NASA/CP-2010-216215

# Proceedings of the Second NASA Formal Methods Symposium

*Edited by*
*César Muñoz*
*NASA Langley Research Center, Hampton, Virginia*

April 2010

**NASA STI Program . . . in Profile**

Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA scientific and technical information (STI) program plays a key part in helping NASA maintain this important role.

The NASA STI program operates under the auspices of the Agency Chief Information Officer. It collects, organizes, provides for archiving, and disseminates NASA's STI. The NASA STI program provides access to the NASA Aeronautics and Space Database and its public interface, the NASA Technical Report Server, thus providing one of the largest collections of aeronautical and space science STI in the world. Results are published in both non-NASA channels and by NASA in the NASA STI Report Series, which includes the following report types:

- TECHNICAL PUBLICATION. Reports of completed research or a major significant phase of research that present the results of NASA programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA counterpart of peer-reviewed formal professional papers, but having less stringent limitations on manuscript length and extent of graphic presentations.

- TECHNICAL MEMORANDUM. Scientific and technical findings that are preliminary or of specialized interest, e.g., quick release reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.

- CONTRACTOR REPORT. Scientific and technical findings by NASA-sponsored contractors and grantees.

- CONFERENCE PUBLICATION. Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or co-sponsored by NASA.

- SPECIAL PUBLICATION. Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.

- TECHNICAL TRANSLATION. English-language translations of foreign scientific and technical material pertinent to NASA's mission.

Specialized services also include creating custom thesauri, building customized databases, and organizing and publishing research results.

For more information about the NASA STI program, see the following:

- Access the NASA STI program home page at *http://www.sti.nasa.gov*

- E-mail your question via the Internet to help@sti.nasa.gov

- Fax your question to the NASA STI Help Desk at 443-757-5803

- Phone the NASA STI Help Desk at 443-757-5802

- Write to:
  NASA STI Help Desk
  NASA Center for AeroSpace Information
  7115 Standard Drive
  Hanover, MD 21076-1320

NASA/CP-2010-216215

# Proceedings of the Second NASA Formal Methods Symposium

*Edited by*
*César Muñoz*
*NASA Langley Research Center, Hampton, Virginia*

Proceedings of a symposium sponsored by the
National Aeronautics and Space Administration
and held in Washington, D.C.
April 13-15, 2010

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

April 2010

The use of trademarks or names of manufacturers in this report is for accurate reporting and does not constitute an official endorsement, either expressed or implied, of such products or manufacturers by the National Aeronautics and Space Administration.

Available from:

NASA Center for AeroSpace Information (CASI)
7115 Standard Drive
Hanover, MD 21076-1320
443-757-5802

# Preface

This NASA conference publication contains the proceedings of the Second NASA Formal Methods Symposium (NFM 2010), held at NASA Headquarters, in Washington D.C., USA, on April 13 - 15, 2010.

NFM 2010 is a forum for theoreticians and practitioners from government, academia, and industry, with the goals of identifying challenges and providing solutions to achieving assurance in safety-critical systems. Within NASA, for example, such systems include autonomous robots, separation assurance algorithms for aircraft, and autonomous rendezvous and docking for spacecraft. Moreover, emerging paradigms such as code generation and safety cases are bringing with them new challenges and opportunities. The focus of the symposium is on formal techniques, their theory, current capabilities, and limitations, as well as their application to aerospace, robotics, and other safety-critical systems.

The NASA Formal Methods Symposium (NFM) is a new annual event intended to highlight the state of formal methods art and practice. It follows the earlier Langley Formal Methods Workshop (LFM) series and aims to foster collaboration between NASA researchers and engineers, as well as the wider aerospace, safety-critical and formal methods communities. Since 1990, the proceedings of the symposium and the previous workshop series have been published as NASA Conference Publications:

| | | | |
|---|---|---|---|
| 1990 | First LFM Workshop | Hampton, VA | NASA-CP-10052 |
| 1992 | Second LFM Workshop | Hampton, VA | NASA-CP-10110 |
| 1995 | Third LFM Workshop | Hampton, VA | NASA-CP-10176 |
| 1997 | Fourth LFM Workshop | Hampton, VA | NASA-CP-3356 |
| 2000 | Fifth LFM Workshop | Williamsburg, VA | NASA/CP-2000-210100 |
| 2008 | Sixth LFM Workshop | Newport News, VA | NASA/CP-2008-215309 |
| 2009 | First NFM Symposium | Moffett Field, CA | NASA/CP-2009-215407 |
| 2010 | Second NFM Symposium | Washington, DC | NASA/CP-2010-216215 |

The specific topics covered by NFM 2010 included but were not limited to: formal verication, including theorem proving, model checking, and static analysis; automated testing and simulation techniques; model-based development; techniques and algorithms for scaling formal methods, such as abstraction and symbolic methods, compositional techniques, as well as parallel and/or distributed techniques; code generation; safety cases; accident/safety analysis; formal approaches to fault tolerance; theoretical advances and empirical evaluations of formal methods techniques for safety-critical systems, including hybrid and embedded systems.

Two types of papers were considered: regular papers describing fully developed work and complete results, and short papers describing interesting work in progress or preliminary results. The symposium received 50 submissions (35 regular papers and 15 short papers) out of which 24 were accepted (20 regular, 4 short). All submissions went through a rigorous reviewing process, where each paper was read by 4 reviewers. Submissions and program selections were performed through the EasyChair conference meeting system. In addition to the refereed papers, the symposium featured a welcome speech given by John Kelly (NASA), and three invited talks given by Guillaume Brat (Carnegie-Mellon/NASA) on Verification and Validation of Flight-Critical Systems, John Harrison (Intel) on Formal Methods at Intel - An Overview, and Nikolaj Bjorner (Microsoft) on Decision Engines for Software Analysis using Satisfiability Modulo Theories Solvers. The organizers are grateful to the authors for submitting their work to NFM 2010 and to the keynote speakers for accepting the invitation to present their insights into their research areas.

NFM 2010 would not have been possible without the collaboration of the program committee members and the external reviewers in paper selection, and the support of the NASA Formal Methods community, in particular Ben Di Vito, Jim Rash, and Kristin Rozier, in setting up this event. Special thanks go to Raymond Meyer for the graphical dessign of NFM 2010 visual material.

The NFM 2010 website can be found at `http://shemesh.larc.nasa.gov/NFM2010`.

April 2010

César Muñoz (Program Chair)
Mike Hinchey (Conference Chair)

## Conference Chair

Mike Hinchey (Lero-the Irish Software Engineering Research Centre, Ireland)

## Program Chair

César Muñoz (NASA)

## Local Organization

Ben Di Vito (NASA)
Jim Rash (NASA)

## Program Committee

Gilles Barthe (IMDEA, Spain)
Jonathan Bowen (London South Bank University, UK)
Ricky Butler (NASA, USA)
Charles Consel (INRIA, France)
Ewen Denney (NASA, USA)
Ben Di Vito (NASA, USA)
Jin Song Dong (U. of Singapore, Singapore)
Gilles Dowek (Ecole Polytechnique, France)
Matt Dwyer (U. Nebraska, USA)
Dimitra Giannakopoulou (NASA, USA)
Klaus Havelund (JPL, USA)
Mats Heimdahl (U. Minnesota, USA)
Gerard Holzmann (JPL, USA)
Mike Lowry (NASA, USA)
Josh McNeil (US Army, USA)
John Matthews (Galois Inc., USA)
Natasha Neogi (UIUC, USA)
Corina Pasareanu (NASA, USA)
Charles Pecheur (U. de Louvain, Belgium)
John Penix (Google, USA)
Jim Rash (NASA, USA)
Chris Rouff (Lockheed Martin, USA)
Kristin Rozier (NASA, USA)
Wolfram Schulte (Microsoft, USA)
Koushik Sen (UC Berkeley, USA)
Natarajan Shankar (SRI, USA)
Radu Siminiceanu (NIA, USA)
Doug Smith (Kestrel Institute, USA)
Luis Trevino (Draper Lab, USA)
Caroline Wang (NASA, USA)
Mike Whalen (Rockwell Collins, USA)
Virginie Wiels (ONERA, France)

# External Reviewers

Emilie Balland, Paul Brauner, Jacob Burnim, Damien Cassou, Chunqing Chen, Phan Cong-Vinh, Juan Manuel Crespo, Bruno Dutertre, Quentin Enard, Indradeep Ghosh, Nick Jalbert, Pallavi Joshi, Sudeep Juvekar, César Kunz, Yang Liu, Federico Olmedo, Sam Owre, Chang-Seo Park, Lee Pike, Matt Staats, Christos Stergiou, Jun Sun, Ashish Tiwari, José Vander Meulen, Arnaud Venet, Shaojie Zhang.

# Table of Contents

**Invited Talks** 1

Decision Engines for Software Analysis using Satisfiability Modulo Theories Solvers . . . . . . . . . 1
*Nikolaj Bjørner*

Verification and Validation of Flight-Critical Systems . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2
*Guillaume Brat*

Formal Methods at Intel – An Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
*John Harrison*

**Regular Papers** 4

Automatic Review of Abstract State Machines by Meta Property Verification . . . . . . . . . . . . 4
*Paolo Arcaini, Angelo Gargantini, Elvinia Riccobene*

Hardware-independent Proofs of Numerical Programs . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
*Sylvie Boldo, Thi Minh Tuyen Nguyen*

Slice-based Formal Specification Measures – Mapping Coupling and Cohesion Measures to Formal Z 24
*Andreas Bollin*

How Formal Methods Impels Discovery: A Short History of an Air Traffic Management Project . . 34
*Ricky Butler, George Hagen, Jeffrey Maddalon, César Muñoz, Anthony Narkawicz, Gilles Dowek*

A Machine-Checked Proof of A State-Space Construction Algorithm . . . . . . . . . . . . . . . . . 47
*Nestor Catano, Radu Siminiceanu*

Automated Assume-Guarantee Reasoning for Omega-Regular Systems and Specifications . . . . . 57
*Sagar Chaki, Arie Gurfinkel*

Modeling Regular Replacement for String Constraint Solving . . . . . . . . . . . . . . . . . . . . . . 67
*Xiang Fu, Chung-Chih Li*

Using Integer Clocks to Verify the Timing-Sync Sensor Network Protocol . . . . . . . . . . . . . . 77
*Xiaowan Huang, Anu Singh, Scott A. Smolka*

Can Regulatory Bodies Expect Efficient Help from Formal Methods? . . . . . . . . . . . . . . . . . 87
*Eduardo Rafael López Ruiz, Michel Lemoine*

Synthesis of Greedy Algorithms Using Dominance Relations . . . . . . . . . . . . . . . . . . . . . . 97
*Srinivas Nedunuri, Douglas R. Smith, William R. Cook*

A New Method for Incremental Testing of Finite State Machines . . . . . . . . . . . . . . . . . . . 109
*Lehilton L. C. Pedrosa, Arnaldo V. Moura*

Verification of Faulty Message Passing Systems with Continuous State Space in PVS . . . . . . . . 119
*Concetta Pilotto, Jerome White*

Phase Two Feasibility Study for Software Safety Requirements Analysis Using Model Checking . . 129
*Petra Price, Greg Turgeon*

A Prototype Embedding of Bluespec SystemVerilog in the PVS Theorem Prover . . . . . . . . . . 139
*Dominic Richards, David Lester*

SimCheck: An Expressive Type System for Simulink . . . . . . . . . . . . . . . . . . . . . . . . . . . 149
*Pritam Roy, Natarajan Shankar*

Coverage Metrics for Requirements-Based Testing: Evaluation of Effectiveness . . . . . . . . . . . 161
*Matt Staats, Michael Whalen, Ajitha Rajan, Mats Heimdahl*

Software Model Checking of ARINC-653 Flight Code with MCP . . . . . . . . . . . . . . . . . . . . 171
*Sarah Thompson, Guillaume Brat, Arnaud Venet*

Evaluation of a Guideline by Formal Modelling of Cruise Control System in Event-B . . . . . . . . 182
*Sanaz Yeganefard, Michael Butler, Abdolbaghi Rezazadeh*

Formal Verification of Large Software Systems . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 192
*Xiang Yin, John Knight*

Symbolic Computation of Strongly Connected Components Using Saturation . . . . . . . . . . . . 202
*Yang Zhao, Gianfranco Ciardo*

**Short Papers** **212**

Towards the Formal Verification of a Distributed Real-Time Automotive System . . . 212
*Erik Endres, Christian Müller, Andrey Shadrin, Sergey Tverdyshev*

Slicing AADL Specifications for Model Checking . . . 217
*Viet Yen Nguyen, Thomas Noll, Max Odenbrett*

Model Checking with Edge-valued Decision Diagrams . . . 222
*Pierre Roux, Radu Siminiceanu*

Data-flow based Model Analysis . . . 227
*Christian Saad, Bernhard Bauer*

# Decision Engines for Software Analysis using Satisfiability Modulo Theories Solvers

Nikolaj Bjørner
Microsoft Research
Redmond, Washington, USA

**Abstract**

The area of software analysis, testing and verification is now undergoing a revolution thanks to the use of automated and scalable support for logical methods. A well-recognized premise is that at the core of software analysis engines is invariably a component using logical formulas for describing states and transformations between system states. The process of using this information for discovering and checking program properties (including such important properties as safety and security) amounts to automatic theorem proving. In particular, theorem provers that directly support common software constructs offer a compelling basis. Such provers are commonly called satisfiability modulo theories (SMT) solvers.

Z3 is a state-of-the-art SMT solver. It is developed at Microsoft Research. It can be used to check the satisfiability of logical formulas over one or more theories such as arithmetic, bit-vectors, lists, records and arrays. The talk describes some of the technology behind modern SMT solvers, including the solver Z3. Z3 is currently mainly targeted at solving problems that arise in software analysis and verification. It has been applied to various contexts, such as systems for dynamic symbolic simulation (Pex, SAGE, Vigilante), for program verification and extended static checking (Spec#/Boggie, VCC, HAVOC), for software model checking (Yogi, SLAM), model-based design (FORMULA), security protocol code (F7), program run-time analysis and invariant generation (VS3). We will describe how it integrates support for a variety of theories that arise naturally in the context of the applications. There are several new promising avenues and the talk will touch on some of these and the challenges related to SMT solvers.

# Verification and Validation of Flight-Critical Systems

Guillaume Brat
Carnegie-Mellon University/NASA
Moffett Field, California, USA

**Abstract**

For the first time in many years, the NASA budget presented to congress calls for a focused effort on the verification and validation (V&V) of complex systems. This is mostly motivated by the results of the VVFCS (V&V of Flight-Critical Systems) study, which should materialize as a a concrete effort under the Aviation Safety program. This talk will present the results of the study, from requirements coming out of discussions with the FAA and the Joint Planning and Development Office (JPDO) to technical plan addressing the issue, and its proposed current and future V&V research agenda, which will be addressed by NASA Ames, Langley, and Dryden as well as external partners through NASA Research Announcements (NRA) calls. This agenda calls for pushing V&V earlier in the life cycle and take advantage of formal methods to increase safety and reduce cost of V&V. I will present the on-going research work (especially the four main technical areas: Safety Assurance, Distributed Systems, Authority and Autonomy, and Software-Intensive Systems), possible extensions, and how VVFCS plans on grounding the research in realistic examples, including an intended V&V test-bench based on an Integrated Modular Avionics (IMA) architecture and hosted by Dryden.

# Formal Methods at Intel - An Overview

John Harrison
Intel Corporation
Hillsboro, Portland, USA

**Abstract**

Since the 1990s, Intel has invested heavily in formal methods, which are now deployed in several domains: hardware, software, firmware, protocols etc. Many different formal methods tools and techniques are in active use, including symbolic trajectory evaluation, temporal logic model checking, SMT-style combined decision procedures, and interactive higher-order logic theorem proving. I will try to give a broad overview of some of the formal methods activities taking place at Intel, and describe the challenges of extending formal verification to new areas and of effectively using multiple formal techniques in combination.

# Automatic Review of Abstract State Machines by Meta-Property Verification*

Paolo Arcaini
University of Milan
paolo.arcaini@unimi.it

Angelo Gargantini
University of Bergamo
angelo.gargantini@unibg.it

Elvinia Riccobene
University of Milan
elvinia.riccobene@unimi.it

**Abstract**

A model review is a validation technique aimed at determining if a model is of sufficient quality and allows defects to be identified early in the system development, reducing the cost of fixing them. In this paper we propose a technique to perform *automatic* review of Abstract State Machine (ASM) formal specifications. We first detect a family of typical vulnerabilities and defects a developer can introduce during the modeling activity using the ASMs and we express such faults as the violation of meta-properties that guarantee certain quality attributes of the specification. These meta-properties are then mapped to temporal logic formulas and model checked for their violation. As a proof of concept, we also report the result of applying this ASM review process to several specifications.

## 1 Introduction

Using formal methods, based on rigorous mathematical foundations, for system design and development is of extreme importance, especially for high-integrity systems where safety and security are important. By means of abstract models, faults in the specification can be detected as early as possible with limited effort. Validation should precede the application of more expensive and accurate verification methods, that should be applied only when a designer has enough confidence that the specification really reflects the user perceptions. Otherwise (right) properties could be proved true for a wrong specification.

*Model review*, also called "model walk-through" or "model inspection", is a validation technique in which modeling efforts are critically examined to determine if a model not only fulfills the intended requirements, but also are of sufficient quality to be easy to develop, maintain, and enhance. This process should, therefore, assure a certain degree of quality. The assurance of quality, namely ensuring readability and avoiding error-prone constructs, is one of the most essential aspects in the development of safety-critical reactive systems, since the failure of such systems – often attributable to modeling and, therefore, coding flaws – can cause loss of property or even human life [13]. When model reviews are performed properly, they can have a big payoff because they allow defects to be detected early in the system development, reducing the cost of fixing them.

Usually model review, which comes from the code-review idea, is performed by a group of external qualified people. However, this review process, if done by hand, requires a great effort that might be tremendously reduced if performed in an automatic way – as allowed by using formal notations – by systematically checking specifications for known vulnerabilities or defects. The question is *what* to check on and *how* to automatically check the model. In other words, it is necessary to identify classes of faults and defects to check, and to establish a process by which to detect such deficiencies in the underlying model. If these faults are expressed in terms of formal statements, these can be assumed as a sort of measure of the *model quality assurance*.

In this paper, we tackle the problem of automatically reviewing formal specifications given in terms of Abstract State Machines (ASMs) [4]. We first detect a family of typical vulnerabilities and defects a developer can introduce during the modeling activity using the ASMs and we express such faults as the violation of formal properties. These properties refer to model *attributes* and characteristics that should hold in any ASM model, independently from the particular model to analyze. For this reason they are

*This work was partially supported by the Italian Government under the project PRIN 2007 D-ASAP (2007XKEHFA)

called *meta-properties*. They should be true in order for an ASM model to have the required quality attributes. Therefore, they can be assumed as measures of model quality assurance. Depending on the meta-property, its violation indicates the presence of actual faults, or only of potential faults.

These meta-properties are defined in terms of temporal logic formulas that use two operators, *Always* and *Sometime*, to capture properties that must be true in every state or eventually true in at least one state of the ASM under analysis. Then, we translate these temporal formulas to Computational Tree Logic (CTL) formulas and we exploit the model checking facilities of AsmetaSMV [2, 1], a model checker for ASM models based on NuSMV [5], to check the meta-property violation.

The choice of defining a model review process for the ASM formal method is due to several reasons. First, the ASMs are powerful extensions of the Finite State Machines (FSMs), and it has been shown [4] that they capture the principal models of computation and specification in the literature. Therefore, the results obtained for the ASMs can be adapted to other state-transition based formal approaches. Furthermore, the ASMs are endowed with a set of tools [6, 1] (among which a model checker) which makes it possible to handle and to automate our approach. Finally, ASMs have been widely applied as a formal method for system specification and development, which makes available a certain number of nontrivial specifications on which to test our process.

The Abstract State Machine formal method is briefly presented in Section 2. Section 3 defines a function, later used in the meta-properties definition, that statically computes the firing condition of a transition rule occurring in the model. Meta-properties that are able to guarantee certain quality attributes of a specification are introduced in Section 4. In Section 5, we describe how it is possible to automate our model review process by exploiting the use of a model checker to check the possible violation of meta-properties. As a proof of concept, in Section 6 we report the results of applying our ASM review process to a certain number of specifications, going from benchmark models to test the meta-properties, to ASM models of real case studies of various degree of complexity. In Section 7, we present other works related to the model review process. Section 8 concludes the paper and indicates some future directions of this work.

## 2 Abstract State Machines

Abstract State Machines (ASMs), whose complete presentation can be found in [4], are an extension of FSMs [3], where *states* are multi-sorted first-order structures, i.e. domains of objects with functions and predicates (boolean functions) defined on them, and the *transition relation* is specified by "rules" describing how functions change from one state to the next.

Basically, a transition rule has the form of *guarded update* "**if** *Condition* **then** *Updates*" where *Updates* are a set of function updates of the form $f(t_1,\ldots,t_n) := t$ which are simultaneously executed when *Condition* is true. $f$ is an arbitrary $n$-ary function and $t_1,\ldots,t_n,t$ are first-order terms.

To fire this rule in a state $s_i$, $i \geq 0$, all terms $t_1,\ldots,t_n,t$ are evaluated at $s_i$ to their values, say $v_1,\ldots,v_n,v$, then the value of $f(v_1,\ldots,v_n)$ is updated to $v$, which represents the value of $f(v_1,\ldots,v_n)$ in the next state $s_{i+1}$. Such pairs of a function name $f$, which is fixed by the signature, and an optional argument $(v_1,\ldots,v_n)$, which is formed by a list of dynamic parameter values $v_i$ of whatever type, are called *locations*. They represent the abstract ASM concept of basic object containers (memory units), which abstracts from particular memory addressing and object referencing mechanisms. Location-value pairs $(loc,v)$ are called *updates* and represent the basic units of state change.

There is a limited but powerful set of *rule constructors* that allow to express simultaneous parallel actions (`par`) or sequential actions (`seq`). Appropriate rule constructors also allow non-determinism (existential quantification `choose`) and unrestricted synchronous parallelism (universal quantification `forall`).

A *computation* of an ASM is a finite or infinite sequence $s_0, s_1, \ldots, s_n, \ldots$ of states of the machine, where $s_0$ is an initial state and each $s_{n+1}$ is obtained from $s_n$ by firing simultaneously all of the transition

rules which are enabled in $s_n$. The (unique) *main rule* is a transition rule and represents the starting point of the computation. An ASM can have more than one *initial state*. A state $s$ which belongs to a computation starting from an initial state $s_0$, is said to be *reachable* from $s_0$.

For our purposes, it is important to recall how functions are classified in an ASM model. A first distinction is between *basic* functions which can be *static* (never change during any run of the machine) or *dynamic* (may be changed by the environment or by machine *updates*), and *derived* functions, i.e. those coming with a specification or computation mechanism given in terms of other functions. Dynamic functions are further classified into: *monitored* (only read, as events provided by the environment), *controlled* (read and write (i.e. updated by transaction rules)), *shared* and *output* (only write) functions.

The ASMETA tool set [1] is a set of tools around the ASMs. Among them, the tools involved in our model review process are: the textual notation *AsmetaL*, used to encode fragments of ASM models, and the model checker *AsmetaSMV* [2], which is based on the NuSMV model checker [5] to prove temporal properties on ASM models.

## 3 Rule Firing Condition

In the following we introduce a method to compute, for each rule of the specification under review, the firing condition under which the rule is executed. We introduce a function *Rule Firing Condition* which returns this condition.

$$RFC : Rules \rightarrow Conditions$$

where *Rules* is the set of the rules of the ASM $M$ under review and *Conditions* are boolean predicates over the state of $M$. *RFC* can be statically computed as follows. First we build a static directed graph, similar to a program control flow graph. Every node of the graph is a rule of the ASM and every edge has label $[u]c$ representing the conditions under which the target rule is executed. $c$ is a boolean predicate and $[u]$ is a sequence of logical assignments of the form $v = t$, being $v$ a variable and $t$ a term. The condition $c$ must be evaluated under every logical assignment $v = t$ listed in $u$. Figure 1 reports how to incrementally build the graph, together with the labels for the edges. By starting from the main rule, the entire graph is built, except for the rules that are never used or are not reachable from the main rule and for which the *RFC* evaluates to *false*. We assume that there are no recursive calls of ASM rules, so the graph is *acyclic*. In general, an ASM rule can call itself (directly or indirectly), but rule recursion is seldom used. However, recursion is still supported in derived functions, which are often used in ASM specifications.

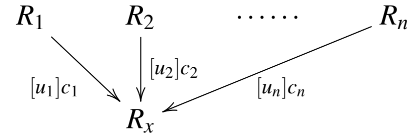
| Rule | Graph | Rule | Graph |
|---|---|---|---|
| Parallel rule $R$<br>$R_1$ **par** $R_2$ | $R \xrightarrow{[]true} R_1$<br>$R \xrightarrow{[]true} R_2$ | Conditional rule $R$<br>**if** $c$ **then** $R_1$ **else** $R_2$ **endif** | $R \xrightarrow{[]c} R_1$<br>$R \xrightarrow{[]\neg c} R_2$ |
| Let rule $R$<br>**let** $x = t$ **in** $R_1$ | $R \xrightarrow{[x=t]true} R_1$ | Forall rule $R$<br>**forall** $x$ **in** $D$ **with** $a$ **do** $R_x$ | $R \xrightarrow{[x=d_1]a} R_x$<br>$R \xrightarrow{[x=d_n]a} R_x$ |
| Macro call rule $R$<br>$R_m[t_1,..,t_n]$ | $R \xrightarrow{[x_1=t_1,\dots,x_n=t_n]true} R_m$ | Choose rule $R$<br>**choose** $x$ **in** $D$ **with** $a$ **do** $R_x$ | $R \xrightarrow{[x=d_1]a} R_x$<br>$R \xrightarrow{[x=d_n]a} R_x$ |

Figure 1: Schemas for building the graph for *RFC*

For this reason the lack of recursive rules does not prevent to write realistic specifications.

To compute the $RFC$ for a rule $R$, one should start from the rule $R$ and visit the graph backward until the main rule is reached. The condition $RFC(R)$ is obtained by applying the following three steps. Initially, $R_x = R$ holds.

1. Expand every occurrence of $RFC(R_x)$ by substituting it with the conditions under which $R_x$ is reached, i.e. the labels of the edges entering the node of $R_x$. If the graph has the schema shown besides, one must substitute $RFC(R_x)$ with $[u_1](RFC(R_1) \wedge c_1) \vee \cdots \vee [u_n](RFC(R_n) \wedge c_n)$

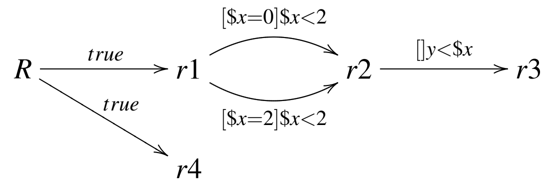



2. Eliminate every logical assignment by applying the following rules:
   - Distribute the $\vee$ (or) over the $\wedge$ (and):
     $$([u_1]A_1 \vee \cdots \vee [u_n]A_n) \wedge B \equiv [u_1](A_1 \wedge B) \vee \cdots \vee [u_n](A_n \wedge B)$$
   - Distribute the assignments: $[u](A \wedge B) \equiv [u]A \wedge [u]B$
   - Apply the assignments: $[u, x = t]A \equiv [u]A[x \leftarrow t]$

3. Apply again 1 until you reach a rule with no entering edges (main rule).

**Example** Consider the following example in which $y$ and $z$ are nullary functions of the machine and $\$x$ is a logical variable. The inner rules are labeled for their concise representation in the graph.

```
main rule R =
 par
  r1: forall $x in {0,2} with $x < 2 do
  r2:     if y < $x then
  r3:             z := y endif
  r4: skip
 endpar
```




To compute the condition under which rule `r3` fires, $RFC(r3)$, one must perform the following steps:

1. Apply the expansion of $RFC(r3)$: $RFC(r3) \equiv RFC(r2) \wedge y < \$x$
2. No assignment to eliminate, expand $RFC(r2)$:
   $$RFC(r3) \equiv ([\$x = 0](RFC(r1) \wedge \$x < 2) \vee [\$x = 2](RFC(r1) \wedge \$x < 2)) \wedge y < \$x$$
3. Distribute the $\vee$ over the $\wedge$:
   $$RFC(r3) \equiv [\$x = 0](RFC(r1) \wedge \$x < 2 \wedge y < \$x) \vee [\$x = 2](RFC(r1) \wedge \$x < 2 \wedge y < \$x)$$
4. Apply the assignments:
   $$RFC(r3) \equiv (RFC(r1) \wedge 0 < 2 \wedge y < 0) \vee (RFC(r1) \wedge 2 < 2 \wedge y < 2)$$
   $$RFC(r3) \equiv (RFC(r1) \wedge y < 0) \vee false$$
5. Expand the definition of $RFC(r1)$ which is *true*:
   $$RFC(r3) \equiv y < 0$$

## 4 Meta-properties

In this section we introduce some properties that should be proved in order to ensure that an ASM specification has some quality attributes. These properties refer to attributes that are defined independently from the particular ASM specification to be analyzed and they should be true in order to guarantee certain degree of quality for the ASM model. For this reason we call them *meta-properties*, and they are formally defined in the following.

The violation of a meta-property always means that a quality attribute is not met and may indicate a potential/actual fault in the model. The severity of such violation depends on the meta-property, each of which measures the degree of model adequacy to the guidelines of ASM modeling style we introduce in this paper in order to use the ASM method for safety critical systems. We have identified the following categories of model quality attributes.

**Consistency** guarantees that locations (memory units) are never simultaneously updated to different values (MP1). This fault is known as *inconsistent updates* and must be removed in order to have a correct model.

**Completeness** requires that every behavior of the system is explicitly modeled. This enforces explicit listing of all the possible conditions in conditional rules (MP2) and the actual updating of controlled locations (MP7).

**Minimality** guarantees that the specification does not contain elements – i.e. transition rules, domain elements, locations, etc. – defined or declared in the model but never used (MP3, MP4, MP5, MP6). Minimality of the state requires that only the necessary state functions are introduced (MP7). These defects are also known as *over-specification*.

### 4.1 Meta-property definition

To formally specify the above attributes in terms of meta-properties we have identified properties that must be true in every state and properties that must be eventually true in at least one state of the ASM under analysis. Given an ASM $M$ and a predicate $\phi$ over a state of $M$, we define the operators *Always* and *Sometime* as follows:

$$
\begin{aligned}
M \models Always(\phi) &= \forall s_0 \in S_0\ \forall s \in \mathscr{R}(s_0):\ \phi(s) \\
M \models Sometime(\phi) &= \exists s_0 \in S_0\ \exists s \in \mathscr{R}(s_0):\ \phi(s)
\end{aligned}
$$

where $S_0$ is the set of initial states of $M$, and $\mathscr{R}(s_0)$ is the set of all the states reachable from $s_0$. In the following we present the meta-properties we have introduced, currently support, and use for automatic review of ASM models.

**MP1. No inconsistent update is ever performed**

An inconsistent update occurs when *two updates clash, i.e. they refer to the same location but are distinct* [7]. If a location is updated by only one rule, no inconsistent update occurs. Otherwise an inconsistent update is possible. Let's see these two examples:

```
main rule r_inc0 =
  par
    l := 1
    l := 2
  endpar
```

```
main rule r_inc1 =
  par
    if cond1 then l(a1) := t1 endif
    if cond2 then l(a2) := t2 endif
  endpar
```

In the first example, the same location $l$ is updated to two different values (1 and 2) in two rules having both conditions *RFC* equal to *true*; in this case, the inconsistent update is apparent. In the second example, instead, to prove that the two updates are consistent, one should prove:

$$Always((cond1 \land cond2 \land a1 = a2) \rightarrow t1 = t2)$$

In general, for every pair of rules $R_1$ and $R_2$ that update two locations $(f, a_1)$ and $(f, a_2)$ to the values $t_1$ and $t_2$ respectively, the property:

$$Always((RFC(R_1) \land RFC(R_2) \land a_1 = a_2) \rightarrow t_1 = t_2) \tag{1}$$

states that the two updates are never inconsistent. The violation of property (1) means that there exists a state in which $R_1$ and $R_2$ fire, $a_1 = a_2$, and $t_1 \neq t_2$.

```
if x > 0 then skip
else
  if x <= 0 then skip endif
endif
```

```
if a and b then skip
else
  if not a then skip endif
endif
```

Figure 2: Complete and incomplete if

**MP2. Every conditional rule must be complete**

In a conditional rule R = **if** $c$ **then** $R_{then}$ **endif**, without *else*, the condition $c$ must be true if $R$ is evaluated. Therefore, in a nested conditional rule, if one does not use the else branch, the last condition must be true. In Fig. 2 the inner conditional rule is complete in the left-hand code, incomplete in the right-hand one, since if $a$ is *true* but $b$ is false, then no branch in the conditional statements is chosen. Property

$$Always(RFC(R) \rightarrow c) \tag{2}$$

states that, when the conditional rule $R$ is executed, its condition $c$ is evaluated to true. A violation of property 2 means that there exists a behavior of the system that satisfies $RFC(R) \wedge \neg c$ but it is not explicitly captured by the model.

**Corollary 1: Every Case Rule without otherwise must be complete** Since the case rule can be reduced, by definition [4], to a series of conditional rules, the computation of $RFC$ is straightforward. The meta-property *MP2* is applied to case rules as follows. Let $R =$ **switch** $t$ **case** $t_1 : R_1$ ... **case** $t_n : R_n$ **endswitch** be a case rule. Its completeness is given by the following property:

$$Always(RFC(R) \rightarrow c_1 \vee c_2 \cdots \vee c_n) \tag{3}$$

where $c_j$ is $t = t_j$ for each $j = 1 \ldots n$. The violation of the property (3) means that there is a state in which the case rule $R$ is executed and none of its conditions is true.

**MP3. Every rule can eventually fire**

Let $R$ be a rule of our ASM model (forall, choose, conditional, update, ...); to verify that $R$ is eventually executed, we must prove the following property:

$$Sometime(RFC(R)) \tag{4}$$

If the property is proved false, it means that rule $R$ is contained in an unreachable model fragment.

**Corollary 2: Every condition in a conditional rule is eventually evaluated to true (and false if the else branch is given)** For every conditional rule, MP3 requires that there exists a path in which its guard is eventually true and, if the else is given, also a path in which its guard is eventually false. In the following example the guard of the inner conditional rule is never true.

```
if x > 0 then
    if x < 0 then skip endif
endif
```

Let $Q =$ **if** $c$ **then** $R_{then}$ $[$**else** $R_{else}]$ **endif** be a conditional rule. The property 4 becomes, for the **then** and **else** part, respectively:

$$Sometime(RFC(Q) \wedge c) \tag{5}$$

$$Sometime(RFC(Q) \wedge \neg c) \tag{6}$$

```
  enum domain State = { AWAITCARD | AWAITPIN | CHOOSE | OUTOFSERVICE | OUTOFMONEY}
  dynamic controlled atmState: State
  dynamic controlled atmInitState: State
  dynamic controlled atmErrState: State
  dynamic monitored pinCode: Integer
  main rule r_Main =
    par
      if(atmState = atmInitState) then atmState := AWAITPIN endif
      if(atmState=AWAITPIN)       then atmState := CHOOSE endif
      if(atmState=CHOOSE)         then atmState := AWAITCARD endif
    endpar

default init s0:
  function atmInitState = AWAITCARD
  function atmErrState = OUTOFSERVICE
  function atmState = atmInitState
```

Figure 3: Over-specified ATM

**MP4. No assignment is always trivial**

An update $l := t$ is trivial [7] if $l$ is already equal to $t$, even before the update is applied. This property requires that each assignment which is eventually performed, will not be always trivial. Let $R = \; l := t$ be an update rule. Property

$$Sometime(RFC(R)) \rightarrow Sometime(RFC(R) \wedge l \neq t) \tag{7}$$

states that, if eventually updated, the location $l$ will be updated to a new value at least in one state. The more simple property $Sometime(RFC(R) \wedge l \neq t)$ would be false if the update is never performed.

**MP5. For every domain element $e$ there exists a location which has value $e$**

Every domain element should be used at least once as location value. In the example of Fig. 3, the element `OUTOFMONEY` of the domain `State` is never used. To check that a domain element $e_j \in D$ is used as location value, if $l_1, \dots, l_n$ are all the locations (possibly defined by different function names) taking value in the domain $D$, the property

$$Sometime(l_1 = e_j \vee l_2 = e_j \vee \ldots \vee l_n = e_j) \tag{8}$$

states that at least a location once takes the value $e_j$. Note that this property must be restricted to domains that are only function co-domains: if the domain $D$ is used as domain of an $n$-ary function with $n > 0$, all its elements have to be considered useful, even if property 8 would be false for some $e_j \in D$. Otherwise, if property 8 is false, the element $e_j$ may be removed from the domain.

**MP6. Every controlled function can take any value in its co-domain**

Every controlled function is assigned at least once to each element in its co-domain; otherwise it could be declared over a smaller co-domain. Let $l_1 \ldots l_m$ be the locations of a controlled function $f$ with co-domain $D = \{e_1, \dots, e_n\}$. Property

$$Sometime(l_1 = e_1 \vee \cdots \vee l_m = e_1) \wedge \ldots \wedge Sometime(l_1 = e_n \vee \ldots \vee l_m = e_n) \tag{9}$$

states that $f$ takes all the values of its co-domain $D$.

**MP7. Every controlled location is updated and every location is read**

This property is obtained combining the results of the previous meta-properties and a static inspection of the model. It is defined by the following table, which also shows what actions the various results suggest.

| **controlled**[1] | **initialized**[2] | **updated**[3] | **always trivial update**[4] | **read**[5] | **Possible actions** |
|---|---|---|---|---|---|
| false | N/A | N/A | N/A | false | remove |
| true | - | false | N/A | false | remove |
| true | true | false | N/A | true | declare static/add an update |
| true | true | true | true | - | declare static |

In the example in Fig. 3 the monitored location *pinCode* is never read; it could be removed. The controlled *atmErrState* location is initialized, but never updated nor read; it could be removed. The controlled *atmInitState* location is initialized, read, but never updated; it could be declared static. Note that if a controlled location is never updated, that may suggest that the specification is incomplete and it misses an update to a part of the controlled state.

## 5 Meta-Property Verification by Model Checking

To verify (or falsify) the meta-properties introduced in the previous section, we use the AsmetaSMV tool [2] which is able to prove temporal properties of ASM specifications by using the model checker NuSMV [5]. The ASM specification $M$ is translated to a NuSMV machine $M_{NuSMV}$ (as explained in [2]) representing the Kripke structure which is model checked to verify a given temporal property. In this paper we use the CTL (Computation Tree Logic) language to express the properties to be verified by NuSMV. In NuSMV, a CTL property $\psi$ is true if and only if $\psi$ is true in every initial state of the machine $M_{NuSMV}$, i.e. $M_{NuSMV} \models \psi$ iff $(M_{NuSMV}, s_0) \models \psi, \forall s_0 \in S_0$, where $S_0$ is the set of initial states of $M_{NuSMV}$. Since the Kripke structure obtained from the ASM may have many initial states, the translation of the meta-properties as defined in Sect. 4.1 into CTL formulas is not straightforward. The translation of $Always(\phi)$ is simply $\texttt{AG}(\phi)$, since $M_{NuSMV} \models \texttt{AG}(\phi)$ means that along all paths starting from each initial state, $\phi$ is true in every state (globally), which corresponds to the definition of *Always*. However, the translation of $Sometime(\phi)$ is not $\texttt{EF}(\phi)$, since $M_{NuSMV} \models \texttt{EF}(\phi)$ means that there exists at least one path starting from *each* initial state containing a state in which $\phi$ is true, while *Sometime* requires only that there exists *at least* an initial state from which $\phi$ will eventually hold. This means that there are cases in which $\texttt{EF}(\phi)$ is false, since not from every initial state $\phi$ will eventually be true, while $Sometime(\phi)$ is true. To prove $Sometime(\phi)$ we use the following equivalence:

$$M \models Sometime(\phi) \Leftrightarrow M_{NuSMV} \not\models \texttt{AG}(\neg\phi)$$

that means that $Sometime(\phi)$ is true if and only if $\texttt{AG}(\neg\phi)$ is false. We run the model checker with the property $P = \texttt{AG}(\neg\phi)$ and if a counter example of $P$ is found, then $Sometime(\phi)$ holds, while if $P$ is proved true, then $Sometime(\phi)$ is false.

## 6 Experimental results

We have implemented a prototype tool, available at [1], that has allowed us to apply our model review process to three different sets of ASM specifications. The first set `Bench` contains only the benchmarks we have explicitly designed to expose the violations of the introduced meta-properties. The set `AsmRep` contains models taken from the ASMETA repository which are also available at [1]. Many ASM case studies of various degree of complexity and several specifications of classical software engineering systems (like ATM, Dining Philosophers, Lift, etc.) are included in `AsmRep`. The last set, `Stu`, contains the

[1] *true* if it is a controlled location, *false* if it is a monitored/static/derived location. Statically checked.
[2] *true* if the location is initialized. It is applicable only to controlled locations. Statically checked.
[3] *true* if the location is updated. It is applicable only to controlled locations. Checked by MP3.
[4] *true* if the update is always trivial. It is applicable only to controlled locations. Checked by MP4.
[5] *true* if the location is read at least in one state. Checked by MP3.

| Spec Set | # spec. | # rules | # violations | violated MPs (# violations) |
|---|---|---|---|---|
| Bench | 21 | 384 | 61 | All |
| AsmRep | 18 | 506 | 29 | MP4(11), MP6(8), MP5(5), MP7(4), MP3(1) |
| Stu | 6 | 172 | 38 | MP7(11), MP5(9), MP6(9), MP1(3), MP3(3), MP4(3) |

Table 1: Experimental results and violations found

models written by our students in a master course in which the ASM method is taught. The results of our experiments are reported in Table 1 which shows the name of the set, the number of models in it, the total number of rules in those models, the number of violations we detected, and the violations found in terms of meta-properties.

