the probability associated with the correct operation of the relay, i.e., operating when required. Therefore, the states 2, 3, 4, and 5 in Figure 3 can be classified as the *dependability* states because in these states, the relay has operated due to a component fault. *Security*, on the other hand, is the probability of an unnecessary operation of the relay. Undesired operation of the relay occurs due to mal-trip of the relay and thus the states 10, 11, 15 and 16 in Figure 3 are the *security* states. The states in which the relay is either faulty or unavailable due to testing procedures (self-checking or routine test) are labeled as the *unavailability* states because of the fact that the relay is unavailable for operation. The states 6, 7, 8, and 9 of Figure 3 fall into the category of *unavailability* states.

The steady state probabilities of a component model can now be determined using the labels defined above. For example, the steady state probability of *unavailability* state can be obtained by verifying the following property in PRISM:

$S_{=?}$ ["*unavailability*"] – "steady state probability of *unavailability* state"

where, in the case of Figure 3, the *unavailability* would represent the states 6, 7, 8 and 9. When determining the steady state probabilities at the system level, the AND(&) operator is used for *normal* and *dependability* states. For example for the system to be in the healthy condition, all of its component models must be in their respective *normal* states at the same time:

$$S_{=?} \text{ ["}normal_1\text{" \& "}normal_2\text{" \& ... "}normal_i\text{"]}$$

On the other hand, the steady state probabilities of *security* and *unavailability* states are determined using the OR(|) operator because even a single component fault can compromise the security of the whole system:

$$S_{=?} \text{ ["}unavailability_1\text{" | "}unavailability_2\text{" | ... "}unavailability_i\text{"]}$$

Time based properties, such as "fault clearing time", can be analyzed using the reward/cost structures. The reward based properties are defined in combination with the *filter* operator to calculate maximum, minimum and average values of fault clearing time:

$$filter\ (max, R^{time}_{=?}[F \text{"}healthy\_state\text{"}], \text{"}fault\_states\text{"})$$

where *healthy_state* is the state in which all the components are in the normal states and *fault_states* is the state in which one or more components are faulted.

5 Case Study: A Power Distribution Substation

In order to illustrate the effectiveness and utilization of the proposed method for the formal reliability analysis of relay-based protective systems, we present the analysis of a typical power distribution substation, shown in Figure 4. The system consists of three transmission lines and a transformer with one relay associated with every component. The relays are assumed to be digital and equipped with the facilities of routine testing, self-checking and continuous monitoring. We have used the version 4.0.3 of the PRISM model checker running on a i7-2600 3.4 GHz processor, with 4 GB memory, as our verification platform.

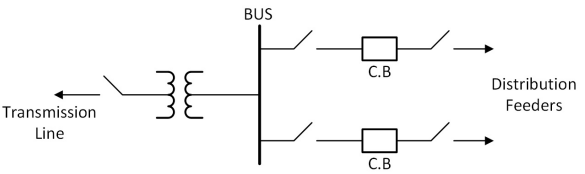


Fig. 4. A Typical Power Distribution Substation

The transmission line model, shown in Figure 5, is obtained by including all three testing procedures in the basic model of Figure 2 according to the modeling approach, described in Section 3. The transformer model is the same as the transmission line model except for the difference in the transition rates and omission of states associated with temporary faults due to their rare occurrence in the case of transformers. In this system, it is assumed that the routine test and self-checking intervals will be the same for all of the four components. The transition rates, based on typical and experimental data, used in our experiments for these models are given in the Appendix. The states of both the transmission line and the transformer models are classified according to the labeling scheme, described in Section 4.2, as shown in Table 1.

Table 1. State Labels

State Label	Transmission Line	Transformer
Normal	1	1
Dependability	2,3,4,5	2,3
Unavailability	6,7,8,9,12,13,14,15,16,17	6,7,8,9,12,13,14,15,16,17
Security	10,11,18,19	10,11,18,19

The optimum value of routine test interval, without taking into consideration self-checking and continuous monitoring, is determined by minimizing the steady state probabilities of states labeled *security* and *unavailability* and maximizing

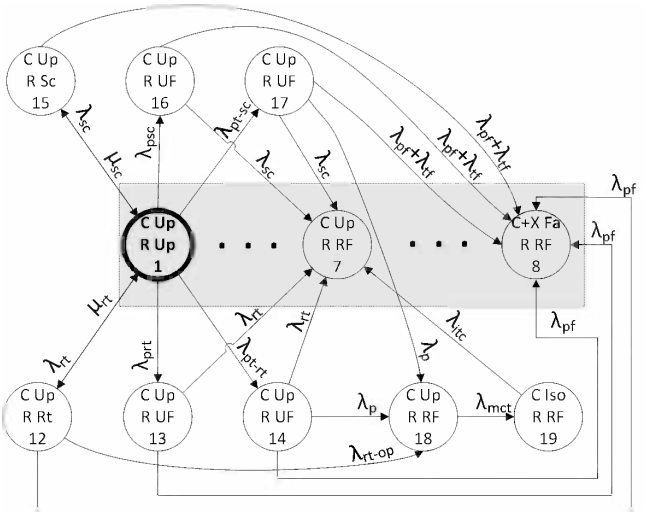


Fig. 5. Transmission Line Model

the steady state probabilities of states labeled *dependability* and *normal* in all of the four modules. This value comes out to be 1228 hours and the results for *normal* and *unavailability* states are shown in Figure 6.

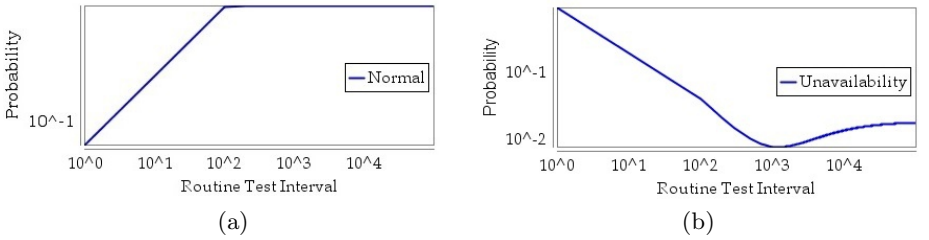


Fig. 6. Optimum Routine Test Interval (a)Normal (b)Unavailability

Self-checking (no continuous monitoring) is included in the system model by assuming the self-checking interval to be 20 hours and the self-checking effectiveness to be 80%. Optimum value of the routine test interval is then determined by minimizing the steady state probability of the undesired states. This value comes out to be 4398 hours which is greater than the case of no self-checking. The routine test is then assumed constant at its optimum value, i.e., 4398, and the optimum value of self-checking interval is determined, which is found to be 41 hours as shown in Figure 7(a). Continuous monitoring (no self-checking) can be included in the model of Figure 5 by adding a transition from state 1 to state 7 as mentioned in Section 3. The impact of including the continuous monitoring

only is that the optimum value of routine test interval comes out the same, i.e., 4398 hours but the probability of the system being in the *unavailability* state is now decreased as shown in Figure 7(b).