As expected our tool was able to detect all the violations in the benchmarks. The student projects contained several faults, most regarding the model minimality but also some inconsistencies which were not detected by model simulation. We found also several violations in the models of AsmRep, all of them regarding model minimality. Note that not all the models in AsmRep could be analyzed, since AsmetaSMV does not support all the AsmetaL constructs and it can analyze only finite models. We plan to use SMT solvers and Bounded Model Checking to analyze ASMs with infinite states.

## 7 Related work

Typical automatic reviews of formal specifications include simple syntax checking and type checking. This kind of analysis is performed by simple algorithms which are able to immediately detect faults like wrong use of types, misspelled variables, and so on. Some complex type systems may require proving of real theorems, like the non-emptiness of PVS types [11]. The review we propose in this paper is more similar to the kind of reviews proposed by Parnas and his colleagues. In a report about the certification of a nuclear plant, he observed that "reviewers spent too much of their time and energy checking for simple, application-independent properties" (like our meta-properties) which distracted them from the more difficult, safety-relevant issues." [12]. Tools that automatically perform such checks can save reviewers considerable time and effort, liberating them to do more creative work.

Our approach has been greatly influenced by the work done by the group lead by Heitmeyer with the Software Cost Reduction (SCR) method. SCR features a tabular notation which can be checked for *completeness* and *consistency* [8]: completeness guarantees that each function is totally defined by its table and consistency guarantees that every value of controlled and internal variables is uniquely defined at every step. In [9] is described a method, similar to ours, to automatically verify the consistency of a software requirements specification (SRS) written in an SCR-style; properties that describe the consistency of the model are defined *structural properties*. The SRS document is translated into a PVS model where, for each structural property, a PVS theorem is declared. The verification of structural properties is carried out through the proof of PVS theorems and, for one property, through the model checking of a CTL property.

Other approaches try to apply analyses similar to those performed in SCR to non-tabular notations. In [13], the authors present a set of robustness rules (like UML well-formedness rules) that seek to avoid common types of errors by ruling out certain modeling constructs for UML state machines or Statecharts. Structural checks over Statecharts models can be formulated by OCL constraints which, if complex, must be proved by theorem proving. Their work and ours extend the use of meta-properties not only to guarantee correctness but also to assure high quality standards in case the models are to be used for safety critical applications.

## 8 Conclusions and Future work

We have presented a method to perform automatic model review of ASM specifications. This process has the aim of guarantee certain quality attributes of models. A given quality attribute is captured by a meta-property expressed in terms of a CTL formula. The AsmetaSMV model checker for ASMs is used to detect a possible violation of this meta-property and, therefore, the presence of a possible defect in the model. These meta-properties can be assumed as measures of model quality assurance.

In the future we plan to improve our process in the following directions. One of the typical shortcomings introduced by a not ASM-expert when modeling with ASMs is the use of the `seq` rule constructor when `par` could be used, instead. This is usually due by a wrong understanding of the simultaneous parallel execution of function updates. The correct use of a `par` instead of a `seq` can improve the quality of a model in terms of abstraction and minimality. So we plan to investigate this kind of defect and define suitable meta-properties able to detect it. Another future plan regards the *vacuity detection* [10] of (temporal) properties which can be specified for an ASM model. We plan to investigate if it is possible to detect property vacuity before proving properties. To this purpose, an integration of the AsmetaSMV system with existing tools able to detect vacuity could be possible.

## References

[1] The ASMETA website. `http://asmeta.sourceforge.net/`, 2010.

[2] P. Arcaini, A. Gargantini, and E. Riccobene. AsmetaSMV: a way to link high-level ASM models to low-level NuSMV specifications. In M. Frappier, U. Glässer, S. Khurshid, R. Laleau, and S. Reeves, editors, *Abstract State Machines, Alloy, B and Z, Second Inter. Conference, ABZ 2010*, volume 5977 of *Lecture Notes in Computer Science*, pages 61–74. Springer, 2010.

[3] E. Börger. The ASM method for system design and analysis. A tutorial introduction. In B. Gramlich, editor, *Frontiers of Combining Systems, 5th International Workshop, FroCoS 2005, Vienna, Austria, September 19-21, 2005, Proceedings*, volume 3717 of *Lecture Notes in Computer Science*, pages 264–283. Springer, 2005.

[4] E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer Verlag, 2003.

[5] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proc. International Conference on Computer-Aided Verification (CAV 2002)*, volume 2404 of *LNCS*. Springer, July 2002.

[6] A. Gargantini, E. Riccobene, and P. Scandurra. Model-driven language engineering: The ASMETA case study. In *International Conference on Software Engineering Advances, ICSEA*, pages 373–378, 2008.

[7] Y. Gurevich. Sequential abstract-state machines capture sequential algorithms. *ACM Trans. Comput. Logic*, 1(1):77–111, 2000.

[8] C. Heitmeyer, R. Jeffords, and B. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, July 1996.

[9] T. Kim and S. D. Cha. Automated structural analysis of SCR-style software requirements specifications using PVS. *Softw. Test, Verif. Reliab*, 11(3):143–163, 2001.

[10] O. Kupferman and M. Y. Vardi. Vacuity detection in temporal model checking. *International Journal on Software Tools for Technology Transfer (STTT)*, 4(2):224–233, 2003.

[11] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In *11th International Conference on Automated Deduction (CADE-11)*, pages 748–752, London, UK, 1992. Springer-Verlag.

[12] D. L. Parnas. Some theorems we should prove. In *HUG '93: 6th International Workshop on Higher Order Logic Theorem Proving and its Applications*, pages 155–162, London, UK, 1994. Springer-Verlag.

[13] S. Prochnow, G. Schaefer, K. Bell, and R. von Hanxleden. Analyzing robustness of UML state machines. In *Workshop on Modeling and Analysis of Real-Time and Embedded Systems (MARTES 06)*, 2006.

# Hardware-independent proofs of numerical programs *

Sylvie Boldo Thi Minh Tuyen Nguyen
INRIA Saclay – Île-de-France, F-91893 Orsay cedex, France
LRI, Univ. Paris-Sud, F-91405 Orsay cedex, France

**Abstract**

On recent architectures, a numerical program may give different answers depending on the execution hardware and the compilation. Our goal is to formally prove properties about numerical programs that are true for multiple architectures and compilers. We propose an approach that states the rounding error of each floating-point computation whatever the environment. This approach is implemented in the Frama-C platform for static analysis of C code. Small case studies using this approach are entirely and automatically proved.

## 1 Introduction

Floating-point computations often appear in current critical systems from domains such as physics, aerospace system, energy, etc. For such systems, hardwares and softwares play an important role.

All current microprocessor architectures support IEEE-754 floating-point arithmetic [1]. However, there exist some architecture-dependent issues. For example, the x87 floating-point unit uses the 80-bit internal floating-point registers on the Intel platform. The fused multiply-add (FMA) instruction, supported by the PowerPC and the Intel Itanium architectures, computes $xy \pm z$ with a single rounding. These issues can introduce subtle inconsistencies between program executions. This means that the floating-point computations of a program running on different architectures may be different [2].

Static analysis is an approach for checking a program without running it. Deductive verification techniques which perform static analysis of code, rely on the ability of theorem provers to check validity of formulas in first-order logic or even more expressive logics. They usually come with expressive specification languages such as JML [3, 4] for Java, ACSL [5] for C, Spec# [6] for C#, etc. to specify the requirements. For automatic analysis of floating-point codes, a successful approach is abstract interpretation based static analysis, that includes Astrée [7, 8] and Fluctuat [9].

Floating-point arithmetic has been formalized since 1989 in order to formally prove hardware components or algorithms [10, 11, 12]. There exist less works on specifying and proving behavioral properties of floating-point programs in deductive verification systems. A work on floating-point in JML for Java is presented in 2006 by Leavens [13]. Another proposal has been made in 2007 by Boldo and Filliâtre [14]. Ayad and Marché extended this to increase genericity and handle exceptional behaviors [15].

However, these works only follow the strict IEEE-754 standard, with neither FMA, nor extended registers. Correctly defining the semantics of the common implementations of floating-point is tricky, because semantics may change according to arguments of compilers and processors. As a result, formal verification of such programs is a challenge. The purpose of this paper is to present an approach to prove numerical programs with few restrictions on the compiler and the processor.

More precisely, we require the compiler to preserve the order of operations of the C language and we only consider rounding-to-nearest mode, double precision numbers and computations. Our approach is implemented in the Frama-C platform[1] associated with Why [16] for static analysis of C code.

This paper is organized as follows. Section 2 presents some basic knowledge needed about floating-point arithmetic, including the x87 unit and the FMA. Section 3 presents a bound on the rounding error

*This work was supported by the Hisseo project, funded by Digiteo (`http://hisseo.saclay.inria.fr/`).

[1] `http://frama-c.cea.fr/`

of a computation in all possible cases (extended registers, FMA). Two small case studies are presented in Section 4. These examples illustrate our approach and show the difference of the results between the usual (but maybe incorrect) model and our approach.

## 2 Floating-point Arithmetic

### 2.1 The IEEE-754 floating-point standard

The IEEE-754 standard [1] for floating-point arithmetic was developed to define formats and behaviors for floating-point numbers and computations. A floating-point number $x$ in a format $(p, e_{min}, e_{max})$, where $e_{min}$ and $e_{max}$ are the minimal and maximal unbiased exponents and $p$ is the precision, is represented by the triplet $(s, m, e)$ so that

$$x = (-1)^s \times 2^e \times m \tag{1}$$

where $s \in \{0, 1\}$ is the sign of $x$, $e$ is any integer so that $e_{min} \leq e \leq e_{max}$, $m$ $(0 \leq m < 2)$ is the significand (in $p$ bits) of the representation.

We only consider the binary 64-bit format (usually *double* in C or Java language), that satisfies the format (1) with $(53, -1022, 1023)$, as it concentrates all the problems. Our ideas could be re-used in other formats.

When approximating a real number $x$ by its rounding $\circ(x)$, a rounding error happens. We here consider only round-to-nearest mode, that includes both the default rounding mode (round-to-nearest, ties to even) and the new round-to-nearest, ties away from zero, of the revision of the IEEE-754 standard. In radix 2 and round-to-nearest mode, a bound on the error is known [17].

If $|\circ(x)| \geq 2^{e_{min}}$, $x$ is called normal number. In other words, it is said to be in the normal range. We then bound the relative error:

$$\left|\frac{x - \circ(x)}{x}\right| \leq 2^{-p}. \tag{2}$$

For smaller $|\circ(x)|$, the value of the relative error becomes large (up to 0.5). In that case, $x$ is in the subnormal range and we prefer a bound based on the absolute error:

$$|x - \circ(x)| \leq 2^{e_{min} - p}. \tag{3}$$

### 2.2 Floating-point computations depend on the architecture

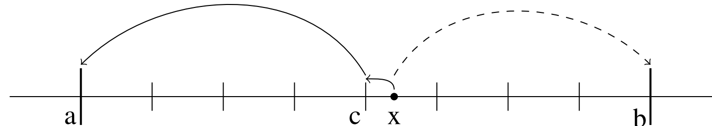
With the same program containing floating-point computations, the result may be different depending on the compiler and the processor. We present in this section some architecture-dependent issues resulting in such problems.

A first cause is the fact that some processors (IBM PowerPC or Intel/HP Itanium) have a *fused multiply-add* (FMA) instruction which computes $(x \times y) \pm z$ as if with unbounded range and precision, and rounds only once to the destination format. This operation can speed up and improve the accuracy of dot product, matrix multiplication and polynomial evaluation, but few processors now support it. But how should $a \times b + c \times d$ be computed? When a FMA is available, the compiler may choose either $\circ(a \times b + \circ(c \times d))$, or $\circ(\circ(a \times b) + c \times d)$, or $\circ(\circ(a \times b) + \circ(c \times d))$ which may give different results.

Another well-known cause of discrepancy happens in the IA32 architecture (Intel 386, 486, Pentium etc.) [2]. The IA32 processors feature a floating-point unit called "x87". This unit has 80-bit registers in "double extended" format (64-bit significand and 15-bit exponent), often associated to the *long double* C type. When using the x87 mode, the intermediate calculations are computed and stored in the x87 registers (80 bits). The final result is rounded to the destination format. Extended registers may also lead to double rounding, where floating-point results are rounded twice. For instance, the operations are

computed in the *long double* type of x87 floating-point registers, then rounded to IEEE double precision type for storage in memory. Double rounding may yield different result from direct rounding to the destination type.

An example is given in the following figure: we assume $x$ is near the midpoint $c$ of two consecutive floating-point numbers $a$ and $b$ in the destination format. Using round-to-nearest, with single rounding, $x$ is rounded to $b$. However, with double rounding, it may firstly be rounded towards the middle $c$ and then be rounded to $a$ (if $a$ is even). The results obtained in the two cases are different.




This is illustrated by the program of Figure 1. In this example, $y = 2^{-53} + 2^{-64}$ and $x$ are exactly representable in double precision. The values 1 and $1+2^{-52}$ are two consecutive floating-point numbers. With strict IEEE-754 double precision computations for `double` type, the result obtained is $z = 1+2^{-52}$. Otherwise, on IA32, if the computations on `double` is performed in the `long double` type inside x87 unit, then converted to double precision, $z = 1.0$.

```
int main(){
  double x = 1.0;
  double y = 0x1p-53 + 0x1p-64; // y = 2^-53 + 2^-64
  double z = x + y;
  printf("z=%a\n",z);
}
```

Figure 1: A simple program giving different answers depending on the architecture.

Another example which gives inconsistencies in result between x87 and SSE[2] is presented in Figure 2. This example will be presented and reused in Section 4. In this example, we have a function `int sign(double x)` which returns a value which is either $-1$ if $x < 0$, or 1 if $x \geq 0$. The function `int eps_line(double sx, double sy, double vx, double vy)` then makes a direction decision depending on the sign of a few floating-point computations. We execute this program on SSE unit and obtain that `Result = 1`. When it is performed on IA32 inside x87 unit, the result is `Result = -1`.

```
int sign(double x) {
  if (x >= 0) return 1;
  else return -1;
}
int eps_line(double sx,double sy,double vx,double vy){
  return sign(sx*vx+sy*vy)*sign(sx*vy-sy*vx);
}
int main(){
  double sx = -0x1.0000000000001p0;   // sx = -1 - 2^-52
  double vx = -1.0;
  double sy = 1.0;
  double vy =  0x1.fffffffffffffp -1; // vy = 1 - 2^-53
  int r = eps_line(sx,sy,vx,vy);
  printf("Result = %d\n",r);
}
```

Figure 2: A more complex program giving different answers depending on the architecture.

# 3 Hardware-independent bounds for floating-point computations

As the result of floating-point computations may depend on the compiler and the architecture, static analysis is the perfect tool, as it will verify the program without running it, therefore without enforcing the architecture or the compiler. As we want both correct and interesting properties on a floating-point computation without knowing which rounding will be in fact executed, the chosen approach is to consider only the rounding error. This will be insufficient in some cases, but we believe this can give useful and sufficient results in most cases.

## 3.1 Rounding error in 64-bit rounding, 80-bit rounding and double rounding

We know that the choice between 64-bit, 80-bit and double rounding is the main reason that causes the discrepancies of result. We prove a rounding error bound that is valid whatever the hardware, and the chosen rounding. We denote by $\circ_{64}$ the round-to-nearest in the `double` 64-bit type. We denote by $\circ_{80}$ the round-to-nearest to the extended 80-bit registers.

**Theorem 1.** *For a real number $x$, let $\Box(x)$ be either $\circ_{64}(x)$, or $\circ_{80}(x)$, or the double rounding $\circ_{64}(\circ_{80}(x))$.*

$$\text{We have either } \left(|x| \geq 2^{-1022} \text{ and } \left|\frac{x-\Box(x)}{x}\right| \leq 2050 \times 2^{-64} \text{ and } |\Box(x)| \geq 2^{-1022}\right)$$
$$\text{or } \left(|x| \leq 2^{-1022} \text{ and } |x-\Box(x)| \leq 2049 \times 2^{-1086} \text{ and } |\Box(x)| \leq 2^{-1022}\right).$$

This theorem is the basis of our approach to correctly prove numerical programs whatever the hardware. These bounds are tight as they are reached in all cases where $\Box$ is the double rounding. They are a little bigger than the ones for 64-bit rounding (2050 and 2049 instead of 2048) for both cases. These bounds are therefore both correct, very tight, and just above the 64-bit's.

As $2^{-1022}$ is a floating-point number, we have $\Box(2^{-1022}) = 2^{-1022}$. As all rounding modes are monotone[2], $\Box(x)$ is also monotone. Then $|x| \geq 2^{-1022}$ implies $|\Box(x)| \geq 2^{-1022}$ and vice versa.

Now let us prove the bounds of Theorem 1 on the rounding error for all possible values of $\Box$.

### 3.1.1 Case 1, $\Box(x) = \circ_{64}(x)$: Rounding error in 64-bit rounding

Here, we have $p = 53$ and $e_{min} = -1022$. Therefore the results of Section 2.1 give the following error bounds that are smaller than the desired ones. Therefore Theorem 1 holds in 64-bit rounding.

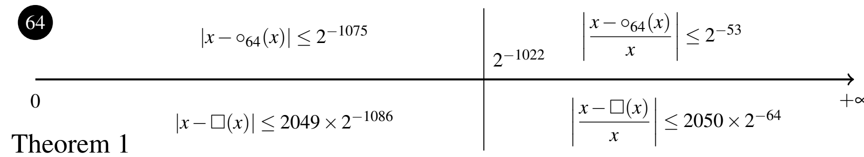



Figure 3: Rounding error in 64-bit rounding vs. Theorem 1

### 3.1.2 Case 2, $\Box(x) = \circ_{80}$: Rounding error in 80-bit rounding

We consider here the 80-bit registers used in x87. They have a 64-bit significand and a 15-bit exponent. Thus, $p = 64$ and the smallest positive number in normal range is $2^{-16382}$.

The error bounds of Section 2.1 are much smaller in this case than Theorem 1's except in the case where $|x|$ is between $2^{-16382}$ and $2^{-1022}$. There, we have $|x - \circ(x)| \leq 2^{-64} \times |x| \leq 2^{-64} \times 2^{-1022} = 2^{-1086}$.

[2] A monotone function $f$ is a function such that, for all $x$ and $y$, $x \leq y$ implies $f(x) \leq f(y)$.

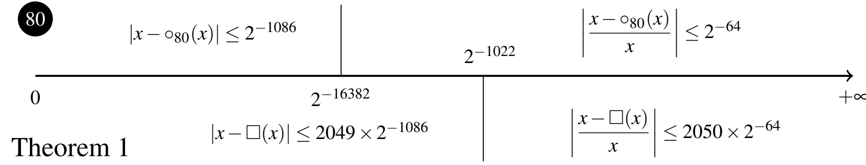



Figure 4: Rounding error in 80-bit rounding vs. Theorem 1

So, all bounds are much smaller than that of Theorem 1 so Theorem 1 holds in 80-bit rounding.

#### 3.1.3 Case 3, $\square(x) = \circ_{64}(\circ_{80}(x))$: Rounding error in double rounding

The bounds here will be that of Theorem 1. We split in two cases depending on the value of $|x|$.

**Normal range.** We first assume that $|x| \geq 2^{-1022}$. We bound the relative error by some computations and the previous formulas:

$$\left|\frac{x-\circ_{64}(\circ_{80}(x))}{x}\right| \leq \left|\frac{x-\circ_{80}(x)}{x}\right| + \left|\frac{\circ_{80}(x)-\circ_{64}(\circ_{80}(x))}{x}\right| \leq 2^{-64} + \left|\frac{\circ_{80}(x)-\circ_{64}(\circ_{80}(x))}{\circ_{80}(x)} \times \frac{\circ_{80}(x)}{x}\right|$$
$$\leq 2^{-64} + 2^{-53} \times (2^{-64}+1) \leq 2050 \times 2^{-64}$$

**Subnormal range.** We now assume that $|x| \leq 2^{-1022}$. The absolute error to bound is $|x-\circ_{64}(\circ_{80}(x))|$. We have two cases depending on whether $x$ is in the 80-bit normal or subnormal range.

If $x$ is in the 80-bit subnormal range, then $|x| < 2^{-16382}$ and
$|x-\circ_{64}(\circ_{80}(x))| \leq |x-\circ_{80}(x)| + |\circ_{80}(x)-\circ_{64}(\circ_{80}(x))| \leq 2^{-1086} + 2^{-1075} \leq 2049 \times 2^{-1086}$.

If $x$ is in the 80-bit normal range, then $2^{-16382} \leq |x| < 2^{-1022}$ and
$|x-\circ_{64}(\circ_{80}(x))| \leq |x-\circ_{80}(x)| + |\circ_{80}(x)-\circ_{64}(\circ_{80}(x))| \leq 2^{-64} \times |x| + 2^{-1075} \leq 2049 \times 2^{-1086}$.

Then Theorem 1 holds in double rounding. In conclusion, Theorem 1 is proved for all 3 roundings.

We use the Coq library developed with the Gappa tool with the help of the Gappa tactic [18] to prove the correctness of Theorem 1. The corresponding theorem and proof (228 lines) in Coq is available at `http://www.lri.fr/~nguyen/research/rnd_64_80_post.html`. The formal proof exactly corresponds to the one described in the preceding Section. It is not very difficult, but needs many computations and a very large number of subcases. The formal proof gives a very strong guarantee on this result, allowing its use in the Frama-C platform.

### 3.2 Hardware and compiler-independent proofs of numerical programs

#### 3.2.1 Rounding error in presence of FMA

Theorem 1 gives rounding error formulas for various roundings denoted by $\square$ (64-bit, 80-bit and double rounding). Now, let us consider the FMA that computes $x \times y + z$ with one single rounding. The question is whether a FMA was used in a computation. We therefore need an error bound that covers all the possible cases.

The idea is very simple: we consider a FMA as a *rounded* multiplication followed by a rounded addition. And we only have to consider another possible "rounding" that is the identity: $\square(x) = x$.

This specific "rounding" magically solves the FMA problem: the result of a FMA is $\square_1(x \times y + z)$, that may be considered as $\square_1(\square_2(x \times y) + z)$ with $\square_2$ being the identity. So we handle in the same way all operations even in presence of FMA or not, by considering one rounding for each basic operation (addition, multiplication...). Of course, this "rounding" easily verifies the formulas of Theorem 1.

#### 3.2.2 Proofs of programs

What is the use of this odd rounding? The idea is that each *basic* operation (addition, subtraction, multiplication, division and square root) will be considered as rounded with a $\square$ that may be one of the four possible roundings. Let us go back to the computation of `a*b+c*d`: it becomes $\square(\square(a \times b) + \square(c \times d))$ with each $\square$ being one of the 4 roundings. It gives us 64 possibilities. In fact, only 45 possibilities are allowed (for example, the addition cannot be exact). But *all* the real possibilities are *included* in all the considered possibilities. And all considered possibilities have a rounding error bounded by Theorem 1.

So, by considering the identity as a rounding like the others, we handle all the possible uses of the FMA in the same way as we handle multiple roundings. Note that absolute value and negation may produce a rounding if we put a 80-bit number into a 64-bit number.

The idea now is to do forward analysis of the rounding errors, that is to say propagate the errors and bound them at each step of the computation. Therefore, we have put the formulas of Theorem 1 as postconditions of each floating-point operation in the Frama-C platform to look into the rounding error of the whole program. Based on the work of [15], we create a new "pragma" called `multirounding` to implements this. Ordinarily, the pragma directive is the method specified by the C standard for providing additional information to the compiler, beyond what is conveyed in the language itself; here, it lets Frama-C know that floating-point computations may be done with extended registers and/or FMA.

In our pragma, each floating-point number is represented by two values, an exact one (a real value, as if no rounding occurred) and a rounded one (the true floating-point value). At each computation, we are only able to bound the difference between these two values, without knowing the true rounded value.

**Theorem 2.** *Let $\odot$ be an operation among addition, subtraction, multiplication, division, square root, negation and absolute value. Let $x = \odot(y,z)$ be the exact result of this operation (without rounding). Then, whatever the architecture and the compiler, the computed result $\tilde{x}$ is such that*

*If $|x| \geq 2^{-1022}$, then $\tilde{x} \in \left[x - 2050 \times 2^{-64} \times |x|\ ,\ x + 2050 \times 2^{-64} \times |x|\right]\ \setminus\ \left]-2^{-1022}\ ,\ 2^{-1022}\right[$.*

*If $|x| \leq 2^{-1022}$, then $\tilde{x} \in \left[x - 2049 \times 2^{-1086}\ ,\ x + 2049 \times 2^{-1086}\right] \cap \left[-2^{-1022}\ ,\ 2^{-1022}\right]$.*

This is straightforward as the formulas of Theorem 1 subsume all possible roundings (64 or 80-bit) and operations (using FMA or not), whatever the architecture and the compiler choices.

**Theorem 3.** *If we define each result of an operation by the formulas of Theorem 2, and if we are able to deduce from these intervals an interval $\mathscr{I}$ for the final result, then the really computed final result is in $\mathscr{I}$ whatever the architecture and the compiler that preserves the order of operations.*

This is proved by using Theorem 2 and minding the FMA and the order of the operations.

The next question is the convenience of this approach. We have a collection of inequalities that might be useless. They are indeed useful and practical as we rely on the Gappa tool [19, 20] that is intended to help verifying and formally proving properties on numerical programs. Formulas of Theorem 1 have been chosen so that Gappa may take advantage of them and give an adequate final rounding error.

We may have chosen other formulas for Theorem 1 (such as $|\square(x) - x| \leq 2050 \times 2^{-64}|x| + 2049 \times 2^{-1086}$ or $|\square(x) - x| \leq \max\left(2050 \times 2^{-64}|x|, 2049 \times 2^{-1086}\right)$ but those are not as conveniently handled by Gappa as the chosen ones.

## 4 Case Studies

### 4.1 Double rounding example

The first example is very easy. It concerns the program of Figure 1. In this program, the result may either be 1 or $1 + 2^{-52}$, depending on the arguments of compiler we use. We add to the original program an

assertion written in ACSL [5]. By using Frama-C/Why and thanks to the Gappa prover, we automatically prove that in every architecture or compiler, the result of this program is always in $[1,\, 1+2^{-52}]$.

### 4.2 Avionics example

We now present a more complex case study containing floating-point computations to illustrate our approach. This example is part of KB3D [21][3], an aircraft conflict detection and resolution program. The aim is to make a decision corresponding to value $-1$ and 1 to decide if the plane will go to its left or its right. Note that KB3D is formally proved correct using PVS and assuming the calculations are exact [21]. However, in practice, when the value of the computation is small, the result may be inconsistent or incorrect. The original code is in Figure 2 and may give various answers depending on the architecture/compilation. To prove the correctness of this program which is independent to the architecture/compiler, we need to modify this program to know whether the answer is correct or not.

The modified program (See Figure 5) provides an answer that may be 1, $-1$ or 0. The idea is that, if the result is nonzero, then it is correct (meaning the same as if the computations were done on real numbers). If the result is 0, it means that the result may be under the influence of the rounding errors and the program is unable to give a certified answer.

In the original program, the inconsistency of the result is derived from the function `int sign(double x)`. To use this function only at the specification level, we define a logic function `logic integer l_sign (real x)` with the same meaning. Then we define another function `int sign (double x,`

```
#pragma JessieFloatModel(multirounding)
#pragma JessieIntegerModel(math)

//@ logic integer l_sign(real x) = (x >= 0.0) ? 1 : -1;

/*@ requires e1<= x-\exact(x) <= e2;
  @ ensures  \abs(\result) <= 1 &&
  @          (\result != 0 ==> \result == l_sign(\exact(x)));
  @*/
int sign(double x, double e1, double e2) {
  if (x > e2)  return  1;
  if (x < e1)  return -1;
  return 0;
}

/*@ requires
  @   sx == \exact(sx)  && sy == \exact(sy) &&
  @   vx == \exact(vx)  && vy == \exact(vy) &&
  @   \abs(sx) <= 100.0 && \abs(sy) <= 100.0 &&
  @   \abs(vx) <= 1.0   && \abs(vy) <= 1.0;
  @ ensures
  @    \result != 0
  @      ==> \result == l_sign(\exact(sx)*\exact(vx)+\exact(sy)*\exact(vy))
  @            * l_sign(\exact(sx)*\exact(vy)-\exact(sy)*\exact(vx));
  @*/
int eps_line(double sx, double sy,double vx, double vy){
  int s1,s2;
  s1=sign(sx*vx+sy*vy, -0x1.90641p-45, 0x1.90641p-45);
  s2=sign(sx*vy-sy*vx, -0x1.90641p-45, 0x1.90641p-45);
  return s1*s2;
}
```

Figure 5: Avionics program

[3] See also `http://research.nianet.org/fm-at-nia/KB3D/`.

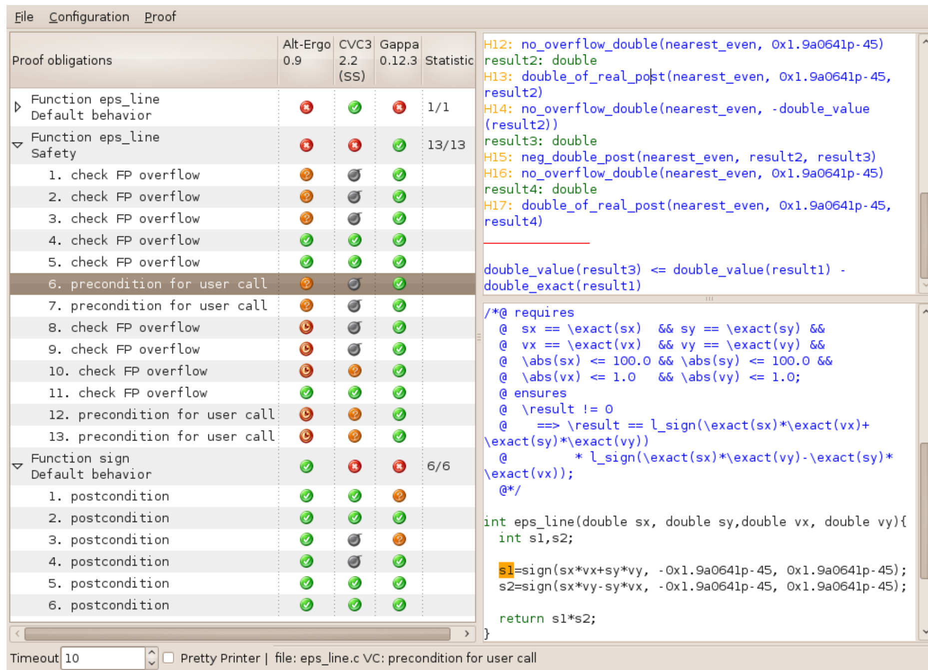



Figure 6: Result of Figure 5 program

`double e1, double e2)` that gives the sign of $x$ provided we know its rounding error is between $e1$ and $e2$. In the other cases, the result is zero.

The function `int eps_line (double sx, double sy, double vx, double vy)` of Figure 5 then does the same computations as the one of Figure 2, but the result may be different. More precisely, if the modified function gives a nonzero answer, it is the correct one (it gives the correct sign). But it may answer zero (contrary to the original program) when it is unable to give a certified answer. As in interval arithmetic, the program does not lie, but it may not answer.

About the other assertions, the given value of $sx$, $vx$... are reasonable for the position and the speed of the plane. The assertions about $s1$ and $s2$ are here to help the automatic provers.

The most interesting parts are the value chosen for $e1$ and $e2$: they need to bound the rounding error of the computation $sx*vx+sy*vy$ (and its counterpart). For this, we will rely on the Gappa tool. In particular, it will solve all the required proofs that no overflow occur.

In the usual formalization where all computations directly round to 64 bits, the values $e2 = -e1 = 0x1p-45$ are correct (it has been proved using the Gappa tool). With our approach and a generic rounding, we have proved that the values $e2 = -e1 = 0x1.90641p-45$ are correct. This means that the rounding error of $sx*vx+sy*vy$ will always be smaller than this value whatever the architecture and the compiler choices. This means that, even if a FMA is used or if extended registers are used somewhere, this function *does not lie*.

The analysis of this program (obtained from the verification condition viewer gWhy [16]) is given in Figure 6. By using different automatic theorem prover: Alt-Ergo [22], CVC3 [23], Gappa, we successfully prove all proof obligations in this program.

Nearly all the proof obligations are quick to prove. The proof that the values $e1$ and $e2$ bound the rounding error is slightly longer (about 10 seconds). This is due to the fact that, for each operation, we have to split into 2 cases: normal and subnormal and this increases the number of proof obligations to solve (exponential in the number of computations).

## 5 Conclusions and further work

We have proposed an approach to give correct rounding errors whatever the architecture and the choices of the compiler (preserving the order). This is implemented in the Frama-C framework from the Beryllium release for all basic operations: addition, subtraction, multiplication, division, square root, negation, absolute value and we have proved its correctness in Coq.

As we use the same conditions for all basic operations, it is both simple and efficient. Moreover, it handles both rounding according to 64-bit rounding in IEEE-754 double precision, 80-bit rounding in x87, double rounding in IA-32 architecture, and FMA in Itanium and PowerPC processors.

Nevertheless, the time to run a program needs to be taken into account. With a program containing few floating-point operations, it works well. However, it will be a little slower with programs containing a large number of floating-point operations because of the disjunction in Theorem 1 . The scalability could be achieved by several means: either modifying Gappa so that it may handle other kinds of formulas as good as those of Theorem 1, or replacing the formulas in Theorem 1 by other ones which do not contain disjunctions.

Another drawback is that we may only prove rounding errors. There is no way to prove, for example, that a computation is correct (even if it would be correct in all possible roundings). This means that some subtle floating-point properties may be lost but bounding the final rounding error is usually what is wanted by engineers and this does not appear to be a big flaw.

Note that we only consider double precision numbers as they are the most used. This is easily applied to single precision computations the same way (with single rounding, 80-bit rounding or double rounding). The idea would be to give similar formulas with the same case splitting and to provide the basic operations with those post-conditions.

We only handle rounding-to-nearest (ties to even and ties away from zero). The reason is that directed roundings do not suffer from these problems: double rounding gives the correct answer and if some intermediate computations are done in 80-bit precision, the final result is more accurate, but still correct as it is always rounded in the correct direction. Only rounding to nearest causes discrepancies.

This work is at the boundary between software and hardware for floating-point programs and this aspect of formal verification is very important. Moreover, this work deals both with normal and subnormal numbers, the latter ones being usually dismissed.

Another interesting point is that our error bounds may be used by other tools. As shown here, considering a slightly bigger error bound for each operation suffices to give a correct final error. This means that if Fluctuat for example would use them, it would also consider all possible cases of hardware and of compilation (preserving the order).

The next step would be to allow the compiler to do anything, including re-organizing the operations. This is a challenge as it may give very different results. For example, if $|e| \ll |x|$, then $(e+x)-x$ gives zero while $e+(x-x)$ gives $e$. Nevertheless, some ideas could probably be reused to give a loose bound on the rounding error.

### Acknowledgments

We are grateful to Claude Marché for his suggestions, his help in creating a new pragma in Frama-C/Why and his constructive remarks. We thank Guillaume Melquiond for his help in the use of Gappa tool. We are grateful to César Muñoz for providing the case study and explanations. We also thank Pascal Cuoq for his helpful suggestions about the FMA.

## References


[1] Microprocessor Standards Subcommittee: IEEE Standard for Floating-Point Arithmetic. IEEE Std. 754-2008 (August 2008) 1–58

[2] Monniaux, D.: The pitfalls of verifying floating-point computations. TOPLAS **30**(3) (May 2008) 12

[3] JML-Java Modeling Language `www.jmlspecs.org`.

[4] Burdy, L., Cheon, Y., Cok, D.R., Ernst, M.D., Kiniry, J.R., Leavens, G.T., Leino, K.R.M., Poll, E.: An overview of JML tools and applications. International Journal on Software Tools for Technology Transfer (STTT) **7**(3) (June 2005) 212–232

[5] Baudin, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language. (2008) `http://frama-c.cea.fr/acsl.html`.

[6] Barnett, M., Leino, K.R.M., Rustan, K., Leino, M., Schulte, W.: The Spec# Programming System: An Overview, Springer (2004) 49–69

[7] Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The ASTRÉE analyzer. In: ESOP. Number 3444 in Lecture Notes in Computer Science (2005) 21–30

[8] Monniaux, D.: Analyse statique : de la théorie à la pratique. Habilitation to direct research, Université Joseph Fourier, Grenoble, France (June 2009)

[9] Delmas, D., Goubault, E., Putot, S., Souyris, J., Tekkal, K., Védrine, F.: Towards an industrial use of fluctuat on safety-critical avionics software. In: FMICS. Volume 5825 of LNCS., Springer (2009) 53–69

[10] Carreño, V.A., Miner, P.S.: Specification of the IEEE-854 floating-point standard in HOL and PVS. In: HOL95: 8th International Workshop on Higher-Order Logic Theorem Proving and Its Applications, Aspen Grove, UT (September 1995)

[11] Russinoff, D.M.: A mechanically checked proof of IEEE compliance of the floating point multiplication, division and square root algorithms of the AMD-K7 processor. LMS Journal of Computation and Mathematics **1** (1998) 148–200

[12] Harrison, J.: Formal verification of floating point trigonometric functions. In: Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design, Austin, Texas (2000) 217–233

[13] Leavens, G.T.: Not a number of floating point problems. Journal of Object Technology **5**(2) (2006) 75–83

[14] Boldo, S., Filliâtre, J.C.: Formal Verification of Floating-Point Programs. In: 18th IEEE International Symposium on Computer Arithmetic, Montpellier, France (June 2007) 187–194

[15] Ayad, A., Marché, C.: Multi-prover verification of floating-point programs. In Giesl, J., Hähnle, R., eds.: Fifth International Joint Conference on Automated Reasoning, Edinburgh, Scotland, Springer (July 2010)

[16] Filliâtre, J.C., Marché, C.: The Why/Krakatoa/Caduceus platform for deductive program verification. In: Computer Aided Verification (CAV). Volume 4590 of LNCS., Springer (July 2007) 173–177

[17] Goldberg, D.: What every computer scientist should know about floating point arithmetic. ACM Computing Surveys **23**(1) (1991) 5–47

[18] Boldo, S., Filliâtre, J.C., Melquiond, G.: Combining Coq and Gappa for Certifying Floating-Point Programs. In: 16th Symposium on the Integration of Symbolic Computation and Mechanised Reasoning. Volume 5625 of Lecture Notes in Artificial Intelligence., Grand Bend, Canada, Springer (July 2009) 59–74

[19] Daumas, M., Melquiond, G., Muñoz, C.: Guaranteed proofs using interval arithmetic. In Montuschi, P., Schwarz, E., eds.: 17th IEEE Symposium on Computer Arithmetic, Cape Cod, MA, USA (2005) 188–195

[20] Daumas, M., Melquiond, G.: Certification of bounds on expressions involving rounded operators. Transactions on Mathematical Software **37**(1) (2009)

[21] Dowek, G., Muñoz, C., Carreño, V.: Provably safe coordinated strategy for distributed conflict resolution. In: Proceedings of the AIAA Guidance Navigation, and Control Conference and Exhibit 2005, AIAA-2005-6047, San Francisco, California (2005)

[22] Conchon, S., Contejean, E., Kanig, J.: CC(X): Efficiently Combining Equality and Solvable Theories without Canonizers. In: SMT 2007: 5th International Workshop on Satisfiability Modulo. (2007)

[23] Barrett, C., Tinelli, C.: CVC3. In: Proceedings of the $19^{th}$ International Conference on Computer Aided Verification (CAV '07). Volume 4590 of LNCS., Springer-Verlag (July 2007) 298–302 Berlin, Germany.

# Slice-based Formal Specification Measures – Mapping Coupling and Cohesion Measures to Formal Z

Andreas Bollin
Alpen-Adria Universität Klagenfurt
Klagenfurt, Austria
Andreas.Bollin@uni-klu.ac.at

**Abstract**

This paper demonstrates that existing slice-based measures can reasonably be mapped to the field of state-based specification languages. By making use of Z specifications this contribution renews the idea of slice-profiles and derives coupling and cohesion measures for them. The measures are then assessed by taking a critical look at their sensitiveness in respect to modifications on the specification source. The presented study shows that slice-based coupling and cohesion measures have the potential to be used as quality indicators for specifications as they reflect the changes in the structure of a specification as accustomed from their program-related pendants.

## 1 Introduction

In one of the rare articles concerning the relation between specifications and code, Samson, Nevill, and Dugard [13] demonstrate a strong quantitative correlation between size-based specification metrics and the related pendant of software code. Their assumption is that a meaningful set of (complexity and quality) measures could help in estimating product measures and development effort at a much earlier stage. Complexity can be described by size-based attributes, but is it reasonable to measure the quality of a specification? This contribution takes a closer look at this problem.

Quality considerations are sophisticated. Besides the question of what a "good specification" looks like, quality-based measures (as in use with program code) are not so easily transformed to specifications. One reason is that such measures are usually based on control/data dependency considerations – concepts that are either not at all or only implicitly available. However, various authors demonstrated in [11, 4, 10, 17] that a reconstruction of the necessary dependencies ameliorates the situation and enables continuative techniques like slicing and chunking. And with that, slice-based measures (which are often taken as the basis for quality considerations) can be mapped to formal specifications, too. What would be the benefits of such measures?

As presumed by Samson et. al., with experiences from a large collection of specifications and implementations at hand, product and development estimates could be calculated at much earlier stages. But there is another benefit. When the measures are sensitive and react instantly to changes in the specifications, considerations, e.g. concerning deterioration effects, could be made, too.

The main objective of this contribution is now to investigate whether slice-based quality measures can reasonably be transformed to formal specifications. It does not invent new measures, but it maps the ideas behind the definitions of coupling and cohesion measures to the world of formal specification languages. Additionally, it looks for possible limitations. Based on Z [14], the mapping is described in some details and the outcome is assessed in respect to its expressiveness and sensitiveness.

This paper is structured as follows: Section 2 introduces specification slices and takes them as the basis for the slice-based measures mentioned above. Section 3 discusses the effects on the measures by making use of sample specifications, and Section 4 concludes the work with a short outlook.

# 2 Background

The motivation behind analyzing slice-based coupling and cohesion measures goes back to a paper of Meyers and Binkley [8]. In their empirical study they take a closer look at these measures and demonstrate that the values of coupling and cohesion can also be used for assessing deterioration effects. As formal specifications evolve, too, it would be interesting to see whether changes in the specification code show a similar behavior of these measures. As a necessary first step, a reasonable transformation of the original definitions of the measures to the world of formal specifications has to be found. This section demonstrates how this can be done for Z [14].

## 2.1 Slice-based Coupling and Cohesion

Coupling is a measure for the strength of inter-component connections, and cohesion is a measure for the mutual affinity of sub-components of a component. Within the range of this contribution we are interested in *how* these measures are calculated and *what* they indicate. As adumbrated in the introduction, a practical way in calculating coupling and cohesion measures is to make use of slices.

Weiser [15, 16] introduced five slice-based measures for cohesion: *Tightness*, *Coverage*, *Overlap*, *Parallelism*, and *Clustering*. Ott and Thuss [12] partly formalized these measures, and this contribution makes use of their formalization. *Coupling* was originally defined as the number of local information flow entering (fan-in) and leaving (fan-out) a procedure [7]. Harman et. al demonstrate in [6] that it can also be calculated via slicing. Furthermore, they show that the use of slices not only enables the detection of coupling, it can also be used to determine the ”bandwidth” of the existing information flow. Their notion of information flow is also used in this contribution.

## 2.2 Specification Slices and Slice Profiles

For the calculation of coupling and cohesion measures, sets of slices and their intersections (comparable to the use of slice profiles in [12]) are needed. For state-based specifications the technique of slicing was introduced by Oda and Araki [11], informally redefined by Chang and Richardson [4], and then refined by Bollin [1]. His Java prototype has been extended in the recent years. It now supports slicing, chunking, and concept location of Z specifications [3]. The technical details of the identification of dependencies are not relevant within the scope of this paper, but the basic idea is quite simple:

First, the specification is dismantled into its basic elements called primes[1] by making use of the *CZT* parser [9]. The primes are mapped to a graph called *SRN* (for *Specification Relationship Net*). Then, by following the approach of Chang and Richardson and Bollin [4, 1] control and data dependencies are reconstructed (via a syntactical approximation to the semantical analysis). The *SRN* gets annotated by this dependency information, yielding an *Augmented SRN* (*ASRN* for short).

The *ASRN* serves the same purpose as the system dependence graph used by the approaches described in [8, p.4]. Based on this data structure, slicing works as follows: a set of vertices (representing the point of interest) in the *ASRN* is taken as starting point, and, by following the dependencies existing in the graph, further primes are aggregated, resulting in the designated specification slice. The transformation between a specification $\Psi$ and its *ASRN* is defined in a bijective manner. So, when talking about a specification it can be either the textual representation (consisting of a set of primes) or its *ASRN* representation (consisting of vertices representing the primes).

[1] Basically, primes are the predicates of the specification and are later represented as vertices in an augmented graph. When they represent predicates of the precondition of a schema they are called precondition primes, and when they form predicates that represent after-states they are called postcondition primes.

Harman et. al [6] and Ott and Thuss [12] use different types of slices for their calculation of coupling and cohesion values. This situation is dealt with hereinafter by generating two variants of the static specification slices: for *coupling* the slices are calculated by following the dependencies in a transitive backward manner, for the values of *cohesion* the slices are calculated by combining the dependencies in a forward and backward manner. Specification slices and slice profiles (the collection of slices for a specific schema operation) are then defined as follows:

**Definition 1. Static Specification Slice**. *Let $\Psi$ be a formal Z specification, $\psi$ one schema out of $\Psi$, and V a set of primes v out of $\psi$. $SSlice_{fb}(\psi, V)$ is called static forward/backward specification slice of $\psi$ for primes V. It is calculated by generating a transitive forward and backward slice with V as the starting point of interest. When the slice is generated in a transitive and backward manner, it is called static backward slice $SSlice_b(\psi, V)$.*

**Definition 2. Slice Profile, Slice Intersection, Slice Union**. *Let $\Psi$ be a formal Z specification, $\psi$ one schema out of $\Psi$, and V the set of primes v representing all postcondition primes in $\psi$. The set of all possible static specification slices ($SSlice_{fb}(\psi, \{v\})$ or $SSlice_b(\psi, \{v\})$, with $v \in V$) is called Slice Profile ($SP(\psi)$). The intersection of the slices in $SP(\psi)$ is called Slice Intersection ($SP_{int}(\psi)$). The union of all slices in $SP(\psi)$ is called Slice Union ($SU(\psi)$).*

## 2.3 Cohesion

With the introduction of slice profiles it is possible to provide the definitions of cohesion measures (as introduced in the work of Ott and Thuss [12]). The values for cohesion are calculated only for a given schema. As slices and slice profiles might contain primes from other schemata (due to inter-schema dependencies), the following definitions restrict the set of primes in the slice profile to the schema.

**Definition 3. Tightness**. *Let $\Psi$ be a formal Z specification, $\psi$ one schema out of $\Psi$, $SP(\psi)$ its slice profile, and $SP_{int}(\psi)$ its slice intersection. Then Tightness $\tau(\psi)$ is the ratio of the size of the slice intersection to the size of $\psi$. It is defined as follows:*

$$\tau(\psi) = \frac{\mid SP_{int}(\psi) \cap \psi \mid}{\mid \psi \mid}$$

**Definition 4. MinCoverage, Coverage, MaxCoverage**. *Let $\Psi$ be a formal Z specification, $\psi$ one schema out of $\Psi$, and $SP(\psi)$ its slice profile containing n slices. MinCoverage $Cov_{min}(\psi)$ expresses the ratio between the smallest slice $SP_{i-min}$ in $SP(\psi)$ and the number of predicate vertices in $\psi$. Coverage $Cov(\psi)$ relates the sizes of all possible specification slices $SP_i$ ($SP_i \in SP(\psi)$) to the size of $\psi$. MaxCoverage $Cov_{max}(\psi)$ expresses the ratio of the largest slice $SP_{i-max}$ in the slice profile $SP(\psi)$ and the number of vertices in $\psi$. They are defined as follows:*

$$Cov_{min}(\psi) = \frac{1}{\mid \psi \mid} \mid SP_{i-min} \cap \psi \mid \qquad Cov(\psi) = \frac{1}{n} \sum_{i=1}^{n} \frac{\mid SP_i \cap \psi \mid}{\mid \psi \mid} \qquad Cov_{max}(\psi) = \frac{1}{\mid \psi \mid} \mid SP_{i-max} \cap \psi \mid$$

**Definition 5. Overlap**. *Let $\Psi$ be a formal Z specification, $\psi$ one schema out of $\Psi$, $SP(\psi)$ its slice profile containing n slices, and $SP_{int}$ its slice intersection. Then Overlap $O(\psi)$ measures how many primes are common to all possible specification slices $SP_i$ ($SP_i \in SP(\psi)$). It is defined as follows:*

$$O(\psi) = \frac{1}{n} \sum_{i=1}^{n} \frac{\mid SP_{int} \cap \psi \mid}{\mid SP_i \cap \psi \mid}$$

*Tightness* measures the number of primes that are common to every slice. The definition is based on the size[2] of the slice intersection. Coverage is split into three different measures: *Minimum Coverage* looks at the size of the smallest slice and relates it to the size of the specification, *Coverage* looks at the size of the slices, but it takes all slices and compares them to the size of the specification, and *Maximum Coverage* looks at the size of the largest slice and relates it to the size of the specification. Finally, *Overlap* looks at the slice intersection and determines how many primes are common to all slices.

[2] Please note that within the context of all definitions *size* counts the number of primes in the *ASRN*.

### 2.4 Coupling

The calculation of coupling follows the definitions to be found in [6]. First, Inter-Schema Flow $\digamma$ is specified. It describes how many primes of the slices in the slice union are outside of the schema. Inter-Schema Coupling then computes the normalized ratio of this flow in both directions.

**Definition 6. Inter-schema Flow and Coupling**. *Let $\Psi$ be a formal Z specification and $\psi_s$ and $\psi_d$ two schemata out of $\Psi$. Inter-Schema Flow between the two schemata $\digamma(\psi_s, \psi_d)$ is the ratio of the primes of $SU(\psi_d)$ that are in $\psi_s$ and that of the size of $\psi_s$. Inter-Schema Coupling between the two schemata $C(\psi_s, \psi_d)$ measures the Inter-Schema Flow in both directions. They are defined as follows:*

$$\digamma(\psi_s, \psi_d) = \frac{\mid SU(\psi_d) \cap \psi_s \mid}{\mid \psi_s \mid} \qquad C(\psi_s, \psi_d) = \frac{\digamma(\psi_s, \psi_d) \times \mid \psi_s \mid \ + \digamma(\psi_d, \psi_s) \times \mid \psi_d \mid}{\mid \psi_s \mid + \mid \psi_d \mid}$$

Schema coupling is calculated by considering the Inter-Schema Coupling values to all other schemata.

**Definition 7. Schema Coupling**. *Let $\Psi$ be a formal Z specification and $\psi_i$ one schema in $\Psi$. Then Schema Coupling $\chi(\psi_i)$ is the weighted measures of the Inter-Schema Coupling of $\psi_i$ to all other n schemata $\psi_j$ in $\Psi \setminus \psi_i$. It is defined as follows:*

$$\chi(\psi_i) = \frac{\sum_{j=1}^{n} C(\psi_i, \psi_j) \times \mid \psi_j \mid}{\sum_{j=1}^{n} \mid \psi_j \mid}$$

With the measures in this section it is possible to assign attributes to a formal Z specification. However, with the mapping a connection to quality has been so far not empirically justified. On the other hand, the slice-based measures have carefully been transformed to Z. There is a chance that, when observing *changes* of these values for a given specification, one might defer useful properties.

## 3 Sensitivity of Slice-based Measures

By following the strategy that Thuss and Ott [12] used for their validations, we now investigate the *sensitivity* of the measures with respect to *representative changes* of the specifications. The advantage is that for such a study only small-sized sample specifications are necessary to explore the effects.

### 3.1 Sensitivity of Cohesion

The first objective is to determine whether the transformed measures for cohesion are sensitive to changes in the internal structure of the specification. The following operations are considered:

O1 Adding a precondition-prime. This means that this prime ”controls” the evaluation of the other primes in the schema. With it, the internal semantic connections are extended. Mapped to a potential implementation, this could mean that an if-clause is added to the code, enveloping all other statements in the method. This operation should slightly increase coverage.

O2 Adding a prime that specifies an after-state and that is not related to all the other predicates in the schema. In this case the predicate introduces new ”trains of thought”. Mapped to a subsequent implementation, this could mean that a new output- or state-relevant statement (not or only fractionally related to the other statements) is added. With it, a new slice is added to the slice-profile. The slice intersection is very likely smaller than before, thus reducing the values for cohesion.

O3 Adding a prime that specifies an after-state and that is furthermore related to all other primes in the schema. In this case the predicate extends existing ”trains of thought” (as there are references to all existing ones). Mapped to a possible implementation, it is very likely that a new output- or state-relevant statement, related to all other statements, is added. If at all, this increases the set of intersection slices. And with that, it also raises the values of coupling.

| $SP(\psi)$ | | | Specifications | Measures |
|---|---|---|---|---|
| <br><br><br> | <br><br><br> | <br><br><br>— | *Test1*<br>$n, n' : \mathbb{N}$<br>$m, m' : \mathbb{N}$<br>$n' = n + 1$ | $\#SP_{int}(\psi) = 1$<br>$\tau(\psi) = 1.00$<br>$Cov_{min}(\psi) = 1.00$<br>$Cov(\psi) = 1.00$<br>$Cov_{max}(\psi) = 1.00$<br>$O(\psi) = 1.00$ |
| <br><br><br> | <br><br><br>—<br> | <br><br><br><br>— | *Test2*<br>$n, n' : \mathbb{N}$<br>$m, m' : \mathbb{N}$<br>$n' = n + 1$<br>$m' = m + 1$ | $\#SP_{int}(\psi) = 0$<br>$\tau(\psi) = 0.00$<br>$Cov_{min}(\psi) = 0.50$<br>$Cov(\psi) = 0.50$<br>$Cov_{max}(\psi) = 0.50$<br>$O(\psi) = 0.00$ |
| <br><br><br> | <br><br><br><br>—<br>—<br> | <br><br><br><br>—<br><br>— | *Test3*<br>$n, n' : \mathbb{N}$<br>$m, m' : \mathbb{N}$<br>$delta? : \mathbb{N}$<br>$delta? > 0$<br>$n' = n + delta?$<br>$m' = m - delta?$ | $\#SP_{int}(\psi) = 1$<br>$\tau(\psi) = 0.33$<br>$Cov_{min}(\psi) = 0.67$<br>$Cov(\psi) = 0.67$<br>$Cov_{max}(\psi) = 0.67$<br>$O(\psi) = 0.50$ |
| <br><br><br> | <br><br><br><br><br>—<br>—<br>—<br> | <br><br><br><br><br>—<br>—<br><br>— | *Test4*<br>$n, n' : \mathbb{N}$<br>$m, m' : \mathbb{N}$<br>$delta? : \mathbb{N}$<br>$set? : \mathbb{P}\mathbb{N}$<br>$delta? > 0$<br>$set? \neq \varnothing$<br>$n' = n + delta?$<br>$m' = m - delta?$ | $\#SP_{int}(\psi) = 2$<br>$\tau(\psi) = 0.50$<br>$Cov_{min}(\psi) = 0.75$<br>$Cov(\psi) = 0.75$<br>$Cov_{max}(\psi) = 0.75$<br>$O(\psi) = 0.67$ |
| <br><br><br><br><br><br>—<br>—<br><br><br>— | <br><br><br><br><br><br>—<br>—<br><br>—<br> | <br><br><br><br><br><br>—<br>—<br>—<br><br> | *Test5*<br>$n, n' : \mathbb{N}$<br>$m, m' : \mathbb{N}$<br>$delta? : \mathbb{N}$<br>$set? : \mathbb{P}\mathbb{N}$<br>$p, p' : \mathbb{N}$<br>$delta? > 0$<br>$set? \neq \varnothing$<br>$n' = n + delta?$<br>$m' = m - delta?$<br>$p' = p + delta?$ | $\#SP_{int}(\psi) = 2$<br>$\tau(\psi) = 0.40$<br>$Cov_{min}(\psi) = 0.60$<br>$Cov(\psi) = 0.60$<br>$Cov_{max}(\psi) = 0.60$<br>$O(\psi) = 0.33$ |
| <br><br><br><br><br><br>—<br>—<br>—<br><br>— | <br><br><br><br><br><br>—<br>—<br><br>—<br> | <br><br><br><br><br><br>—<br>—<br>—<br><br>— | *Test6*<br>$n, n' : \mathbb{N}$<br>$m, m' : \mathbb{N}$<br>$delta? : \mathbb{N}$<br>$set? : \mathbb{P}\mathbb{N}$<br>$p, p' : \mathbb{N}$<br>$delta? > 0$<br>$set? \neq \varnothing$<br>$n' = n + delta?$<br>$m' = m - delta?$<br>$p' = p + n$ | $\#SP_{int}(\psi) = 2$<br>$\tau(\psi) = 0.40$<br>$Cov_{min}(\psi) = 0.60$<br>$Cov(\psi) = 0.73$<br>$Cov_{max}(\psi) = 0.80$<br>$O(\psi) = 0.56$ |
| <br><br><br><br> | <br><br><br><br><br>—<br>—<br>—<br>— | <br><br><br><br><br>—<br>—<br>—<br>— | *Test7*<br>$n, n' : \mathbb{N}$<br>$m, m' : \mathbb{N}$<br>$delta? : \mathbb{N}$<br>$set? : \mathbb{P}\mathbb{N}$<br>$delta? > 0$<br>$set? \neq \varnothing$<br>$n' = n + delta?$<br>$m' = m - n$ | $\#SP_{int}(\psi) = 4$<br>$\tau(\psi) = 1.00$<br>$Cov_{min}(\psi) = 1.00$<br>$Cov(\psi) = 1.00$<br>$Cov_{max}(\psi) = 1.00$<br>$O(\psi) = 1.00$ |

Figure 1: Z specifications of raising sizes. On the left side of the table the slices (and thus the slice-profile) are visualized, on the right side the values for cohesion are presented.

Based on the assumption that schema operations are often mapped to methods (or procedures) as described in operations $O1$ to $O3$, the following working hypothesis can be posted:

**Hypothesis 1.** *A structural change of type $O1$, $O2$ or $O3$ in a schema operation influences the values for cohesion. Adding a predicate prime to the schema according to operations $O1$ or $O3$ increases the values (or leaves them unchanged), adding a prime according to operation $O2$ decreases the values (or leaves them unchanged). Reversing the operations also reverses the effect on the measures.*

There are situations where, due to a large number of dependencies, a method or a schema operation already has reached the maximum values for cohesion. These special cases are the reason why the values might also be unchanged (and Sec. 3.3 reconsiders this issue in more details).

Hypothesis 1 is now checked by using small sample schema operations (called *Test*1 to *Test*7 in Fig. 1). At first let us start with a simple Z schema operation called *Test*1. It contains a prime that increases the value of $n$ by one. As there is only one slice, the slice intersection only contains one element. The values of cohesion are all 1. Then another prime ($m' = m + 1$, prescribing an after-state) is

added to the schema (which is an operation of class $O2$), yielding operation $Test2$. With this new prime a new "functionality" has been introduced to the schema. The values for cohesion are reduced as the slice intersection is empty. Tightness and Overlap are zero, the rest of the values are equal to $\frac{1}{2}$. Then, in $Test3$, the prime $delta? > 0$ is added to the schema. This prime is a precondition prime and thus the operation belongs to class $O1$. With this, all the values of cohesion increase. As the slice intersection contains one prime only ($delta? > 0$), its size is $\#SP_{int}(\psi) = 1$. With that, the values for cohesion result in: $\tau = \frac{1}{3}$, $Cov_{min} = \frac{1}{3} \times 2$, $Cov = \frac{1}{2} \times (\frac{2}{3} + \frac{2}{3})$, $Cov_{max} = \frac{1}{3} \times 2$, and $O = \frac{1}{2} \times (\frac{1}{2} + \frac{1}{2})$.

$Test4$ adds another precondition prime $set? \neq \varnothing$ to the schema (operation $O1$). This yields an increase in the values of cohesion. The reason is the increase in size of the slice intersection. $Test5$ adds another prime containing an after state identifier to the schema operation. The prime $p' = p + delta?$ is not related to the other primes prescribing after-states, so this change is an operation of class $O3$. The size of the slice intersection stays the same, only the size of the schema increases. As a result the values for cohesion decrease.

Now let us take a look at situations when predicates that are partly related to existing predicates are added to the schema. $Test6$ is an extension of $Test4$, but this time the new prime $p' = p + n$ uses the identifier $n$ which is also defined by the postcondition prime $n' = n + delta?$. On the other hand it does not refer to the third postcondition prime $m' = m - delta?$, and so the operation belongs to class $O2$. The values for cohesion consequently decrease. On contrary, $Test7$ is a modification of $Test3$. The prime $m' = m - n$ is added, and so it is related to all other postcondition primes in the schema. With this operation the values for cohesion increase, again.

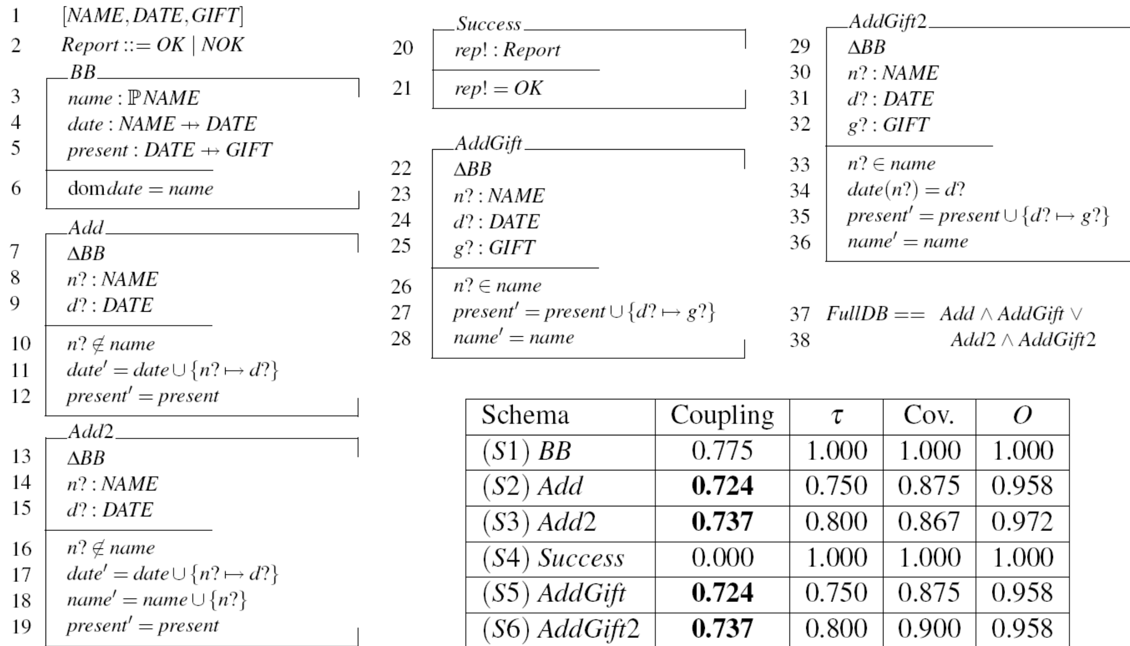
Due to reasons of space the example schemata above only contain simple predicates. But they are sufficient to demonstrate the influence of structural changes. In Z there are several schema operators that complicate the situations, but by further analyzing the formulas of the measures one observes the following behavior:

- Cohesion will increase when (a) at least one postcondition exists and a precondition prime is added, (b) a postcondition prime that is related to some, but not to all, other postcondition primes in the schema is added, (c) a postcondition prime that is not related to the other postcondition primes is removed.
- Cohesion stays the same when (a) a postcondition prime that is related to all other existing postcondition primes is added or removed and the other existing primes are already fully inter-related, (b) a pre- or postcondition prime is added and there is no postcondition prime.
- Cohesion will decrease when (a) a postcondition prime that is not related to the other postcondition primes is added, (b) a precondition prime is removed and there is at least one postcondition prime, (c) a postcondition prime that is related to the other postcondition primes is removed.

The values for a derived implementation would very likely react to these changes in the same way, so the sample specifications and the analysis of the formulas seems to confirm our working hypothesis. The measures are sensitive to changes in the underlying specification, and the observed changes are in such a way that an increase in internal connectivity also leads to an increase of the values of the measures. Conversely, a decrease in connectivity also leads to a decrease of the values of cohesion.

### 3.2 Sensitivity of Coupling

The next step is the evaluation of coupling. As mentioned above, specification coupling measures the mutual interdependence between different schemata. According to Harman et. al [6] it is an advantage to use slice-based measures as they measure the "bandwidth" of the information flow. In our case, this




| Schema | Coupling | $\tau$ | Cov. | $O$ |
|---|---|---|---|---|
| $(S1)$ *BB* | 0.775 | 1.000 | 1.000 | 1.000 |
| $(S2)$ *Add* | **0.724** | 0.750 | 0.875 | 0.958 |
| $(S3)$ *Add2* | **0.737** | 0.800 | 0.867 | 0.972 |
| $(S4)$ *Success* | 0.000 | 1.000 | 1.000 | 1.000 |
| $(S5)$ *AddGift* | **0.724** | 0.750 | 0.875 | 0.958 |
| $(S6)$ *AddGift2* | **0.737** | 0.800 | 0.900 | 0.958 |

Figure 2: Z example for analyzing the effect of slice-based coupling and values for coupling and cohesion for the six Z schemata (omitting the *FullDB* operation schema).

flow is defined as the relative amount of the size of a set of slices of one schema that lies inside another schema. This flow depends upon control- and data-relationships, so an increase in the number of these dependencies should increase the value of coupling. Reducing the number of relations should decrease the value of coupling. There are no differences between the definitions of the measure, be it for traditional programs or be it for formal specifications.

The next hypothesis focuses on the sensitiveness to structural changes within a formal Z specification. Especially, an increase (or decrease) in inter-schema relations should be reflected correctly[3].

**Hypothesis 2.** *A change in the number of relations between two schemata in a formal specification generally leads to a change in the value of coupling. Adding a predicate prime to one schema that introduces new dependencies to primes in the other schema increases the value of coupling (or leaves it unchanged). Reversing the change also reverses the effect on the measure.*

For our considerations we now make use of a small specification (see Fig. 2) which is an extension (and intentionally unfinished version) of the birthday-book specification out of Spivey [14, pp.3–6].

The specification consists of one state space called *BB* (for *Birthday Book*) which stores the names of friends, their birthday dates, *and* a small gift. Consequently, there are two operations (*Add* and *AddGift*) for adding them to the database. The operations are not total, but are sufficient for our examinations. In order to analyze the effect of added pre- and postcondition primes, these operations are available in two versions. Finally, there is an operation called *Success* which should (later on) indicate the success of an operation. However, at the moment this operation is not related to any of the other operations.

The values of schema coupling are summarized in Fig. 3. As expected, the *Success* operation has a value of coupling equal to zero. There are no connections to the state space *BB* and also no connections

[3] Again, there are situations where, due to a high number of dependencies, the value of coupling might stay unchanged.

| $\digamma$ | $(S1)$ | $(S2)$ | $(S3)$ | $(S4)$ | $(S5)$ | $(S6)$ |
|---|---|---|---|---|---|---|
| $(S1)$ | 1.000 | **1.000** | 1.000 | 0.000 | 1.000 | 1.000 |
| $(S2)$ | **0.750** | 1.000 | 0.750 | 0.000 | 0.750 | 0.750 |
| $(S3)$ | 0.800 | 0.800 | 1.000 | 0.000 | 0.800 | 0.800 |
| $(S4)$ | 0.000 | 0.000 | 0.000 | 1.000 | 0.000 | 0.000 |
| $(S5)$ | 0.750 | 0.750 | 0.750 | 0.000 | 1.000 | 0.750 |
| $(S6)$ | 0.800 | 0.800 | 0.800 | 0.000 | 0.800 | 1.000 |

| $C$ | $(S1)$ | $(S2)$ | $(S3)$ | $(S4)$ | $(S5)$ | $(S6)$ |
|---|---|---|---|---|---|---|
| $(S1)$ | 1.000 | **0.800** | **0.833** | 0.000 | 0.800 | 0.833 |
| $(S2)$ | 0.800 | 1.000 | 0.778 | 0.000 | **0.750** | **0.778** |
| $(S3)$ | 0.833 | 0.778 | 1.000 | 0.000 | 0.778 | 0.800 |
| $(S4)$ | 0.000 | 0.000 | 0.000 | 1.000 | 0.000 | 0.000 |
| $(S5)$ | **0.800** | 0.750 | 0.778 | 0.000 | 1.000 | 0.778 |
| $(S6)$ | **0.833** | 0.778 | 0.818 | 0.000 | **0.778** | 1.000 |

Figure 3: Values for Inter-Schema Flow $\digamma(\psi_1, \psi_2)$ and Inter-Schema Coupling $C(\psi_1, \psi_2)$. The abbreviations $S1$ to $S6$ refer to the schema names mentioned in Fig. 2.

to the other operation schemata. On the other hand, the values for the other operations differ (though their syntactical composition is more or less equivalent). With $n = 6$ operations there are 15 different combinations and thus possibly 15 values for Inter-Schema Coupling. Within the scope of this contribution we will focus on three combinations that are of most interest in this schema constellation.

As a first example we look at the operations $Add$ and $Add2$. The difference between them is made up by just one prime, namely $name' = name \cup \{n?\}$ at line 18. In fact, in the context of the specification this postcondition prime is redundant (as the state invariant at line 6 would guarantee that the added name is also in the set of names). But syntactically this prime is sufficient to increase the set of dependencies. Both operations include the state-space, which means that there is a relation between the postcondition primes and the invariant. This introduces a new "flow of control", which increases the bandwidth and thus the value for coupling (from 0.724 to 0.737 in Fig. 2).

Fig. 3 presents the values for Inter-Schema Flow and Coupling, and they make this difference more visible. Inter-Schema Flow $\digamma(BB, Add)$ is $\frac{|SU(Add) \cap BB|}{|BB|}$, which is 1 (so the slices of $Add$(S2) cover 100% of the state space $BB$(S1)). $\digamma(Add, BB)$ is $\frac{|SU(BB) \cap Add|}{|Add|}$, where $SU(BB) \cap Add$ covers the primes at lines $\{6, 10, 11\}$ (due to data dependency between line 6 and line 11). With this, the Inter-Schema Flow is $\frac{3}{4}$. (Please note that due to the schema inclusion the $Add$ schema consists of 4 predicates!) Now, looking at Inter-Schema Coupling, the value is $\frac{1 \times 1 + 3/4 \times 4}{1+4}$, which is 0.8. Similarly, one can calculate the value for the coupling between $BB(S1)$ and $Add2(S3)$, which is 0.833. The new dependency between the invariant at line 6 and line 18 led to the situation that the slice contains one more prime. For similar situations we might infer that the introduction of postcondition primes will (very likely) raise the value of coupling.

Another situation occurs when looking at the operation schemata $AddGift$ and $AddGift2$. In relation to the state schema the second variant of the operation contains an additional prime at line 35. However, the point of departure is not the same. In this case the prime is a precondition prime that does not influence any primes outside the schema – at first sight. On closer examination it is clear that the postcondition primes in this schema are control dependent upon this prime, and a slice "reaching" the schema operation will have to include this prime, too. It grows in size, meaning that more "bandwidth" is spent on it. In similar situation we can infer that the value of coupling will also increase as the value for the Inter-Schema Flow increases from $\frac{n}{m}$ to $\frac{n+1}{m+1}$.

The above specification does not show that the value for coupling can also decrease. This is the case when we introduce a postcondition prime that is not related to the primes in the other schema(ta). Then, in case of a state schema, there is no data-dependency between the primes. And in the case of another operation schema there is neither control nor data-dependency. As the size of the schema increases, Inter-Schema Flow decreases from $\frac{n}{m}$ to $\frac{n}{m+1}$.

On the syntactical level there is no difference between $Add$ and $AddGift$. Both consist of a precondition prime, three postcondition primes, and include the state. This equivalence can be seen in Fig. 2

as the values for cohesion are the same. But it gets interesting when comparing them to *AddGift*2. The value for Inter-Schema Coupling between *Add* (*S*2) and *AddGift* (*S*5) is 0.750, whereas the value for *Add* (*S*2) and *AddGift*2 (*S*6) is 0.778. So, there is a slightly higher value of coupling between *Add* and *AddGift*2. The reason for this is a semantic difference: the data relationship between the lines 11 and 34. This simple example demonstrates that the idea of the ”bandwidth” is quite applicable in this situation.

Though the above example is simple, it is able to demonstrate the effects on the value for coupling in the case of structural changes of schema operations. The second working hypothesis seems to be confirmed, at least from the analytical part of view.

### 3.3 Limitations

Though the above hypotheses seem to be confirmed, there are limitations. More precisely, the problems are that (a) the specifications might be too dense, (b) only part of the ”bandwidth” is regarded, and (c) the dependency reconstruction does not work correctly. What seems to corrupt the measures is in fact not a real problem.

In [2] the effect and efficiency of slicing has been investigated, and it turned out that slicing has disadvantages when the specifications are too dense. In about 60-70% of all cases slicing led to a reduction in the size of the specification, which also means that in some situations there has been no reduction at all. The slices then contained all other primes – indicating that nearly every prime is related to all other primes. However, this effect decreases on average with raising sizes of the specifications (our experience relies on more than 45.000 lines of specification text), and it is only a problem with text-book examples that consist of a view schema operations only.

The next concern is that coupling is not sensitive to changes that lead to an increase in the number of relations between the same primes. Irrespective the number of dependencies between two primes, only the occurrence of the primes is counted by the size-operator. Bandwidth does not increase then. The presented notion of coupling works well on a syntactical level, but not necessarily as expected on a semantic level. The last measure (comparing *AddGift* and *AddGift*2) was sensitive because one prime has (intentionally) been omitted from both schemata: normally, an author of these operations would have added the line $date' = date$ as predicates to the schemata. This would have introduced data-dependencies from both schemata to the *Add* schema, and there would have been no difference in Inter-Schema Coupling anymore. In fact, this can not be seem as a real problem, it is as coupling is defined. Nevertheless, one has to keep in mind that there might be more dependencies as expected.

Finally, slicing only works fine when the specifications are ”well-formed”. This means that they consist of separable pre- and postconditions primes. When such a separation is not possible, then the outcome is vitiated. Diller mentions in [5, p.165] that in most cases this separation can be done (which means that a syntactical approximation to the semantic analysis is possible), but this does not prevent from cases where pre- and postconditions are interwoven and not separable.

## 4 Conclusion and Outlook

This contribution introduces the notion of specification slice-profiles that are then used for the calculation of slice-based specification measures. The way of calculating these measures for Z (namely coupling and cohesion) is new and it is based on the use of (reconstructed) control and data dependency information. The objective of this work is to investigate if such a mapping is meaningful. For this, the contribution takes a set of small specifications as a basis, and the sensitivity of the measures is then analyzed by changing internal and external properties of the specifications.

The evaluation shows that the measures reflect the changes in the structure of the specification as expected. Especially the values for cohesion seem to be a good indicator for changes in internal properties. Coupling is, due to the use of unions of slices a bit less sensitive, but it also reacts when there are dramatic structural changes. All in all, the measures proof useful.

The understanding of the *behavior* of the measures was a first, necessary step. The next steps will be to include different ”real-life” specifications and to perform an empirical study that demonstrates that the measures are not just proxies for other, eventually size- based, measures. In case of confirming their unique features again, this could be a step towards taking specifications as quality indicators quite at the beginning of the SW-development cycle.

## References

[1] Andreas Bollin. *Specification Comprehension – Reducing the Complexity of Specifications*. PhD thesis, Universität Klagenfurt, April 2004.

[2] Andreas Bollin. The Efficiency of Specification Fragments. In *Proceedings of the 11th IEEE Working Conference on Reverse Engineering*, 2004.

[3] Andreas Bollin. Concept Location in Formal Specifications. *Journal of Software Maintenance and Evolution: Research and Practice*, 20(2):77–104, March/April 2008.

[4] Juei Chang and Debra J. Richardson. Static and Dynamic Specification Slicing. Technical report, Department of Information and Computer Science, University of California, 1994.

[5] Antoni Diller. *Z – An Introduction to Formal Methods*. John Wiley and Sons, 2nd edition, 1999.

[6] Mark Harman, Margaret Okulawon, Bala Sivagurunathan, and Sebastian Danicic. Slice-based measurement of coupling. In *Proceedings of the IEEE/ACM ICSE workshop on Process Modelling and Empirical Studies of Software Evolution. Boston, Massachusetts*, pages 28–32, 1997.

[7] S. Henry and D. Kafura. Software structure metrics based on information flow. *IEEE Transactions on Software Engineering*, 7(5):510–518, 1981.

[8] Timothy M. Meyers and David Binkley. An empirical study of slice-based cohesion and coupling metrics. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 17(1), December 2009.

[9] Tim Miller, Leo Freitas, Petra Malik, and Mark Utting. CZT Support for Z Extensions. In *Proc. 5th International Conference on Integrated Formal Methods (IFM 2005)*, pages 227 – 245. Springer, 2005.

[10] Roland T. Mittermeir and Andreas Bollin. Demand-driven Specification Partitioning. *Lecture Notes in Computer Science*, 2789:241–253, 2003.

[11] Tomohiro Oda and Keijiri Araki. Specification slicing in a formal methods software development. In $17^{th}$ *Annual International Computer Software and Applications Conference*, IEEE Computer Socienty Press, pages 313–319, November 1993.

[12] Linda M. Ott and Jeffrey J. Thuss. Slice based metrics for estimating cohesion. In *In Proceedings of the IEEE-CS International Metrics Symposium*, pages 71–81. IEEE Computer Society Press, 1993.

[13] W.B. Samson, D.G. Nevill, and P.I. Dugard. Predictive software metrics based on a formal specification. In *Information and Software Technology*, volume 29 of *5*, pages 242–248, June 1987.

[14] J.M Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International, $2^{nd}$ edition, 1992.

[15] Mark Weiser. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, University of Michigan, 1979.

[16] Mark Weiser. Program slicing. In *Proceedings of the* $5^{th}$ *International Conference on Software Engineering*, pages 439–449. IEEE, 1982.

[17] Fangjun Wu and Tong Yi. Slicing Z Specifications. *SIGPLAN Not.*, 39(8):39–48, 2004.

# How Formal Methods Impels Discovery: A Short History of an Air Traffic Management Project

Ricky W. Butler, George Hagen, Jeffrey M. Maddalon, César A. Muñoz, Anthony Narkawicz
NASA Langley Research Center
Hampton, VA 23681, USA
{r.w.butler,george.hagen,j.m.maddalon,c.a.munoz,anthony.narkawicz}@nasa.gov
Gilles Dowek
École polytechnique and INRIA
LIX, École polytechnique
91128 Palaiseau Cedex, France
gilles.dowek@polytechnique.fr

**Abstract**

In this paper we describe a process of algorithmic discovery that was driven by our goal of achieving complete, mechanically verified algorithms that compute conflict prevention bands for use in en route air traffic management. The algorithms were originally defined in the PVS specification language and subsequently have been implemented in Java and C++. We do not present the proofs in this paper: instead, we describe the process of discovery and the key ideas that enabled the final formal proof of correctness.

## 1 Introduction

The formal methods team at NASA Langley has developed air traffic separation algorithms for the last 10 years. Given the safety-critical nature of these algorithms, we have emphasized formal verification of the correct operation of these algorithms. In February of 2008, Ricky Butler and Jeffrey Maddalon started a project to develop and formally verify algorithms that compute conflict prevention bands for en route aircraft.

In air traffic management systems, a conflict prevention system senses nearby aircraft and provides ranges of maneuvers that avoid conflicts with these aircraft The maneuvers are typically constrained to ones where only one parameter is varied at a time: track angles, vertical speeds, or ground speeds. Such maneuvers are easy for pilots to fly and have the advantage that they can be presented in terms of prevention bands, i.e. ranges of parameter values that should be avoided. Prevention bands display the maneuvers that result in conflict within a specified lookahead time as a function of one parameter. Without conflict prevention information, a pilot might create another conflict while seeking to solve a primary conflict or otherwise changing the flight plan.

The National Aerospace Laboratory (NLR) in the Netherlands refers to their conflict prevention capability as Predictive Airborne Separation Assurance System or Predictive ASAS [3]. The NLR approach provides two sets of bands: near-term conflicts are shown in red, while intermediate-term conflicts are shown in amber as illustrated in Figure 1. We did not directly analyze the NLR system because the algorithms were not available to us; however, we did use some of their interface ideas.

When we began this project, we had no idea that this project would take almost two years to complete and that four additional formal methods researchers would join our effort before we were done. This project has been one of the most interesting and enjoyable projects that we have worked on in our careers. The reason for this is manifold: (1) the work resulted in a very elegant algorithm that is implemented in Java and C++, (2) the final algorithm was very different from our first ideas, (3) there were many, many discoveries that were surprising. (At some points in the project we were having daily insights that improved the algorithm or a proof), (4) on the surface the problem looks simple, but looks can

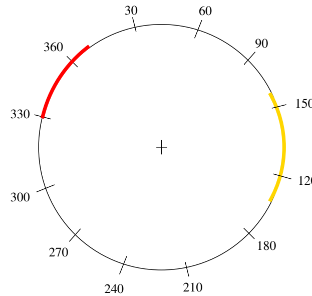



Figure 1: Compass Rose with Conflict Prevention Bands

be deceiving and the problem is actually very subtle with many special cases. After studying the problem for over a year, we developed an algorithm and rigorously documented this algorithm in a NASA Technical Memorandum [8]. We also formalized much of the mathematical development in the paper. We planned to follow up this paper with another paper that documented a complete formal proof of this algorithm. Much to our surprise this final formal proof step found some deficiencies in our algorithm. These deficiencies were repaired and a formal proof in the Prototype Verification System (PVS) [9] was finally completed in November 2009.

In this paper we will present a brief history of this project and highlight how the goal of formally verifying the algorithm in the PVS theorem prover pushed us to new discoveries. In addition to finding some subtle problems in the initial approach, we are confident that many of the discoveries would not have been made if we had taken a more traditional approach of constructing algorithms, testing, and revising them until they *worked*.

## 2 Notation

We consider two aircraft, the *ownship* and the *traffic* aircraft, that are potentially in conflict in a 3-dimensional airspace. The conflict prevention algorithm discussed here only uses state-based information, e.g,. initial position and velocity and straight line trajectories, i.e., constant velocity vectors in an Euclidean airspace. These approximations of real aircraft behavior are valid for short lookahead times (typically less than 5 minutes).