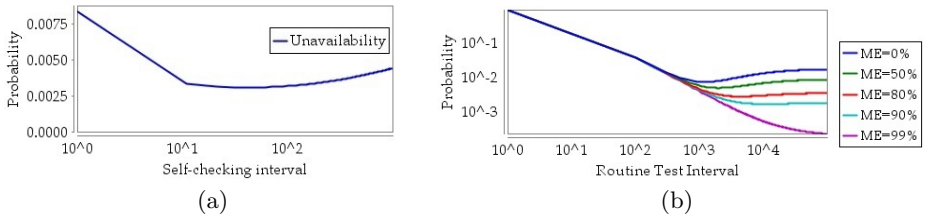


Fig. 7. (a)Optimum Self-checking interval (b)Impact of monitoring effectiveness

Figure 8(a) shows the probability of the *unavailability* state when both self-checking and continuous monitoring facilities are included simultaneously. Self-checking interval of 41 hours is assumed with its effectiveness of 50%. The effectiveness of continuous monitoring is assumed to be 30%. The optimum routine test interval comes out to be 4398 hours. This result is same as obtained previously with only self-checking or only continuous monitoring but the difference is that in this case, the probability of the *unavailability* state is lesser. Sensitivity analysis can also be performed for different values of self-checking effectiveness and the results are shown in Figure 8(b). The probability of being in the *unavailability* state decreases as the effectiveness increases.

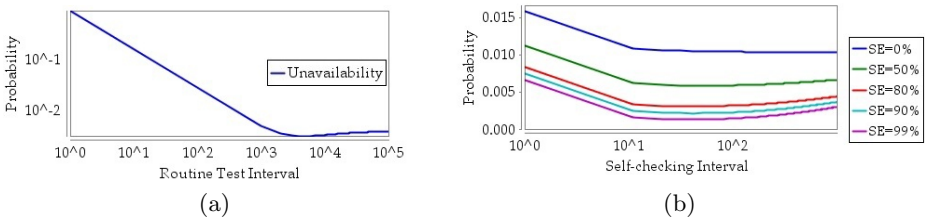


Fig. 8. (a)All three testing procedures included (b)Sensitivity analysis

The permanent fault clearing time is calculated using the *filter* operator in combination with the reward properties. The minimum, average and maximum permanent fault clearing times have been found to be 1.005212032615363, 670.3588473481418 and 4026.7267711779245 hours, respectively. We can also find the steady state probabilities associated with the cases when the system is not in the *healthy_state* due to different number of components not in their *normal* states. Table 2 summarizes these results.

The verification of properties, like the permanent fault clearing time and probability of the system not in the *healthy state*, can only be done using probabilistic

Table 2. Probability of the system not in *healthy_state*

Module in normal state	Steady State Probability
Transmission line 1	$4.249870787 \times 10^{-9}$
Transmission line 1,2	$4.168868359 \times 10^{-6}$
Transmission line 1,2,3	$4.096023769 \times 10^{-3}$
Transmission line 1,2,3 and Transformer	$9.928568273 \times 10^{-1}$

model checking due to its exhaustive state-exploration. Thus, despite providing useful information, these properties, to the best of our knowledge, have not been verified by any other existing reliability analysis approach. Moreover, traditional techniques, like numerical methods and simulation, cannot match the precision of results obtained by the proposed approach for large systems like the one analyzed in this section. The complexity of the analysis can be judged by the fact that the verification of some of the properties, mentioned above, required exploring up to 116603 states and 983364 state-transitions. Finally, the plotting capabilities of PRISM were also found to be very handy.

6 Conclusion

This paper presents a formal reliability analysis approach for power distribution systems that use relays for protection. The main contributions of the paper include a modular approach to construct Markovian models for individual relay-based protected components of a power distribution system and using probabilistic model checking to analyze the reliability of power distribution systems at the system level. The paper presents a foundational model for a relay-based protected component that can have both component and relay faults. This foundational model can be extended to construct the models of most relay-based protected components and the paper presents its extensions for the most commonly used testing procedures, i.e., self-checking, routine test and continuous monitoring. Once the models of all the components of a power distribution system are obtained, they can be translated to the PRISM language to be executed in parallel to judge the reliability of the overall power distribution system using the PRISM model checker. The paper also identifies some key reliability assessment properties of power distribution systems that can be formally verified by PRISM. For illustration purposes, we presented the reliability analysis of a typical power distribution substation. The proposed approach has been found to be more accurate and scalable and it also allows us to verify many novel reliability aspects compared to the other existing reliability analysis approaches for power distribution systems. We plan to use the proposed approach to analyze the reliability of other power distribution systems while considering potential failures in the back-up relays as well. Moreover, the extensions of the proposed foundational model to include other components of power protection system, such as current transformer (CT), voltage transformer (VT), Power Supply Unit (PSU), circuit breaker and the trip coil [1] are also worth exploring.

References

1. Abbarin, A., Fotuhi-Firuzabad, M.: A novel routine test schedule for protective systems using an extended component-based reliability model. In: International Conference on Electrical and Electronics Engineering, pp. 97–102 (2009)
2. Abrial, J.R.: Faultless systems: Yes we can! Computer 42(9), 30–36 (2009)
3. Anderson, P., Agarwal, S.: An improved model for protective-system reliability. IEEE Transactions on Reliability 41(3), 422–426 (1992)
4. Bakshi, A.S., Velayutham, A., Sirivastava, S.C., Agrawal, K.K.: Report of the enquiry committee on grid disturbance in northern region on 30th july 2012 and in northern, eastern & north-eastern region on 31st July. Tech. rep., Ministry of Power Government of India (2012)
5. Billinton, R., Allan, R.N.: Reliability Evaluation of Power Systems. Plenum, New York (1996)
6. Brown, R.: Electric Power Distribution Reliability. CRC Press (2008)
7. Damchi, Y., Sadeh, J.: Considering failure probability for back-up relay in determination of the optimum routine test interval in protective system using markov model. In: IEEE Power Energy Society General Meeting, pp. 1–5 (2009)
8. Udren, E.A., Zipp, J.A., Michel, G.L., Mustaphi, K.K., Nilsson, S.L., Phadke, A.G., Ramaswami, R., Rockefeller, G.D., Sachdev, M.S., Strang, W.M., Thorp, J.S., Tziouvaras, D.A., Varneckas, V., Wagner, C.L.: Proposed statistical performance measures for microprocessor-based transmission-line protective relays. i. explanation of the statistics. IEEE Transactions on Power Delivery 12(1), 134–143 (1997)
9. Endrenyi, J.: Three-state models in power system reliability evaluations. IEEE Transactions on Power Apparatus and Systems 90(4), 1909–1916 (1971)
10. Etemadi, A., Fotuhi-Firuzabad, M.: New considerations in modern protection system quantitative reliability assessment. IEEE Transactions on Power Delivery 25(4), 2213–2222 (2010)
11. Euronews (2013), `http://www.euronews.com/2005/05/25/moscow-hit-by-power-blackout/`
12. Force, U.C.P.S.O.T.: Final report on the 14th august 2003 blackout in the United States and Canada: Causes and recommendations. Tech. rep., U.S. Department of Energy and Natural Resources Canada (2003)
13. Maas, G.A., Mial, M., Fijalkowski, J.: Final report-system disturbance on 4 november 2006. Tech. rep., Union for the Co-ordination of Transmission of Electricity (2006)
14. Grimes, J.: On determining the reliability of protective relay systems. IEEE Transactions on Reliability 19(3), 82–85 (1970)
15. Hermanns, H., Katoen, J., Meyer-Kayser, J., Siegle, M.: Model checking performability properties of markov chains. In: International Conference on Dependable Systems and Networks, pp. 673–673 (2003)
16. Seyedi, H., Fotuhi-Firuzabad, M., Sanaye-Pasand, M.: An extended markov model to determine the reliability of protective system. In: IEEE Power India Conference, pp. 10–12 (2006)
17. Katoen, J., Khattri, M., Zapreev, I.: A markov reward model checker. In: International Conference on the Quantitative Evaluation of Systems, pp. 243–244 (2005)
18. Rutten, J., Kwiatkowska, M., Norman, G., Parker, D.: Mathematical techniques for analyzing concurrent and probabilistic systems. American Mathematical Society (2004)