We use the following notations:

| | | |
|---|---|---|
| $\mathbf{s}_o$ | 3D vector | Initial position of the ownship aircraft |
| $\mathbf{v}_o$ | 3D vector | Initial velocity of the ownship aircraft |
| $\mathbf{s}_i$ | 3D vector | Initial position of the traffic aircraft |
| $\mathbf{v}_i$ | 3D vector | Initial velocity of the traffic aircraft |

The components of each vector are scalar values, so they are represented without the bold-face font, for example $\mathbf{s}_o = (s_{ox}, s_{oy}, s_{oz})$. As a simplifying assumption, we regard the position and velocity vectors as accurate and without error. For notational convenience, we use $\mathbf{v}^2 = \mathbf{v} \cdot \mathbf{v}$ and we denote by $\mathrm{gs}(\mathbf{v})$ the ground speed of $\mathbf{v}$, i.e., the norm of the 2-dimensional projection of $\mathbf{v}$: $\mathrm{gs}(\mathbf{v}) = \sqrt{v_x^2 + v_y^2}$.

In the airspace system, the separation criteria are specified as a minimum horizontal separation $D$ and a minimum vertical separation $H$ (typically, $D$ is 5 nautical miles and $H$ is 1000 feet). It is convenient to develop the theory using a translated coordinate system. The relative position $\mathbf{s}$ is defined to be $\mathbf{s} = \mathbf{s}_o - \mathbf{s}_i$ and relative velocity of the ownship with respect to the traffic aircraft is denoted by $\mathbf{v} = \mathbf{v}_o - \mathbf{v}_i$. With

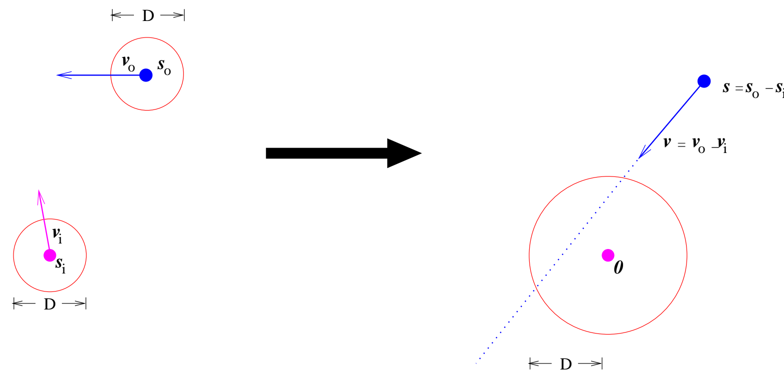



Figure 2: Translated Coordinate System

these vectors the traffic aircraft is at the center of the coordinate system and does not move. For example in the left side of Figure 2, the blue (upper) point represents the ownship with its velocity vector and its avoidance area (circle of diameter $D$ around the aircraft). The magenta (lower) point represents the traffic aircraft. The right side represents the same information in the translated coordinate system. In a 3-dimensional airspace, the separation criteria defines a cylinder of radius $D$ and half-height $H$ around the traffic aircraft. This cylinder is called the *protected zone*.

In Figure 2, the two aircraft are potentially in conflict because the half-line defined by the relative velocity vector $\mathbf{v}$ intersects the protected area, meaning that in some future time the ownship will enter the protected zone around the traffic. If this future time is within the lookahead time $T$, the two aircraft are said to be in *conflict*.

## 3 The Start of the Project

We began our project by first surveying the literature for previous solutions. Hoekstra [4] describes some algorithms developed at NLR with some diagrams [3], but unfortunately he did not provide much detail about how the algorithms actually worked. We decided to develop our own track angle, ground speed, and vertical speed algorithms. In this paper, we will only present the track band algorithm. These are the most challenging and interesting of the three and the other bands are computed and verified using analogous methods. We adopted the NLR idea of introducing two parameters, $T_{red}$ (typically three minutes) and $T_{amber}$ (typically five minutes), which divide the set of conflicts based on their nearness (in time) to a loss of separation (see figure 1). If a loss of separation will occur within $T_{red}$, then the region is colored red. On the other hand, if a loss of separation will occur after $T_{red}$, but before $T_{amber}$, then the region is colored amber, otherwise it is painted green.

We first recognized that each aircraft's contribution to the prevention band is independent of all other aircraft; thus, the problem neatly decomposes into two steps:

1. Solve the bands problem for the ownship relative to each other aircraft separately.

2. Merge all of the pairwise regions.

We also quickly realized that an iterative solution was possible for the first step. We already had a formally proven, efficient algorithm available to us named `CD3D` that decides if a conflict occurs for specific values of $\mathbf{s}_o$, $\mathbf{v}_o$, $\mathbf{s}_i$, and $\mathbf{v}_i$, and parameters $D$, $H$, and $T$. More formally, `CD3D` determines whether there exists a future time $t$ where the aircraft positions $\mathbf{s}_o + t\mathbf{v}_o$ and $\mathbf{s}_i + t\mathbf{v}_i$ are within a horizontal distance $D$ of each other *and* where the aircraft are within vertical distance $H$ of each other. In other words there

is a predicted loss of separation within the lookahead time. Therefore, one need only to execute `CD3D` iteratively, varying the track angle from 0 to 360° per traffic aircraft. By evaluating different scenarios, we determined that a step size of 0.1° would be adequate for ranges of up to 200 nautical miles. An iterative approach would consume more computational resources than an analytical one where the edges of the bands are computed directly. An iterative approach may not scale well where such separation assurance algorithms must be executed for many different traffic aircraft every second. Despite these disadvantages, the existence of an iterative approach provided a fall-back position: if we were not able to discover a formally verifiable analytical solution, then we knew we could use the interative approach.

# 4 Search for an Analytical Solution

To solve the prevention bands problem in an analytical way, it is useful to define separate horizontal and vertical notions of conflict. In the relative coordinate system, we define that a *horizontal conflict* occurs if there exists a future time $t$ within the lookahead time $T$ where the aircraft are within horizontal distance $D$ of each other, i.e., $(s_x+tv_x)^2+(s_y+tv_y)^2<D^2$. where $\mathbf{s}$ and $\mathbf{v}$ are, respectively, the relative position and the relative velocity of the ownship with respect to the traffic aircraft. A *vertical conflict* occurs if there exists a future time $t$ within the lookahead time $T$ where the aircraft are within horizontal distance $H$ of each other, i.e., $|s_z+tv_z|<H$. We say that two aircraft are in *conflict* if there is a time $t$ where they are in horizontal and vertical conflict. Formally, we define the predicate `conflict?` as follows

$$\texttt{conflict?}(\mathbf{s},\mathbf{v}) \quad \equiv \quad \exists\, 0 \le t \le T : (s_x+tv_x)^2+(s_y+tv_y)^2 < D^2 \text{ and } |s_z+tv_z|<H. \tag{1}$$

## 4.1 Track Only Geometric Solution

We begin with a non-translated perspective shown in Figure 3. The track angle is denoted by $\alpha$[1]. For

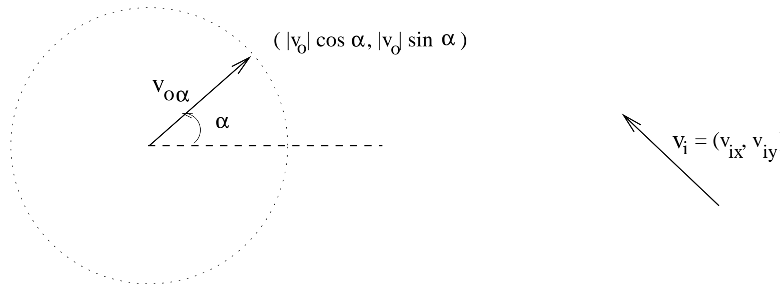



Figure 3: Two Dimensional Version of Track Bands Problem

given vectors $\mathbf{v}_o$ and $\mathbf{v}_i$, we need to find the track angles $\alpha$ such that the relative vector $\mathbf{v}_\alpha = \mathbf{v}_{o\alpha} - \mathbf{v}_i$:

$$\mathbf{v}_\alpha = (|\mathbf{v}_o|\cos\alpha - v_{ix},\ |\mathbf{v}_o|\sin\alpha - v_{iy}), \tag{2}$$

is not in conflict.

Our initial approach was to divide the conflict prevention problem into simplifying cases. We discovered later that this division was unnecessary and more elegant formulations provided the necessary leverage to formally verify the algorithm. In any case, we decided to first solve the track bands problem in two dimensions without consideration of the lookahead time. We also decided to ignore vertical speed

[1] In air traffic management, the track angle is calculated from true north in a clockwise direction, but here we have followed the traditional mathematical definition.

considerations and look at the horizontal plane only. The problem thus reduced to finding the tangent lines to the horizontal protection zone (in the relative frame of reference) as a function of $\alpha$. We begin with the observation that in order for a vector to be tangent it must intersect the circle of the protection zone. In other words, we need solutions of $|\mathbf{s}+t\mathbf{v}_\alpha| = D$ or equivalently

$$(\mathbf{s}+t\mathbf{v}_\alpha)^2 = D^2 \tag{3}$$

where the vectors are two-dimensional. Expanding we obtain a quadratic equation $at^2+bt+c=0$ with $a=\mathbf{v}_\alpha{}^2$, $b=2(\mathbf{s}\cdot\mathbf{v}_\alpha)$, and $c=\mathbf{s}^2-D^2$. The tangent lines are precisely those where the discriminant of this equation is zero. In other words, where $b^2-4ac=0$. But, expanding the dot products yield:

$$b^2 = 4[s_x(\omega\cos\alpha - v_{ix}) + s_y(\ \omega\sin\alpha - v_{iy})]^2$$
$$4ac = 4(\omega^2 - 2\omega(v_{ix}\cos\alpha + v_{iy}\sin\alpha) + v^2)(\mathbf{s}\cdot\mathbf{s} - D^2)$$

The discriminant finally expands into a complex second-order polynomial in $\sin\alpha$ and $\cos\alpha$. But to solve for $\alpha$, we need to eliminate the $\cos\alpha$ using the equation

$$\cos\alpha = \sqrt{1-\sin^2\alpha}$$

The net result is an unbelievably complex fourth order polynomial in $\sin\alpha$. Solving for $\alpha$ analytically would require the use of the quartic formulas. Although these formulas are complicated, such a program could probably be written in a day or two. But, how would we verify these solutions? After all, the quartic equations involve the use of complex analysis. Therefore, we began to look for simplifications.

We found a simplification of the discriminant that had been used in the verification of the KB3D algorithm [6]:

$$b^2 - 4ac = 0 \text{ if and only if } (\mathbf{s}\cdot\mathbf{v}) = R\,\varepsilon\,\mathtt{det}(\mathbf{s},\mathbf{v}), \tag{4}$$

where $\varepsilon \in \{-1,+1\}$, $\mathtt{det}(\mathbf{s},\mathbf{v}) \equiv \mathbf{s}^\perp\cdot\mathbf{v}$, $\mathbf{s}^\perp = (-s_y, s_x)$, and $R = \frac{\sqrt{\mathbf{s}^2-D^2}}{D}$. The beauty of the final form is that the equation is linear on $\mathbf{v}$. The two solutions are captured in the two values of $\varepsilon$. When we instantiate $\mathbf{v}_\alpha$ in this formula, we end up with a quadratic equation in $\sin\alpha$.

Using this approach, we were able to derive the following solutions for $\alpha$. If $\frac{|G|}{\sqrt{E^2+F^2}} \leq 1$ then in some $2\pi$ range, we have

$$\alpha_1 = \mathtt{asin}\left(\frac{G}{\sqrt{E^2+F^2}}\right) - \mathtt{atan}(E,F),$$
$$\alpha_2 = \pi - \mathtt{asin}\left(\frac{G}{\sqrt{E^2+F^2}}\right) - \mathtt{atan}(E,F),$$

where

$$E = \omega(R\varepsilon s_x - s_y),\ \ F = -\omega(R\varepsilon s_y + s_x),\ \ G = \mathbf{v}_i\cdot(R\varepsilon\,\mathbf{s}^\perp - \mathbf{s}),$$

Since $E$, $F$, and $G$ are all functions of $\varepsilon$, we have two pairs of $\alpha_1$ and $\alpha_2$ or a total of four total angles. These angles are the places where the track prevention band changes color, assuming no lookahead time. This result was formalized in the PVS theorem prover and implemented in Java.

### 4.2 Solution with Lookahead Time

The solution presented so far only considers the 2-dimensional case with no lookahead time. Figure 4 illustrates three distinct cases that appear when the lookahead time is considered: (a) the protection zone is totally within lookahead time, (b) the zone is partially within, and (c) the zone is totally beyond the

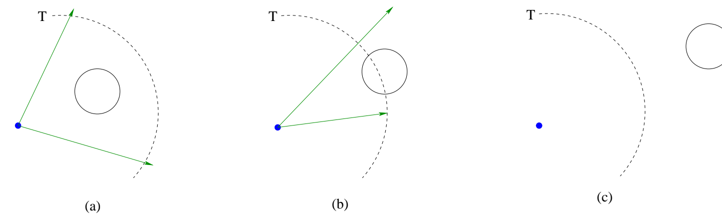



Figure 4: Relationship of Encounter Geometry and Lookahead Time

lookahead time. Cases (a) and (c) were easy to handle, but we realized that case (b) was going to take some additional analysis. We were quite pleased with our initial geometric result and decided to present the result to our branch head and research director. During the presentation, a member of the original KB3D team announced, "I think you can solve this problem without trigonometry," and he urged us to defer the use of trigonometry until the last possible moment. In other words, he suggested that we solve for $(\mathbf{v}_\alpha)$ without expanding its components. Only after the appropriate abstract solution vector is found, should the conversion to a track angle, $\alpha$, be made. This was a key idea that had been used in the development of the KB3D algorithms, which resulted in very efficient and elegant algebraic solutions [1]. Indeed, we realized that the geometric problem was solvable by a particular kind of KB3D resolutions called *track lines*, computed by the function `track_line`:

`track_line(`$\mathbf{s},\mathbf{v}_o,\mathbf{v}_i,\varepsilon,\iota$`): Vect2`

The function `track_line` returns the vector $\mathbf{0}$ when all track angles for the ownship yield a potential conflict. Otherwise, the vector returned by this function is a velocity vector for the ownship that is tangent to the 2-dimensional protected zone. Since $\varepsilon$ and $\iota$ are $\pm 1$, there are four possible track line solutions for given $\mathbf{s}$, $\mathbf{v}_o$, and $\mathbf{v}_i$ .

The key to solving track bands with a lookahead time is to find where the *projected* lookahead time intersects the protected zone. That is, plot where the relative position of the aircraft will be after $T$ units of time in every possible direction given an unchanged ground speed and find the intersection points with the protection zone. The function `track_circle`, also available in KB3D, provides these solutions, which are called *track circle* solutions.

The function `track_circle`

`track_circle(`$\mathbf{s},\mathbf{v}_o,\mathbf{v}_i,T,\iota$`): Vect2`

returns the vector $\mathbf{0}$ when there are no track circle solutions, i.e., when the lookahead time boundary $T$ and the protected zone do not intersect, or when there are an infinite number of solutions. Otherwise, the vector returned by this function is a velocity vector for the ownship that intersects the 2-dimensional protected zone at a time later than $T$. Since $\iota$ is $\pm 1$, for given $\mathbf{s}$, $\mathbf{v}_o$, and $\mathbf{v}_i$ there are two possible track circle solutions. The `track_line` and `track_circle` functions are derived and discussed in [8].

We believe that the lookahead problem would not have been analytically tractable using the trigonometric approach pursued at first. This switch to a pure algebraic approach was fundamental to achieving the final proof of the 3-dimensional bands algorithm.

## 4.3 The Track Bands Algorithm

The two functions, `track_line` and `track_circle`, form the basis of the track bands algorithm. We define a *critical vector* as a relative velocity vector where the color of the bands *may* change. These

critical vectors are

$$
\begin{aligned}
\mathbf{R}_{mm} &= \texttt{track_line}(\mathbf{s},\mathbf{v}_o,\mathbf{v}_i,-1,-1),\\
\mathbf{R}_{mp} &= \texttt{track_line}(\mathbf{s},\mathbf{v}_o,\mathbf{v}_i,-1,+1),\\
\mathbf{R}_{pm} &= \texttt{track_line}(\mathbf{s},\mathbf{v}_o,\mathbf{v}_i,+1,-1),\\
\mathbf{R}_{pp} &= \texttt{track_line}(\mathbf{s},\mathbf{v}_o,\mathbf{v}_i,+1,+1),\\
\mathbf{C}_{rm} &= \texttt{track_circle}(\mathbf{s},\mathbf{v}_o,\mathbf{v}_i,T_{red},-1),\\
\mathbf{C}_{rp} &= \texttt{track_circle}(\mathbf{s},\mathbf{v}_o,\mathbf{v}_i,T_{red},+1),\\
\mathbf{C}_{am} &= \texttt{track_circle}(\mathbf{s},\mathbf{v}_o,\mathbf{v}_i,T_{amber},-1),\\
\mathbf{C}_{ap} &= \texttt{track_circle}(\mathbf{s},\mathbf{v}_o,\mathbf{v}_i,T_{amber},+1),\\
\mathbf{C}_{em} &= \texttt{track_circle}(\mathbf{s},\mathbf{v}_o,\mathbf{v}_i,t_{entry},-1),\\
\mathbf{C}_{ep} &= \texttt{track_circle}(\mathbf{s},\mathbf{v}_o,\mathbf{v}_i,t_{entry},+1),\\
\mathbf{C}_{xm} &= \texttt{track_circle}(\mathbf{s},\mathbf{v}_o,\mathbf{v}_i,t_{exit},-1),\\
\mathbf{C}_{xp} &= \texttt{track_circle}(\mathbf{s},\mathbf{v}_o,\mathbf{v}_i,t_{exit},+1).
\end{aligned}
$$

Some of these vectors may be zero vectors in which case they are ignored. The times $t_{entry}$ and $t_{exit}$ are the calculated entry and exit times into the protection zone.

In the track bands algorithm, these vectors are calculated, the corresponding track angles (using `atan`) are computed and sorted into a list of angles. Next, the angles 0 and $2\pi$ are added to the list to provide appropriate bounding. Then, a conflict probe (such as CD3D) us applied to an angle between each of the critical angles to characterize the whole region (i.e., determine which color the region should be painted: green, amber, or red). This procedure is iterated between the ownship and all traffic aircraft. Finally, the resulting bands are merged to get the display as in Figure 1.

# 5 Formal Verification of Pairwise Prevention Bands Algorithms

The functions `track_line` and `track_circle` discussed in Section 4.2 have been verified *correct* for conflict resolution, i.e., they compute vectors that yield conflict free trajectories for the ownship, and *complete* for track prevention bands, i.e., they compute all critical vectors where the track bands change colors. These functions are slightly different from the original ones presented in [8]. Indeed, the functions presented in that report, while correct for conflict resolution, failed to compute all the critical vectors. For conflict prevention bands this is a safety issue, because a region that should be colored red can be colored green instead. Interestingly, those functions, which had been tested on over 10,000 test cases without any error manifestations, were in fact incorrect. The deficiencies in these functions were only found during the formal verification process!

The general idea of the correctness proof of the prevention bands algorithms is as follows:

1. For a given parameter of the ownship, e.g., track angle, define a function $\Omega_{\mathrm{trk}} : \mathbb{R} \to \mathbb{R}$, parametrized by $\mathbf{s}$, $\mathbf{v}_o$, and $\mathbf{v}_o$, that characterizes conflicts in the following way: $\Omega_{\mathrm{trk}}(\alpha) < 0$ if and only if $\texttt{conflict?}(\mathbf{s},\mathbf{v}_\alpha)$, where $\mathbf{v}_\alpha$ is defined as in Formula 2.
2. Prove that the critical vectors computed in Section 4.3 are *complete*, i.e., they compute all of the zeros of the function $\Omega_{\mathrm{trk}}$,
3. Prove that the function $\Omega_{\mathrm{trk}}$ is continuous.
4. Use the Intermediate Value theorem to deduce that any point in an open bands region, e.g., the mid-point, determines the color of the whole band. This last step requires the existence of a conflict probe algorithm that is proven correct, which we have already developed and verified.

This approach is illustrated in figure 5.

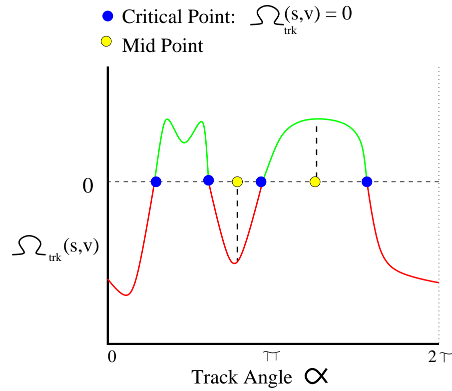



Figure 5: $\Omega_{\text{trk}}$ Proof Approach

The discovery of this $\Omega_{\text{trk}}$ proof approach also directly influenced the final bands algorithms. Originally we expected to compute the color of a region. This proof method lead us to the idea of using the `CD3D` conflict probe on the midpoint of the region in order to color the region.

We first tried this proof approach on a simplified version of the problem: the two-dimensional ground speed bands with infinite lookahead time. In this case, the function $\Omega_{\text{gs}}$ must characterize $\texttt{conflict?}(\mathbf{s}, \mathbf{v}_k)$, where $\mathbf{v}_k = \frac{k}{\text{gs}(\mathbf{v})}\mathbf{v}$.

The following formula provided the needed relationship between horizontal conflict that does not include a quantification over time:

$$\texttt{horizontal_conflict?}(\mathbf{s}, \mathbf{v}) \iff \mathbf{s} \cdot \mathbf{v} < R\,\texttt{det}(\mathbf{v}, \mathbf{s}) < -\mathbf{s} \cdot \mathbf{v}, \tag{5}$$

where, $R$ is defined as in Formula 4.

The function $\Omega_{\text{gs}}$ was constructed based on this theorem. The resulting function required the use of if-then-else logic so the proof that it was continuous was tedious. After much effort, we were able to prove in PVS that the ground-speed functions `gs_line` and `gs_circle`, which are analogous to `track_line` and `track_circle`, were complete assuming no look-ahead time.

The lesson learned from this first attempt was that we needed a more abstract way of defining the functions $\Omega_{\text{gs}}$ and $\Omega_{\text{trk}}$ so that the complexity of the continuity proofs could be untangled from the subtleties of the track and ground speed resolutions. With this in mind, we defined a function $\Omega : \mathbb{R}^n \to \mathbb{R}$, parametrized by $\mathbf{s}$, $\mathbf{v}_o$, and $\mathbf{v}_o$, such that $\Omega_{\text{trk}} = \Omega \circ \mathbf{v}_\alpha$ and $\Omega_{\text{gs}} = \Omega \circ \mathbf{v}_k$. The continuity of $\Omega_{\text{trk}}$ and $\Omega_{\text{gs}}$ is a consequence of the continuity of $\Omega$, which was proved once and for all for all kinds of bands, and the continuity of $\mathbf{v}_\alpha$ and $\mathbf{v}_k$. All this seems straightforward except that there are several technical difficulties.

The function $\Omega$ is closely related to the function that computes the minimum distance between two aircraft. That function is, in general, noncontinuous for an infinite lookahead time. Interestingly, it is continuous when a lookahead time is considered, but the general proof of this fact requires the use of vector variant of the Heine-Cantor Theorem, i.e., if $M$ is a compact metric space, then every continuous function $f : M \to N$, where $N$ is a metric space, is uniformly continuous. Furthermore, the minimum distance function may have flat areas. Therefore, special attention has to be paid to the definition of $\Omega$ to guarantee that the set of critical points is finite. Otherwise, it cannot be proven that the critical vector functions are complete.

The next sections discuss the formal verification of the prevention bands algorithms with lookahead time for both the 2-D and 3-D cases.

### 5.1 Verification of 2D Prevention Bands

In the 2-dimensional case, a direct definition of $\Omega$ is possible by using $\tau$, the time of minimal horizontal separation between two aircraft:

$$\tau(\mathbf{s},\mathbf{v}) \quad = \quad -\frac{\mathbf{s}\cdot\mathbf{v}}{\mathbf{v}^2}. \tag{6}$$

From $\tau$, we can define $\Omega$ as follows:

$$\Omega(\mathbf{v}) \quad = \quad (\mathbf{s}+\min(\max(0,\tau(\mathbf{s},\mathbf{v})),T)\mathbf{v})^2 - D^2, \tag{7}$$

where $\mathbf{s}$ is the relative distance between the ownship and the traffic aircraft.

The use of square distances in Formula 7 avoids the use of the square root function. Since the minimum and maximum of a continuous function is continuous, the use of min and max is easier to handle than the if-then-logic used in our first attempt.

The function $\Omega$ is not defined when $\mathbf{v}$ is $\mathbf{0}$. Therefore, rather than using $\Omega$ directly, we used the function $\mathbf{v} \mapsto \mathbf{v}^2\Omega(\mathbf{v})$, which is defined everywhere, and proved that it is continuous and that it correctly characterizes conflicts, i.e., `conflict?`$(\mathbf{s},\mathbf{v})$ if and only if $\mathbf{v}^2\Omega(\mathbf{v}) < 0$.

The function $\mathbf{v}^2\Omega(\mathbf{v})$ has an infinite number of zeroes in some special cases, e.g., when $\mathbf{s}$ is at the border of the protected zone, i.e, when $\mathbf{s}^2 = D^2$. In those, special cases, we use an alternative characterization of conflicts that has the required properties. In August 2009, we completed the proof of the 2-dimensional track and ground speed bands with finite lookahead time. For additional technical details on this formal development, we refer the reader to [7].

### 5.2 Verification of 3D Prevention Bands

The verification of the 3D conflict prevention bands algorithm is similar to that of the 2D algorithm. Indeed, many of the geometrical concepts critical to the verification in the 2D case can be generalized to the 3D case. However, these generalizations are typically nontrivial because, geometrically, a circle (a 2D protected zone) is much easier to work with than a cylinder (a 3D protected zone). The $\Omega$ function used in the verification of the 2D algorithm uses the horizontal time of minimum separation $\tau$, which is easy to compute analytically. In contrast, the fact that a cylinder is not a smooth surface indicates that a 3D generalization of the $\Omega$ function will not be as simply defined.

Despite these geometric challenges, a concept was discovered that can be used to simplify geometry problems involving distance on cylinders. This concept is the notion of a *normalized cylindrical length* [2]:

$$||\mathbf{u}||_{\text{cyl}} \quad = \quad \max(\frac{\sqrt{u_x^2+u_y^2}}{D},\frac{|u_z|}{H}). \tag{8}$$

This metric nicely reduces horizontal and vertical loss of separation into a single value. Indeed, if $\mathbf{s}$ is the relative position vector of two aircraft, then $||\mathbf{s}||_{\text{cyl}} < 1$ if and only if the aircraft are in 3D loss of separation.

Using the cylindrical distance metric, the $\Omega$ function can be defined in the 3D case as follows.

$$\Omega_{3D}(\mathbf{v}) \quad = \quad \min_{t\in[0,T]} ||\mathbf{s}+t\cdot\mathbf{v}||_{cyl} - 1, \tag{9}$$

where $S$ is the relative position vector between the ownship and traffic aircraft. An immediate consequence of this definition is that two aircraft are in conflict if and only if $\Omega_{3D}(\mathbf{v}) < 0$.

The correctness of the prevention bands algorithms relies on the fact that $\Omega_{3D}$ is a continuous function of $\mathbf{v}$, that the set of critical vectors, i.e., the zeroes of the function, is finite, and that the critical vector algorithms are complete.

For many functions, a proof of continuity follows immediately from definitions. In this case, the function $\Omega_{3D}$ is a minimum over the closed interval $[0,T]$. While standard methods from differentiable calculus are often employed in similar problems, this function is a minimum of a non-differentiable function, namely the cylindrical length. Its closed form involves several if-else statements and it would be difficult to use directly in a proof of continuity. Thus, somewhat more abstract results from real analysis were needed to be extended to vector analysis, e.g., the notion of limits, continuity, compactness, and finally the Heine-Cantor Theorem.

As in the 2-dimensional case, the function $\Omega_{3D}$ may have flat areas and, consequently, in some special cases, may have an infinite number of critical zeros. We carefully identified these special cases and then used an alternative definition of $\Omega_{3D}$. These special cases are extremely rare, indeed all the missing critical vectors in the original algorithms presented in [8] were due to these special cases. Although they are rare, dealing with them is necessary for the correctness proof of the algorithms. If one critical vector is missing, the coloring of the bands will be potentially switch from red to green or vice-versa.

Finally, the PVS proof that the algorithms find all of the critical points is less abstract but more tedious than the proof of continuity. It required the development of several PVS theories about the $\Omega_{3D}$ function, which are general enough to be used in other state-based separation assurance algorithms. The proof of the correctness of the 3-dimensional algorithms for track, ground speed, and vertical speed with finite lookahead time was complete in December 2009.

# 6 Verification of the Merge Algorithm

Having developed methods to calculate pairwise solutions (see section 4.3), we turned to the problem of merging all of the pairwise solutions into a single set of bands for all aircraft (second problem listed in section 3). Our original approach relied on complex reasoning about overlapping regions coupled with precedence rules: an amber region take precedence over a green region but not a red region. We developed a Java version that "worked," but it was soon obvious that this solution was complex enough that it could not be implicitly trusted. We decided that a formal verification of the merge algorithm was necessary.

As we began considering how to formally specify and analyze this merge algorithm, we recognized two problems with our approach. First, our algorithm was specialized to the precise problem we were working; almost any change to the system would require a new algorithm and therefore a new verification. The other problem was that our algorithm was monolithic; there was no obvious decomposition into general pieces that could be verified once and used in different contexts. To resolve these problems, we soon realized that standard set operations (set union, set difference, etc.) could be used to implement not only the multiple-aircraft merge problem, but also the different colors of conflict prevention information.

Suppose we had a way of determining the set of track angles that have a loss of separation within time $T$, denoted $\mathscr{G}_{<T}$. Then since $T_{red} < T_{amber}$, we can define the colored bands of track angles in terms of this new set:

$$
\begin{aligned}
\mathscr{G}_{red} &= \mathscr{G}_{<T_{red}} \\
\mathscr{G}_{amber} &= \mathscr{G}_{<T_{amber}} - \mathscr{G}_{<T_{red}} \\
\mathscr{G}_{green} &= \{\alpha \,|\, 0^\circ \le \alpha < 360^\circ\} - \mathscr{G}_{<T_{amber}}
\end{aligned}
$$

This observation simplified the analysis, because only one set, $\mathscr{G}_{<T}$ needed to be analyzed.

Next, we observed that each aircraft's contribution to the set $\mathscr{G}_{<T}$ is independent of all other traffic; thus, the problem neatly divides into a series of aircraft pairs: the ownship and each traffic aircraft. If we use $\mathscr{G}^{o,i}_{<T}$ to represent the set of track angles which cause a loss of separation within time $T$ between traffic aircraft $i$ and the ownship $o$, then the set of track angles for all traffic is then be formed by

$$\mathscr{G}_{<T} = \bigcup_{i \in traffic} \mathscr{G}^{o,i}_{<T}$$

This observation simplified the analysis again, because now we only needed to find the track angles which cause a conflict between two aircraft, denoted by the set $\mathscr{G}^{o,i}_{<T}$. The set $\mathscr{G}^{o,i}_{<T}$ corresponds to output of the algorithm presented in section 4.3 except the two lookahead times are replaced with one time, $T$.

By using set operations we had a well-defined specification of the key parts of our merge algorithm. However common implementations of sets in programming languages do not include efficient ways to deal with ranges of floating point numbers; therefore, we chose to implement our own. We then performed a code-level verification of the set union and set difference operations that were used in the merge algorithm.

Each band is represented by an interval describing its minimal and maximal values, with the set of all bands of one color being an interval set. These interval sets were internally represented by arrays of (ordered) intervals. Necessary properties for the implementation would be that the data structures representing the bands remained ordered and preserved the proper value ranges within a set of bands.

Set union combines overlapping bands as appropriate and set difference involves breaking larger bands into smaller ones. There were two complications in the verification. The first complication arose because the implementation used a subtle notion of ordering where a zero or a positive value represents an actual position in the array of intervals, but a negative value represents a point between (or beyond) the intervals currently in the set. Although tedious, this verification was completed without issues. The second complication resulted from the boundary conditions. The original Java implementation did not clearly indicate whether the endpoints of a band of green angles were part of that the green band, or if they were part of the next band. The formal verification brought this issue forward. It was not possible to exclusively use closed or open intervals for both union and difference operations: the use of one necessitates the use of the other. For instance, removing a closed interval, which includes the endpoints, leaves us with open intervals—everything up to, but not including, these end points. Also, removing two adjacent open intervals leads to a left over point between them.

At this point we had an implementation issue. Should we use fully general set operations with open and closed intervals, or should we use set operations with well-defined, but non-standard semantics. In the interest of having a consistent interface and eliminating redundant code, we decided to take the second route. The union operation would assume that inputs would be closed intervals and therefore, the result would be closed intervals. The difference operations would assume that the set to be subtracted would consist of only open intervals and therefore, the result would be only of closed intervals. As mentioned above, this lead to the possibility of introducing artifacts of *singleton* intervals, where both endpoints have the same value. After consideration, however, we realized these points could be safely eliminated, as they are equivalent to a critical point at a local minimum or maximum. Green singletons could be eliminated without introducing additional danger, and a red singleton would represent a brush against (but not a cross into) the traffic aircraft's protected zone.

The formal verification of merge algorithm required us to think deeply about what the merge algorithm was trying to accomplish. During this analysis process we were able to develop an elegant solution which can be presented in a paragraph of text, instead of a complicated 400 line Java program with many special cases. In addition the formal verification process required us to clearly specify how our algorithm would behave at the points where there is a transition from one color to another.

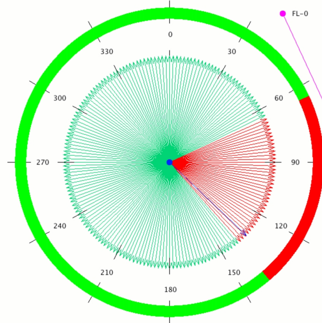



Figure 6: Screenshot of Track Prevention Band Display

## 7 Java and C++ Implementations

Ultimately these algorithms will be brought into large simulation environments where they will be evaluated for performance benefits (improvements to airspace capacity or aircraft efficiency). Some of these simulation environments are in Java and some are in C++. Therefore a requirement of this project was not only to develop an algorithm and verify it, but to also produce Java and C++ implementations of the algorithm. The initial Java version of the algorithm was available in December 2008 (see Figure 6). This version successfully passed the limited test suite we developed. By the summer of 2009 we had a C++ version and the testing apparatus to verify exact agreement between the Java and C++ versions, along with a regression suite of 100 test scenarios.

Since PVS is a specification language and contains non-computable functions, we deliberately restricted our use of PVS in the specification of our algorithms to only the computable elements. In this way there was a direct translation of PVS into Java or C++. We are currently developing a tool to automatically convert PVS specifications into Java [5], but the tool is not yet mature enough to handle the specification of these kinds of algorithms. However, we ran into problems even in our manual conversion of the algorithm. For instance, the PVS libraries contain all the appropriate vector operations (addition, dot-product, etc.) but libraries do not exist for these operations in standard Java or C++. We searched for a third-party library to offer these functions, but there were two downsides to these libraries. First, we wanted the same algorithm in both Java and C++, but most libraries were targeted to only one programming language. Second, we desired a library with identical semantics in both languages. But even when we found libraries that supported both languages, we inevitably discovered certain quirks in their implementation. For example, One vector library took full advantage of the imperative nature of the Java language, implementing functions on vectors which would change the components of the vector. While this results in efficient code, because object creation is not necessary, it does not closely relate to the functional style of PVS. Because of these incompatibilities, we chose to implement our own vector libraries. For similar reasons, we developed our own set operations (union and intersection).

Even with this hand translation, we still do not have an exact behavioral replica of the PVS in Java or C++. The most glaring difference is that Java and C++ use floating point numbers while PVS uses actual real numbers. All of our verifications in PVS are accomplished with vectors defined over the real numbers. This can be thought of as computation using infinite precision arithmetic. Clearly, our Java and C++ implementations execute on less powerful machines than this. There are several places where we must be especially careful:

- Calculation of quadratic discriminants. Since we are often computing tangents, the theoretical

value is zero, but the floating point answer can easily be a small negative number near zero. In this case, we would miss a critical point.

- The possibility of the mid-point of a region being very close to zero.

Finally, another aspect related to this issue is that the data input into the algorithm is not precise. A standard engineering assumption is that the error in the input data will overwhelm any error introduced by floating point computations. However, we would like to make a formal statement that includes both data and computational errors.

## 8 Conclusions

In this paper, we have presented a short history of the development and formal verification of prevention bands algorithms. The resulting track-angle, ground speed, and vertical speed bands algorithms are far more simple than our earlier versions. The goal of completing a formal proof forced us to search for simplifications in the algorithms and in the underlying mathematical theories. A key insight that enabled the completion of this work, is that trigonometric analysis should be deferred until the latest possible time. Although, the project took far longer than we expected, we are very pleased with the elegance and efficiencies of the discovered algorithms.

## References

[1] G. Dowek, A. Geser, and C. Muñoz. Tactical conflict detection and resolution in a 3-D airspace. In *Proceedings of the 4th USA/Europe Air Traffic Management R&DSeminar, ATM 2001*, Santa Fe, New Mexico, 2001. A long version appears as report NASA/CR-2001-210853 ICASE Report No. 2001-7.

[2] Gilles Dowek and C. Muñoz. Conflict detection and resolution for 1,2,...,N aircraft. In *Proceedings of the 7th AIAA Aviation, Technology, Integration, and Operations Conference, AIAA-2007-7737*, Belfast, Northern Ireland, 2007.

[3] J. Hoekstra, R. Ruigrok, R. van Gent, J. Visser, B. Gijsbers, M. Valenti, W. Heesbeen, B. Hilburn, J. Groeneweg, and F. Bussink. Overview of NLR free flight project 1997-1999. Technical Report NLR-CR-2000-227, National Aerospace Laboratory (NLR), May 2000.

[4] J. M. Hoekstra. Designing for safety: The free flight air traffic management concept. Technical Report 90-806343-2-8, Technische Universiteir Delft, November 2001.

[5] Leonard Lensink, César Muñoz, and Alwyn Goodloe. From verified models to verifiable code. Technical Memorandum NASA/TM-2009-215943, NASA, Langley Research Center, Hampton VA 23681-2199, USA, June 2009.

[6] Jeffrey Maddalon, Ricky Butler, Alfons Geser, and César Muñoz. Formal verification of a conflict resolution and recovery algorithm. Technical Report NASA/TP-2004-213015, NASA/Langley Research Center, Hampton VA 23681-2199, USA, April 2004.

[7] Jeffrey Maddalon, Ricky Butler, César Muñoz, and Gilles Dowek. A mathematical analysis of conflict prevention information. In *Proceedings of the AIAA 9th Aviation, Technology, Integration, and Operations Conference, AIAA-2009-6907*, Hilton Head, South Carolina, USA, September 2009.

[8] Jeffrey Maddalon, Ricky Butler, César Muñoz, and Gilles Dowek. A mathematical basis for the safety analysis of conflict prevention algorithms. Technical Report TM-2009-215768, NASA Langley, June 2009.

[9] S. Owre, J. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *Proc. 11th Int. Conf. on Automated Deduction*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer-Verlag, June 1992.

# A Machine-Checked Proof of A State-Space Construction Algorithm

Nestor Catano
The University of Madeira, M-ITI
Campus da Penteada, Funchal, Portugal
ncatano@uma.pt

Radu I. Siminiceanu
National Institute of Aerospace
100 Exploration Way, Hampton, VA 23666, USA
radu@nianet.org

**Abstract**

This paper presents the correctness proof of Saturation, an algorithm for generating state spaces of concurrent systems, implemented in the SMART tool. Unlike the Breadth First Search exploration algorithm, which is easy to understand and formalise, Saturation is a complex algorithm, employing a mutually-recursive pair of procedures that compute a series of non-trivial, nested local fixed points, corresponding to a chaotic fixed point strategy. A pencil-and-paper proof of Saturation exists, but a machine checked proof had never been attempted. The key element of the proof is the characterisation theorem of saturated nodes in decision diagrams, stating that a saturated node represents a set of states encoding a local fixed-point with respect to firing all events affecting only the node's level and levels below. For our purpose, we have employed the Prototype Verification System (PVS) for formalising the Saturation algorithm, its data structures, and for conducting the proofs.

## 1 Introduction

Software systems have become a key part of our lives, controlling or influencing many of the activities that we undertake every day. Software correctness is particularly important for safety-critical systems where people's lives can be at risk. Such systems have to be rigorously checked for correctness before they can be deployed. Several formal techniques and tools for reinforcing the dependability of critical systems exist. Temporal logic model checking is a technique to verify systems algorithmically, using a decision procedure that checks if a (usually finite) abstract model of the system, expressed as a state-transition system, satisfies a formal specification written as a temporal logic formula [7, 13]. Additionally, theorem proving is a technique to establish mathematical theorems and theories with the aid of a computer program, *i.e.*, a theorem prover. Theorem provers usually require the interaction with an experienced user to find a proof. While model checking has been successfully used to find errors, it is seldom used to seek the absolute correctness of a system or application. On the other hand, provers and proof checkers are oftentimes the most reliable means available for establishing a higher level of correctness.

For both approaches however, from the point of view of validation, and even certification, there is the outstanding issue of trustworthiness: if a formal analysis tool is employed, how does one establish the semantic validity of the analysis tool itself – in other words, *who is checking the checker?* While the formal methods community has been functioning on the premise of accumulated trust – some theorem provers and model checkers accumulate a track record of being trustworthy – the issue of certifying verification tools is of undeniable concern, the more so as experience shows that automated verification tools are far from being free of bugs [14].

Regulatory authorities (FAA, CAA, or DOD) require software development standards, such as MIL-STD-2167 for military systems and RTCA DO-178B for civil aircraft. While certified tools for generating code exist, they have been mostly confined to a very narrow segment, such as synchronous languages for embedded systems and are usually of proprietary nature.

In this paper we attempt to establish the correctness of a general purpose model checking algorithm, with the help of a theorem prover. More precisely, we present the PVS proof of correctness for Saturation [5], a non-trivial algorithm for model checking concurrent systems, implemented within the SMART [6] formal analysis tool. We have employed the Prototype Verification System (PVS) [12] for formalising the Saturation algorithm, the Multi-valued Decision Diagrams (MDDs)

[15] data structure, and for conducting the proofs. Unlike the Breadth First Search exploration algorithm which is easy to understand and formalise, Saturation is a high-performance algorithm, employing a mutually-preemptive, doubly-recursive pair of procedures that compute a series of nested local fixed points, corresponding to a *chaotic* global fixed point strategy. The key result of the proof is the characterisation theorem of saturated nodes in decision diagrams, stating that a saturated node represents a set of states encoding a local fixed-point with respect to *firing* all events affecting only the node's level and levels below. Saturation requires a *Kronecker consistent* partition of the system model in sub-models. The PVS proofs presented here take advantage of the Kronecker consistency property to express how the state-spaces generated by local next-state functions relate to the global state-space. In this regard, Saturation's correctness proof outlines an approach that may be re-used in the correctness proofs of an entire class of algorithms that rely on structural properties. Additionally, the PVS formalisation makes the invariant relating Saturation and the firing routine explicit. This is an important requirement for the SMART code. Its implementation, and the implementation of the structures it uses for storing state-spaces, must satisfy this invariant.

## 2 Preliminaries

Saturation employs Multi-valued Decision Diagrams (MDDs) [15], an extension of the classical Binary Decision Diagrams (BDDs), to symbolically encode and manipulate sets of states for constructing the state-space of structured asynchronous systems. MDDs encode characteristic functions for sets of the form $S_K \times \cdots \times S_1$ for systems comprising $K$ subsystems, each with its local state space $S_k$, for $K \geq k \geq 1$. The particular MDD variant used in Saturation (quasi-reduced, ordered) is formally given in Definition 1.

**Definition 1** (MDDs)**.** *MDDs are directed acyclic edge-labelled multi-graphs with the following properties:*

1. *Nodes are organised into* $K+1$ *levels from* $0$ *to* $K$*. The expression* $\langle k|p\rangle$ *denotes a generic node, where* $k$ *is the node's level and* $p$ *is a unique index for a node at that level.*
2. *Level* $0$ *consists of two terminal nodes* $\langle 0|0\rangle$ *and* $\langle 0|1\rangle$*, which we often denote* **0** *and* **1**.
3. *Level* $K$ *contains only a single non-terminal node* $\langle K,r\rangle$*, the root, whereas levels* $K-1$ *through* $1$ *contain one or more non-terminal nodes.*
4. *A non-terminal node* $\langle k|p\rangle$ *has* $|S_k|$ *arcs pointing to nodes at level* $k-1$*. An arc from the position* $i_k \in S_k$ *to the node* $\langle (k-1)|q\rangle$ *is denoted by* $\langle k|p\rangle[i_k] = q$*.*
5. *No two nodes are* duplicate, i.e.*, there are no nodes* $\langle k|p\rangle$ *and* $\langle k|q\rangle$ *such that* $p \neq q$ *and for all* $i_k \in S_k$*,* $\langle k|p\rangle[i_k] = \langle k|q\rangle[i_k]$*.*

In contrast to [15], not fully- but quasi-reduced ordered MDDs are considered in Saturation, hence *redundant* nodes, *i.e.*, nodes $\langle k|p\rangle$ such that $\langle k|p\rangle[i_k] = \langle k|p\rangle[j_k]$ for all $i_k \neq j_k$, are valid according to definitions used in Saturation. This relaxed requirement can potentially lead to a much larger number of nodes, but in practice this rarely occurs. On the other hand, the quasi-reduced form has the property that all the children of a node are at the same level, which can be exploited in the algorithm. Equally important, the quasi-reduced form is still canonical, as the fully-reduced form is. A sequence $\sigma$ of *local states* $(i_k,\ldots,i_1)$ is referred to as a *sub-state*. Given a node $\langle k|p\rangle$, the node reached from it through a sequence $\sigma$ of local states $(i_k,\ldots,i_1)$ is defined as:

$$\langle k|p\rangle[\sigma] = \begin{cases} \langle k|p\rangle & \text{if } \sigma = (), \text{ the empty sequence} \\ \langle (k-1)|\langle k|p\rangle[i_k]\rangle[\sigma'] & \text{if } \sigma = (i_k,\sigma'), \text{ with } i_k \in S_k\,. \end{cases}$$

A sub-state $\sigma$ constitutes a *path* starting at node $\langle k|p\rangle$, if $\langle k|p\rangle[\sigma] = \langle 0|1\rangle$. $\mathscr{B}(k,p)$ is the set of states encoded by $\langle k|p\rangle$, *i.e.*, those states constituting paths starting at node $\langle k|p\rangle$.

$$\mathscr{B}(k,p) = \{\sigma \in S_k \times \cdots \times S_1 : \langle k|p\rangle[\sigma] = \langle 0|1\rangle\}$$

### 2.1 Kronecker Consistency

Saturation requires a *Kronecker consistent* partition of the system model into $K$ sub-models. A next-state function $\mathcal{N}$ of a Kronecker structured asynchronous system model is defined on the potential state-space $\widehat{S}: S_K \times \cdots \times S_1$ decomposed by event, *i.e.*, $\mathcal{N} = \bigcup_{e \in \mathcal{E}} \mathcal{N}_e$. An event $e$ has a *Kronecker representation* for a given model partition if $\mathcal{N}_e$ can be written as the cross-product of $K$ local next-state functions, *i.e.*, $\mathcal{N}_e = \mathcal{N}_e^K \times \cdots \times \mathcal{N}_e^1$, where $\mathcal{N}_e^k : S_k \to 2^{|S_k|}$, for $K \geq k \geq 1$. Since $S_k$ are all assumed to be finite in Saturation, $\widehat{S}: S_K \times \cdots \times S_1$ can be regarded as a structure of the form $\widehat{S}: \{1, \cdots, n_K\} \times \cdots \times \{1, \cdots, n_1\}$, where $n_k = |S_k|$. A partition of a system model into sub-models is *Kronecker consistent* if every event $e$ has a Kronecker representation for that partition. The definition of $\mathcal{N}_e$ can be extended to a set of sub-states $X$, $\mathcal{N}_e(X) = \bigcup_{x \in X} \mathcal{N}_e(x)$, and to a set of events $\mathcal{E}$, $\mathcal{N}_{\mathcal{E}}(X) = \bigcup_{e \in \mathcal{E}} \mathcal{N}_e(X)$. An event *depends* on a particular level if the event affects the (local) state-space generated for that partition at that level. Let $Top(e)$ and $Bottom(e)$ be the highest and lowest levels an event $e$ depends, two cases in the definition of $\mathcal{N}_{\mathcal{E}}$ result when $\mathcal{E} = \{e : Top(e) \leq k\}$ and $\mathcal{E}^k = \{e : Top(e) = k\}$. We write $\mathcal{N}_{\leq k}$ as a shorthand for $\mathcal{N}_{\{e : Top(e) \leq k\}}$, and $\mathcal{N}_{=k}$ as a shorthand for $\mathcal{N}_{\mathcal{E}^k}$. The *reachable state-space* $S \subseteq \widehat{S}$ from an initial system state $X$ is the smallest set containing $X$ and being closed with respect to $\mathcal{N}$, *i.e.*, $S = X \cup \mathcal{N}(X) \cup \mathcal{N}(\mathcal{N}(X)) \cdots = \mathcal{N}^*(X)$, where $*$ denotes the reflexive and transitive closure.

### 2.2 Saturation

Saturation relies on a routine $Fire(e,k,p)$, for exhaustively firing all the events $e$ such that $Top(e) = k$, and recursively calling Saturation of any descendant $\langle k|p \rangle[i_k]$ of $\langle k|p \rangle$. $Fire$, like Saturation, checks whether $e$ is enabled in a node $\langle k|p \rangle$ and then reveals and adds globally reachable states to the MDD representation of the state-space under construction. However, unlike Saturation, $Fire$ operates on a fresh node instead of modifying $\langle k|p \rangle$ in place, since $\langle k|p \rangle$ is already saturated. In Saturation, MDDs are modified locally and only between the levels on which the fired event depends on. The enabling and the outcome of firing an event $e$ only depend on the states of sub-models $Top(e)$ through $Bottom(e)$. Definition 2 formalises the idea of a saturated node. Saturation and the routine for firing events are mutually recursive, related through the following invariant conditions: $Saturate$ is called on a node $\langle k|p \rangle$ whose children are already saturated, $Fire$ is always invoked on a saturated node $\langle l|q \rangle$ with $l < Top(e)$ and $Saturate$ is invoked just before returning from $Fire$.

**Definition 2** (Saturated Node)**.** *An MDD node $\langle k|p \rangle$ is saturated if it encodes a set of states that is a fixed-point with respect to the firing of any event affecting only the node's level or lower levels, that is, if $\mathcal{B}(k,p) = \mathcal{N}_{\leq k}^*(\mathcal{B}(k,p))$.*

### 2.3 Saturation's Correctness

For ease of reference, Figure 1 shows the pseudo-code for the Saturation and event firing algorithms. We reproduce here the original pencil-and-paper correctness proof of the theorem relating both algorithms, for ease of reference, then proceed with the formal PVS proof.

**Correctness.** Let $\langle k|p \rangle$ be a node with $K \geq k \geq 1$ and saturated children, and $\langle l|q \rangle$ be one of its children with $q \neq \mathbf{0}$ and $l = k-1$; let $\mathcal{U}$ stand for $\mathcal{B}(l,q)$ before the call $Fire(e,l,q)$, for some event $e$ with $l < Top(e)$, and let $\mathcal{V}$ represent $\mathcal{B}(l,f)$, where $f$ is the value returned by this call; Then, $\mathcal{V} = \mathcal{N}_{\leq l}^*(\mathcal{N}_e(\mathcal{U}))$.

*Proof.* By induction on $k$. For the induction base, $k = 1$, the only possible call $Fire(e,l,\mathbf{1})$ returns $\mathbf{1}$ because of the test on $l$, which has value 0, in Line 1 in Figure 1. Then, $\mathcal{U} = \mathcal{V} = \{()\}$ and $\{()\} = \mathcal{N}_{\leq 0}^*(\mathcal{N}_e(\{()\}))$.
For the induction step we assume that the call to $Fire(e,l-1,\cdot)$ works correctly. Recall that $l = k-1$. $Fire$ does not add further local states to $\mathcal{L}$, since it modifies "in–place" the new node $\langle l|s \rangle$, and not node $\langle l|q \rangle$ describing the states from where the firing is explored. The call $Fire(e,l,q)$ can be resolved in three ways. If $l < Bottom(e)$, then the returned value is $f = q$ and $\mathcal{N}_e^l(\mathcal{U}) = \mathcal{U}$ for any set $\mathcal{U}$;

since $q$ is saturated, $\mathscr{B}(l,q) = \mathscr{N}^*_{\leq l}(\mathscr{B}(l,q)) = \mathscr{N}^*_{\leq l}(\mathscr{N}_e(\mathscr{B}(l,q)))$. If $l \geq Bottom(e)$ but *Fire* has been called previously with the same parameters, then the call $Find(FCache[l], \{q,e\}, s)$ is successful. Since node $q$ is saturated, it has not been modified further; Finally, we need to consider the case where the call $Fire(e,l,q)$ performs "real work." First, a new node $\langle l|s\rangle$ is created, having all its arcs initialised to $\mathbf{0}$. We explore the firing of $e$ in each state $i$ satisfying $\langle l|q\rangle[i] \neq \mathbf{0}$ and $\mathscr{N}_l^e(i) \neq \emptyset$. By induction hypothesis, the recursive call $Fire(e,l-1,\langle l|q\rangle[i])$ returns $\mathscr{N}^*_{\leq l-1}(\mathscr{N}_e(\mathscr{B}(l-1,\langle l|q\rangle[i])))$. Hence, when the "while $\mathscr{L} \neq \emptyset$" loop terminates, $\mathscr{B}(l,s) = \bigcup_{i\in\mathscr{S}^l} \mathscr{N}_e^l(i) \times \mathscr{N}^*_{\leq l-1}(\mathscr{N}_e(\mathscr{B}(l-1,\langle l|q\rangle[i]))) = \mathscr{N}^*_{\leq l-1}(\mathscr{N}_e(\mathscr{B}(l,q)))$ holds. Thus, all children of node $\langle l|s\rangle$ are saturated. According to the induction hypothesis, the call $Saturate(l,s)$ correctly saturates $\langle l|s\rangle$.

Therefore, we have $\mathscr{B}(l,s) = \mathscr{N}^*_{\leq l}(\mathscr{N}^*_{\leq l-1}(\mathscr{N}_e(\mathscr{B}(l,q))) = \mathscr{N}^*_{\leq l}(\mathscr{N}_e(\mathscr{B}(l,q)))$ after the call. □

# 3 The PVS Formalisation

We used the Prototype Verification System (PVS) [12] for formalising *Saturate*, the MDD data structure it uses to store state-spaces, and for conducting the correctness proofs. Our formalisation is purely functional, e.g., we do not formalise memory and memory operations. The main goal of our PVS formalisation is to machine-check the pencil-and-paper correctness proof of *Saturate*, introduced in Section 2.3 and Figure 1. Modelling memory is necessary when one is interested in generating actual code from the formalisation. We are planning to port our PVS formalisation to AtelierB [2] and use refinement calculus techniques to generate Saturation actual code from the B model. Yet, this is future work.

We started formalising the basic concepts employed by the definition of Kronecker consistency such as events, states, local state values, next-state functions, local next-state functions. Then, we formalised the Saturation algorithm and the routine for firing events, and conducted their correctness proof in PVS following the pencil-and-paper proof. In the following, we present the definition of some of those basic concepts in PVS. Predicate `local_value?(m)` below formalises local state values at level `m`, where `nk` is the function $n_k = |S_k|$. The reader should remember that *Saturate* generates state spaces of values on a level basis. This is a direct consequence of the definition of Kronecker consistency. `local_value(m)` is the type of all elements satisfying the predicate `local_value?(m)`, and the type `state(k)` formalises sub-states of size `k` as sequences `s` whose elements `` s`sq(m) `` at any position `m` ≤ `k` are restricted by `nk`. We further define `s_t(k,s,m)` (not shown here) to be a sub-state of `s` of size `m` ≤ `k`, where `k` is the size of `s`.

```
local_value?(m)(n): bool = (m=0 ∧ n=0) ∨ (m > 0 ∧ n > 0 ∧ n ≤ nk(m))
local_value(m): type = (local_value?(m))
state(k): type = { s:Seq(k) | ∀(m:upto(k)): (m=0 ∧ s`sq(m)=0) ∨
                                   (m > 0 ∧ s`sq(m) > 0 ∧ s`sq(m) ≤ nk(m)) }
```

We use PVS datatypes to model MDDs, formed using *type constructors*, and define predicates `ordered?` and `reduced?` (not shown here), modelling ordered and reduced MDDs. Predicate `reduced?` subsumes `ordered?`. The type `OMDD` below formalises Definition 1. We further define function `level`, which returns the height of an MDD node, `child(p,i)`, which returns the `i`-th child of an MDD `p`, and predicate `trivial?` that holds of an MDD if it is **0** or **1**.

```
OMDD: type = (reduced?)
```

Events are formalised as the type `event` below with two functions `Top` and `Bottom` returning the highest and lowest level an event depends upon. The symbol `+` in the type definition indicates that the type `event` is non-empty. `TopLesser(k)` models the set $\mathscr{E} = \{e \colon Top(e) \leq k\}$.

```
event: type+

Top: [event -> posnat]
Bottom: [event -> posnat]

TopLesser(k): setof[event] = {e:event | Top(e) ≤ k}
```

*Saturate*(in $k$:*level*, $p$:*index*)

> Update $\langle k|p\rangle$ in–place, to encode $\mathcal{N}^{*}_{\leq k}(\mathcal{B}(\langle k|p\rangle))$.

declare $e$:*event*;
declare $\mathcal{L}$:set of *local*;
declare $f,u$:*index*;
declare $i,j$:*local*;
declare *pChanged*:*bool*;

1. repeat
2. *pChanged* $\leftarrow$ *false*;
3. foreach $e \in \mathcal{E}^k$ do
4. $\mathcal{L} \leftarrow Locals(e,k,p)$;
5. while $\mathcal{L} \neq \emptyset$ do
6. $i \leftarrow Pick(\mathcal{L})$;
7. $f \leftarrow Fire(e,k-1,\langle k|p\rangle[i])$;
8. if $f \neq \mathbf{0}$ then
9. foreach $j \in \mathcal{N}^k_e(i)$ do
10. $u \leftarrow Union(k-1,f,\langle k|p\rangle[j])$;
11. if $u \neq \langle k|p\rangle[j]$ then
12. $\langle k|p\rangle[j] \leftarrow u$;
13. *pChanged* $\leftarrow$ *true*;
14. if $\mathcal{N}^k_e(j) \neq \emptyset$ then
15. $\mathcal{L} \leftarrow \mathcal{L} \cup \{j\}$;
16. until *pChanged* $=$ *false*;

*Union*(in $k$:*level*, $p$:*index*, $q$:*index*):*index*

> Build an MDD rooted at $\langle k|s\rangle$ encoding $\mathcal{B}(\langle k|p\rangle) \cup \mathcal{B}(\langle k|q\rangle)$. Return $s$.

declare $i$:*local*;
declare $s,u$:*index*;

1. if $p = \mathbf{1}$ or $q = \mathbf{1}$ then return $\mathbf{1}$;
2. if $p = \mathbf{0}$ or $p = q$ then return $q$;
3. if $q = \mathbf{0}$ then return $p$;
4. if $Find(UCache[k],\{p,q\},s)$ then return $s$;
5. $s \leftarrow NewNode(k)$;
6. for $i = 0$ to $n^k - 1$ do
7. $u \leftarrow Union(k-1,\langle k|p\rangle[i],\langle k|q\rangle[i])$;
8. $\langle k|s\rangle[i] \leftarrow u$;
9. $Check(k,s)$;
10. $Insert(UCache[k],\{p,q\},s)$;
11. return $s$;

*Fire*(in $e$:*event*, $l$:*level*, $q$:*index*):*index*

> Build an MDD rooted at $\langle l|s\rangle$ encoding $\mathcal{N}^{*}_{\leq l}(\mathcal{N}_e(\mathcal{B}(\langle l|q\rangle)))$. Return $s$.

declare $\mathcal{L}$:set of *local*;
declare $f,u,s$:*index*;
declare $i,j$:*local*;
declare *sChanged*:*bool*;

1. if $l < Bottom(e)$ then return $q$;
2. if $Find(FCache[l],\{q,e\},s)$ then
3. return $s$;
4. $s \leftarrow NewNode(l)$;
5. *sChanged* $\leftarrow$ *false*;
6. $\mathcal{L} \leftarrow Locals(e,l,q)$;
7. while $\mathcal{L} \neq \emptyset$ do
8. $i \leftarrow Pick(\mathcal{L})$;
9. $f \leftarrow Fire(e,l-1,\langle l|q\rangle[i])$;
10. if $f \neq \mathbf{0}$ then
11. foreach $j \in \mathcal{N}^l_e(i)$ do
12. $u \leftarrow Union(l-1,f,\langle l|s\rangle[j])$;
13. if $u \neq \langle l|s\rangle[j]$ then
14. $\langle l|s\rangle[j] \leftarrow u$;
15. *sChanged* $\leftarrow$ *true*;
16. if *sChanged* then
17. $Saturate(l,s)$;
18. $Check(l,s)$;
19. $Insert(FCache[l],\{q,e\},s)$;
20. return $s$;

*Locals*(in $e$:*event*, $k$:*level*, $p$:*index*): set of *local*

> Return $\{i \in \mathcal{S}^k : \langle k|p\rangle[i] \neq \mathbf{0}, \mathcal{N}^k_e(i) \neq \emptyset\}$, the local states in $p$ locally enabling $e$. Return $\emptyset$ or $\{i \in \mathcal{S}^k : \mathcal{N}^k_e(i) \neq \emptyset\}$, respectively, if $p$ is $\mathbf{0}$ or $\mathbf{1}$.

*Check*(in $k$:*level*, inout $p$:*index*)

> If $\langle k|p\rangle$ is the duplicate of an existing $\langle k|q\rangle$ delete $\langle k|p\rangle$ and set $p$ to $q$. Else, insert $\langle k|p\rangle$ in the unique table. If $\langle k|p\rangle[0] = \cdots = \langle k|p\rangle[n^k-1] = \mathbf{0}$ or $\mathbf{1}$, delete $\langle k|p\rangle$ and set $p$ to $\mathbf{0}$ or $\mathbf{1}$, since $\mathcal{B}(\langle k|p\rangle)$ is $\emptyset$ or $\mathcal{S}^k \times \cdots \times \mathcal{S}^1$, respectively.

Figure 1: Pseudo–code for the node–saturation and event firing algorithms.

The type `next(k)` describes $\mathcal{N}_e$ for some event $e$. Local next-state functions at level `k` are introduced by the type `Localnext(k)`. A finite sequence of local next-state functions `fs` makes a next-state function `N` Kronecker consistent, `Kronecker?(k)(N)(fs)`, if for each event `e`, and sub-states `x` and `y` of size `k`, every `y`'s local state `y`sq(m)` with $\mathtt{m} \leq \mathtt{k}$ is an image of `x`'s local state `x`sq(m)` through the local next-state function `fs`sq(m)(e)` (modelling $\mathcal{N}_e^m$).

```
next(k): type = [ event ->  [state(k) -> setof[state(k)]] ]
Localnext(k): type = [ event  -> [local_value(k)  -> setof[local_value(k)]] ]

Kronecker?(k)(N)(fs): bool =
 ∀(e:event, x,y:state(k)):
   N(e)(x)(y) ⇔ ∀(m:upto(k)): fs`sq(m)(e)(x`sq(m))(y`sq(m))
```

`NextLesser(k)(N,fs)(x)` formalises $\mathcal{N}_{\leq k}$ applied over a sub-state $x$ consisting of `k` levels. The formalisation assumes that `fs` is a sequence of local next-state functions that make `N` Kronecker consistency. That is, $\mathcal{N}_e$ has a Kronecker representation for all event $e$ such that $Top(e) \leq k$. We further define `NextLesser(k,m)(N,fs)` similar to `NextLesser(m)(N,fs)` (not shown here) for levels less than or equal to `m`, and equals to the identity function from `m+1` to `k`. Definition of `NextLesser(m)(N,fs)` is extended to a set `X` of sub-states in the natural way.

```
NextLesser(k)(N, fs)(x): setof[state(k)] =
 { y:state(k) | ∃(e:(TopLesser(k))):
                 ∀(m:upto(k)): fs`sq(m)(e)(x`sq(m))(y`sq(m)) }
```

$\mathcal{N}^*_{\leq k}(X)$ is formalised as `Apply(k)(N,fs)(w)(X)` below, where `w` represents the number of iterations after which applying $\mathcal{N}_{\leq k}$ on $X$ does not generate any new state. That is, $X$ is a fixed-point of $\mathcal{N}_{\leq k}$. The existence of such `w` is guaranteed by our formalisation since $X$ is a finite set, and every $\mathcal{N}_e$ in $\mathcal{N}_{\leq k}$ with $Top(e) \leq k$ is an increasing function. A similar definition for `Apply(k,m)(N,fs)(w)(X)` exists (not shown here) that uses `NextLesser(k,m)(N,fs)(X)` instead of `NextLesser(k)(N,fs)(X)`. As a consequence of the definition of $\mathcal{N}^*_{\leq k}$, and because we are considering Kronecker consistency next-state functions, we have that $\mathcal{N}^*_{\leq k}(\mathcal{N}^*_{\leq k-1}(X)) = \mathcal{N}^*_{\leq k}(X)$ (Corollary `Apply_cor1`).

```
Apply(k)(N,fs)(w)(X): recursive setof[state(k)] =
 if w=0 then X elsif w=1 then union(X,NextLesser(k)(N,fs)(X))
 else union(X,Apply(k)(N,fs)(w-1)(X)) endif
measure w

Apply_cor1: corollary
 ∀(k:{n:upto(K) | n > 0}, N:next(k), fs:(kronecker?(k)(N)),
   w:posnat, X:setof[state(k)], s:state(k)):
 Apply(k)(N,fs)(w)(Apply(k,k-1)(N,fs)(w)(X))(s) ⇔ Apply(k)(N,fs)(w)(X)(s)
```

Finally, we formalise the routine for firing events and model its usage in Saturation. We employed an axiomatic approach rather than writing a definition for the routine. Although axioms might potentially introduce inconsistencies, the use of definitions may force PVS to generate additional proof obligations all over the lemmas and theorems using the definitions, cluttering their proofs. Furthermore, an axiomatic approach is preferable when one is not interested in generating code for the algorithm directly. The firing of an event $e$ on a node $\langle l|q\rangle$, `fire(l,e,N,fs,w,q)` is described by axioms `fire_trivial`, `fire_nontrivial`, `fire_recursive`, and `fire_ saturated` below, with `l` $\leq$ `K`, `e:{ev:event | l < Top(ev)}`, `N:next(l)`, `fs:(Kronecker?(l )(N))`, and `q:{u:OMDD | level(u)=l ∧ saturated?(l,u)(N,fs)(w)}`. Therefore, to be able to call `fire(l,e,N,fs,w,q)`, event `e` should be such that `l < Top(e)`, and `q` should be such that `saturated?(l,q)(N, fs)(w)`, in accordance with the "*Fire* is always invoked on a saturated node $\langle l|q\rangle$ with $l < Top(e)$" invariant condition. `fire_trivial` states that `fire(l,e,N,fs,w,q)` returns `q` unaffected when either `l < Bottom(e)` or `l = 0`. These two conditions describe the cases when the recursive call to `fire(l,e,N,fs,w,q)` ends. `fire_nontrivial` states that if `l > 0`, then `fire(l,e,N,fs,w,q)`'s

call returns a node whose level is the same as q's level, that is l. trivial? holds of an MDD if it is **0** or **1**. fire_recursive states that the set Below(l,fire(l,e,N,fs,w,q)) of states of size l encoded by a call to fire(l,e,N,fs,w,q) is recursively generated on a Kronecker structured level-basis. That is, at level l, the local next-state function $\mathcal{N}_e^l$ (see fs`sq(l)(e)) generates all the local states for every local state value i at level l, and recursively, fire(l-1,e,$N_{l'}$,$fs_{l'}$,w,child(q,i)) generates all the sub-states of size l-1. The final challenge in our formalisation comes from modelling the mutual recursion between *Saturate* and *Fire*. Mutual recursion is not directly supported by PVS. The axiom fire_saturated formalises the invariant "*Saturate* is invoked just before returning from *Fire*" in which we model Saturated as a property of a node that is fired on a particular event. Notice that fire_saturated cannot directly be expressed as a definition.

```
fire(l,e,N,fs,w,q): OMDD

fire_trivial: axiom l < Bottom(e) ∨ l=0 ⇒ fire(l,e,N,fs,w,q)=q

fire_nontrivial: axiom
 l > 0 ⇒ ¬trivial?(fire(l,e,N,fs,w,q)) ∧ level(fire(l,e,N,fs,w,q))=l

fire_recursive: axiom
 Below(l,fire(l,e,N,fs,w,q)) =
  { s:state(l) | ∃(i:local_value(l)): fs`sq(l)(e)(i)(s`sq(l)) ∧
            Below(l-1,fire(l-1,e,N_l',fs_l',w,child(q,i)))(s_t(l,s,l-1)) }

fire_saturated: axiom saturated?(l,fire(l,e,N,fs,w,q))(N,fs)(w)
```

## 3.1 Saturation's Formalisation

Theorem 1 formalises $\mathcal{N}^*_{\leq k-1}(\mathcal{N}_e(\mathcal{B}(k,p))) = \bigcup_{i \in S^k} \mathcal{N}_e^k(i) \times \mathcal{N}^*_{\leq k-1}(\mathcal{N}_e(\mathcal{B}(\langle k-1 | \langle k|p \rangle [i] \rangle)))$, which is true in Kronecker systems.

**Theorem 1** (Applying Kronecker Consistent Next-State Functions)**.** *This theorem is used in Saturation's correctness proof (Theorem 2).*

```
kronecker_apply: theorem
 ∀(k:{n:upto(K) | n > 0}, p:{u:OMDD | ¬trivial?(u) ∧ level(u)=k}, ev:event,
   N:next(k), fs:(kronecker?(k)(N)), w:posnat, w_1:posnat, s:state(k)):
    ( ∃(i:local_value(k)):
        fs`sq(k)(ev)(i)(s`sq(k)) ∧
        Apply(k-1)(N_k',fs_k')(w_1)(Next(k-1,ev)(N_k',fs_k')(Below(k-1,child(p,i))))
                              (s_t(k,s,k-1)) )
         ⇔
    Apply(k,k-1)(N,fs)(w)(Next(k,ev)(N,fs)(Below(k,p)))(s)
```

The proof of Theorem 1 is conducted under the well_defined assumption below, which states that since "the enabling and the outcome of firing an event *e* only depend on the states of sub-models *Top*(*e*) through *Bottom*(*e*)", if $k < Bottom(e)$ then applying $\mathcal{N}_{\leq k}$ to $\mathcal{B}(k,p)$ does not generate any new state, and the non_decreasing assumption, which states that all considered local next-state functions f are non-decreasing. Below(k,p) formalises $\mathcal{B}(k,p)$, the set of states encoded by $\langle k|p \rangle$.

```
well_defined: assumption
 ∀(k:upto(K), p:{u:OMDD | level(u)=k}, e:event,
   N:next(k), fs:(Kronecker?(k)(N))):
  k < Bottom(e) ⇒ Below(k,p) = Next(k,e)(N,fs)(Below(k,p))

non_decreasing: assumption
  ∀(k:upto(K), e:event, f:Localnext(k), i:local_value(k)): f(e)(i)(i)
```

Theorem 1 is proved by induction on w and $w_1$. The base case, w=1 and $w_1$=1, is proved from the lemma below_incremental below. This lemma states that a state s of size k>0 belongs to Below(k,p) if and only if p is not trivial (p is different to **0** and **1**), and s_t(k,s,k-1) belongs to Below(k-1,child(n,s`sq(s`ln))).

```
below_incremental: lemma
  Below(k,p)(s) ⇔
  ( ¬trivial?(p) ∧ Below(k-1,child(p,s`sq(s`ln)))(s_t(k,s,k-1)) )
```

The inductive step for Theorem 1 is shown below. It reduces after expanding the definition of Apply in the w+1 part of the logical equivalence.

```
Apply(k,k-1)(N,fs)(w)(Next(k,ev)(N,fs)(Below(k,p)))(s)
  ⇔
Apply(k,k-1)(N,fs)(w+1)(Next(k,ev)(N,fs)(Below(k,p)))(s)
```

Theorem 2 below formalises Saturation's correctness condition in PVS. The PVS proof of this theorem follows the pencil-and-paper proof in Section 2.3.

**Theorem 2** (Saturation's Correctness)**.** $\mathscr{B}(l,f) = \mathscr{N}^*_{\leq l}(\mathscr{N}_e(\mathscr{U}))$*, where f is the value returned by Fire*$(l,e,q)$*'s call, and* $\mathscr{U}$ *stands for* $\mathscr{B}(l,q)$ *before calling Fire*$(l,e,q)$*, for some event e such that* $l < Top(e)$*.*

```
saturation_correctness: theorem
 ∀(l:upto(K), e:{ev:event | l < Top(ev)}, N:next(l), fs:(Kronecker?(l)(N),
   w:posnat, q:{u:OMDD | level(u)=l ∧ saturated?(l,u)(N,fs)(w)})):
  Below(l,fire(l,e,N,fs,w,q)) =
   Apply(l)(N,fs)(w)(Next(l,e)(N,fs)(Below(l,q)))
```

*Proof.* By induction on l.

(*i.*) Base case (l=0). If l=0 then fire(l,e,N,fs,w,q) = q (Axiom fire_trivial). Below(0,q) equals TSeq (the empty sequence), and Next(0,e)(N,fs)({TSeq}) = {TSeq}. Because Apply(0)(N,fs)(w)({TSeq}) = {TSeq}, the base case reduces trivially.

```
level(q)=0 ∧ 0 < Top(e) ∧ reduced?(q) ∧
Kronecker?(0)(N)(fs) ∧ saturated?(0,q)(N,fs)(w)
  ⇒
Below(0,fire(0,e,N,fs,w,q)) =
 Apply(0)(N,fs)(w)(Next(0,e)(N,fs)(Below(0,q)))
```

(*ii.*) Inductive step with f = fire(l+1,e,N,fs,w,q).

```
l+1 < Top(e) ∧ level(q)=l+1 ∧ reduced?(q) ∧
Kronecker?(l+1)(N)(fs) ∧ saturated?(l+1,q)(N,fs)(w) ∧
(∀(ee:{ev:event | l < Top(ev)}, N1:next(l), fs1:(Kronecker?(l)(N1)),
   ww:nat, qq:{u:OMDD | level(u)=l ∧ saturated?(l,u)(N1,fs1)(ww)}):
  Below(l,fire(l,ee,N1,fs1,ww,qq)) =
    Apply(l)(N1,fs1)(ww)(Next(l,ee)(N1,fs1)(Below(l,qq))))
  ⇒
Below(l+1,f) = Apply(l+1)(N,fs)(Next(l+1,e)(N,fs)(Below(l+1,q)))
```

(*ii.i*) Let us suppose l+1 < Bottom(e). From axiom fire_trivial, f = fire(l+1,e,N,fs, w,q) = q. The proof is discharged from this, saturated?(l+1,q)(N,fs)(w) in the antecedent of the proof, and the assumption well_defined.

(*ii.ii*) Let us suppose l+1 $\geq$ Bottom(e). If trivial?(q) then level(q) = 0, which contradicts the hypothesis level(q) = l+1. We hence assume ¬trivial?(q) afterwards. From the axiom fire_saturated and saturated?(l,f)(N,fs)(w) in the hypothesis of the proof, Below(l+1,f) = Apply( l+1)(N,fs)(w)(Below(l+1,f)). Because

`Below(l+1,f) = Apply(l+1,l)(N,fs)(w)(Next(l+1,e)(N,fs)(Below(l+1,q)))`[1], then `Below(l+1,f) = Apply(l+1)(N,fs)(w)(Apply(l+1,r)(N,fs)(w)(Next( l+1, e)(N,fs)(Below(l+1,q))))`. Therefore, from Corollary `Apply_cor1` in Section 3, `Below(l+1,f) = Apply(l+1)(N,fs)(Next(l+1, e)(N,fs) (Below(l+1,q)))`.

□

# 4 Conclusion and Future Work

Saturation is a high-performance non-trivial algorithm with an existing pencil-and-paper correctness proof. Conducting Saturation's correctness proof in PVS allowed us to verify correct the existing pencil-and-paper proof. Additionally, the PVS type-checker ensures that all the definitions in Saturation are type-correct, and that details are not overlooked. The Kronecker consistency property of systems considered in Saturation allows a separation of concerns so that proof-constraints did not clutter the actual structural proofs we conducted. In this regard, Saturation's correctness proof outlines a proof approach for an entire family of algorithms relying on structural properties. However, there is still a missing link. We proved the correctness of a model of Saturation. But, how do we know that Saturation's implementation faithfully attests to this model? As future work, we will pursue research in generating Java or C code from the PVS formalisation of Saturation, in the spirit of C. Muñoz and L. Lensink's work in [11], and comparing this code with the existing implementation of Saturation in the SMART formal analysis tool.

The full formalisation of Saturation in PVS consists of 7 theories, 10 lemmas, 7 corollaries, 2 main theorems, and 107 Type-Correctness Conditions (TCCs). The full formalisation can be reached at `http://www.uma.pt/ncatano/satcorrectness/saturation-proofs.htm`. Our formalisation is purely functional, e.g., we do not formalise memory, or memory operations.

**Future Work.** In [11], Muñoz and Lensink present a prototype code generator for PVS which translates a subset of PVS functional specifications into the Why language [8] then to Java code annotated with JML specifications [3, 4]. However, the code generator is still a proof of concept so that many of its features have to be improved. We will pursue research in that direction so as to generate Java certified code from the PVS formalisation of Saturation, and compare this with the existing implementation of Saturation in the SMART formal analysis tool.

In a complementary direction, our PVS formalisation of Saturation can be ported into B [1]. Then, using refinement calculus techniques [9, 10], e.g., implemented in the AtelierB tool [2], code implementing Saturation can be generated. This code is ensured to comply with the original formalisation of Saturation. A predicate calculus definition would require that axiomatisation for the routine for firing events (Section 3) is replaced by a more definitional style of modelling.

# References


[1] J. R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[2] AtelierB. `http://www.atelierb.eu/index_en.html`.

[3] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G.T. Leavens, K.R.M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(3):212–232, June 2005.

[4] N. Catano, F. Barraza, D.García, P. Ortega, and C. Rueda. A case study in JML-assisted software development. In *SBMF: Brazilian Symposium on Formal Methods*, 2008.

[5] G. Ciardo, G. Lüttgen, and R. Siminiceanu. Saturation: An efficient iteration strategy for symbolic state-space generation. In *Tools and Algorithms for the Construction and Analysis of*


[1] This result is not proved here.

*Systems (TACAS)*, volume 2031 of *Lecture Notes in Computer Science*, pages 328–342, Genova, Italy, April 2001. Springer-Verlag.

[6] Gianfranco Ciardo, R. L. Jones III, Andrew S. Miner, and Radu Siminiceanu. Logic and stochastic modeling with SMART. *Journal of Performance Evaluation*, 63(6):578–608, 2006.

[7] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite state concurrent system using temporal logic specifications. In *Proceedings of TOPLAS*, pages 244–263, 1986.

[8] J.-C. Filliâtre and Claude Marché. Multi-prover verification of c programs. In *International Conference on Formal Engineering Methods (ICFEM)*, volume 3308 of *Lecture Notes in Computer Science*, pages 15–29, 2004.

[9] J. He, C. A. R. Hoare, and J. W. Sanders. Data refinement refined. In *European Symposium on Programming (ESOP)*, pages 187–196, 1986.

[10] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

[11] Leonard Lensink, César Muñoz, and Alwyn Goodloe. From verified models to verifiable code. Technical Memorandum NASA/TM-2009-215943, NASA, Langley Research Center, Hampton VA 23681-2199, USA, June 2009.

[12] S. Owre, J. M. Rushby, , and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, Jun 1992. Springer-Verlag.

[13] A. Pnueli. The temporal logic of programs. In *Symposium on the Foundations of Computer Science (FOCS)*, pages 46–57, Providence, Rhode Island, USA, 1977. IEEE Computer Society Press.

[14] K. Y. Rozier and M. Vardi. LTL satisfiability checking. In *SPIN*, pages 149–167, 2007.

[15] R. K. Brayton Timothy Y.K. Kam, T. Villa and A. L. Sangiovanni-Vincentelli. Multi-valued decision diagrams: Theory and applications. *Multiple-Valued Logic*, 4(1–2), 1998.

# Automated Assume-Guarantee Reasoning for Omega-Regular Systems and Specifications

Sagar Chaki Arie Gurfinkel
Software Engineering Institute, Carnegie Mellon University

**Abstract**

We develop a learning-based automated Assume-Guarantee (AG) reasoning framework for verifying $\omega$-regular properties of concurrent systems. We study the applicability of non-circular (**AG-NC**) and circular (**AG-C**) AG proof rules in the context of systems with infinite behaviors. In particular, we show that **AG-NC** is incomplete when assumptions are restricted to strictly infinite behaviors, while **AG-C** remains complete. We present a general formalization, called LAG, of the learning based automated AG paradigm. We show how existing approaches for automated AG reasoning are special instances of LAG. We develop two learning algorithms for a class of systems, called $\infty$-regular systems, that combine finite and infinite behaviors. We show that for $\infty$-regular systems, both **AG-NC** and **AG-C** are sound and complete. Finally, we show how to instantiate LAG to do automated AG reasoning for $\infty$-regular, and $\omega$-regular, systems using both **AG-NC** and **AG-C** as proof rules.

## 1 Introduction

Compositional reasoning [8, 13] is a widely used technique for tackling the *statespace explosion* problem while verifying concurrent systems. Assume-Guarantee (AG) is one of the most well-studied paradigms for compositional reasoning [19, 14]. In AG-style analysis, we infer global properties of a system from the results of local analysis on its components. Typically, to analyze a system component $C$ locally, we use an appropriate "assumption", a model of the rest of the system that reflects the behavior expected by $C$ from its environment in order to operate correctly. The goal of the local analyses is then to establish that every assumption made is also "guaranteed" – hence Assume-Guarantee.

Since its inception [18, 16], the AG paradigm has been explored in several directions. However, a major challenge in automating AG reasoning is constructing appropriate assumptions. For realistic systems, such assumptions are often complicated, and, therefore, constructing them manually is impractical. In this context, Cobleigh et al. [9] proposed the use of learning to automatically construct appropriate assumptions to verify a system composed of finite automata against a finite automaton specification (i.e., to verify safety properties). They used the following sound and complete AG proof rule:

$$\frac{M_1 \parallel A \sqsubseteq S \qquad M_2 \sqsubseteq A}{M_1 \parallel M_2 \sqsubseteq S}$$

where $M_1, M_2, A$ and $S$ are finite automata, $||$ is a parallel composition, and $\sqsubseteq$ denotes language containment. The essential idea is to use the $\mathbf{L}^*$ algorithm [2] to learn an assumption $A$ that satisfies the premises of the rule, and implement the minimally adequate teacher required by $\mathbf{L}^*$ via model-checking.