19. Sen, K., Viswanathan, M., Agha, G.: Vesta: A statistical model-checker and analyzer for probabilistic systems. In: International Conference on the Quantitative Evaluation of Systems, pp. 251–252 (2005)
20. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 585–591. Springer, Heidelberg (2011)
21. Paithankar, Y., Bhide, S.: Fundamentals of Power System Protection. Prentice-Hall (2003)
22. Billinton, R., Fotuhi-Firuzabad, M., Sidhu, T.: Determination of the optimum routine test and self-checking intervals in protective relaying using a reliability model. IEEE Transactions on Power Systems 17(3), 663–669 (2002)
23. Singh, C., Patton, A.D.: Protection system reliability modeling: Unreadiness probability and mean duration of undetected faults. IEEE Transactions on Reliability 29(4), 339–340 (1980)
24. United States Department of Agriculture: Design Guide for rural substations (2001)
25. Younes, H.L.S.: Ymer: A statistical model checker. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 429–433. Springer, Heidelberg (2005)

Appendix: Transition Rates

Rate Parameter	Transformer	Transmission Line
λ_{pf}	3 fault/yr	2 fault/yr
λ_{tf}	—	15 fault/yr
λ_p	0.01 failure/yr	0.01 failure/yr
λ_{ext}	0.0001 failure/yr	0.00001 failure/yr
λ_{inr}	0.0001 failure/yr	—
λ_{exc}	0.0001 failure/yr	—
λ_{cp}	0.000001 failure/hr	0.000001 failure/hr
λ_{rct}	—	10,800 operation/hr
λ_{mct}	30857.14 operation/hr	30857.14 operation/hr
λ_{bct}	21600 operation/hr	8640 operation/hr
λ_{rt-op}	0.001 failure/routine test interval	0.001 failure/routine test interval
λ_{itc}	0.5 operation/hr	0.5 operation/hr
λ_{itx}	0.5 operation/hr	0.5 operation/hr
η	0.1	0.1
μ_{sc}	720 test/hr	720 test/hr
μ_{rt}	1 test/hr	1 test/hr
μ_c	0.1 repair/hr	1 repair/hr
μ_p	1 repair/hr	1 repair/hr
λ_{psc}	$(1-\eta)\lambda_p \times SE$	$(1-\eta)\lambda_p \times SE$
λ_{pmn}	$\lambda_p \times ME$	$\lambda_p \times ME$
λ_{prt}	$(1-\eta)\lambda p \times (1-SE-ME)$	$(1-\eta)\lambda p \times (1-SE-ME)$
λ_{pt-sc}	$\eta\lambda_p \times SE$	$\eta\lambda_p \times SE$
λ_{pt-rt}	$\eta\lambda_p \times (1-SE-ME)$	$\eta\lambda_p \times (1-SE-ME)$

Specification and Verification Using Alloy of Optimistic Access Control for Distributed Collaborative Editors*

Aurel Randolph[1], Abdessamad Imine[2],
Hanifa Boucheneb[1], and Alejandro Quintero[1]

[1] École Polytechnique de Montréal, Montréal, Canada
{aurel.randolph,hanifa.boucheneb,alejandro.quintero}@polymtl.ca
[2] Lorraine University and INRIA Nancy-Grand-Est., France
abdessamad.imine@loria.fr

Abstract. Distributed Collaborative Editors are interactive systems where several and dispersed users edit concurrently shared documents. Generally, these systems rely on data replication and use safe coordination protocol which ensures data consistency even though the users's updates are executed in any order on different copies. Controlling access in such systems is a challenging problem, as they need dynamic access changes and low latency access to shared documents. In [1], a flexible access control protocol is proposed; it is based on replicating the shared document and its authorization policy at the local memory of each user. To deal with latency and dynamic access changes, an optimistic access control technique is used where enforcement of authorizations is retroactive. However, verifying whether the combination of access control and coordination protocols preserves the data consistency is a hard task since it requires examining a large number of situations. In this paper, we specify this access control protocol in the first-order relational logic with Alloy, and we verify that it preserves the correctness of the system on which it is deployed in such a way that the access control policy is enforced identically at all participating user sites and, accordingly, the data consistency remains still maintained.

Keywords: Access control policies, distributed collaborative editors, data consistency, formal specification, formal verification, Alloy.

1 Introduction

Distributed Collaborative Editors (DCE) enable several and dispersed users to form a group for editing simultaneously shared documents, such as articles, wiki pages and program source code (e.g. Google Docs). To achieve data availability, each user owns a local copy of the shared documents. Thus, the collaboration is

* This work is supported by grant number 138732 awarded by the Fonds de Recherche du Québec - Nature et Technologies (FQRNT-Équipe).

C. Pecheur and M. Dierkes (Eds.): FMICS 2013, LNCS 8187, pp. 184–198, 2013.

performed as follows: each user site's updates are locally executed in a non blocking manner and then are propagated to the other sites in order to be executed on remote copies. Although being distributed applications, DCE are specific in the sense they must consider human factors. Moreover, they are characterized by the following features: (i) High local responsiveness: the system has to be as responsive as its single-user editors [2–4]; (ii) High concurrency: the users must be able to concurrently and freely modify any part of the shared document at any time [2, 3]; (iii) Consistency: the users must eventually see a converged view of all copies [2, 3]; (iv) Scalability: a group must be dynamic in the sense that users may join or leave the group at any time. Due to data replication and arbitrary exchange of updates, consistency preservation is one of the most critical properties in DCE. Accordingly, each DCE is endowed with *Coordination Protocol* (CP) to maintain globally consistent state.

Balancing the computing goals of collaboration and access control to shared information is a challenging problem in DCE [5]. Indeed, interaction in collaborative editors is aimed at making shared document available to all who need it, whereas access control seeks to ensure this availability only to users with proper authorization. To preserve the above cited DCE's features and avoid a central authority, a *flexible Access Control Protocol* (ACP) is proposed in [1] where all updates will be checked at each user site without resorting to a central authority. In this model, a user will own two copies: the shared document and its authorization policies. This replication allows for high availability since when users want to read or update the shared document, this manipulation will be granted or denied by controlling only the local copy of the authorization policies. Due to the out-of-order execution of the shared document's updates and the authorization policy's updates, an optimistic approach is used that tolerates momentary violation of access rights but then ensures the copies to be restored in valid states (by undoing invalid document's updates) w.r.t the stabilized access control policy.