The learning-based automated AG paradigm has been extended in several directions [6, 1, 21]. However, the question of whether this paradigm is applicable to verifying $\omega$-regular properties (i.e., liveness and safety) of reactive systems is open. In this paper, we answer this question in the affirmative. An automated AG framework requires: (i) an algorithm that uses queries and counterexamples to learn an appropriate assumption, and (ii) a set of sound and complete AG rules. Recently, a learning algorithm for $\omega$-regular languages has been proposed by Farzan et al. [10]. However, to our knowledge, the AG proof rules have not been extended to $\omega$-regular properties. This is the problem we address in this paper.

*First*, we study the applicability of non-circular (**AG-NC**) and circular (**AG-C**) AG proof rules in the context of systems with infinite behaviors. We assume that processes synchronize on shared events and proceeding asynchronously otherwise, i.e., as in CSP [15]. We prove that, in this context, **AG-NC** is sound but *incomplete* when restricted to languages with strictly infinite behaviors (e.g., $\omega$-regular). This is surprising and interesting. In contrast, we show that **AG-C** is both sound and complete for $\omega$-regular languages. *Second*, we extend our AG proof rules to systems and specifications expressible in $\infty$-regular

languages (i.e., unions of regular and $\omega$-regular languages). We show that both **AG-C** and **AG-NC** are sound and complete in this case. To the best of our knowledge, these soundness and completeness results are new. We develop two learning algorithms for $\infty$-regular languages – one using a learning algorithm for $\omega$-regular languages (see Theorem 8(a)) with an augmented alphabet, and another combining a learning algorithm for $\omega$-regular languages with $\mathbf{L}^*$ (see Theorem 8(b)) without alphabet augmentation. *Finally*, we present a very general formalization, called LAG, of the learning based automated AG paradigm. We show how existing approaches for automated AG reasoning are special instances of LAG. Furthermore, we show how to instantiate LAG to develop automated AG algorithms for $\infty$-regular, and $\omega$-regular, languages using both AG-NC and AG-C as proof rules.

The rest of the paper is structured as follows. We present the necessary background in Section 2. In Section 3, we review our model of concurrency. In Section 4, we study the soundness and completeness of AG rules, and present our LAG framework in Section 5. We conclude the paper with an overview of related work in Section 6.

## 2 Preliminaries

We write $\Sigma^*$ and $\Sigma^\omega$ for the set of all finite and infinite words over $\Sigma$, respectively, and write $\Sigma^\infty$ for $\Sigma^* \cup \Sigma^\omega$. We use the standard notation of regular expressions: $\lambda$ for empty word, $a \cdot b$ for concatenation, $a^*$, $a^+$, and $a^\omega$ for finite, finite and non-empty, and infinite repetition of $a$, respectively. When $a \in \Sigma^\omega$, we define $a \cdot b = a$. These operations are extended to sets in the usual way, e.g., $X \cdot Y = \{x \cdot y \mid x \in X \wedge y \in Y\}$.

**Language.** A language is a pair $(L, \Sigma)$ such that $\Sigma$ is an alphabet and $L \subseteq \Sigma^\infty$. The alphabet is an integral part of a language. In particular, $(\{a\},\{a\})$ and $(\{a\},\{a,b\})$ are different languages. However, for simplicity, we often refer to a language as $L$ and mention $\Sigma$ separately. For instance, we write "language $L$ over alphabet $\Sigma$" to mean the language $(L,\Sigma)$, and $\Sigma(L)$ to mean the alphabet of $L$. Union and intersection are defined as usual, but only for languages over the same alphabet. The complement of $L$, denoted $\overline{L}$, is defined as: $\overline{L} = \Sigma(L)^\infty \setminus L$. A finitary language ($\Sigma^*$-language) is a subset of $\Sigma^*$. An infinitary language ($\Sigma^\omega$-language) is a subset of $\Sigma^\omega$. For $L \subseteq \Sigma^\infty$, we write $*(L)$ for the finitary language $L \cap \Sigma^*$ and $\omega(L)$ for the infinitary language $L \cap \Sigma^\omega$. Note that $\Sigma(\overline{L}) = \Sigma(*(L)) = \Sigma(\omega(L)) = \Sigma(L)$.

**Transition Systems.** A labeled transition system (LTS) is a 4-tuple $M = (S, \Sigma, \mathit{Init}, \delta)$, where $S$ is a finite set of states, $\Sigma$ is an alphabet, $\mathit{Init} \subseteq S$ is the set of initial states, and $\delta \subseteq S \times \Sigma \times S$ is a transition relation. We write $s \xrightarrow{\alpha} s'$ for $(s, \alpha, s') \in \delta$, and $\Sigma(M)$ for $\Sigma$. $M$ is deterministic if $|\mathit{Init}| \leq 1$, and $\forall s \in S \centerdot \forall \alpha \in \Sigma \centerdot |\{s' \mid s \xrightarrow{\alpha} s'\}| \leq 1$. A run $r$ over a word $w = \alpha_0, \alpha_1, \ldots, \in \Sigma(M)^\infty$ is a sequence of states $s_0, s_1, \ldots$, such that $\forall i \geq 0 \centerdot s_i \xrightarrow{\alpha_i} s_{i+1}$. We write $\mathit{First}(r)$, $\mathit{Last}(r)$, and $\mathit{Inf}(r)$ to denote the first state of $r$, the last state of $r$ (assuming $r \in S^*$), and states that occur infinitely often in $r$ (assuming $r \in S^\omega$), respectively. We write $\mathit{Run}(w, M)$ for the set of runs of $w$ on $M$.

**Automata.** A Finite Automaton (FA) is a 5-tuple $A = (S, \Sigma, \mathit{Init}, \delta, F)$, where $(S, \Sigma, \mathit{Init}, \delta)$ is an LTS and $F \subseteq S$ is a set of accepting states. The language accepted by $A$, $\mathscr{L}(A)$, is the set of all words $w \in \Sigma^*$ s.t. there exists a run $r$ of $w$ on $A$, with $\mathit{First}(r) \in \mathit{Init} \wedge \mathit{Last}(r) \in F$. A BüchiAutomaton (BA) is a 5-tuple $B = (S, \Sigma, \mathit{Init}, \delta, F)$, where $(S, \Sigma, \mathit{Init}, \delta)$ is an LTS and $F \subseteq S$ is a set of accepting states. The language accepted by $B$, $\mathscr{L}(B)$, is the set of all words $w \in \Sigma^\omega$ s.t. there exists a run $r$ of $w$ on $A$ with $\mathit{First}(r) \in \mathit{Init} \wedge \mathit{Inf}(r) \cap F \neq \emptyset$. A BA or FA is deterministic if its underlying LTS is deterministic.

**Regularity.** A language is regular ($\omega$-regular) iff it is accepted by a FA (BA). A language $L \subseteq \Sigma^\infty$ is $\infty$-regular iff $*(L)$ is regular and $\omega(L)$ is $\omega$-regular. Deterministic FA (DFA) and non-deterministic FA (NFA) are equally expressive. Deterministic BA are strictly less expressive than non-deterministic BA.

**Learning.** A learning algorithm for a regular language is any algorithm that learns an unknown, but fixed, language $U$ over a known alphabet $\Sigma$. Such an algorithm is called *active* if it works by querying a Minimally Adequate Teacher (MAT). The MAT can answer "Yes/No" to two types of queries about $U$:

**Membership Query** Given a word $w$, is $w \in U$?

**Candidate Query** Given an automaton $B$, is $\mathscr{L}(B) = U$? If the answer is "No", the MAT returns a counterexample ($CE$), which is a word such that $CE \in \mathscr{L}(B) \ominus U$, where $X \ominus Y = (X \setminus Y) \cup (Y \setminus X)$.

An active learning algorithm begins by asking membership queries of the MAT until it constructs a candidate, with which it make a candidate query. If the candidate query is successful, the algorithm terminates; otherwise it uses the $CE$ returned by the MAT to construct additional membership queries. The family of active learning algorithms was originated by Angluin via $\mathbf{L}^*$ [2] for learning a minimal DFA that accepts an unknown regular language. $\mathbf{L}^*$ was further optimized by Rivest and Schapire [20].

The problem of learning a *minimal* automaton which accept an unknown $\omega$-regular language is still open. It is known [17] that for any language $U$ one can learn in the limit an automaton that accepts $U$ via the *identification by enumeration* approach proposed by Gold [12]. However, the automaton learned via enumeration may, in the worst case, be exponentially larger than the minimal automaton accepting $U$. Furthermore, there may be multiple minimal automata [17] accepting $U$. Maler et al. [17] have shown that $\mathbf{L}^*$ can be extended to learn a minimal (Müller) automaton for a fragment of $\omega$-regular languages.

Farzan et al. [10] show how to learn a Büchi automaton for an $\omega$-regular language $U$. Specifically, they use $\mathbf{L}^*$ to learn the language $U_{\$} = \{u\$v \mid u \cdot v^{\omega} \in U\}$, where \$ is a fresh letter not in the alphabet of $U$. The language $U_{\$}$ was shown to be regular by Calbrix et al. [4]. In the sequel, we refer to this algorithm as $\mathbf{L}^{\$}$. The complexity of $\mathbf{L}^{\$}$ is exponential in the minimal BA for $U$. Our LAG framework can use any active algorithm for learning $\omega$-regular languages. In particular, $\mathbf{L}^{\$}$ is an existing candidate.

# 3 Model of Concurrency

Let $w$ be a word and $\Sigma$ an arbitrary alphabet. We write $w \downharpoonright \Sigma$ for the projection of $w$ onto $\Sigma$ defined recursively as follows (recall that $\lambda$ denotes the empty word):

$$\lambda \downharpoonright \Sigma = \lambda \qquad\qquad (a \cdot u) \downharpoonright \Sigma = \begin{cases} a \cdot (u \downharpoonright \Sigma) & \text{if } a \in \Sigma \\ u \downharpoonright \Sigma & \text{otherwise} \end{cases}$$

Clearly, both $\Sigma^*$ and $\Sigma^\infty$ are closed under projection, but $\Sigma^\omega$ is not. For example, $(a^* \cdot b^\omega \downharpoonright \{a\}) = a^*$, and $a^*$ consists only of finite words. Projection preservers regularity. If $L$ is a regular ($\infty$-regular) language and $\Sigma$ is any alphabet, then $L \downharpoonright \Sigma$ is also regular ($\infty$-regular).

A process is modeled by a language of all of its behaviors (or computations). *Parallel composition* ($||$) of two processes/languages synchronizes on common actions while executing local actions asynchronously. For languages $(L_1, \Sigma_1)$ and $(L_2, \Sigma_2)$, $L_1 || L_2$ is the language over $\Sigma_1 \cup \Sigma_2$ defined as follows:

$$L_1 \mid\mid L_2 = \{w \in (\Sigma_1 \cup \Sigma_2)^\infty \mid w \downharpoonright \Sigma_1 \in L_1 \wedge w \downharpoonright \Sigma_2 \in L_2\} \qquad \text{(def. of } ||\text{)}$$

Intuitively, $L_1 || L_2$ consists of all permutations of words from $L_1$ and $L_2$ that have a common synchronization sequence. For example, $(b^* \cdot a \cdot b^*) || (c^* \cdot a \cdot c^*)$ is $(b+c)^* \cdot a \cdot (b+c)^*$. Note that when $L_1$ and $L_2$ share an alphabet, the composition is their intersection; when their alphabets are disjoint, the composition is their language shuffle. The set of $\Sigma^*$, $\Sigma^\omega$, and $\Sigma^\infty$ languages are all closed under parallel composition.

**Theorem 1.** *The $||$ operator is associative, commutative, distributive over union and intersection. It is also monotone, i.e., for any two languages $L_1$, $L_2$, and $L_3$: $L_2 \subseteq L_3 \Rightarrow (L_1 || L_2) \subseteq (L_1 || L_3)$.*

Let $L_1$ and $L_2$ be two languages such that $\Sigma(L_1) \supseteq \Sigma(L_2)$. We say that $L_1$ is subsumed by $L_2$, written $L_1 \preccurlyeq L_2$, if $L_1 \downharpoonright \Sigma(L_2) \subseteq L_2$. Let $L_S$ be the language of a specification $S$, and $L_M$ be the language of a system $M$. Then, $M$ satisfies $S$, written $M \models S$, iff $L_M \preccurlyeq L_S$.

# 4 Proof Rules for Assume-Guarantee Reasoning

In this section, we study the applicability of a non-circular and a circular AG rule to proving properties of processes with infinite behaviors (e.g., reactive systems that neither terminate nor deadlock). These rules were shown to be sound and complete for systems with finite (i.e., in $\Sigma^*$) behaviors by Barringer et al. [3]. In Section 4.1, we show that the non-circular AG rule is sound for both $\Sigma^\infty$ and $\Sigma^\omega$ behaviors. However, it is complete only when the assumptions are allowed to combine *both* finite and infinite behaviors (i.e., in $\Sigma^\infty$). In Section 4.2, we show that the circular AG rule is sound and complete for $\Sigma^\omega$ and $\Sigma^\infty$ behaviors.

## 4.1 Non-Circular Assume-Guarantee Rule

The non-circular AG proof rule (**AG-NC** for short) is stated as follows:

$$\frac{(L_1 \parallel L_A) \preccurlyeq L_S \qquad L_2 \preccurlyeq L_A}{(L_1 \parallel L_2) \preccurlyeq L_S}$$

where $L_1$, $L_2$, $L_S$, and $L_A$ are languages with the alphabets $\Sigma_1$, $\Sigma_2$, $\Sigma_S$, $\Sigma_A$, respectively, $\Sigma_S \subseteq (\Sigma_1 \cup \Sigma_2)$, and $\Sigma_A = (\Sigma_1 \cup \Sigma_S) \cap \Sigma_2$. **AG-NC** is known to be sound and complete for $\Sigma^*$-languages. Intuitively, it says that if there exists an assumption $L_A$ such that: (a) $L_1$ composed with $L_A$ is contained in $L_S$, and (b) $L_2$ is contained in $L_A$, then the composition of $L_1$ with $L_2$ is contained in $L_S$ as well. Note that the alphabet $\Sigma_A$ is the smallest alphabet containing: (a) actions at the interface between $L_1$ and $L_2$, i.e., actions common to the alphabets of $L_1$ and $L_2$, and (b) external actions of $L_2$, i.e., actions common to the alphabets of $L_2$ and $L_S$. Any smaller alphabet makes the rule trivially incomplete; any larger alphabet exposes internal (i.e., non-external) actions of $L_2$. It is not surprising that **AG-NC** remains sound even when applied to languages with infinite words. However, **AG-NC** is *incomplete* when $L_A$ is restricted to $\Sigma^\omega$-languages:

**Theorem 2.** *There exists $L_1, L_2, L_S \subseteq \Sigma^\omega$ such that $(L_1 || L_2) \preccurlyeq L_S$, but there does not exists an assumption $L_A \subseteq \Sigma^\omega$ that satisfies all of the premises of* **AG-NC**.

*Proof.* By example. Let $L_1$, $L_2$, $L_S$, and their alphabets be defined as follows:

$$\Sigma_1 = \{a,b\} \qquad \Sigma_2 = \{a,c\} \qquad \Sigma_S = \{a,b\} \qquad L_1 = (a+b)^\omega \qquad L_2 = a^* c^\omega \qquad L_S = (a+b)^* b^\omega$$

The conclusion of **AG-NC** rule is satisfied since $(L_1 || L_2) \downharpoonright \Sigma_S = (a+b)^* b^\omega = L_S$. The alphabet $\Sigma_A$ of $L_A$ is $(\Sigma_1 \cup \Sigma_S) \cap \Sigma_2 = \{a\}$. Since $L_A \subseteq \Sigma_A^\omega$, it can only be $a^\omega$ or $\emptyset$. The only way to satisfy the first premise of **AG-NC** is to let $L_A = \emptyset$, but this is too strong to satisfy the second premise. □

Note that the proof of Theorem 2 shows that **AG-NC** is incomplete even for $\infty$-regular languages.

**Remark 1.** *One may conjecture that the* **AG-NC** *rule becomes complete for $\Sigma^\omega$ if subsumption is redefined to only consider infinite words. That is, by redefining subsumption as: $L_1 \preccurlyeq L_2 \Leftrightarrow \omega(L_1 \downharpoonright \Sigma(L_2)) \subseteq L_2$. However, under this interpretation,* **AG-NC** *is no longer sound. For example, let the languages $L_1$, $L_2$, $L_S$, and their alphabets be defined as follows:*

$$\Sigma_1 = \{a,b\} \qquad \Sigma_2 = \{a,c\} \qquad \Sigma_S = \{a,b\} \qquad L_1 = (a+b)^\omega \qquad L_2 = a^* c^\omega \qquad L_S = b^\omega$$

*Then, the conclusion of* **AG-NC** *does not hold: $\omega((L_1 || L_2) \downharpoonright \Sigma_S) = (a+b)^* b^\omega \nsubseteq b^\omega$. But $L_A = \emptyset$ satisfies both premises: $(L_1 || L_A) = b^\omega$, and $\omega(L_2 \downharpoonright \{a\}) = L_A$.*

**Remark 2.** **AG-NC** *is complete if the alphabet $\Sigma_A$ is redefined to be $\Sigma_1 \cup \Sigma_2$. However, in this case the rule is no longer "compositional" since the assumption $L_A$ can be as expressive as the component $L_2$.*

Intuitively, **AG-NC** is incomplete for $\Sigma^\omega$ because $\Sigma^\omega$ is not closed under projection. However, we show that the rule is complete for $\Sigma^\infty$ – the smallest projection-closed extension of $\Sigma^\omega$. We first show that for any languages $L_1$ and $L_S$, there always exists a unique weakest assumption $L_A$, such that $L_1 || L_A \preccurlyeq L_S$.

**Theorem 3.** *Let $L_1$ and $L_S$ be two languages, and $\Sigma_A$ be any alphabet s.t. $\Sigma(L_1) \cup \Sigma_A = \Sigma(L_1) \cup \Sigma(L_S)$. Then, $L_A = \{w \in \Sigma_A^\infty \mid (L_1 || \{w\}) \preccurlyeq L_S\}$ satisfies $L_1 || L_A \preccurlyeq L_S$, and is the weakest such assumption.*

*Proof.* Let us write $\Sigma_1$, $\Sigma_S$ and $\Sigma_{1S}$ to mean $\Sigma(L_1)$, $\Sigma(L_S)$ and $\Sigma(L_1) \cup \Sigma(L_S)$ respectively. To show that $L_A$ is a valid assumption, pick any $w \in L_1 \parallel L_A$. Then $w \downharpoonright \Sigma_A \in L_A$. This implies that $w \downharpoonright \Sigma_S \in (L_1 \parallel \{w \downharpoonright \Sigma_A\}) \downharpoonright \Sigma_S \subseteq L_S$. Since $w$ is any word in $L_1 \parallel L_A$, we have $L_1 \parallel L_A \preccurlyeq L_S$. To show that $L_A$ is the weakest assumption, let $L'_A \subseteq \Sigma_A^\infty$ be any language such that $L_1 \parallel L'_A \preccurlyeq L_S$ and let $w$ be any word in $L'_A$. Then, $(L_1 \parallel \{w\}) \subseteq (L_1 \parallel L'_A) \preccurlyeq L_S$. But this implies that $w \in L_A$, and, therefore, $L'_A \subseteq L_A$. □

Note that $\Sigma_A^\infty$ subsumes both finite ($\Sigma_A^*$) and infinite ($\Sigma_A^\omega$) words. Thus, if $L_A$ is a $\Sigma_A^\infty$ weakest assumption, then $*(L_A)$ and $\omega(L_A)$ are the weakest $\Sigma_A^*$ and $\Sigma_A^\omega$ assumptions, respectively.

**Theorem 4.** *Let $L_1$, $L_2$, $L_S$, and $L_A$ be in $\Sigma^\infty$. Then, the* **AG-NC** *rule is sound and complete.*

*Proof.* The proof of soundness is trivial and is omitted. For the proof of completeness we only show the key step. Assume that $L_1 || L_2 \preccurlyeq L_S$, and let $L_A$ be the weakest assumption such that $L_1 || L_A \preccurlyeq L_S$. By Theorem 3, $L_A$ is well-defined and satisfies the first premise of **AG-NC**. The second premise holds because $L_2 \downharpoonright \Sigma_A \subseteq \Sigma_A^\infty$, and $L_A$ is the weakest $\Sigma_A^\infty$ assumption (see Theorem 3). □

Theorem 4 implies that **AG-NC** is sound for any fragment of $\Sigma^\infty$. Of course, this is not true for completeness of the rule. For practical purposes, we would like to know that the rule remains complete when its languages are restricted to the regular subset. We show that this is so by showing that under the assumption that $L_1$ and $L_S$ are regular, the weakest assumption is regular as well.

**Theorem 5.** *Let $L_1$ and $L_S$ be two languages, and $\Sigma_A$ be any alphabet such that $\Sigma(L_1) \cup \Sigma_A = \Sigma(L_1) \cup \Sigma(L_S)$. Then, $L_A \subseteq \Sigma_A^\infty$ is the weakest assumption such that $L_1 || L_A \preccurlyeq L_S$ iff $L_A = \overline{(L_1 \parallel \overline{L_S}) \downharpoonright \Sigma_A}$.*

*Proof.* Let us write $\Sigma_1$, $\Sigma_S$ and $\Sigma_{1S}$ to mean $\Sigma(L_1)$, $\Sigma(L_S)$ and $\Sigma(L_1) \cup \Sigma(L_S)$, respectively. For any $w \in \Sigma_A^\infty$:

$$
\begin{aligned}
& w \in \overline{(L_1 \parallel \overline{L_S}) \downharpoonright \Sigma_A} \ \textbf{iff}\ \forall w' \in \Sigma_{1S}^\infty \centerdot \{w'\} \preccurlyeq \{w\} \implies w' \notin (L_1 \parallel \overline{L_S}) \\
\textbf{iff} \quad & \forall w' \in \Sigma_{1S}^\infty \centerdot \{w'\} \preccurlyeq \{w\} \implies (\{w'\} \not\preccurlyeq L_1 \vee \{w'\} \preccurlyeq L_S) \\
\textbf{iff} \quad & \forall w' \in \Sigma_{1S}^\infty \centerdot (\{w'\} \preccurlyeq \{w\} \wedge \{w'\} \preccurlyeq L_1) \implies \{w'\} \preccurlyeq L_S \\
\textbf{iff} \quad & \forall w' \in \Sigma_{1S}^\infty \centerdot (\{w'\} \preccurlyeq (L_1 \parallel \{w\})) \implies \{w'\} \preccurlyeq L_S \ \textbf{iff}\ L_1 \parallel \{w\} \preccurlyeq L_S
\end{aligned}
$$

Together with Theorem 3, this completes the proof. □

Theorem 5 implies **AG-NC** is complete for any class of languages closed under complementation and projection, e.g., regular and $\infty$-regular languages. In addition, Theorem 5 implies that learning-based automated AG reasoning is effective for any class of languages whose weakest assumptions fall in a "learnable" fragment. In particular, this holds for regular, $\omega$-regular and $\infty$-regular languages.

## 4.2 Circular Assume-Guarantee Rule

The Circular Assume-Guarantee proof rule (**AG-C** for short) is stated as follows:

$$
\frac{(L_1 \parallel L_{A1}) \preccurlyeq L_S \qquad (L_2 \parallel L_{A2}) \preccurlyeq L_S \qquad (\overline{L_{A1}} \parallel \overline{L_{A2}}) \preccurlyeq L_S}{(L_1 \parallel L_2) \preccurlyeq L_S}
$$

where $L_1$, $L_2$, and $L_S$ are languages over alphabets $\Sigma_1$, $\Sigma_2$, $\Sigma_S$, respectively; $\Sigma_S \subseteq \Sigma_1 \cup \Sigma_2$, and $L_{A1}$ and $L_{A2}$ share a common alphabet $\Sigma_A = (\Sigma_1 \cap \Sigma_2) \cup \Sigma_S$. **AG-C** is known to be sound and complete for $\Sigma^*$-languages. Note that in comparison with **AG-NC**, there are two assumptions $L_{A1}$ and $L_{A2}$ over a larger alphabet $\Sigma_A$. Informally, the rule is sound for the following reason. Let $w$ be a word in $L_1 || L_2$, and $u = w \downharpoonright \Sigma_A$. Then $u \in L_{A1}$, or $u \in L_{A2}$, or $u \in \overline{L_{A1} \cup L_{A2}} = (\overline{L_{A1}} || \overline{L_{A2}})$. If $u \in L_{A1}$ then the first premise implies that $\{w\} \preccurlyeq L_1 || \{u\} \preccurlyeq L_S$; if $u \in L_{A2}$ then the second premise implies that $\{w\} \preccurlyeq L_2 || \{u\} \preccurlyeq L_S$; otherwise, the third premise implies that $\{w\} \preccurlyeq \{u\} \preccurlyeq L_S$.

**Remark 3.** *Note that the assumption alphabet for* **AG-C** *is larger than* **AG-NC**. *In fact, using* $\Sigma_{A1} = (\Sigma_1 \cup \Sigma_S) \cap \Sigma_2$ *and* $\Sigma_{A2} = (\Sigma_2 \cup \Sigma_S) \cap \Sigma_1$ *makes* **AG-C** *incomplete. Indeed, let* $L_1 = \{aa\}$ *with* $\Sigma_1 = \{a\}$, $L_2 = \{bb\}$ *with* $\Sigma_2 = \{b\}$ *and* $\overline{L_S} = \{aab, abb, ab\}$. *Note that* $L_1 || L_2 \preccurlyeq L_S$. *We show that no* $L_{A_1}$ *and* $L_{A_2}$ *can satisfy the three premises of* **AG-C**. *Premise 1* $\Rightarrow b \notin L_{A_1} \Rightarrow b \in \overline{L_{A_1}}$. *Similarly, premise 2* $\Rightarrow a \notin L_{A_2} \Rightarrow a \in \overline{L_{A_2}}$. *But then* $ab \in \overline{L_{A1}} || \overline{L_{A_2}}$, *violating premise 3.*

In this section, we show that **AG-C** is sound and complete for both $\Sigma^\omega$ and $\Sigma^\infty$ languages. First, we illustrate an application of the rule to the example from the proof of Theorem 2. Let $L_1$, $L_2$, and $L_S$ be $\Sigma^\omega$ languages as defined in the proof of Theorem 2. In this case, the alphabet $\Sigma_A$ is $\{a, b\}$. Letting $L_{A1} = (a+b)^* b^\omega$, and $L_{A2} = (a+b)^\omega$ satisfies all three premises of the rule.

**Theorem 6.** *Let* $L_1$, $L_2$, $L_S$, $L_{A1}$, *and* $L_{A2}$ *be in* $\Sigma^\omega$ *or* $\Sigma^\infty$. *Then, the* **AG-C** *rule is sound and complete.*

*Proof.* The proof of soundness is sketched in the above discussion. For the proof of completeness we only show the key steps. Assume that $L_1 || L_2 \preccurlyeq L_S$. Let $L_{A1}$ and $L_{A2}$ be the weakest assumptions such that $L_1 || L_{A1} \preccurlyeq L_S$, and $L_2 || L_{A2} \preccurlyeq L_S$, respectively. By Theorem 3, both $L_{A1}$ and $L_{A2}$ are well-defined and satisfy the first and the second premises of **AG-C**, respectively. We prove the third premise by contradiction. Since $L_{A1}$ and $L_{A2}$ have the same alphabet, $(\overline{L_{A1}} || \overline{L_{A2}}) = (\overline{L_{A1}} \cap \overline{L_{A2}})$. Assume that $(\overline{L_{A1}} \cap \overline{L_{A2}}) \not\preccurlyeq L_S$. Then, there exists a word $w \in (\overline{L_{A1}} || \overline{L_{A2}})$ such that $w \notin L_{A1}$, and $w \notin L_{A2}$, and $w \downharpoonright \Sigma_S \notin L_S$. By the definition of weakest assumption (see Theorem 3), $L_1 || \{w\} \not\preccurlyeq L_S$ and $L_2 || \{w\} \not\preccurlyeq L_S$. Pick any $w_1 \in L_1 || \{w\}$ and $w_2 \in L_2 || \{w\}$. Let $w'_1 = w_1 \downharpoonright \Sigma_1$ and $w'_2 = w_2 \downharpoonright \Sigma_2$. We know that $\{w'_1\} || \{w'_2\} \subseteq L_1 || L_2$. Also, $w \in (\{w'_1\} || \{w'_2\}) \downharpoonright \Sigma_A$. Now since $\{w'_1\} || \{w'_2\} \subseteq L_1 || L_2$, we have $w \in (L_1 || L_2) \downharpoonright \Sigma_A$. Since $\Sigma_S \subseteq \Sigma_A$, $w \downharpoonright \Sigma_S \in (L_1 || L_2) \downharpoonright \Sigma_S$. But $w \downharpoonright \Sigma_S \notin L_S$, which contradicts $L_1 || L_2 \preccurlyeq L_S$. □

The completeness part of the proof of Theorem 6 is based on the existence of the weakest assumption. We already know from Theorem 5, that the weakest assumption is ($\infty$-,$\omega$-)regular if $L_1$, $L_2$, and $L_S$ are ($\infty$-,$\omega$-)regular, respectively. Thus, **AG-C** is complete for ($\infty$-,$\omega$-)regular languages. Since **AG-NC** is incomplete for $\omega$-regular languages, a learning algorithm for $\omega$-regular languages (such as $\mathbf{L}^\$$) cannot be applied directly for AG reasoning for $\omega$-regular systems and specifications. In the next section, we overcome this challenge by developing automated AG algorithms for $\infty$-regular and $\omega$-regular languages.

# 5 Automated Assume-Guarantee Reasoning

In this section, we present our LAG framework, and its specific useful instances. LAG uses membership oracles, learners, and checkers, which we describe first.

**Definition 1** (Membership Oracle and Learner)**.** *A membership oracle* $Q$ *for a language* $U$ *over alphabet* $\Sigma$ *is a procedure that takes as input a word* $u \in \Sigma^\infty$ *and returns* 0 *or* 1 *such that* $Q(u) = 1 \iff u \in U$. *We say that* $Q \models U$. *The set of all membership oracles is denoted by* **Oracle**. *Let* $\mathscr{A}$ *be any set of automata. We write* $\mathbf{Learner}_{\mathscr{A}}$ *to denote the set of all learners of type* $\mathscr{A}$. *Formally, a learner of type* $\mathscr{A}$ *is a pair* $(\mathbf{Cand}, \mathbf{LearnCE})$ *such that: (i)* $\mathbf{Cand} : \mathbf{Oracle} \mapsto \mathscr{A}$ *is a procedure that takes a membership oracle as input and outputs a candidate* $C \in \mathscr{A}$, *and (ii)* $\mathbf{LearnCE} : \Sigma^\infty \mapsto \mathbf{Learner}_{\mathscr{A}}$ *is a procedure that takes a counterexample as input and returns a new learner of type* $\mathscr{A}$. *For any learner* $P = (\mathbf{Cand}, \mathbf{LearnCE})$ *we write* $P.\mathbf{Cand}$ *and* $P.\mathbf{LearnCE}$ *to mean* **Cand** *and* **LearnCE** *respectively.*

Intuitively, a membership oracle is the fragment of a MAT that only answers membership queries, while a learner encapsulates an active learning algorithm that is able to construct candidates via membership queries, and learn from counterexamples of candidate queries.

**Learning.** Let $U$ be any unknown language, $Q$ be an oracle, and $P$ be a learner. We say that $(P, Q)$ learns $U$ if the following holds: if $Q \models U$, then there does not exist an infinite sequence of learners $P_0, P_1, \ldots$ and an infinite sequence of counterexamples $CE_1, CE_2, \ldots$ such that: (i) $P_0 = P$, (ii) $P_i = P_{i-1}.\mathbf{LearnCE}(CE_i)$ for $i > 0$, and (iii) $CE_i \in \mathscr{L}(P_{i-1}.\mathbf{Cand}(Q)) \ominus U$ for $i > 0$.

**Input:** $P_1 \dots P_k$ : **Learner**$_{\mathscr{A}}$; $Q_1, \dots, Q_k$ : **Oracle**; $V$ : **Checker**$_{(\mathscr{A},k)}$
**forever do**
  **for** $i = 1$ to $k$ **do** $C_i := P_i$.**Cand**$(Q_i)$
  $R := V(C_1, \dots, C_k)$
  **if** $(R = ($**FEEDBACK**$, i, CE))$ **then** $P_i := P_i$.**LearnCE**$(CE)$ **else return** $R$

Figure 1: Algorithm for overall LAG procedure.

**Definition 2** (Checker)**.** *Let $\mathscr{A}$ be a set of automata, and $k$ be an integer denoting the number of candidates. A checker of type $(\mathscr{A}, k)$ is a procedure that takes as input $k$ elements $A_1, \dots, A_k$ of $\mathscr{A}$ and returns either (i)* **SUCCESS***, or (ii) a pair* $($**FAILURE**$, CE)$ *such that $CE \in \Sigma^\infty$, or (iii) a triple* $($**FEEDBACK**$, i, CE)$ *such that $1 \leq i \leq k$ and $CE \in \Sigma^\infty$. We write* **Checker**$_{(\mathscr{A},k)}$ *to mean the set of all checkers of type $(\mathscr{A}, k)$.*

Intuitively, a checker generalizes the fragment of a MAT that responds to candidate queries by handling multiple (specifically, $k$) candidates. This generalization is important for circular proof rules. The checker has three possible outputs: (i) **SUCCESS** if the overall verification succeeds; (ii) $($**FAILURE**$, CE)$ where $CE$ is a real counterexample; (iii) $($**FEEDBACK**$, i, CE)$ where $CE$ is a counterexample for the $i$-th candidate.

## 5.1 LAG Procedure

Our overall LAG procedure is presented in Fig. 1. We write $X : T$ to mean that “$X$ is of type $T$”. LAG accepts a set of $k$ membership oracles, $k$ learners, and a checker, and repeats the following steps:

1. Constructs candidate automata $C_1, \dots, C_k$ using the learners and oracles.
2. Invokes the checker with the candidates constructed in Step 1 above.
3. If the checker returns **SUCCESS** or $($**FAILURE**$, CE)$, then exits with this result. Otherwise, updates the appropriate learner with the feedback and repeats from Step 1.

**Theorem 7.** *LAG terminates if there exists languages $U_1, \dots, U_k$ such that: (i) $Q_i \models U_i$ for $1 \leq i \leq k$, (ii) $(P_i, Q_i)$ learns $U_i$ for $1 \leq i \leq k$, and (iii) if $V(C_1, \dots, C_k) =$* $($**FEEDBACK**$, i, CE)$*, then $CE \in \mathscr{L}(C_i) \ominus U_i$.*

*Proof.* By contradiction. If LAG does not terminate there exists some $P_i$ such that $P_i$.**LearnCE** is called infinitely often. This, together with assumptions (i) and (iii), contradicts (ii), i.e., $(P_i, Q_i)$ learns $U_i$. □

## 5.2 Oracle, Learner, and Checker Instantiations

We now describe various implementations of oracles, learners and checkers. We start with the notion of an oracle for weakest assumptions.

**Oracle for Weakest Assumption.** Let $L_1$, $L_S$ be any languages and $\Sigma$ be any alphabet. We write $Q(L_1, L_S, \Sigma)$ to denote the oracle such that $Q(L_1, L_S, \Sigma) \models \overline{(L_1 \parallel \overline{L_S})} \downharpoonright \Sigma$. $Q(L_1, L_S, \Sigma)$ is typically implemented via model checking since, by Theorems 3 and 5, $Q(L_1, L_S, \Sigma)(u) = 1 \iff u \in \Sigma^\infty \wedge L_1 \parallel \{u\} \preccurlyeq L_S$.

**Learner Instantiations.** In general, a learner $P(\mathbf{L})$ is derived from an active learning algorithm $\mathbf{L}$ as follows: $P(\mathbf{L}) = ($**Cand**$,$ **LearnCE**$)$ s.t. **Cand** $=$ part of $\mathbf{L}$ that constructs a candidate using membership queries, and **LearnCE** $=$ part of $\mathbf{L}$ that learns from a counterexample to a candidate query.

**Non-circular Checker.** Let $\mathscr{A}$ be a type of automata, and $L_1$, $L_2$ and $L_S$ be any languages. Then $V_{NC}(L_1, L_2, L_S)$ is the checker of type $(\mathscr{A}, 1)$ defined in Fig. 2. Note that $V_{NC}(L_1, L_2, L_S)$ is based on the **AG-NC** proof rule. The following proposition about $V_{NC}(L_1, L_2, L_S)$ will be used later.

**Proposition 1.** *If $V_{NC}(L_1, L_2, L_S)(A)$ returns* **SUCCESS***, then $L_1 \parallel L_2 \preccurlyeq L_S$. Otherwise, if $V_{NC}(L_1, L_2, L_S)(A)$ returns* $($**FAILURE**$, CE)$*, then $CE$ is a valid counterexample to $L_1 \parallel L_2 \preccurlyeq L_S$. Finally, if $V_{NC}(L_1, L_2, L_S)(A)$ returns* $($**FEEDBACK**$, 1, CE)$*, then $CE \in \mathscr{L}(A) \ominus \overline{(L_1 \parallel \overline{L_S})} \downharpoonright \Sigma$.*

**Checker:** $V_{NC}(L_1, L_2, L_S)$

**Input:** $A$: $\mathscr{A}$
**if** $(L_1 \parallel \mathscr{L}(A)) \preccurlyeq L_S$ **then**
  **if** $L_2 \preccurlyeq \mathscr{L}(A)$ **then return SUCCESS**
  **else**
    **let** $w$ be a CEX to $L_2 \preccurlyeq \mathscr{L}(A)$
    **if** $L_1 \parallel \{w\} \preccurlyeq L_S$ **then**
      **return** $(\textbf{FEEDBACK}, 1, w \downharpoonright \Sigma(A))$
    **else**
      **let** $w'$ be a CEX to $L_1 \parallel \{w\} \preccurlyeq L_S$
      **return** $(\textbf{FAILURE}, w')$
**else**
  **let** $w$ be a CEX to $(L_1 \parallel \mathscr{L}(A)) \preccurlyeq L_S$
  **return** $(\textbf{FEEDBACK}, 1, w \downharpoonright \Sigma(A))$

**Checker:** $V_C(L_1, L_2, L_S)$

**Input:** $A_1, A_2 : \mathscr{A}$
**for** $i = 1, 2$ **do**
  **if** $L_i \parallel \mathscr{L}(A_i) \not\preccurlyeq L_S$ **then**
    **let** $w$ be a CEX to $L_i \parallel \mathscr{L}(A_i) \preccurlyeq L_S$
    **return** $(\textbf{FEEDBACK}, i, w \downharpoonright \Sigma_A)$
**if** $\overline{\mathscr{L}(A_1)} \parallel \overline{\mathscr{L}(A_2)} \preccurlyeq L_S$ **then return SUCCESS**
**else**
  **let** $w$ be a CEX to $\overline{\mathscr{L}(A_1)} \parallel \overline{\mathscr{L}(A_2)} \preccurlyeq L_S$
  **for** $i = 1, 2$ **do**
    **if** $L_i \parallel \{w\} \preccurlyeq L_S$ **then**
      **return** $(\textbf{FEEDBACK}, i, w \downharpoonright \Sigma_A)$
    **else let** $w_i$ be a CEX to $L_i \parallel \{w\} \preccurlyeq L_S$
  **pick** $w' \in \{w_1\} \parallel \{w_2\}$
  **return** $(\textbf{FAILURE}, w')$

Figure 2: $V_{NC}$ – a checker based on **AG-NC**; $V_C$ – a checker based on **AG-C**.

**Circular Checker.** Let $\mathscr{A}$ be a type of automata, and $L_1$, $L_2$ and $L_S$ be any languages. Then $V_C(L_1, L_2, L_S)$ is the checker of type $(\mathscr{A}, 2)$ defined in Fig. 2. Note that $V_C(L_1, L_2, L_S)$ is based on the **AG-C** proof rule. The following proposition about $V_C(L_1, L_2, L_S)$ will be used later.

**Proposition 2.** *If $V_C(L_1, L_2, L_S)(A_1, A_2)$ returns* **SUCCESS***, then $L_1 \parallel L_2 \preccurlyeq L_S$. Otherwise, if $V_C(L_1, L_2, L_S)(A_1, A_2)$ returns $(\textbf{FAILURE}, CE)$, then $CE$ is a valid counterexample to $L_1 \parallel L_2 \preccurlyeq L_S$. Finally, if $V_C(L_1, L_2, L_S)(A_1, A_2)$ returns $(\textbf{FEEDBACK}, i, CE)$, then $CE \in \mathscr{L}(A_i) \ominus \overline{(L_i \parallel \overline{L_S})} \downharpoonright \Sigma$.*

### 5.3 LAG Instantiations

In this section, we present several instantiations of LAG for checking $L_1 \parallel L_2 \preccurlyeq L_S$. Our approach extends to systems with finitely many components, as for example in [9, 3].

**Existing Work as LAG Instances: Regular Trace Containment.** Table 1 instantiates LAG for existing learning-based algorithms for AG reasoning. The first row corresponds to the work of Cobleigh et al. [9]; its termination and correctness follow from Theorem 7, Proposition 1, and the fact that $(P_1, Q_1)$ learns the language $\overline{(L_1 \parallel \overline{L_S})} \downharpoonright \Sigma$. The second row corresponds to Barringer et al. [3]; its termination and correctness follow from Theorem 7, Proposition 2, and the fact that $(P_i, Q_i)$ learns $\overline{(L_i \parallel \overline{L_S})} \downharpoonright \Sigma$ for $i \in \{1, 2\}$.