To ensure a safe access control in DCE (i.e. permitting legal updates and rejecting illegal updates on the shared document), a protocol stack is built by integrating an ACP on the top of any CP based on data replication and update logging [1]. If we combine a correct CP (i.e. satisfying separately the consistency property) with an ACP: can we verify that the consistency property is preserved by the new protocol? This verification turns out a hard task and unmanageably complicated. Indeed, it requires examining a large number of situations since the updates are performed in different orders on different copies of the shared document and the authorization policy. Consequently, the verification of the combination correctness must be assisted by an automatic checker tool.

Contributions. We propose here a model which specifies concisely the ACP and verify the consistency property of any DCE integrating an ACP on the top of a consistent CP. We use the first-order logic "à la Alloy" to describe symbolically ACP and its environment. This choice is motivated by the possibility to handle symbolically bounded and unbounded variables such as queues of messages, logs, number of sites, number of operations generated by each site, etc. The consistency property is also specified in Alloy language and verified by Alloy analyzer,

using a SAT-based bounded model checking. This technique is established as a good alternative to the classical symbolic model checking using binary decision diagrams (BDDs), as it can often handle much larger systems, by searching for counterexamples of bounded length.

Outline. This paper is organized as follows: Section 2 presents the flexible access control protocol. Section 3 is devoted to the formal specification of ACP and its environment. Section 4 discusses related work. Finally, the conclusion is presented in Section 5.

2 Optimistic Access Control Protocol for DCE

Shared documents are objects whose state can be altered by a set of *cooperative operations* generated by sites. For instance, a shared text document is modified by operations such as inserting a new section, deleting an existing paragraph and replacing an old line by new one. In [1], an access policy is described as an indexed list of authorization rules, where each rule is a quadruple $\langle S, O, R, \omega \rangle$ with (i) S is set of subjects (sites or users), (ii) O is a set of objects (e.g. paragraphs or chapters), (iii) is R a set of access rights (e.g. deleting or updating paragraphs) and (iv) $\omega \in \{-, +\}$. The sign "+" represents a right *attribution* and the sign "−" represents a right *revocation*.

The state of the policy object can be altered by a set of *administrative operations* such as adding and removing authorizations. Administrative operations are generated by the administrator, at any time, and aimed to manage dynamically the right access to the shared documents. These operations are next broadcast to other sites, in order to modify their local copies of the policy object. Thus, on each site, cooperative operations are granted or denied by using the local copy of the policy and applying the first-match semantics: when an operation o is generated, the system checks o against its authorizations one by one, starting from the first authorization and stopping when it reaches the first authorization l that matches o. If no matching authorizations are found, o is rejected. Note that every local policy copy maintains a monotonically increasing version counter that is incremented by every administrative operation performed on this copy.

The collaboration happens in optimistic approach and modifications could be applied in different orders at different sites. The messages are assumed to be exchanged via secure and reliable communication network: each message sent is received by each others without alteration. The flow of messages exchanged during the collaboration is illustrated in Fig. 1.

2.1 Generation of Local Cooperative Requests

Locally, each site can generate some cooperative operations. Each generated cooperative operation is first checked against the local policy. If the operation is revoked then it is said to be invalid and its execution is aborted. When the operation is granted, it is set to valid status in the case of administrator site

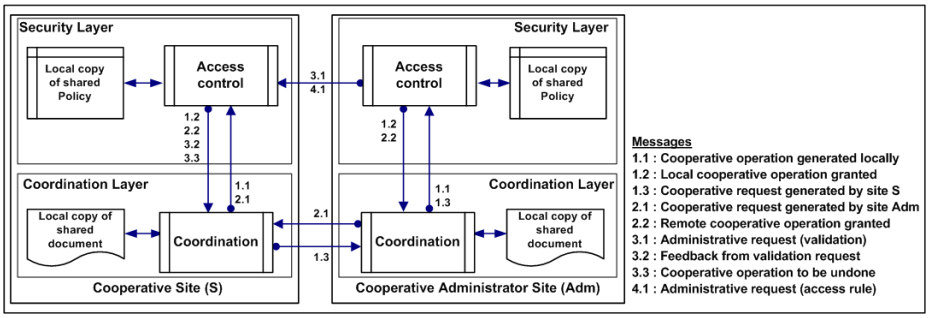


Fig. 1. Flow of collaboration messages

and to tentative status otherwise. The operation is then performed immediately on the local copy of the shared document. A resulting cooperative request is generated and attached with the number version of the policy copy on which the operation is granted. This cooperative request is finally broadcast to other sites.

2.2 Reception of Remote Cooperative Requests

When a remote cooperative request is received, it is first stored in a dedicated queue before being extracted. The request is extracted if it is causally-ready, when its attached version number of policy is less or equal than the current version of the local policy and its precedent cooperative request have been already integrated to the local copy of the shared document. This mechanism is setup to ensure that the access control protocol preserves the causality dependency with respect to precedent administrative requests and precedent cooperative requests.

After its extraction, the remote cooperative request is checked against the local administrative log to verify whether or not it is granted. If the request is granted, its status is set to valid, if the receiver is the administrator, otherwise, its status is tentative. If the receiver is the administrator then the policy version is incremented and a validation request is generated in order to broadcast it to other sites. The new version number is attached to the validation request before its broadcasting. Once, the cooperative operation is performed on the shared document with regard to the collaborative editor's procedures.

2.3 Generation of Administrative Operations

To manage the access control, the administrator produces some access rules called administrative operations. When an administrative operation is generated, the version number is incremented for the administrator's local policy, which is immediately updated by performing on, the generated administrative operation. Once, an administrative request with the corresponding new version number is generated and broadcast to other sites to enforce their own policy.

2.4 Reception of Remote Administrative Requests

There exists two kinds of remote administrative request: validation request and access rule based request. Each received remote administrative request is first stored in a dedicated queue then, extracted when it is causally-ready. The administrative request is said to be causally-ready if the value of its attached policy version number is the next value of the version number of the local policy (the difference is one) and in case of validation request, the corresponding cooperative operation is already executed on this site. Each extracted access rule based request, is performed on the local policy. Thereafter, if the access rule is restrictive, then all tentative cooperative operations, locally generated or received, which are concerned by the rule with regard to the rights, are undone. For the extracted validation request, the status of the corresponding cooperative operation is updated from tentative to valid. At the end of the treatment of the administrative request, the version number of the local policy is incremented.