**New Contribution: Learning Infinite Behavior.** Let $\mathbf{L}^\omega$ be any active learning algorithm for $\omega$-regular languages (e.g., $\mathbf{L}^\$$). Since **AG-NC** is incomplete for $\omega$-regular languages, $\mathbf{L}^\omega$ is not applicable directly in this context. On the other hand, both **AG-NC** and **AG-C** are sound and complete for $\infty$-regular languages. Therefore, a learning algorithm for $\infty$-regular languages yields LAG instances for systems with infinite behavior. We now present two such algorithms. The first (see Theorem 8 (a)) uses $\mathbf{L}^\omega$ only, but augments the assumption alphabet. The second (see Theorem 8(b)) combines $\mathbf{L}^\omega$ and $\mathbf{L}^*$, but leaves the assumption alphabet unchanged. We present both schemes since neither is objectively superior.

**Theorem 8.** *We can learn a $\infty$-regular language $U$ using a MAT for $U$ in two ways: (a) using only $\mathbf{L}^\omega$ but with alphabet augmentation, and (b) without alphabet augmentation, but using both $\mathbf{L}^*$and $\mathbf{L}^\omega$.*

*Proof.* Part(a): Let $\Sigma$ be the alphabet of $U$. We use $\mathbf{L}^\omega$ to learn an $\omega$-regular language $U'$ over the alphabet $\Sigma' = \Sigma \cup \{\tau\}$ such that $U' \downharpoonright \Sigma = U$, and $\tau \notin \Sigma$. Let $U' = U \cdot \tau^\omega$. We assume that the MAT $X$ for $U$ accepts membership queries of the form $(M_1, M_2) \in$ DFA $\times$ BA, and returns "Yes" if $U = \mathscr{L}(M_1) \cup \mathscr{L}(M_2)$, and a $CE$ otherwise. Then, a MAT for $U'$ is implemented using $X$ as follows: (i) **Membership**: $u \in U'$ iff $u \in \Sigma^\infty \cdot \tau^\omega \wedge u \downharpoonright \Sigma \in U$, where $u \downharpoonright \Sigma \in U$ is decided using $X$; (ii) **Candidate**

| Conformance | Rule | $\mathscr{A}$ | Learner(s) | Oracle(s) | Checker |
|---|---|---|---|---|---|
| Regular Trace Containment | **AG-NC** [9] | DFA | $P_1 = P(\mathbf{L}^*)$ | $Q_1 = Q(L_1, L_S, \Sigma_{NC})$ | $V_{NC}(L_1, L_2, L_S)$ |
| Regular Trace Containment | **AG-C** [3] | DFA | $P_1 = P_2 = P(\mathbf{L}^*)$ | $Q_1 = Q(L_1, L_S, \Sigma_C)$ $Q_2 = Q(L_2, L_S, \Sigma_C)$ | $V_C(L_1, L_2, L_S)$ |
| $\infty$-regular Trace Containment | **AG-NC** | DFA $\times$ BA | $P_1 = P(\mathbf{L})$ | $Q_1 = Q(L_1, L_S, \Sigma_{NC})$ | $V_{NC}(L_1, L_2, L_S)$ |
| $\infty$-regular Trace Containment | **AG-C** | DFA $\times$ BA | $P_1 = P_2 = P(\mathbf{L})$ | $Q_1 = Q(L_1, L_S, \Sigma_C)$ $Q_2 = Q(L_2, L_S, \Sigma_C)$ | $V_C(L_1, L_2, L_S)$ |
| $\omega$-regular Trace Containment | **AG-NC** | DFA $\times$ BA | $P_1 = P(\mathbf{L})$ | $Q_1 = Q(L_1, L_S, \Sigma_{NC})$ | $V_{NC}(L_1, L_2, L_S)$ |
| $\omega$-regular Trace Containment | **AG-C** | BA | $P_1 = P_2 = P(\mathbf{L}^\omega)$ | $Q_1 = Q(L_1, L_S, \Sigma_C)$ $Q_2 = Q(L_2, L_S, \Sigma_C)$ | $V_C(L_1, L_2, L_S)$ |

Table 1: Existing learning-based AG algorithms as instances of LAG; $\Sigma_{NC} = (\Sigma(L_1) \cup \Sigma(L_S)) \cap \Sigma(L_2)$; $\Sigma_C = (\Sigma(L_1) \cap \Sigma(L_2)) \cup \Sigma(L_S)$; $\mathbf{L}$ is a learning algorithm from Theorem 8.

with $C'$: If $\mathscr{L}(C') \not\subseteq \Sigma^\infty \cdot \tau^\omega$, return $CE' \in \mathscr{L}(C') \setminus \Sigma^\infty \cdot \tau^\omega$. Otherwise, make a candidate query to $X$ with $(M_1, M_2)$ such that $\mathscr{L}(M_1) = *(C' \downharpoonright \Sigma)$ and $\mathscr{L}(M_2) = \omega(C' \downharpoonright \Sigma)$, and turn any $CE$ to $CE' = CE \cdot \tau^\omega$.

Part(b): We use $\mathbf{L}^*$ to learn $*(U)$ and $\mathbf{L}^\omega$ to learn $\omega(U)$. We assume that the MAT $X$ for $U$ accepts membership queries of the form $(M_1, M_2) \in$ DFA $\times$ BA, and returns "Yes" if $U = \mathscr{L}(M_1) \cup \mathscr{L}(M_2)$, and a $CE$ otherwise. We run $\mathbf{L}^*$ and $\mathbf{L}^\omega$ concurrently, and iterate the two next steps: (1) answer membership queries with $X$ until we get candidates $M_1$ and $M_2$ from $\mathbf{L}^*$ and $\mathbf{L}^\omega$ respectively; (2) make candidate query $(M_1, M_2)$ to $X$; return any finite (infinite) $CE$ back to $\mathbf{L}^*$ ($\mathbf{L}^\omega$); repeat from Step 1. □

**LAG instances for $\infty$-regular Trace Containment.** Suppose that $L_1, L_2$ and $L_S$ are $\infty$-regular and we wish to verify $L_1 \parallel L_2 \preccurlyeq L_S$. The third row of Table 1 show how to instantiate LAG to solve this problem using **AG-NC**. This instance of LAG terminates with the correct result due to Theorem 7, Proposition 1, and the fact that $(P_1, Q_1)$ learns $\overline{(L_1 \parallel \overline{L_S})} \downharpoonright \Sigma$. The fourth row of Table 1 show how to instantiate LAG to solve this problem using **AG-C**. This instance of LAG terminates correctly due to Theorem 7, Proposition 2, and because $(P_i, Q_i)$ learns $\overline{(L_i \parallel \overline{L_S})} \downharpoonright \Sigma$ for $i \in \{1, 2\}$.

**LAG instances for $\omega$-regular Trace Containment.** Suppose that $L_1, L_2$ and $L_S$ are $\omega$-regular and we wish to check $L_1 \parallel L_2 \preccurlyeq L_S$. When using **AG-NC**, restricting assumptions to $\omega$-regular languages is incomplete (cf. Theorem 2). Hence, the situation is the same as for $\infty$-regular languages (cf. row 5 of Table 1). When using **AG-C**, restricting assumptions to be $\omega$-regular is complete (cf. Theorem 6). Hence, we use $\mathbf{L}^\omega$ without augmenting the assumption alphabet, as summarized in row 6 of Table 1. This is a specific benefit of the restriction to $\omega$-regular languages. This instance terminates with the correct result due to Theorem 7, Proposition 2, and because $(P_i, Q_i)$ learns $\overline{(L_i \parallel \overline{L_S})} \downharpoonright \Sigma$ for $i \in \{1, 2\}$.

# 6 Related Work and Conclusion

Automated AG reasoning with automata-based learning was pioneered by Cobleigh et al. [9] for checking safety properties of finite state systems. In this context, Barringer et al. [3] investigate the soundness and completeness of a number of decomposition proof rules, and Wang [23] proposed a framework for automatic derivation of sound decomposition rules. Here, we extend the AG reasoning paradigm to arbitrary $\omega$-regular properties (i.e., both safety and liveness) using both non-circular and circular rules.

The idea behind (particular instances of) Theorem 5 is used implicitly in almost all existing work on automated assume-guarantee reasoning [9, 6, 7]. However, we are not aware of an explicit closed-form treatment of the weakest assumption in a general setting such as ours.

The learning-based automated AG reasoning paradigm has been extended to check simulation [5] and

deadlock [6]. Alur et al. [1], and Sinha et al. [21], have investigated symbolic and lazy SAT-based implementations, respectively. Tsay and Wang [22] show that verification of safety properties of $\infty$-regular systems is reducible the standard AG framework. In contrast, our focus is on the verification of arbitrary $\omega$-regular-properties of $\omega$-regular-systems.

In summary, we present a very general formalization, called LAG, of the learning-based automated AG paradigm. We instantiate LAG to verify $\omega$-regular properties of reactive systems with $\omega$-regular behavior. We also show how existing approaches for automated AG reasoning are special instances of LAG. In addition, we prove the soundness and completeness of circular and non-circular AG proof rules in the context of $\omega$-regular languages. Recently, techniques to reduce the number of queries [7], and refine the assumption alphabet [11], have been proposed in the context of using automated AG to verify safety properties. We believe that these techniques are applicable for $\omega$-regular-properties as well.

## References


[1] R. Alur, P. Madhusudan, and W. Nam. Symbolic Compositional Verification by Learning Assumptions. In *Procs. of CAV '05*, volume 3576 of *LNCS*, pages 548–562. Springer, July 2005.

[2] D. Angluin. Learning Regular Sets from Queries and Counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.

[3] H. Barringer, D. Giannakopoulou, and C. S. Păsăreanu. Proof Rules for Automated Compositional Verification Through Learning. In *Procs. of SAVCBS '03*, pages 14–21, Sept. 2003.

[4] H. Calbrix, M. Nivat, and A. Podelski. Ultimately Periodic Words of Rational $\omega$-Languages. In *Proc. of MPFS'93*, 1993.

[5] S. Chaki, E. M. Clarke, N. Sinha, and P. Thati. Automated Assume-Guarantee Reasoning for Simulation Conformance. In *Procs. of CAV '05*, volume 3576 of *LNCS*, pages 534–547. Springer, July 2005.

[6] S. Chaki and N. Sinha. Assume-Guarantee Reasoning for Deadlock. In *Procs. of FMCAD '06*.

[7] S. Chaki and O. Strichman. Optimized L* for Assume-Guarantee Reasoning. In *Procs. of TACAS '07*.

[8] E. Clarke, D. Long, and K. McMillan. Compositional Model Checking. In *Procs. of LICS '89*.

[9] J. M. Cobleigh, D. Giannakopoulou, and C. S. Păsăreanu. Learning Assumptions for Compositional Verification. In *Procs. of TACAS '03*, volume 2619 of *LNCS*, pages 331–346. Springer, Apr. 2003.

[10] A. Farzan, Y. Chen, E. Clarke, Y. Tsan, and B. Wang. Extending Automated Compositional Verification to the Full Class of Omega-Regular Languages. In *Procs. of TACAS '08*. Springer, 2008.

[11] M. Gheorghiu, D. Giannakopoulou, and C. S. Păsăreanu. Refining Interface Alphabets for Compositional Verification. In *Procs. of TACAS '07*, volume 4424 of *LNCS*, pages 292–307. Springer, Mar. 2007.

[12] E. M. Gold. Language Identification in the Limit. *Information and Control*, 10(5):447–474, May 1967.

[13] O. Grumberg and D. Long. Model Checking and Modular Verification. *TOPLAS*, 16(3):843–871, May 1994.

[14] T. A. Henzinger, S. Qadeer, and S. K. Rajamani. Decomposing Refinement Proofs Using Assume-Guarantee Reasoning. In *Procs. of ICCAD '00*, pages 245–252. IEEE, Nov. 2000.

[15] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[16] C. B. Jones. Specification and Design of (Parallel) Programs. In *Proceedings of the 9th IFIP World Congress*, volume 83 of *Information Processing*, pages 321–332, September 1983.

[17] O. Maler and A. Pnueli. On the Learnability of Infinitary Regular Sets. *Inf. Comput.*, 118(2):316–326, 1995.

[18] J. Misra and K. M. Chandy. Proofs of Networks of Processes. *TSE*, 7(4):417–426, July 1981.

[19] A. Pnueli. In Transition from Global to Modular Temporal Reasoning About Programs. *Logics and Models of Concurrent Systems*, 13:123–144, 1985.

[20] R. Rivest and R. Schapire. Inference of Finite Automata Using Homing Sequences. *Inf. Comput.*, 103, 1993.

[21] N. Sinha and E. Clarke. SAT-based Compositional Verification Using Lazy Learning. In *Procs. of CAV '07*.

[22] Y.-K. Tsay and B.-Y. Wang. Automated Compositional Reasoning of Intuitionistically Closed Regular Properties. In *Procs. of CIAA'08*, pages 36–45, 2008.

[23] B.-Y. Wang. Automatic Derivation of Compositional Rules in Automated Compositional Reasoning. In *Procs. of CONCUR'07*, pages 303–316, 2007.

# Modeling Regular Replacement for String Constraint Solving

Xiang Fu
Hofstra University
Xiang.Fu@hofstra.edu

Chung-Chih Li
Illinois State University
cli2@ilstu.edu

**Abstract**

Bugs in user input sanitation of software systems often lead to vulnerabilities. Among them many are caused by improper use of regular replacement. This paper presents a precise modeling of various semantics of regular substitution, such as the declarative, finite, greedy, and reluctant, using finite state transducers (FST). By projecting an FST to its input/output tapes, we are able to solve atomic string constraints, which can be applied to both the forward and backward image computation in model checking and symbolic execution of text processing programs. We report several interesting discoveries, e.g., certain fragments of the general problem can be handled using less expressive deterministic FST. A compact representation of FST is implemented in SUSHI, a string constraint solver. It is applied to detecting vulnerabilities in web applications.

## 1 Introduction

User input sanitation has been widely used by programmers to assure robustness and security of software. *Regular replacement* is one of the most frequently used approaches by programmers. For example, at both client and server sides of a web application, it is often used to perform format checking and filtering of command injection attack strings. As software bugs in user input sanitation can easily lead to vulnerabilities, it is desirable to employ automated analysis techniques for revealing such security holes. This paper presents the finite state transducer models of a variety of regular replacement operations, geared towards automated analysis of text processing programs.

One application of the proposed technique is symbolic execution [10]. In [3] we outlined a unified symbolic execution framework for discovering command injection vulnerabilities. The target system under test is executed as usual except that program inputs are treated as symbolic literals. A path condition is used to record the conditions to be met by the initial input, so that the program will execute to a location. At critical points, e.g., where a SQL query is submitted, path conditions are paired with attack patterns. Solving these constraints leads to attack signatures.

```
1 <?php
2   $msg = $_POST["msg"];
3   $sanitized =  preg_replace("/\<script.*?\>.*?\<\/script.*?\>/i","", $msg);
4   save_to_db($sanitized)
5 ?>
```

Listing 1: Vulnerable Sanitation against XSS Attack

In the following, we use an example to demonstrate the idea of the above research and motivate the modeling of regular replacement in this paper. Consider a PHP snippet in Listing 1, which takes a message as input and posts it to a bulletin. To prevent the Cross-Site Scripting (XSS) attack, the programmer calls preg_replace() to remove any pair of <script> and </script> tags. Unfortunately, the protection is insufficient. Readers can verify that <<script></script>script>alert('a')</script> is an attack string. After preg_replace(), it yields <script>alert('a')</script>.

We now show how the attack signature is generated, assuming the availability of symbolic execution. By symbolically executing the program, variable $msg is initialized with a symbolic literal and let it be $x$. Assume $\alpha$ is the regular expression <script.*?>.*?</script.*?> and $\varepsilon$ is the empty string. After

line 3, variable `$sanitized` has a symbolic value represented by string expression $x^{-}_{\alpha\to\varepsilon}$, and it is a replacement operator that denotes the effects of `preg_replace`, using the *reluctant* semantics (see "$*$?" in formula). Then at line 4 where the SQL query is submitted, a string constraint can be constructed as below, using an existing attack pattern. The equation asks: can a JavaScript snippet be generated after the `preg_replace` protection?

$$x^{-}_{\alpha\to\varepsilon} \equiv \texttt{<script.*?>alert('a')</script.*?>}$$

To solve the above equation, we first model the reluctant regular replacement $x^{-}_{\alpha\to\varepsilon}$ as a finite state transducer (FST) and let it be $\mathscr{A}_1$. The right hand side (RHS) of the equation is a regular expression, and let it be $r$. It is well known that the identity relation $Id(r) = \{(w,w) \mid w \in L(r)\}$ is a regular relation that can be recognized by an FST (let it be $\mathscr{A}_2$). Now let $\mathscr{A}$ be the composition of $\mathscr{A}_1$ and $\mathscr{A}_2$ (by piping the output tape of $\mathscr{A}_1$ to input tape of $\mathscr{A}_2$). Projecting $\mathscr{A}$ to its input tape results in a finite state automaton (FSA) that represents the solution of $x$.

Notice that a precise modeling that distinguishes the various regular replacement semantics is necessary. For example, a natural question following the above analysis is: *If we approximate the reluctant semantics using the greedy semantics, could the static analysis be still effective?* The answer is negative: When the `*?` operators in Listing 1 are treated as `*`, the analysis reports no solution for the equation, i.e., a false negative report on the actually vulnerable program.

In this paper, we present the modeling of regular replacement operations. §2 covers preliminaries. §3 and §4 present the modeling of various regular replacement semantics. §5 introduces tool support. §6 discusses related work. §7 concludes.

## 2 Preliminaries

This section formalizes several typical semantics of regular substitution, and then introduces a variation of the standard finite state transducer model. We introduce some notations first. Let $\Sigma$ represent the alphabet and $R$ the set of regular expressions over $\Sigma$. If $\omega \in \Sigma^*$, $\omega$ is called a word. Given a regular expression $r \in R$, its language is denoted as $L(r)$. When $\omega \in L(r)$ we say $\omega$ is an instance of $r$. We sometimes abuse the notation as $\omega \in r$ when the context is clear that $r$ is a regular expression. A regular expression $r$ is said to be *finite* if $L(r)$ is finite. Clearly, $r \in R$ is finite if and only if there exists a constant length bound $n \in N$ s.t. for any $\omega \in L(r)$, $|w| \le n$. We assume $\# \notin \Sigma$ is the *begin marker* and $\$ \notin \Sigma$ is the *end marker*. They will be used in modeling procedural regular replacement in §4. Let $\Sigma_2 = \Sigma \cup \{\#,\$\}$. Assume $\Psi$ is a second alphabet which is disjoint with $\Sigma$. Given $\omega \in (\Sigma \cup \Psi)^*$, $\pi(\omega)$ denotes the projection of $\omega$ to $\Sigma$ s.t. all the symbols in $\Psi$ are removed from $\omega$. Let $0 \le i < j \le |\omega|$, $\omega[i,j]$ represents a substring of $\omega$ starting from index $i$ and ending at $j-1$ (index counts from 0). Similarly, $\omega[i]$ refers to the element at index $i$. We use NFST, DFST to denote the nondeterministic and deterministic FST, respectively. Similar are NFSA and DFSA for finite state automata.

There are three popular semantics of regular replacement, namely *greedy*, *reluctant*, and *possessive*, provided by many programming languages, e.g., in `java.utils.regex` of J2SE. We concentrate on two of them: the greedy and the reluctant. The greedy semantics tries to match a given regular expression pattern with the longest substring of the input while the reluctant semantics works in the opposite way. From the theoretical point of view, it is also interesting to define a *declarative* semantics for string replacement. A declarative replacement $\gamma_{r\to\omega}$ replaces every occurrence of a regular pattern $r$ with $\omega$.

**Definition 2.1.** Let $\gamma, \omega \in \Sigma^*$ and $r \in R$ (with $\varepsilon \notin r$). The *declarative replacement*, denoted as $\gamma_{r\to\omega}$, is defined as:

$$\gamma_{r\to\omega} = \begin{cases} \{\gamma\} & \text{if } \gamma \notin \Sigma^* r \Sigma^* \\ \{\nu_{r\to\omega}\,\omega\,\mu_{r\to\omega} \mid \gamma = \nu\beta\mu \text{ and } \beta \in r\} & \text{otherwise} \end{cases}$$

∎

The greedy and reluctant semantics are also called *procedural*, because both of them enforce a *left-most* matching. The replacement procedure is essentially a loop which examines each index of a word, from left to right. Once there is a match of the regular pattern $r$, the greedy replacement performs the longest match, and the reluctant replaces the shortest.

**Definition 2.2.** Let $\gamma, \omega \in \Sigma^*$ and $r \in R$ (with $\varepsilon \notin r$). The *reluctant replacement* of $r$ with $\omega$ in $\gamma$, denoted as $\gamma^-_{r\to\omega}$, is defined recursively as $\gamma^-_{r\to\omega} = \{\nu\omega\mu^-_{r\to\omega}\}$ where $\gamma = \nu\beta\mu$, $\nu \notin \Sigma^* r \Sigma^*$, $\beta \in r$, and for every $\nu_1, \nu_2, \beta_1, \beta_2, \mu_1, \mu_2 \in \Sigma^*$ with $\nu = \nu_1\nu_2$, $\beta = \beta_1\beta_2$, $\mu = \mu_1\mu_2$: if $\nu_2 \neq \varepsilon$ then $\nu_2\beta_1 \notin r$ and $\nu_2\beta\mu_1 \notin r$; and, if $\beta_2 \neq \varepsilon$ then $\beta_1 \notin r$. ▮

Note that in the above definition, "if $\nu_2 \neq \varepsilon$ then $\nu_2\beta_1 \notin r$ and $\nu_2\beta\mu_1 \notin r$" enforces left-most matching", i.e., there does not exist an earlier match of $r$ than $\beta$; similarly, "if $\beta_2 \neq \varepsilon$ then $\beta_1 \notin r$" enforces shortest matching, i.e., there does not exist a shorter match of $r$ than $\beta$.

**Definition 2.3.** Let $\gamma, \omega \in \Sigma^*$ and $r \in R$ (with $\varepsilon \notin r$). The *greedy replacement*, denoted as $\gamma^+_{r\to\omega}$, is defined recursively as $\gamma^+_{r\to\omega} = \{\nu\omega\mu^+_{r\to\omega}\}$ where $\gamma = \nu\beta\mu$, $\nu \notin \Sigma^* r \Sigma^*$, $\beta \in r$, and for every $\nu_1, \nu_2, \beta_1, \beta_2, \mu_1, \mu_2 \in \Sigma^*$ with $\nu = \nu_1\nu_2$, $\beta = \beta_1\beta_2$, $\mu = \mu_1\mu_2$: if $\nu_2 \neq \varepsilon$ then $\nu_2\beta_1 \notin r$ and if $\mu_1 \neq \varepsilon$ then $\nu_2\beta\mu_1 \notin r$. ▮

**Example 2.4.** Let $\gamma = aaa$ with $a \in \Sigma$, (i) $\gamma_{aa\to b} = \{ba, ab\}$, $\gamma^+_{aa\to b} = \{ba\}$, $\gamma^-_{aa\to b} = \{ba\}$. (ii) $\gamma_{a^+\to b} = \{b, bb, bbb\}$, $\gamma^+_{a^+\to b} = \{b\}$, and $\gamma^-_{a^+\to b} = \{bbb\}$. ▮

Notice that in the above definitions, $\varepsilon \notin r$ is required for simplicity. In practice, precise Perl/Java regex semantics is followed for handling $\varepsilon \in r$. For example, in SUSHI, given $\gamma = a$, $r = a^*$, and $\omega = b$, $\gamma^-_{r\to\omega} = \{bab\}$ and $\gamma^+_{r\to\omega} = \{bb\}$. When $\beta \in \gamma^-_{r\to\omega}$, we often abuse the notation and write it as $\beta = \gamma^-_{r\to\omega}$, given the following lemma. Similar applies to $\gamma^+_{r\to\omega}$.

**Lemma 2.5.** For any $\gamma, \omega \in \Sigma^*$ and $r \in R$: $|\gamma^+_{r\to\omega}| = |\gamma^-_{r\to\omega}| = 1$.

In the following, we briefly introduce the notion of FST and its variation, using the terminology in [7]. We demonstrate its application to modeling the declarative replacement.

**Definition 2.6.** Let $\Sigma^\varepsilon$ denote $\Sigma \cup \{\varepsilon\}$. A finite state transducer (FST) is an enhanced two-taped nondeterministic finite state machine described by a quintuple $(\Sigma, Q, q_0, F, \delta)$, where $\Sigma$ is the alphabet, $Q$ the set of states, $q_0 \in Q$ the initial state, $F \subseteq Q$ the set of final states, and $\delta$ is the transition function, which is a total function of type $Q \times \Sigma^\varepsilon \times \Sigma^\varepsilon \to 2^Q$. ▮

It is well known that each FST accepts a regular relation which is a subset of $\Sigma^* \times \Sigma^*$. Given $\omega_1, \omega_2 \in \Sigma^*$ and an FST $\mathscr{M}$, we say $(\omega_1, \omega_2) \in L(\mathscr{M})$ if the word pair is accepted by $\mathscr{M}$. Let $\mathscr{M}_3$ be the composition of two FSTs $\mathscr{M}_1$ and $\mathscr{M}_2$, denoted as $\mathscr{M}_3 = \mathscr{M}_1 || \mathscr{M}_2$. Then $L(\mathscr{M}_3) = \{(\mu, \nu) \mid (\mu, \eta) \in L(\mathscr{M}_1)$ and $(\eta, \nu) \in L(\mathscr{M}_2)$ for some $\eta \in \Sigma^*\}$. We introduce an equivalent definition of FST below.

**Definition 2.7.** An augmented finite state transducer (AFST) is an FST $(\Sigma, Q, q_0, F, \delta)$ with the transition function augmented to type $Q \times \mathscr{R} \to 2^Q$, where $\mathscr{R}$ is the set of regular relations over $\Sigma$. ▮

In practice, we would often restrict the transition function of an AFST to the following two types: (1) $Q \times R \times \Sigma^* \to 2^Q$. In a transition diagram, we label the arc from $q_i$ to $q_j$ for transition $q_j \in \delta(q_i, r : \omega)$

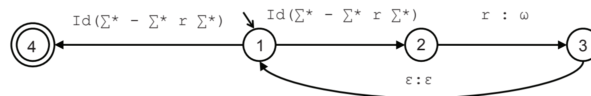



Figure 1: An FST for $s_{r\to\omega}$

by $r:\omega$; and (2) $Q \times \{Id(r) \mid r \in R\} \rightarrow 2^Q$, where $Id(r) = \{(\omega,\omega) \mid \omega \in L(r)\}$. In a transition diagram, an arc of type (2) is labeled as $Id(r)$.

Now, we can use an AFST to model the declarative string replacement $s_{r\rightarrow\omega}$ for any $\omega \in \Sigma^*$ and $r \in R$ (with $\varepsilon \notin r$). Figure 1 shows the construction, which presents an AFST that accepts $\{(s,\eta) \mid s \in \Sigma^* \text{ and } \eta \in s_{r\rightarrow\omega}\}$. In other words, given any two $s,\eta \in \Sigma^*$, we can use the AFST to check if $\eta$ is a string obtained from $s$ by replacing every occurrence of patterns in $r$ with $\omega$. We alternatively use FST and AFST for the time being without loss of generality.

# 3 DFST and Finite Replacement

This section shows that regular replacement with finite language pattern can be modeled using DFST, under certain restrictions. We fix the notation of DFST first. Intuitively, for a DFST, at any state $q \in Q$ the input symbol uniquely determines the destination state and the symbol on output tape. If there is a transition labeled with $\varepsilon$ on the input, then this is the only transition from $q$.

**Definition 3.1.** An FST $\mathscr{A} = (\Sigma, Q, s_0, F, \delta)$ is *deterministic* if for any $q \in Q$ and any $a \in \Sigma$ the following is true. Let $t_1,t_2 \in \{a,\varepsilon\}$, $b_1,b_2 \in \Sigma^\varepsilon$, and $q_1,q_2 \in Q$. $q_1 = q_2$, $b_1 = b_2$, and $t_1 = t_2$ if $q_1 \in \delta(q,t_1:b_1)$ and $q_2 \in \delta(q,t_2:b_2)$. ∎

**Lemma 3.2.** Let $\$ \notin \Sigma$ be an end marker. Given a *finite* regular expression $r \in R$ with $\varepsilon \notin r$ and $\omega_2 \in \Sigma^*$, there exist DFST $\mathscr{A}^-$ and $\mathscr{A}^+$ s.t. for any $\omega,\omega_1 \in \Sigma^*$: $\omega_1 = \omega^-_{r\rightarrow\omega_2}$ iff $(\omega\$,\omega_1\$) \in L(\mathscr{A}^-)$; and, $\omega_1 = \omega^+_{r\rightarrow\omega_2}$ iff $(\omega\$,\omega_1\$) \in L(\mathscr{A}^+)$.

We briefly describe how $\mathscr{A}^+$ is constructed for $\omega^+_{r\rightarrow\omega_2}$, similar is $\mathscr{A}^-$. Given a finite regular expression $r$, and assume its length bound is $n$. Let $\Sigma^{\leq n} = \bigcup_{0\leq i\leq n}\Sigma^i$. Then $\mathscr{A}^+$ is defined as a quintuple $(\Sigma\cup\{\$\}, Q, q_0, F, \delta)$. The set of states $Q = \{q_1,\dots,q_{|\Sigma^{\leq n}|}\}$ has $|\Sigma^{\leq n}|$ elements, and let $\mathscr{B}:\Sigma^{\leq n} \rightarrow Q$ be a bijection. Let $q_0 = \mathscr{B}(\varepsilon)$ be the initial state and the only final state. A transition $(q,q',a:b)$ is defined as follows for any $q \in Q$ and $a \in \Sigma\cup\{\$\}$, letting $\beta = \mathscr{B}^{-1}(q)$: (case 1) if $a \neq \$$ and $|\beta| < n$, then $b = \varepsilon$ and $q' = \mathscr{B}(\beta a)$; or (case 2) if $a \neq \$$ and $|\beta| = n$: if $\beta \notin r\Sigma^*$, then $b = \beta[0]$ and $q' = \mathscr{B}(\beta[1:|\beta|]a)$; otherwise, let $\beta = \mu\nu$ where $\mu$ is the longest match of $r$, then $b = \omega_2$ and $q' = \mathscr{B}(\nu a)$; or (case 3) if $a = \$$, then $b = \beta^+_{r\rightarrow\omega_2}\$$ and $q' = q_0$.

Intuitively, the above algorithm simulates the left-most matching. It buffers the current string processed so far, and the buffer size is the length bound of $r$. Once the buffer is full (case 2), it examines the buffer and checks if there is a match. If not, it emits the first character and produces it as output; otherwise, it produces $\omega_2$ on the output tape. Clearly, $\mathscr{B}$ is feasible because of the bounded length of $r$.

# 4 Procedural Replacement

The modeling of procedural replacement is much more complex than that of the declarative semantics. The general idea is to compose a number of finite state transducers for generating and filtering begin and end markers for the regular pattern in the input word. We start with the reluctant semantics. Given reluctant replacement $S^-_{r\rightarrow\omega}$, the modeling consists of four steps.

## 4.1 Modeling Left-Most Reluctant Replacement

**Step 1 (DFST Marker for End of Regular Pattern):** The objective of this step is to construct a DFST (called $\mathscr{A}_1$) that marks the end of regular pattern $r$, given $S^-_{r\rightarrow\omega}$. We first construct a deterministic FSA $\mathscr{A}$ that accepts $r$ s.t. $\mathscr{A}$ does not have any $\varepsilon$ transition. We use $(q,a,q')$ to denote a transition from

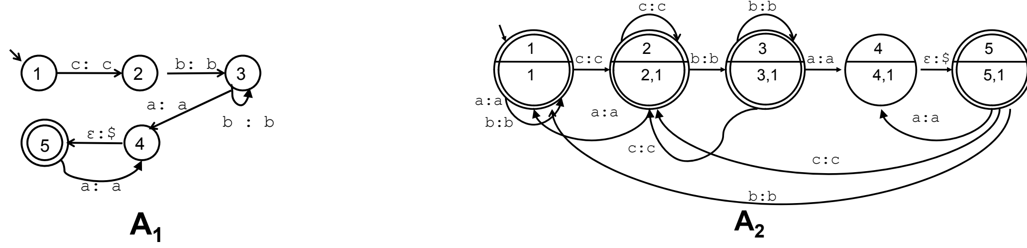



Figure 2: DFST End Marker

state $q$ to $q'$ that is labeled with $a \in \Sigma$. Then we modify each final state $f$ of FSA as below: (1) make $f$ a non-final state, (2) create a new final state $f'$ and establish a transition $\varepsilon$ from $f$ to $f'$, (3) for any outgoing transition $(f,a,s)$ create a new transition $(f',a,s)$ and remove that outgoing transition from $f$ (keeping the $\varepsilon$ transition). Thus the $\varepsilon$ transition is the only outgoing transition of $f$. Then convert the FSA into a DFST $\mathscr{A}_1$ as below: for an $\varepsilon$ transition, its output is \$ (end marker); for every other transition, its output is the same as the input symbol.

**Example 4.1.** $A_1$ in Figure 2 is the DFST generated for regular expression $cb^+a^+$. ▮

**Step 2 (Generic End Marker):** Note that on the input tape, $\mathscr{A}_1$ only accepts $r$. We would like to generalize $\mathscr{A}_1$ so that the new FST (called $\mathscr{A}_2$) will accept any word on its input tape. For example, $A_2$ in Figure 2 is a generalization of $A_1$, and $(ccbbaa, ccbba\$a\$) \in L(A_2)$.

Step 2 is formally defined as follows. Given $\mathscr{A}_1 = (\Sigma \cup \{\$\}, Q_1, q_0^1, F_1, \delta_1)$ as described in Step 1, $\mathscr{A}_2$ is a quintuple $(\Sigma \cup \{\$\}, Q_2, q_0^2, F_2, \delta_2)$. A *labeling function* $\mathscr{B} : Q_2 \to 2^{Q_1}$ is a bijection s.t. $\mathscr{B}(q_0^2) = \{q_0^1\}$. For any $t \in Q_2$ and $a \in \Sigma$: $t' \in \delta_2(t, a:a)$ iff $\mathscr{B}(t') = \{s' \mid \exists s \in \mathscr{B}(t)\ s.t.\ s' \in \delta_1(s, a:a)\} \cup \{q_0^1\}$. Clearly, $\mathscr{B}$ models a collection of states in $\mathscr{A}_1$ that can be reached by the substrings consumed so far on $\mathscr{A}_2$. Note that there is at most one state reached by a substring, because $\mathscr{A}_1$ is deterministic. Hence, the collection of states is always finite. The handling of the only $\varepsilon$ transition in $\mathscr{A}_2$ is similar.

**Example 4.2.** $A_2$ in Figure 2 is the result of applying the above algorithm on $A_1$. Clearly, for $A_2$, $\mathscr{B}(1) = \{1\}$, $\mathscr{B}(2) = \{2,1\}$, and $\mathscr{B}(3) = \{3,1\}$. Running $(ccbb, ccbb)$ on $A_2$ results a partial run to state 3. For $(ccbb, ccbb)$, there are five substring pairs to be observed: $(ccbb, ccbb)$, $(cbb, cbb)$, $(bb, bb)$, $(b, b)$, and $(\varepsilon, \varepsilon)$. Among them, only $(cbb, cbb)$ and $(\varepsilon, \varepsilon)$ can be extended to match $r$ (i.e., $cb^+a^+$). Clearly, if run them on $A_1$, they would result in partial runs that end at states 3 (by $(cbb, cbb)$) and 1 (by $(\varepsilon, \varepsilon)$). This is the intuition of having $\mathscr{B}(3) = \{3,1\}$ in $A_2$. The labeling function $\mathscr{B}$ keeps track of the potential substrings of match by recording those states of $A_1$ that could be reached by the substrings. ▮

The following lemma states that $\mathscr{A}_2$ inserts an end marker \$ after each occurrence of regular pattern $r$, and there are no duplicate end markers inserted (even when empty string $\varepsilon \in r$).

**Lemma 4.3.** For any $r \in R$ there exists a DFST $\mathscr{A}_2$ s.t. for any $\omega \in \Sigma^*$, there is one and only one $\omega_2 \in (\Sigma \cup \{\$\})^*$ with $(\omega, \omega_2) \in L(\mathscr{A}_2)$ and $\omega = \pi(\omega_2)$ such that $\omega_2$ satisfies the following: for any $0 \le x < |\omega_2|$, $\omega_2[x] = \$$ iff $\pi(\omega_2[0,x]) \in \Sigma^* r$; and for any $1 \le x < |\omega_2|$, if $\omega_2[x] = \$$, then $\omega_2[x-1] \neq \$$.

**Step 3 (Begin Marker of Regular Pattern):** From $\mathscr{A}_2$ we can construct a *reverse* transducer $\mathscr{A}_3$ by reversing all transitions in $\mathscr{A}_2$ and replacing the end marker \$ with the begin marker #. Then create a new initial state $s_0$, add $\varepsilon$ transitions from $s_0$ to each final state in $\mathscr{A}_2$, and make the original initial state of $\mathscr{A}_2$ the final state in $\mathscr{A}_3$. For example, the $A_3$ shown in Figure 3 is a reverse of $A_2$ in Figure 2. Clearly, $(aabbcc, \#a\#abbcc) \in L(A_3)$, and $A_3$ marks the beginning for pattern $r = a^+b^+c$.

**Lemma 4.4.** For any $r \in R$ there exists an FST $\mathscr{A}_3$ s.t. for any $\mu \in \Sigma^*$, there exists one and only one $\nu \in (\Sigma \cup \{\#\})^*$ with $(\mu, \nu) \in L(\mathscr{A}_3)$. $\nu$ satisfies the following: (i) $\mu = \pi(\nu)$, and, (ii) for $0 \le i < |\nu|$: $\nu[i] = \#$ iff $\pi(\nu[i, |\nu|]) \in r\Sigma^*$, and (iii) for $1 \le i < |\nu|$: if $\nu[i] = \#$ then $\nu[i-1] \neq \#$,

Figure 3: Begin Marker and Reluctant Replacement Transducers

The beauty of the nondeterminism is that $\mathscr{A}_3$ can always make the "smart" decision to enforce there is one and only one run which "correctly" inserts the label #. Any incorrect insertion will never reach a final state. The nondeterminism gives $\mathscr{A}_3$ the "look ahead" ability.

**Step 4 (Reluctant Replacement):** Next we define an automaton for implementing the reluctant replacement semantics. Given a DFSA $\mathscr{M}$, let $\mathscr{M}_1$ be the new automaton generated from $\mathscr{M}$ by removing all the outgoing transitions from each final state of $\mathscr{M}$. We have the following result: $L(\mathscr{M}_1) = \{s \mid s \in L(\mathscr{M}) \wedge \forall s' \prec s : s' \notin L(\mathscr{M})\}$. Clearly $\mathscr{M}_1$ implements the "shortest match" semantics. Given $r \in R$, let $\texttt{reluc}(r)$ represent the result of applying the above "reluctant" transformation on $r$.

We still need to filter the extra begin markers during the replacement process. Given a regular language $\mathscr{L} = \texttt{reluc}(r)$, let $\mathscr{L}_\#$ represent the language generated from $\mathscr{L}$ by nondeterministically inserting #, i.e., $\mathscr{L}_\# = \{\mu \mid \mu \in (\Sigma \cup \{\#\})^* \wedge \pi(\mu) \in \mathscr{L}\}$. Clearly, to recognize $\mathscr{L}_\#$, an automaton can be constructed from $\mathscr{A}$ (which accepts $\mathscr{L}$) by attaching a self loop transition (labeled with #) to each state of $\mathscr{A}$. Let $\mathscr{L}_\#' = \mathscr{L}_\# \cap \overline{\Sigma_2^*\#}$ (this is to avoid removing the begin marker for the next match). Now given regular language $\mathscr{L}_\#'$ and $\omega \in \Sigma^*$, it is straightforward to construct an FST $\mathscr{A}_{\mathscr{L}_\#' \times \omega}$ s.t. $(\mu, \nu) \in L(\mathscr{A}_{\mathscr{L}_\#' \times \omega})$ iff $\mu \in \mathscr{L}_\#'$ and $\nu = \omega$. Intuitively, given any $\mu$ (interspersed with #) that matches $r$, the FST replaces it with $\omega$.

An automaton $\mathscr{A}_4$ (as shown in Figure 3) can be defined. Intuitively, $\mathscr{A}_4$ consumes a symbol on both the input tape and output tape unless encountering a begin marker #. Once a # is consumed, $\mathscr{A}_4$ enters the replacement mode, which replaces the shortest match of $r$ with $\omega$ (and also removes extra # in the match). Thus, piping it with $\mathscr{A}_3$ directly leads to the precise modeling of reluctant replacement.

**Lemma 4.5.** Given any $r \in R$ and $\omega \in \Sigma^*$, and let $\mathscr{A}_r$ be $\mathscr{A}_3 || \mathscr{A}_4$, then for any $\omega_1, \omega_2 \in \Sigma^*$: $(\omega_1, \omega_2) \in L(\mathscr{A}_r)$ iff $\omega_2 = \omega_{1}{}^{-}_{r \to \omega}$.