2.5 Verification Issues

The DCE consists of several sites. Each of them maintains the shared objects and its access right policy, by generating, exchanging and performing some cooperative and administrative operations. As the numbers of sites, cooperatives operations and administrative operations are arbitrary, the queues of cooperative requests and administrative requests are unbounded. The system is then infinite and parameterisable by the number of sites, the number of cooperative operations to be generated by each site, and the number of administrative operations to be generated by the administrator. On each site, the shared objects are modified with respect to the local access right policy. Meanwhile, the local policy is enforced by taking into account the administrative operations generated and broadcast by the administrator. Thus, if the policy is not enforced identically at all sites, it can result in the security hole on the shared objects by permitting illegal modifications or rejecting legal modifications. In addition, this situation can lead to data inconsistency for the collaborative edition such as the document can diverge at the end of the collaboration.

For instance, consider a group composed of an administrator *adm* and two sites S_1 and S_2. Initially, the three sites have the same shared document abc and the same policy object where S_1 is authorized to insert characters (see Fig. 2). Suppose that *adm* revokes the insertion right of S_1 and sends this administrative operation to S_1 and S_2 so that enforce their local policy copies. Concurrently S_1 executes a cooperative operation $Ins(1, x)$ to derive the state xabc as it is granted by its local policy. When *adm* receives the S_1's operation, it will be ignored (as it is not granted by the *adm*s local policy) and then the final state still remain abc. As S_2 receives the S_1's insert operation before its revocation, he gets the state xabc that will be unchanged even after having executed the revocation operation. We are in presence of data inconsistency (the state of *adm* is different from the state of S_1 and S_2) even though the policy object is same in all sites. In fact, the new policy object is not uniformly enforced among all

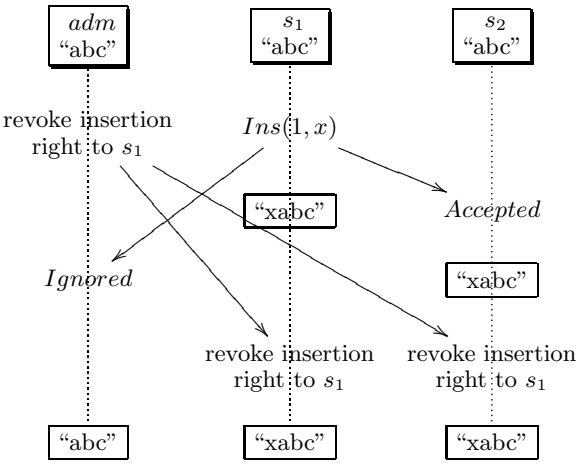


Fig. 2. Divergence caused by introducing administrative operations

sites because of the out-of-order execution of administrative and cooperative operations. Thus, security holes may be created. For instance some sites can accept cooperative operations that are illegal with respect to the new policy (e.g. sites S_1 and S_2).

To solve this problem, the ACP applies the principles of optimistic security [6] in such a way that the enforcement of the new policy may be retroactive with respect to concurrent cooperative operations. In this case, only illegal operations are undone. For instance, $Ins(1, x)$ as shown at 2 should be undone at S_1 and S_2 after the execution of the revocation.

It appears important to verify that the ACP preserves the correctness of the collaborative editing system on which it is deployed with regards to the security issues and data consistency. For this purpose, the sets of legal (valid) cooperative operations must be identical at all sites, when all generated and received cooperative and administrative operations are performed on each site (stable state). Performing such a verification is tricky and hard to do manually. So, the system must be automatically checked using formal methods.

3 Specification and Verification

Several model checking techniques have been proposed in the literature. These techniques can be classified into *explicit state model checker* and *Symbolic model checker*. In explicit state model checker [7], states, sets and relations are explicitly represented, whereas, in symbolic model checker, they are implicitly represented using boolean logic formulas. The category of symbolic model checker can be subdivided into *BDD-based model checkers* [8] and *SAT-based bounded model checkers* [9]. BDD-based model checking allows to prove by considering the whole state space of the model that some property is satisfied but it does not

scale well in practice. SAT-based bounded model checking is considered as a good alternative to BDD-based model checking. It is more appropriate to find bugs in infinite systems. Its basic idea is to search for a counterexample in traces whose length is bounded by some integer k [10]. If no bug is found then k is increased until either a bug is found or the computer resource limits are reached. We propose to use a SAT-based bounded model checker of the tool suite of Alloy, to verify that ACP preserves consistency of DCE.

3.1 Alloy

Alloy [11] is a SAT-based bounded model-checker whose specification language and analyzer are inspired by Z notation [12] and SMV (Symbolic Model Verifier) [13]. The Alloy model consists of signatures, facts, functions and predicates denoted *sig*, *fact*, *fun* and *pred*, respectively. Signatures describe the sets and relations used to specify the system to be verified. Facts represent the constraints of the system that are always assumed to hold. The expected properties of the system are expressed as assertions (constraints) denoted *assert*. The Alloy analyzer is an automatic constraint solver which operates as an instance finder for the specified model that form counterexamples to the assertions. To find such an instance, Alloy proceeds by an exhaustive search over restricted scopes defined by the user [9]. The scope means the maximum number of occurrences assigned to each object of the model, but also means the maximum length of the execution traces. The principle of searching instance is based on the small scope hypothesis which states that an invalid assertion should have a small counterexample [14]. The instance found is reported as counterexample and is guaranteed to be valid. Unfortunately, the failure of finding an instance should not be confused with its absence. Alloy is then useful to specify infinite models and find bugs. The use of signatures and fields like object-oriented programming classes increase its expressiveness [15]. Moreover, Alloy analyzer allows to use several SAT-solvers like SAT4J [16], zChaff [17], MiniSAT [18], Kodkod [19].

3.2 Formal Specification of ACP

The underlying access control model of ACP considers a set of subjects defined as users (or sites) including the administrator, a set of objects denoting a part of or the whole shared document, and a set of access rights. A policy is defined as a function that maps a set of subjects and a set of objects to a set of access rights. On each site, the policy is indexed with a version number which is incremented during the collaboration. In addition to that fundamental components of the model, we have some operations generated and exchanged in the system. There are cooperative operations with three kinds of status (tentative, valid, invalid), access rules and validation requests as administrative requests. For simplification purpose, we consider that the set of objects represents the whole document. Then, it is not necessarily to model the document. Our specification of the fundamental components of the model is shown at **Snippet** 1. From line 1 to 3, we represent the subject. We declare an abstract signature to represent the

generic subject. It is extended to have cooperative site, which is also extended to represent an administrative site. To manage the number of policy, we create the signature *SiteVersion*.

The allowed cooperative operations performed on the shared document could be inserting, deleting, updating, etc. This set of operation types is described with the signature *oper_t* at **Snippet** 2. At the same snippet, cooperative operations are represented from line 6 to 14. Some constraints assumed always to hold are added from line 11. They express that the sender of the operation must not receive it back, the operation has one type and is attached to a version number of policy. To undo an operation we consider a new linked operation. It is specified in our model with the signature *UndoneOp*. The status of cooperative operation is represented by *OpStatus* with the three declinations.

Snippet 1. Specification of subjects

```
1 abstract sig Site {}
2 sig CoopSite extends Site {}
3 one sig AdmSite extends Site {}
4 sig SiteVersion{}
```

Snippet 2. Specification of cooperative operations

```
5  sig oper_t{}//identify a type of operation: insert, delete, etc.
6  sig Coop {
7        from: lone Site, // Site that generates and sends the cooperative operation
8        to: set Site, // Intended recipient(s) of a cooperative operation
9        type:lone oper_t,//Type of operation
10       vers: lone SiteVersion  // Version of local policy which granted the operation
11 }{(from !=none) implies {
12       no from & to and to=Site-from and type!=none and vers!=none and
13       # to > 1 //Allow to have at least 2 sites in the system
14 } }
15 sig UndoneOp{
16       owner: lone Site,
17       linkedOp: lone Coop }
18 abstract sig OpStatus {}
19 one sig tentative, invalid, valid extends OpStatus {}
```

To deal with authorizations, we define a signature *Authorization* extended to have *Plus* for right attribution and *Minus* for right revocation. The object representing the administrative request is abstracted and called *AdReq*. It is extended in validation request and access rule, denoted *Val* and *Rule*, respectively. In the relation *Rule*, rights are of the same type of cooperative operations (*oper_t*). For instance we could have the right of inserting. Rights and authorization (field *signe*) are mapped to the *subject* field to describe an access rule as shown at **Snippet** 3.

In addition to these core elements of the model, we define a global state of the system which consists of the state of each site. We called it *SiteState*. The state of each site is represented by the last number version of its policy, the sending and receiving cooperative operations, the sending and receiving administrative operations (only effective for the administrator), the snapshot view of some queues,

Snippet 3. Specification of administrative requests

```
20 abstract sig Authorization {}
21 one sig Plus, Minus extends Authorization {}
22 abstract sig AdReq{
23       source: lone AdmSite,  // Intended recipient(s) of an administrative request
24       dest: set Site,  //Version of policy when generating the administrative request
25       vers: lone SiteVersion
26 }{(source !=none) implies{
27       no source&dest and dest=Site-source and vers!=none and
28       #dest>1  //Allow to have at least 2 receivers (sites) in the system
29 } }
30 sig Rule extends AdReq{
31       subject: some Site,
32       right: some oper_t,
33       signe: one Authorization }
34 sig Val extends AdReq{op: lone Coop}{ op.from != source }
```

administrative and cooperative operations which are causally-ready at the state. The corresponding signature is described as shown at **Snippet** 4. For the transition system, we create a linear ordering over states by using the Alloy ordering utility module (*open util/ordering[SiteState] as sitesstates*). This module is also used to manage the version number of policy (*open util/ordering[SiteVersion] as versOrder*).

Snippet 4. Specification of the state of the system

```
35 sig SiteState {
36       versions: Site − > one SiteVersion, // Version
37       CoopStatus: Site − > Coop− > OpStatus,  //Status of all cooperative operations
38       sentCoop: Site − > lone Coop, // Cooperative operations sent in this state.
39       sentAdReq: AdmSite − > lone AdReq, // Administrative requests sent.
40       ReceivedCoop: Site − > set Coop,// Cooperative operations received.
41       ReceivedAdReq: Site − > set AdReq, // Administrative requests received.
42       F: Site − > (seq Coop),// Received cooperative operation's queue.
43       Q: Site − > (seq AdReq),// Received administrative request's queue.
44       H: (Site − > set Coop)+(Site − > set UndoneOp), //Cooperative log
45       L: Site − > (seq Rule), // Administrative log.
46       Vr: Site − > (seq Val), // Validation request log.
47       CoopCausallyReady: Site − > lone Coop, // Operations which are causally-ready.
48       AdCausallyReady: Site − > lone AdReq //Causally-ready administrative requests.
49 }
```

The dynamics of the system is specified using *facts*. **Snippet** 5 descibes the generation of cooperative and administrative requests and their reception. We assume that when an operation is sent by a site, it is received by others at the next state of the system. When the cooperative and administrative requests are received, they are stored in appropriate queue and are extracted when there are causally-ready. To define the causally-ready expressions we use functions *FcausallySeq* and *QcausallySeq* for cooperative and administrative requests, respectively. These functions are presented at **Snippet** 6. Once extracted, there are processed.

Snippet 5. Specification of the generation and reception of requests

```
50 fact GenerateCooperativeRequest{
51       all pre: SiteState-sitesstates/last, S:Site, o:pre.sentCoop[S] |
52             let post = sitesstates/next[pre] | {
53                   o.vers = pre.versions[S] //Set the site version to the operation
54                   o in post.H[S] //Add the operation to the owner's H
55                   {(S in CoopSite) and (o in tentative[post.CoopStatus[S]])} or
56                   {(S in AdmSite) and (o in valid[post.CoopStatus[S]])} //Set the status
57                   all Sj:Site-S| o in post.ReceivedCoop[Sj] //Broadcast
58 } }
59 fact GenerateAdministrativeRequest{
60       all pre: SiteState-sitesstates/last, S:AdmSite, a:pre.sentAdReq[S]|
61             let post=sitesstates/next[pre], v=pre.versions[S], vv=versOrder/next[v]|{
62                   vv in post.versions[S] //Incrementation of the version for the next SiteState
63                   a.vers = vv
64                   {(a in Rule) and (post.L[S]= add[pre.L[S], a])} or //Update the policy
65                   {(a in Val) and (post.Vr[S]= add[pre.Vr[S], a])}
66                   all Sj:Site-S| a in post.ReceivedAdReq[Sj] //Broadcast
67 } }
68 fact RequestReception{
69       all post: SiteState-sitesstates/first| let pre = sitesstates/prev[post]|{
70             all S:Site, o:post.ReceivedCoop[S]|{
71                   o not in elems[pre.F[S]] and o in elems[post.F[S]]
72                   all oj: elems[pre.F[S]]|lastIdxOf[pre.F[S],oj]<lastIdxOf[post.F[S],o]}
73             all S:Site, a:post.ReceivedAdReq[S]|{
74                   a not in elems[pre.Q[S]] and a in elems[post.Q[S]]
75                   all aj: elems[pre.Q[S]]|lastIdxOf[pre.Q[S],aj]<lastIdxOf[post.Q[S],a]}}
76 }
```

Snippet 6. Specification of causally-ready expressions

```
77 fun FcausallySeq[st: SiteState, S: Site]: lone Coop {
78       { o: (elems[st.F[S]]- OpStatus[st.CoopStatus[S]])| {
79             versOrder/lte [o.vers, st.versions[S]]
80             all oj: (elems[st.F[S]]- OpStatus[st.CoopStatus[S]]-o)| {
81                   versOrder/lte [oj.vers, st.versions[S]] implies
82                         lastIdxOf [st.F[S],oj]>lastIdxOf [st.F[S],o] }} }
83 }
84 fun QcausallySeq[st: SiteState, S: Site]: lone AdReq {
85       { ad: (elems[st.Q[S]] -elems[st.L[S]]-elems[st.Vr[S]])| {
86             ad.vers= versOrder/next [st.versions[S]]
87             all adj: (elems[st.Q[S]] -elems[st.L[S]]-elems[st.Vr[S]]-ad)| {
88                   (adj.vers= versOrder/next [st.versions[S]]) implies
89                         lastIdxOf [st.Q[S],adj] > lastIdxOf [st.Q[S],ad] }}}
90 }
```

Snippets 7 and 8 present the processing of the causally-ready cooperative operation and administrative request by a non-administrative site, respectively. The processing of a causally-ready cooperative request by the administrative site is shown at **snippet** 9.

Several complementary constraints are defined to control the dynamics of the system. These constraints are not explicitly expressed in the protocol but are necessary from our point of view to prevent unacceptable instances or counterexamples. For instance, at the beginning of the collaboration considered as the initial state, all queues are empty at each site. The complete specification is given in https://sites.google.com/site/laboratoireverifom.

Snippet 7. Processing of causally-ready administrative request by a site

```
fact ReceiveAdminRequest{
    all st: SiteState-sitesstates/first-sitesstates/last,
    S:st.AdCausallyReady.AdReq, ad:st.AdCausallyReady[S]|
        let post=sitesstates/next[st]|{ ad in Val implies {
                ad.op in valid[post.CoopStatus[S]]
                post.Vr[S]= add[st.Vr[S], ad]}
            else {
                post.L[S]= add[st.L[S], ad] //Add to the policy
                ((ad.signe in Minus) and (S in CoopSite)) implies {
                    all oi:st.H[S]&
                    (tentative[st.CoopStatus[S]]-invalid[st.CoopStatus[S]]-
                    valid[st.CoopStatus[S]])|{(oi.type in ad.right and
                        oi.from in ad.subject) implies{
                            oi in invalid[post.CoopStatus[S]]
                            let uo=UndoneOp|{ uo.linkedOp=oi and
                                uo.owner=S and uo in post.H[S]} }}}}
        S->(versOrder/next[ st.versions[S]]) in post.versions }
}
```

Snippet 8. Processing of causally-ready cooperative request by a site

```
fact ReceiveCoopRequestOrdSite{
    all st: (SiteState - sitesstates/first - sitesstates/last),
    o: st.CoopCausallyReady[CoopSite], S:(st.CoopCausallyReady.o)&CoopSite|
        let post= sitesstates/next[st]|{ (o.from in AdmSite) implies {
                o in valid[post.CoopStatus[S]] and o in post.H[S]}
            else{ o.from in CoopSite and
                ((some r: elems[st.L[S]] | {
                    versOrder/lt[o.vers, r.vers] and o.from in r.subject and
                    r.signe in Minus and o.type in r.right and
                    all rj: (elems[st.L[S]]-r) |{{versOrder/lt[o.vers, rj.vers] and
                        o.from in r.subject and o.type in rj.right} implies
                            lastIdxOf [st.L[S],rj] < lastIdxOf [st.L[S],r] }})
                or (no r: elems[st.L[S]] | {
                    versOrder/lt[o.vers, r.vers] and o.from in r.subject and
                    r.signe in Minus and o.type in r.right }))
                implies o in invalid[post.CoopStatus[S]]
                else {o in tentative[post.CoopStatus[S]] and o in post.H[S] }}}
}
```

3.3 Specification of Consistency Property

We deal with an access control model for DCE. Hence, property considered is related to safe and consistent collaboration. Firstly we consider the fundamental concept of stable state of the system. We define it as follows in definition 1.

Snippet 9. Processing of causally-ready cooperative request by the administrator

```
fact fact ReceiveCoopRequestAdm{
    all st: (SiteState - sitesstates/first - sitesstates/last),
    o: st.CoopCausallyReady[AdmSite], S: st.CoopCausallyReady.o&AdmSite|
        let post= sitesstates/next[st]|{(o.from in AdmSite) implies {
                o in valid[post.CoopStatus[S]] and o in post.H[S] }
            else{((some r: elems[st.L[S]] | {
                    versOrder/lt[o.vers, r.vers] and o.from in r.subject and
                    r.signe in Minus and o.type in r.right and
                    all rj: (elems[st.L[S]]-r) |{{versOrder/lt[o.vers, rj.vers] and
                        o.from in r.subject and o.type in rj.right}
                    implies lastIdxOf [st.L[S],rj] < lastIdxOf [st.L[S],r] }})
                or (no r: elems[st.L[S]] | {
                    versOrder/lt[o.vers, r.vers] and o.from in r.subject and
                    r.signe in Minus and o.type in r.right }))
                implies o in invalid[post.CoopStatus[S]]
                else {
                    o in valid[post.CoopStatus[S]]
                    let vad=Val |{
                        vad.op =o and vad.source=S and
                        vad.vers=versOrder/next[ st.versions[S]] and
                        (versOrder/next[st.versions[S]]) in post.versions[S]
                        and vad in st.sentAdReq[S]}
                    o in post.H[S] }}}
}
```

Definition 1. ***(Stable state)*** *The system is said to be in a stable state iff each site completes the processing of all generated and received operations.*

Note that, according to the definition 1, it is possible to have several stable states during the collaboration. The consequence of the definition 1 is that at each stable state of the system, all sites have the same number version of policy. Recall that when an operation is generated, it is immediately sent to others. The broadcasting is considered as part of the generation process unless the operation is denied. Also we have assumed that the communication is message lossless and when an operation is sent by a site, it is received by others at the next state of the system.

Once the stable state is reached by the system, it is possible to verify if the protocol preserves its correctness. The property of interest relies on data consistency. The *data consistency property* allows to know if the protocol is enforced identically at all sites in context of dynamic changes of access rules. The goal is to prevent any security hole with regards to the granted and denied operations and the convergence of the shared document. The data consistency property is satisfied by the model iff for all stable state of the system, for all two disjoint sites, the sets of valid cooperative operations are the same. The data consistency property is specified using assertion (see **Snippet** 10).

As explained in Section 3.1, the scope indicates the maximal number of occurrences of each signature used during the searching, but also the maximal length of the execution trace. In our context, the scope denotes the maximum number of states (*SiteState*), sites, cooperative operations, administrative operations, etc. Considering that the collaboration held with concurrency, at each state, each

site could do an action according to an operation (generate, send, received, process). To analyze the data consistency property, several checking scenarios are used by specifying different scopes. The results state "*No counterexample found. Assertion may be valid*". Note that the SAT-solver used is SAT4J.

Snippet 10. Data Consistency property

```
1 assert ValidPreservation1{
2       all st:Stable[ ]|{ all disj Si, Sj:Site|{
3             valid[st.CoopStatus[Si]] in valid[st.CoopStatus[Sj]]
4             valid[st.CoopStatus[Sj]] in valid[st.CoopStatus[Si]] }}}
```

4 Related Work

Several access control models have been proposed in the literature for collaborative environment. An overview of access control models, including their principles, advantages and potential shortcomings, is available in [5]. Among these models there are Role-Based Access Control (RBAC) [20] and its variants [21–23], which deal with a decentralized authorization. Their drawback is that they do not allow dynamic reassignment of roles. There is very little recent contributions on replicating authorization policies in the literature [24, 25]. These contributions deal with database systems and are not so flexible to support dynamic changes of authorization policies. For instance, Xin T. et al. used in [25], an extension of two-phase locking protocol as concurrency control technique to update policies. This technique is not suitable in the context of DCE.

Alloy has been used in several case study applications in security, access control and security policies domains with regards to model, framework or protocol. In [26], Hu H. et al. presented a case study on verification for the NIST/ANSI standard model for Role Based Access Control (RBAC). The authors verified only one property related to the role deleting. The Alloy analyzer allowed them to conclude that the functional definition of *DeleteRole* function proposed by the NIST/ANSI standard for hierarchical RBAC misses a step for removing inheritance. In [15], authors confirmed the known security vulnerability in oAuth, an open authentication protocol for the Web, using Alloy. Samuel A. et al. [27] specified the Generalized Spatio-Temporal Role Based Access Control (GST-RBAC) model using Alloy. The detection of some conflicts by the analyzer helped them to refine their proposed framework. Similarly, in [28], the authors applied Alloy to specify and verify a spatio-temporal access control correctness. The analyzer showed possible conflict permissions assignment to the same role.

5 Conclusion

In this paper, we have presented a case study on formal specification and verification of a flexible access control protocol running on the top of a DCE. The purpose is to verify that data consistency of the DCE is preserved by the protocol. Proving data consistency of such systems is very challenging as they are

infinite with a high degree of concurrency. To deal with these limitations, we have proposed an abstract model and used Alloy, a SAT-based bounded model-checker. We have shown that ACP preserves the correctness for several scopes. Alloy specification language, based on the first-order relational logic, is very appropriate to describe infinite systems. However, bounded model-checker techniques are known to be useful to find bugs but less appropriate to prove the absence of bugs. In the future, we plan to investigate the existence of a finite abstract model, which preserves data consistency.

References

1. Imine, A., Cherif, A., Rusinowitch, M.: A Flexible Access Control Model for Distributed Collaborative Editors. In: Jonker, W., Petković, M. (eds.) SDM 2009. LNCS, vol. 5776, pp. 89–106. Springer, Heidelberg (2009)
2. Ellis, C.A., Gibbs, S.J.: Concurrency Control in Groupware Systems. In: SIGMOD Conference, vol. 18, pp. 399–407 (1989)
3. Sun, C., Jia, X., Zhang, Y., Yang, Y., Chen, D.: Achieving Convergence, Causality-preservation and Intention-preservation. In: Real-time Cooperative Editing Systems, pp. 63–108. ACM, New York (1998)
4. Sun, C., Xia, S., Sun, D., Chen, D., Shen, H., Cai, W.: Transparent Adaptation of Single-user Applications for Multi-user Real-time Collaboration. ACM Trans. Comput.-Hum. Interact. 13(4), 531–582 (2006)
5. Tolone, W., Ahn, G.J., Pai, T., Hong, S.P.: Access Control in Collaborative Systems. ACM Comput. Surv. 37(1), 29–41 (2005)
6. Povey, D.: Optimistic security: a new access control paradigm. In: Proceedings of the 1999 Workshop on New Security Paradigms, NSPW 1999, pp. 40–45. ACM, New York (2000)
7. Holzmann, G.J.: The Spin Model Checker: Primer and Reference Manual. Addison-Wesley (2004)
8. Cimatti, A., Clarke, E., Giunchiglia, F., Roveri, M.: NUSMV: A New Symbolic Model Checker. International Journal on Software Tools for Technology Transfer 2(4), 410–425 (2000)
9. Schaeffer-Filho, A., Lupu, E., Sloman, M., Eisenbach, S.: Verification of Policy-Based Self-Managed Cell Interactions Using Alloy. In: IEEE International Symposium on Policies for Distributed Systems and Networks, POLICY 2009., pp. 37–40 (2009)
10. Frappier, M., Fraikin, B., Chossart, R., Chane-Yack-Fa, R., Ouenzar, M.: Comparison of Model Checking Tools for Information Systems. In: Dong, J.S., Zhu, H. (eds.) ICFEM 2010. LNCS, vol. 6447, pp. 581–596. Springer, Heidelberg (2010)
11. MIT Sofware Design Group: Alloy : a language and tool for relational models, `http://alloy.mit.edu/alloy/` (accessed : May 5, 2013)
12. Woodcock, J., Davies, J.: Using Z: Specification, Refinement, and Proof. Prentice Hall (1996)
13. Canergie Mellon University: The SMV System, `http://www.cs.cmu.edu/~modelcheck/smv.html` (accessed : May 5, 2013)
14. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. The MIT Press (2006)

15. Pai, S., Sharma, Y., Kumar, S., Pai, R.M., Singh, S.: Formal Verification of OAuth 2.0 Using Alloy Framework. In: 2011 International Conference on Communication Systems and Network Technologies (CSNT), pp. 655–659 (2011)
16. Le Berre, D., Parrain, A.: The Sat4j Library, Release 2.2. Journal on Satisfiability, Boolean Modeling and Computation 7, 59–64 (2010), System description
17. Boolean Satisability Research Group at Princeton: zChaff, `http://www.princeton.edu/~chaff/zchaff.html` (accessed : May 5, 2013)
18. Eén, N., Sörensson, N.: MiniSat, The MiniSat Page, `http://minisat.se/` (accessed : May 5, 2013)
19. Torlak, E., Dennis, G.: Kodkod for Alloy users. In: First ACM Alloy Workshop (2006)
20. Sandhu, R., Coyne, E., Feinstein, H., Youman, C.: Role-Based Access Control Models. Computer 29(2), 38–47 (1996)
21. Joshi, J.B.D., Bhatti, R., Bertino, E., Ghafoor, A.: Access-Control Language for Multidomain Environments. IEEE Internet Computing 8(6), 40–50 (2004)
22. Piromruen, S., Joshi, J.B.D.: An RBAC Framework for Time Constrained Secure Interoperation in Multi-Domain Environments. In: 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems, WORDS 2005, pp. 36–45 (2005)
23. Lee, H.K., Luedemann, H.: Lightweight Decentralized Authorization Model for Inter-Domain Collaborations. In: Proceedings of the 2007 ACM Workshop on Secure Web Services, SWS 2007, pp. 83–89. ACM, New York (2007)
24. Samarati, P., Ammann, P., Jajodia, S.: Maintaining Replicated Authorizations in Distributed Database Systems. Data & Knowledge Engineering 18(1), 55–84 (1996)
25. Xin, T., Ray, I.: A Lattice-Based Approach for Updating Access Control Policies in Real-time. Inf. Syst. 32(5), 755–772 (2007)
26. Hu, H., Ahn, G.: Enabling Verification and Conformance Testing for Access Control Model. In: Proceedings of the 13th ACM Symposium on Access Control Models and Technologies, SACMAT 2008, pp. 195–204. ACM, New York (2008)
27. Samuel, A., Ghafoor, A., Bertino, E.: A framework for specification and verification of generalized spatio-temporal role based access control model. Technical report, Purdue University (2007)
28. Toahchoodee, M., Ray, I., Anastasakis, K., Georg, G., Bordbar, B.: Ensuring Spatio-temporal Access Control for Real-world Applications. In: Proceedings of the 14th ACM Symposium on Access Control Models and Technologies, SACMAT 2009, pp. 13–22. ACM, New York (2009)

Author Index