## 4.2 Modeling Left-Most Greedy Semantics

Handling the greedy semantics is more complex. We have to insert both begin and end markers for the regular pattern and then apply a number of filters to ensure the longest match. The first action is to insert begin markers using $\mathscr{A}_3$ as described in the previous section. Then the second action is to insert an end marker \$ *nondeterministically* after each substring matching $r$. Later, additional markers will be filtered, and improper marking will be rejected. We call this FST $\mathscr{A}_2'$. Given $r \in R$ and $\omega \in \Sigma^*$, $\mathscr{A}_2'$ can be constructed so that for any $\omega_1 \in \Sigma_2^*$ and $\omega_2 \in \Sigma_2^*$: $(\omega_1, \omega_2) \in L(\mathscr{A}_2')$ iff (i) $\pi(\omega_1) = \pi(\omega_2)$, and (ii) for any $0 \le i < |\omega_2|$, $\pi(\omega_2[0,i]) \in \Sigma^* r$ *if* $\omega_2[i] = \$$, and (iii) for any $1 \le i < |\omega_2|$, if $\omega_2[i] = \$$ then $\omega_2[i-1] \neq \$$. Notice that $\mathscr{A}_2'$ is different from $\mathscr{A}_2$ in that the \$ after a match of $r$ is *optional*. Clearly, $\mathscr{A}_2'$ can be modified from $\mathscr{A}_2$ by simply adding an $\varepsilon : \varepsilon$ transition from $f$ (old final state) to $f'$ (new final state) in $\mathscr{A}_2$, e.g., to add an $\varepsilon : \varepsilon$ transition from state 4 to 5 in $A_2$ in Figure 2. Also #:# transitions are needed for each state to keep the # introduced by $\mathscr{A}_3$.

Then we need a filter to remove extra markers so that every \$ is paired with a #. Note we do not yet make sure that between the pair of # and \$, the substring is a match of $r$. We construct the AFST

as follows. Let $\mathscr{A}_f = (\Sigma_2, Q, q_0, F, \delta)$. $Q$ has two states $q_0$ and $q_1$. $F = \{q_0\}$. The transition function $\delta$ is defined as below: (i) $\delta(q_0, Id(\Sigma)) = \{q_0\}$, (ii) $\delta(q_0, \$ : \varepsilon) = \{q_0\}$, (iii) $\delta(q_0, \# : \#) = \{q_1\}$, (iv) $\delta(q_1, \# : \varepsilon) = \{q_1\}$, (v)$\delta(q_1, Id(\Sigma)) = \{q_1\}$, (vi) $\delta(q_1, \$ : \$) = \{q_0\}$.

Now we will apply three FST filters (represented by three identity relations $Id(L_1)$, $Id(L_2)$, and $Id(L_3)$), for filtering the nondeterministic end marking. $L_1$, $L_2$, and $L_3$ are defined as below:

$$L_1 = \overline{\Sigma_2^* \# (\bar{r} \cap \Sigma^*) \$ \Sigma_2^*} \quad (1)$$

$$L_2 = \overline{\Sigma_2^* [\hat{}\#] (\$ \cap r_{\#,\$})} \cap \overline{\Sigma_2^* [\hat{}\#] (\$\Sigma\Sigma_2^* \cap r_{\#,\$}\Sigma_2^*)} \quad (2)$$

$$L_3 = \overline{\Sigma_2^* \# (r_{\#,\$} \cap (\Sigma^* \$ (\Sigma^+)_{\#,\$})) \Sigma_2^*} \quad (3)$$

The intuition of $L_1$ is to make sure that the substring between each pair of # and \$ is a match of $r$. The motivation of $L_2$ is for preventing removing too many # symbols by $\mathscr{A}_f$ (due to improper insertion of end markers by $\mathscr{A}_2'$). $Id(L_2)$ handles two cases: (1) to avoid removing the begin markers at the end of input word if the pattern $r$ includes $\varepsilon$; and (2) to avoid removing begin markers for the next instance of $r$. Consider the following example for case (2): given $S^+_{a^* \to c}$ and the input word *bab*, the correct marking of begin and end markers should be #\$*b*#*a*\$#\$*b*#\$ (which leads to *cbccbc* as output). However the following incorrect marking could pass $Id(L_1)$ and $Id(L_3)$, if not enforcing the $Id(L_2)$ filter: #\$*b*#*a*\$*b*#\$. The cause is that an ending marker \$ (e.g., the one before the last *b*) may trigger $\mathscr{A}_f$ to remove a good begin marker # that precedes an instance of $r$ (i.e., $\varepsilon$). Filter $Id(L_2)$ is thus defined for preventing such cases.

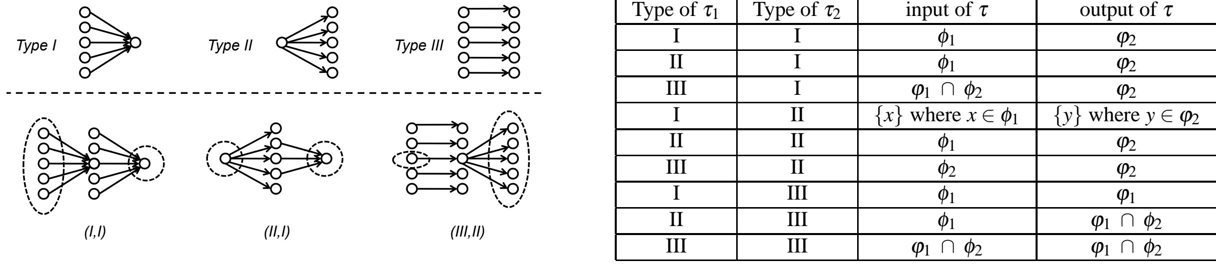
Finally, $L_3$ is defined for ensuring longest match. Note that filter $Id(L_3)$ will be applied after $Id(L_1)$ and $Id(L_2)$ which have guaranteed the pairing of markers and the proper contents between each pair of markers. $L_3$ eliminates cases where starting from # there is a substring (when projected to $\Sigma$) matches $r$ and the string contains at least one \$ inside (implying that there is a longer match than the substring between the # and its matching \$). Note that $(\Sigma^+)_{\#,\$}$ refers to a word in $\Sigma^+$ interspersed with begin/end markers, i.e., for any $\omega \in (\Sigma^+)_{\#,\$}$, $|\pi(\omega)| > 0$. We also need an FST $\mathscr{A}_4'$, which is very similar to $\mathscr{A}_4$. $\mathscr{A}_4'$ enters (and leaves) the replacement mode, once it sees the begin (and the end) marker. Then we have the following:

**Lemma 4.6.** Given any $r \in R$ and $\omega \in \Sigma^*$, let $\mathscr{A}_g$ be $\mathscr{A}_3 || \mathscr{A}_2' || \mathscr{A}_f || \mathscr{A}_{Id(L_1)} || \mathscr{A}_{Id(L_2)} || \mathscr{A}_{Id(L_3)} || \mathscr{A}_4'$, then for any $\omega_1, \omega_2 \in \Sigma^*$: $\omega_2 = {\omega_1}^+_{r \to \omega}$ iff $(\omega_1, \omega_2) \in L(\mathscr{A}_g)$.

# 5 SISE Constraint and SUSHI Solver

This work is implemented as part of a constraint solver called SUSHI [4], which solves SISE (Simple Linear String Equation) constraints. Intuitively, a SISE equation can be regarded as a variation of word equation [13]. It is composed of word literals, string variables, and various frequently seen string operators such as substring, concatenation, and regular replacement. To solve SISE, an automata based approach is taken, where a SISE is broken down into a number of atomic string operations. Then the solution process consists of a number of backward image computation steps. We now briefly describe the part related to regular replacement.

It is well known that projecting an FST to its input tape (by removing the output symbol from each transition) results in a standard finite state machine. Similar applies to the projection to output tape. We use `input`$(\mathscr{A})$ and `output`$(\mathscr{A})$ to denote the input and output projection of an FST $\mathscr{A}$. Given an atomic SISE constraint $x_{r \to \omega} \equiv r_2$, the solution pool of $x$ (backward image of the constraint) is defined as $\{\mu \mid \mu_{r \to \omega} \in L(r_2)\}$. Given a regular expression $\nu$, the forward image of $\nu_{r \to \omega}$ is defined as $\{\mu \mid \mu \in \alpha_{r \to \omega} \text{ and } \alpha \in \nu\}$. Clearly, let $\mathscr{A}$ be the corresponding FST of $x_{r \to \omega}$, the backward image can be computed using `input`$(\mathscr{A} || Id(r_2))$. Similarly, given $\mu_{r \to \omega}$, the forward image is `output`$(Id(\mu) || \mathscr{A})$.




| Type of $\tau_1$ | Type of $\tau_2$ | input of $\tau$ | output of $\tau$ |
|---|---|---|---|
| I | I | $\phi_1$ | $\varphi_2$ |
| II | I | $\phi_1$ | $\varphi_2$ |
| III | I | $\varphi_1 \cap \phi_2$ | $\varphi_2$ |
| I | II | $\{x\}$ where $x \in \phi_1$ | $\{y\}$ where $y \in \varphi_2$ |
| II | II | $\phi_1$ | $\varphi_2$ |
| III | II | $\phi_2$ | $\varphi_2$ |
| I | III | $\phi_1$ | $\varphi_1$ |
| II | III | $\phi_1$ | $\varphi_1 \cap \phi_2$ |
| III | III | $\varphi_1 \cap \phi_2$ | $\varphi_1 \cap \phi_2$ |

Figure 4: SUSHI FST Transition Set

## 5.1 Compact Representation of FST

SUSHI relies on `dk.brics.automaton` [15] for FSA operations. We use a self-made Java package for supporting FST operations [16]. Note that there are existing tools related to FST, e.g., the FSA toolbox [16]. In practice, to perform inspection on user input, FST has to handle a large alphabet represented using 16-bit Unicode. In the following, we introduce a compact representation of FST. A collection of FST transitions can be encoded as a *SUSHI FST Transition Set* (SFTS) in the following form:

$$\mathscr{T} = (q, q', \phi : \varphi)$$

where $q$, $q'$ are the source and destination states, the *input charset* $\phi = [n_1, n_2]$ with $0 \leq n_1 \leq n_2$ represents a range of input characters, and the *output charset* $\varphi = [m_1, m_2]$ with $0 \leq m_1 \leq m_2$ represents a range of output characters. $\mathscr{T}$ includes a set of transitions with the same source and destination states: $\mathscr{T} = \{(q, q', a : b) \mid a \in \phi \text{ and } b \in \varphi\}$. For $\mathscr{T} = (q, q', \phi : \varphi)$, however, it is required that if $|\phi| > 1$ and $|\varphi| > 1$, then $\phi = \varphi$. For $\phi$ and $\varphi$, $\varepsilon$ is represented using $[-1, -1]$. Thus, there are three types of SFTS (excluding the $\varepsilon$ cases), as shown in the following. *Type I:* $|\phi| > 1$ and $|\varphi| = 1$, thus $\mathscr{T} = \{(q, q', a : b) \mid a \in \phi \text{ and } \varphi = \{b\}\}$. *Type II:* $|\phi| = 1$ and $|\varphi| > 1$, thus $\mathscr{T} = \{(q, q', a : b) \mid b \in \varphi \text{ and } \phi = \{a\}\}$. *Type III:* $|\varphi| = |\phi| > 1$, thus $\mathscr{T} = \{(q, q', a : a) \mid a \in \phi\}$. The top-left of Figure 4 gives an intuitive illustration of these SFTS types (which relates the input and output chars).

The algorithms for supporting FST operations (such as union, Kleen star) should be customized correspondingly. In the following, we take FST composition as one example. Let $\mathscr{A} = (\Sigma, Q, q, F, \delta)$ be the composition of $\mathscr{A}_1 = (\Sigma, Q_1, q_0^1, F_1, \delta_1)$ and $\mathscr{A}_2 = (\Sigma, Q_2, s_0^2, F_2, \delta_2)$. Given $\tau_1 = (t_1, t_1', \phi_1 : \varphi_1)$ in $\mathscr{A}_1$ and $\tau_2 = (t_2, t_2', \phi_2 : \varphi_2)$ in $\mathscr{A}_2$, where $\varphi_1 \cap \phi_2 \neq \emptyset$, an SFTS $\tau = (s_1, s_2, \phi : \varphi)$ is defined for $\mathscr{A}$ s.t. $s_1 = (t_1, t_2)$, $s_2 = (t_1', t_2')$, and the input/output charset of $\tau$ is defined as the table in Figure 4 (note all entries except for (I,II) produce one SFTS only). For example, when both $\tau_1$ and $\tau_2$ are type I, we have $\phi = \phi_1$ and $\varphi = \varphi_2$. The bottom left of Figure 4 shows the intuition of the algorithm. The dashed circles represent the corresponding input/output charset.

## 5.2 Evaluation

We are interested in whether the proposed technique is efficient and effective in practice. We list here four SISE equations for stress-testing the SUSHI package. Note that each equation is parametrized by an integer $n$. **eq1**: $x^{+}_{\mathtt{a}^+ \to \mathtt{b}\{n,n\}} \equiv \mathtt{b}\{2n, 2n\}$; **eq2**: $x^{-}_{\mathtt{a}^+ \to \mathtt{b}\{n,n\}} \equiv \mathtt{b}\{2n, 2n\}$; **eq3**: $x^{+}_{\mathtt{a}^* \to \mathtt{b}\{n,n\}} \equiv \mathtt{b}\{2n, 2n\}$; **eq4**: $x^{-}_{\mathtt{a}^* \to \mathtt{b}\{n,n\}} \equiv \mathtt{b}\{2n, 2n\}$. The following table displays the running results when $n$ is 41. (more data in [4]). It displays the max size of FST and FSA used in the solution process.

| Equation | FST States | FST Transitions | FSA States | FSA Transitions | Time (Seconds) |
|---|---|---|---|---|---|
| eq1(41) | 5751 | 16002 | 125 | 207 | 155.281 |
| eq2(41) | 5416 | 5748 | 83 | 124 | 162.469 |
| eq3(41) | 631 | 1565 | 2 | 2 | 492.281 |
| eq4(41) | 126 | 177 | 0 | 0 | 14.016 |

The technique scales well in practice. We applied SUSHI in discovering SQL injection vulnerabilities and XSS attacks in FLEX SDK (see technical report [4]). The running cost ranges from 1.4 to 74 seconds on a 2.1Ghz PC with 2GB RAM (with SISE equation size ranging from 17 to 565). [1]

# 6 Related Work

Recently, string analysis has received much attention in security and compatibility analysis of programs (see e.g., [5, 12]). In general, there are two interesting directions of string analysis: (1) *forward analysis*, which computes the image (or its approximation) of the program states as constraints on strings; and, (2) *backward analysis*, which usually starts from the negation of a property and computes backward. Most of the related work (e.g., [2, 11, 18]) falls into the category of forward analysis. This work can be used for both forward and backward image computation. Compared with forward analysis, it is able to generate attack signatures as evidence of vulnerabilities.

Modeling regular replacement distinguishes our work from several related work in the area. For example, one close work to ours is the HAMPI string constraint solver [9]. HAMPI supports solving string constraints with context-free components, which are unfolded to regular language. HAMPI, however, supports neither constant string replacement nor regular replacement, which limits its ability to reason about sanitation procedures. Similarly, Hooimeijer and Weimer's work [6] in the decision procedure for regular constraints does not support regular replacement. A closer work to ours is Yu's automata based forward/backward string analysis [18]. Yu uses a language based replacement [17], which introduces imprecision in its over-approximation. Conversely, our analysis considers the delicate differences among the typical regular replacement semantics and provides more accurate analysis. In [1], Bjørner *et al.* uses first order formula on bit-vector to model string operations except replacement. We conjecture that it can be extended by using recursion in their first order framework for defining replaceAll semantics.

FST is the major modeling tool in this paper. It is mainly inspired by [7, 14, 8] in computational linguistics, for processing phonological and morphological rules. In [8], an informal discussion was given for the semantics of left-most longest matching of string replacement. This paper has given the formal definition of replacement semantics and has considered the case where $\varepsilon$ is included in the search pattern. Compared with [7] where FST is used for processing phonological rules, our approach is lighter given that we do not need to consider the left and right context of re-writing rules in [7]. Thus more DFST can be used, which certainly has advantages over NFST, because DFST is less expressive. For example, in modeling the reluctant semantics, compared with [7], our algorithm does not have to non-deterministically insert begin markers and it does not need extra filters, thus more efficient. It is interesting to compare the two algorithms and measure the gain in performance by using more DFST in modeling, which remains one of our future work.

**Limitation of the Model:** It is shown in [4] that solving SISE constraint is decidable (with worst complexity 2-EXPTIME). This may seem contradictory with the conclusion in [1]. The decidability is achieved by restricting SISE as described below. SISE requires that each variable appears at most once and all variables must be appear in LHS. This permits a simple recursive algorithm that reduces the solution process into a number of backward image computation steps. However, it may limit the expressiveness of SISE in certain application scenario. SISE supports regular replacement, substring, concatenation operators, however, it does not support operators related to string length, e.g., indexOf and length operators. It is interesting to extend the framework to support mixed numerical and string operators, e.g., encoding numeric constraints using automata as described by Yu *et al.* in [18], or translating string constraints to first order formula on bit-vectors as shown by Bjørner *et al.* [1].

[1] SISE equation size is measured by the combined length of constant words, variables, and operators included in the equation.

# 7 Conclusion

This paper presents the finite state transducer models of various regular substitutions, including the declarative, finite, reluctant, and greedy replacement. A compact FST representation is implemented in a constraint solver SUSHI. The presented technique can be used for analyzing programs that process text and communicate with users using strings. Future directions include modeling mixture of greedy and reluctant semantics, handling hybrid numeric/string constraints, and context free components.

**Acknowledgment:** This paper is inspired by the discussion with Fang Yu, Tevfik Bultan, and Oscar Ibarra. We thank the anonymous reviewers for very constructive comments that help improve the paper.

# References

[1] Nikolaj Bjørner, Nikolai Tillmann, and Andrei Voronkov. Path feasibility analysis for string-manipulating programs. In *Proceedings of the 15th International Conference on Tools AND Algorithms for the Construction and Analysis of Systems (TACAS 2009)*, pages 322–336. Springer, 2009.

[2] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Precise analysis of string expressions. In *Proc. 10th International Static Analysis Symposium, SAS '03*, volume 2694 of *LNCS*, pages 1–18. Springer-Verlag, June 2003. Available from `http://www.brics.dk/JSA/`.

[3] X. Fu, X. Lu, B. Peltsverger, S. Chen, K. Qian, and L. Tao. A Static Analysis Framework for Detecting SQL Injection Vulnerabilities. In *Proceedings of 31st Annual International Computer Software and Applications Conference (COMPSAC 2007)*, pages 87 – 96, 2007.

[4] Xiang Fu, Chung chih Li, and Kai Qian. On simple linear string equations. `http://people.hofstra.edu/Xiang_Fu/XiangFu/publications/techrep09b.pdf`, 2009.

[5] C. Gould, Z. Su, and P. Devanbu. JDBC Checker: A Static Analysis Tool for SQL/JDBC Applications. In *Proceedings of the 26th International Conference on Software Engineering*, pages 697–698, 2004.

[6] Pieter Hooimeijer and Westley Weimer. A decision procedure for subset constraints over regular languagees. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 188–198, 2009.

[7] Ronald M. Kaplan and Martin Kay. Regular models of phonological rule systems. *Computational Linguistic*, 20(3):331–378, 1994.

[8] L. Karttunen, J p. Chanod, G. Grefenstette, and A. Schiller. Regular expressions for language engineering. *Natural Language Engineering*, 2:305–328, 1996.

[9] Adam Kiezun, Vijay Ganesh, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. Hampi: A solver for string constraints. In *Proc. of 2009 International Symposium on Testing and Analysis (ISSTA'09)*, 2009.

[10] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[11] Christian Kirkegaard and Anders Møller. Static analysis for Java Servlets and JSP. In *Proc. 13th International Static Analysis Symposium, SAS '06*, volume 4134 of *LNCS*, August 2006.

[12] V. B. Livshits and M. S. Lam. Finding security vulnerabilities in java applications with static analysis. In *Proceedings of the 14th conference on USENIX Security Symposium*. USENIX Association, 2005.

[13] M. Lothaire. *Algebraic Combinatorics on Words*. Cambridge University Press, 2002.

[14] Mehryar Mohri. Finite-state transducers in language and speech processing. *Computational Linguistics*, 23:269–311, 1997.

[15] A. Møller. The dk.brics.automaton package. *available at* `http://www.brics.dk/automaton/`.

[16] Gertjan van Noord. Fsa utilities: A toolbox to manipulate finite-state automata on simple linear string equations. `http://www.let.rug.nl/~vannoord/papers/fsa/fsa.html`, 1998.

[17] Fang Yu, Tevfik Bultan, Marco Cova, and Oscar H. Ibarra. Symbolic string verification: An automata-based approach. In *Proc. of the 15th SPIN Workshop on Model Checking Software*, pages 306–324, 2008.

[18] Fang Yu, Tevfik Bultan, and Oscar H. Ibarra. Symbolic string verification: Combining string analysis and size analysis. In *Proceedings of the 15th International Conference on Tools AND Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 322–336. Springer, 2009.

# Using Integer Clocks to Verify the Timing-Sync Sensor Network Protocol

Xiaowan Huang, Anu Singh, Scott A. Smolka
Department of Computer Science, Stony Brook University

**Abstract**

We use the UPPAAL model checker for Timed Automata to verify the Timing-Sync time-synchronization protocol for sensor networks (TPSN). The TPSN protocol seeks to provide network-wide synchronization of the distributed clocks in a sensor network. Clock-synchronization algorithms for sensor networks such as TPSN must be able to perform arithmetic on clock values to calculate clock drift and network propagation delays. They must be able to read the value of a local clock and assign it to another local clock. Such operations are not directly supported by the theory of Timed Automata. To overcome this formal-modeling obstacle, we augment the UPPAAL specification language with the *integer clock* derived type. Integer clocks, which are essentially integer variables that are periodically incremented by a global pulse generator, greatly facilitate the encoding of the operations required to synchronize clocks as in the TPSN protocol. With this integer-clock-based model of TPSN in hand, we use UPPAAL to verify that the protocol achieves network-wide time synchronization and is devoid of deadlock. We also use the UPPAAL Tracer tool to illustrate how integer clocks can be used to capture clock drift and resynchronization during protocol execution.

## 1 Introduction

A *sensor network* is a collection of spatially distributed, autonomous sensing devices, used to perform a cooperative task. Sensor networks are widely deployed for unmanaged and decentralized operations such as military operations, surveillance, and health and environmental monitoring. Sensor networks differ from traditional distributed networks in their stringent resource constraints, network dynamics, node failure, and intermittent communication links. Consequently, protocols executing on sensor networks must be robust and reliable.

Many high-level sensor applications depend on an underlying *time-synchronization* mechanism, which provides network-wide time synchronization. Thus, protocols for time synchronization are critical for sensor applications, and ensuring their correctness is of the utmost importance.

In this paper, we use the UPPAAL model checker [7] for Timed Automata to verify the correctness of the *Timing-sync Protocol for Sensor Networks* (TPSN) [3]. TPSN is a time-synchronization protocol that performs pair-wise node synchronization along the edges of a network spanning tree. The local clock of a designated node, typically the *root* of the spanning tree, serves as the reference clock. Nodes may leave or join the network. TPSN enjoys several advantages over other sensor network time-synchronization protocols, including higher precision (less synchronization error) and the fact that its tree-based scheme makes it well-suited for multi-hop networks.

Clock-synchronization algorithms for sensor networks such as TPSN must be able to perform arithmetic on clock values in order to calculate clock drift and network propagation delays. They must be able to read the value of a local clock and assign it to another local clock. Such operations are not directly supported by the theory of Timed Automata. To overcome this formal-modeling obstacle, we augment UPPAAL's input language with the *integer clock* derived type. Integer clocks, which are essentially integer variables that are periodically incremented by a global pulse generator, greatly facilitate the encoding of the operations required to synchronize clocks as in the TPSN protocol.

With this integer-clock-based model of TPSN in hand, we use UPPAAL to verify that the protocol achieves network-wide time synchronization and is devoid of deadlock. Our model additionally takes

into account nodes entering and leaving the network. We also use the UPPAAL Tracer tool to illustrate how integer clocks can be used to capture clock drift and resynchronization during protocol execution.

This paper is organized as follows. Section 2 offers a description of the TPSN [3] protocol. Section 3 introduces the concept of integer clocks. Section 4 presents our UPPAAL model of TPSN; the corresponding verification and simulation results are given in Section 5. Section 6 considers related work, while Section 7 contains our concluding remarks. An online version of the paper, available from `www.cs.sunysb.edu/~sas/NFM10-full.pdf`, contains an Appendix in which a complete UPPAAL source-code listing of our TPSN model can be found.

## 2 The Timing-Sync Protocol for Sensor Networks

The Timing-sync Protocol for Sensor Networks (TPSN) aims to provide network-wide time synchronization in a sensor network [3]. The TPSN protocol first establishes a hierarchical structure in the network. It then achieves pair-wise synchronization along the edges of this structure such that all local clocks synchronize to a reference node. A more detailed discussion of the two phases of TPSN is now given.

- **Level-discovery phase:** Level discovery occurs when the network is first deployed. In this phase, a hierarchical structure (spanning tree) is established for the network in which every node is assigned a level. A distinguished node called the **root node** is assigned level 0 and initiates this phase by broadcasting a *level_discovery* packet containing the identity and level of the sender. The immediate neighbors of the root node receive this packet and assign themselves a level one greater than the level they have received; i.e., level 1. They then broadcast a new *level_discovery* packet containing their own level. This process is continued and eventually every node in the network is assigned a level. After been assigned a level, a node neglects any future *level_discovery* packets, thereby ensuring that no flooding congestion takes place during this phase. Ultimately, a spanning tree is created with the root node at level 0.

- **Synchronization phase:** The root node initiates the synchronization phase, during which pair-wise synchronizations involving a level-$i$ node and a level-$(i-1)$ node are carried out using handshake (two-way) message exchanges. Eventually, all nodes in the network synchronize their clocks to that of the root node. Consider the synchronization of level-$i$ node $A$ and level-$(i-1)$ node $B$; i.e., $A$ is a child of $B$ in the spanning tree and $A$ is attempting to synchronize its clock with that of $B$. The following sequence of messages comprise the handshake: $A$ at local time $T1$ sends a *synchronization_pulse* packet containing its level number and the value of $T1$ to $B$. $B$ receives this packet at local time $T2$, where $T2 = T1 + \Delta + d$, $\Delta$ is the clock drift between the two nodes, and $d$ is the propagation delay. $B$, at local time $T3$, sends back an *acknowledgment* packet to $A$ containing its level number and the values of $T1$, $T2$, and $T3$. Finally, $A$ receives the *acknowledgment* packet at local time $T4$. Assuming that the clock drift and the propagation delay do not change in this small span of time, $A$ can calculate the clock drift and propagation delay as:

  $$\Delta = \frac{(T2-T1)-(T4-T3)}{2} \quad ; \quad d = \frac{(T2-T1)+(T4-T3)}{2}$$

  Knowing the drift, $A$ can correct its clock accordingly, so that it synchronizes to $B$.

  The root node initiates the synchronization phase by broadcasting a *time_sync* packet. Upon receiving this packet, nodes belonging to level 1 wait for a random amount of time before synchronizing with the root node. Randomization is used to avoid contention at the MAC layer of the communications protocol stack. Upon receiving an acknowledgment, these nodes adjust their clock to the

root node. Given that every level-2 node has at least one level-1 node in its neighbor set, nodes belonging to level-2 overhear this message exchange, upon which they wait for some random amount of time before initiating the message exchange with level-1 nodes. This use of randomization is to ensure that level-1 nodes have been synchronized before level-2 nodes begin the synchronization phase. A node sends back an acknowledgment to a *synchronization_pulse* only after it has synchronized itself, thereby ensuring that multiple levels of synchronization do not occur in the network. This process is carried out throughout the network, and consequently every node is eventually synchronized to the root node.

- **Special Cases:** The TPSN algorithm employs heuristics to handle special cases such as a sensor node joining an already established network, and nodes dying randomly. **Node joining:** This case concerns a sensor node that joins an already-established network (in particular, after the level-discovery phase is over) or that does not receive a *level_discovery* packet owing to MAC-layer collisions. In either case, it will not be assigned a level in the spanning tree. Every node, however, needs to be a part of the spanning tree so that it can be synchronized with the root. When a new node is deployed, it waits for some time to be assigned a level, failing which it timeouts and broadcasts a *level_request* message. Neighboring nodes reply to this request by sending their own level. On receiving the levels from its neighbors, the new node assigns itself a level one greater than the smallest level it has received, thereby joining the hierarchy. This is a *local* level-discovery phase. **Node dying:** If a sensor node dies randomly, it may lead to a situation where a level-$i$ node does not possess a level-$(i-1)$ neighbor. In this case, the level-$i$ node would not get back an acknowledgment to its *synchronization_pulse* packet, preventing it from synchronizing to the root node. As part of collision handling, a node retransmits its *synchronization_pulse* after some random amount of time. After a fixed number of retransmissions of the *synchronization_pulse*, a node assumes that it has lost all its neighbors on the upper level, and therefore broadcasts a *level_request* message to discover its level. Assuming the network is connected, the node will have at least one neighboring node and will receive a reply, after which The TPSN protocol considers four retransmissions to be a heuristic for deciding non-availability of an upper-level neighbor.

  If the node that dies is the root node, level-1 nodes will not receive an acknowledgment and hence will timeout. Instead of broadcasting a *level_request* packet, they run a leader-election algorithm to determine a new root node. The newly elected root node re-initiates the level-discovery phase.

# 3 Integer Clocks

UPPAAL is an integrated tool environment for the specification, simulation and verification of real-time systems modeled as networks of Timed Automata extended with data types. Primitive data types in UPPAAL are: clock, bounded integer, boolean and channel. There are also two compound types: array and structure, as in the C language. Timed Automata in UPPAAL are described as *processes*. A process consists of process parameters, local declarations, states and transitions. Process parameters turn into process-private constants when a process is instantiated. Local declarations describe the set of private variables to which a running process has access. States correspond to the vertices of a Timed Automaton in graphical form. Transitions are the edges connecting these vertices. A transition specifies a source and destination state, a guard condition, a synchronization channel, and updates to private or global data. A *system* in UPPAAL is the parallel composition of previously declared processes.

**Clocks in Timed Automata** A Timed Automaton is equipped with a finite set of *clocks*: variables whose valuation is a mapping from variable name to a time domain. In the theory of Timed-Automata [1], an assignment to a clock $x$ is restricted to $x \rightarrow R^{+}$, where $R^{+}$ is a non-negative real domain. That is, a

clock can only be assigned a constant value.

The UPPAAL model checker, which is based on the theory of Timed Automata, places similar restrictions on assignments to clocks. In UPPAAL, a clock is variable of type *clock* and can only be assigned the value of an integer expression. Moreover, UPPAAL transition guards are limited to conjunctions of simple *clock conditions* or *data conditions*, where a clock condition is a clock compared to an integer expression or another clock, and a data condition is a comparison of two integer expressions.

**Integer Clocks** Many real-time systems require more clock manipulations than UPPAAL can provide. For example, it is not possible in UPPAAL to read a clock value and send it as a message along a channel, as is required, for example, in time-sychronization protocols such as TPSN and FTSP [8]. The reason is that in UPPAAL, it is not possible to advance a clock by any arbitrary amount. Instead, when a transition occurs, the resulting clock values, with the exception of those that are reset to zero by the transition, are constrained only to lie within the region (convex polyhedron) formed by the conjunction of the invariant associated with the transition's source state and the transition guard. One therefore cannot in general determine the exact value of a clock when a particular transition occurs.

To address this problem, we introduce the notion of an *integer clock*, which is simply an integer variable whose value advances periodically. To enforce clock advance, we introduce a stand-alone *Universal Pulse Generator* (UPG) process which, at every time unit, broadcasts a signal (pulse) to all other UPPAAL processes in the system. All processes having integer clocks are required to catch this signal and increment their respective integer clocks. Using this mechanism, we mimic discrete-time clocks that possess an exact desired value when a Timed Automaton transition occurs. Because integer clocks are just integers whose valuations are under the control of the UPG, it is possible to directly perform arithmetic operations on them as needed.

To improve code readability, we introduce the new derived type *intclock* as follows: `typedef int intclock;` and the UPG process is given by:

```
broadcast chan time_pulse;
process universal_pulse_generator()
{   clock t;
    state S { t <= 1 };
    init S;
    trans
        S -> S { guard t == 1; sync time_pulse!; assign t = 0; }; }
```

Processes deploying integer clocks must, at every state, specify transitions (one per integer clock) that respond to `time_pulse` events. For example, the following transition specifies a perfect (zero-drift) integer clock `x`: `S -> S { sync time_pulse?; assign x = x+1;}`

In the following example, process `A` sends out its current clock value to another process `B` after waiting a certain amount of time after initialization. Without using integer clocks, `A`'s clock cannot be read.

```
chan AtoB;
meta int msg;
process A()
{
   const int wait = 3;
   meta intclock x;
   state INIT, SENT;
   init INIT;
   trans
       INIT -> INIT { sync time_pulse?; assign x = x+1; },/* time advance */
       SENT -> SENT { sync time_pulse?; assign x = x+1; }, /* time advance */
       INIT -> SENT { guard x >= wait; sync AtoB!; assign msg = x; };
}
```

The drawback of using integer clocks is an increase (albeit modest) in model complexity, through the addition of one process (the UPG) and the corresponding time-advance transitions, one for every state of every process. The additional code needed to implement integer clocks is straightforward, and can be seen as a small price to pay for the benefits integer clocks bring to the specification process.

**Time Drift** Time-synchronization protocols such as TPSN are needed in sensor networks due to the phenomenon of *time drift*: local clocks advancing at slightly different rates. Time drift can be modeled using integer clocks by allowing a process to either ignore or double-count a UPG pulse event after a certain number of pulses determined by the *time-drift rate* (TDR). Integer clocks with time drift are used in Section 4 to model a sensor network containing nodes with a positive TDR (pulse events are double-counted periodically), a negative TDR (pulse events are ignored periodically), and a TDR of zero (no time drift) for the root node. As we show in Section 5, the TPSN protocol achieves periodic network-wide resynchronization with the root node.

For example, an integer clock having a TDR of 5 means that its value is incremented by 2 every 5 time units (and by 1 every other time unit). An integer clock having a TDR of -10 means that its value remains unchanged every 10 time units. A TDR of 0 means that the integer clock strictly follows the UPPAAL unit time-advance rate. The UPPAAL code for implementing integer clocks with TDR-based time drift appears in function `local_time_advance()` of the TPSN specification; see Appendix A in the online version of the paper.

# 4 Uppaal Model of TPSN

In this section, we use integer clocks to model TPSN in UPPAAL. In particular, we use integer clocks for each node's local time. We still, however, rely fundamentally on UPPAAL's built-in clock type for other modeling purposes, including, for example, the modeling of message channel delays. We consider a sensor network whose initial topology is a ring of $N$ nodes indexed from 0 to $N-1$. Each node therefore has exactly two neighbors, and nodes are connected by communication channels. Node 0 is considered to be the root node. Note that the choice of a ring for the initial topology is not essential. The TPSN protocol constructs a spanning tree from the network topology, and the synchronization phase operates over this spanning tree.

Nodes and channels are modeled as UPPAAL processes. A node process maintains information about its neighbors, level in the spanning tree, parent, and local clock. Nodes send and receive packets through the channels processes. We cannot simply use UPPAAL's built-in `chan` type to model channels because we need to model the delay in the network, which is required in the protocol. Upon sending or receiving packets, a node switches its state and alter its data accordingly.

A node process takes three parameters, its id and the id of its two neighbors, and is declared in UPPAAL as follows: `process node(int[0,N-1] id, int[0,N-1] neighbor1, int[0,N-1] neighbor2)`

Channels also have ids. Channel $k$ represents the communication media surrounding node $k$, and therefore is only accessible to node $k$ and its neighbors.

**Using Integer Clocks** The TPSN protocol involves clock-value reading (a parent node sends its local time to a child node during the synchronization phase) and clock-value arithmetic (calculating $\Delta$ and $d$). We use integer clocks to model these operations. In a node process, we declare an integer clock as: `meta intclock local_time;`

A node process has seven states. There are therefore seven time-advance transitions in its specification; e.g. `initial -> initial { sync time_pulse?; assign local_time = local_time+1; }`

**Level-Discovery Phase** Three UPPAAL states are used to model the level-discovery phase: `initial`, `discovered`, `discovered-neighbors`. When the system starts, all nodes except the root enter the

initial state. The root goes directly to the discovered state, which indicates that a node has obtained its level in the hierarchy (zero for the root). After a certain delay, the root initiates the level-discovery phase by broadcasting a level_discovery message to its neighbors and then enters the discovered-neighbors state, means it has done the task to discover its neighbors' level. A node in the initial state enters the discovered state upon receiving a level_discovery message, or simply ignores this message if it is already in discovered or discovered-neighbors state. A node in the discovered state waits for some delay and then emits a level_discovery, changing its state to discovered-neighbors. As described in Section 2, this procedure will occur all over the network until eventually all nodes are in the discovered-neighbors state, indicating the entire network is level-discovered.

In our model, there are only two level-discovery transitions: from initial to discovered and from discovered to discovered-neighbors. Consider, for example, the former, which occurs upon receipt of a level_discovery message:

```
initial -> discovered {
    select i:int[0,N-1];
    guard level == UNDEF && neighbors[i];
    sync lev_recv[i]?;
    assign level = msg1[i]+1, parent = i;
},
```

If a node is not yet discovered (guard level == UNDEF) and the level_discovery message (through channel lev_recv[i]) is sent by its neighbor (guard neighbor[i]), then it sets its level to one greater than its neighbor's level (msg1[i]+1); i.e. this neighbor becomes its parent.

**Synchronization Phase** We consider state discovered-neighbors to be the initial state of the synchronization phase, and introduce four additional states: sync-ready, sync-sent, sync-received and synchronized. The root, which is the absolute standard of time, broadcasts a time_sync message to its neighbors, and then goes directly from discovered-neighbors to synchronized. The time_sync message notifies the root's children that they can now synchronize with their parent. The root's children then transit from discovered-neighbors to sync-ready (see Section 2). A node in the sync-ready state will send a synchronization_pulse to its parent and then enter sync-sent. The parent, which must be already synchronized, goes from synchronized to sync-received, where it will remain for a certain amount of delay. When the delay is over, it returns to the synchronized state and sends an acknowledgment message to the child. The child then adjusts its local clock using the information carried in these messages and goes to the synchronized state. All of these messages are actually broadcasted, which allows a child's child to overhear the synchronization_pulse message and turn itself into sync-ready. The state transition diagram for this phase is depicted in Figure 1.

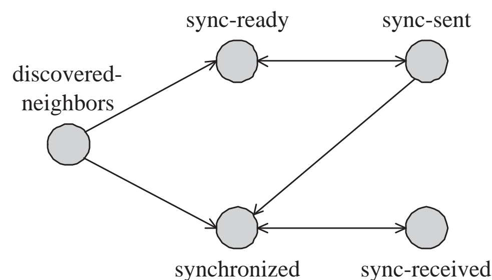



Figure 1: State Transition Diagram for the Synchronization Phase

For example, consider the transition a node makes from `sync-sent` to `synchronized` upon receipt of an `acknowledgment` message:

```
sync_sent -> synchronized {
    select i:int[0,N-1];
    guard id != ROOT_ID && i == parent;
    sync ack_recv[i]?;
    assign adjust_local_time();
}
```

This transition occurs if the node is not the root (`guard id != ROOT_ID`) and the incoming `acknowledgment` message is from its parent as determined during level-discovery (`guard i==parent`). The transition's affect is to adjust the node's local time (an integer clock) by calling function `adjust_local_time()`, which simply calculates the time drift $\Delta$ and adds $\Delta$ to `local_time`.

**Time Drift** TPSN addresses time drift in a sensor network by periodically performing resynchronization to ensure that each node's local clock does not diverge too much from the root. Since we have modeled local clocks using integer clocks, we cannot use arbitrary rational numbers for the TDR. Rather, we let the root advance its local clock at the pace of the UPG-generated pulse events and non-root nodes either skip or double-count a pulse after every certain number of pulses determined by an integer TDR. Since the TDR is fixed for each node, time drift appears linear in the long run. Figure 2 illustrates integer-clock-based time drift in a 5-node sensor network having respective TDRs $[0,-8,10,16,-10]$.

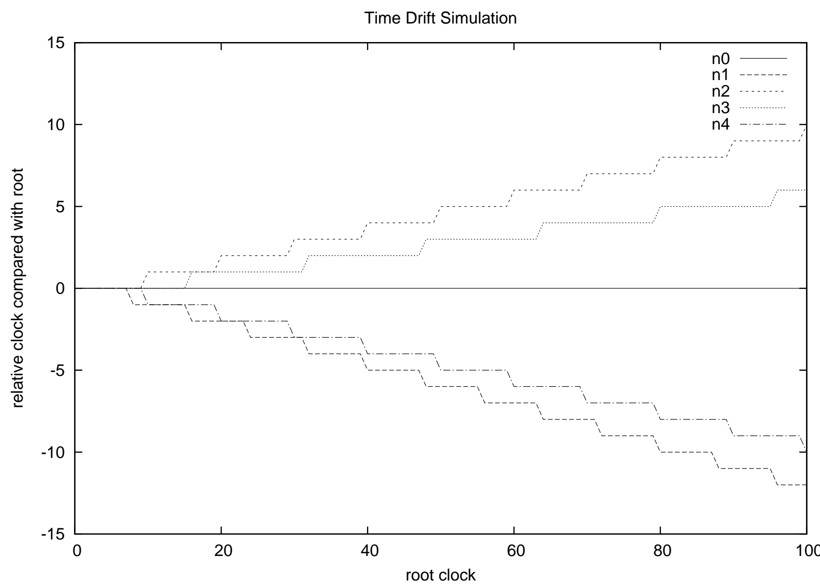



Figure 2: Integer-clock-based Time Drift

**Resynchronization** The resynchronization phase is almost identical to the synchronization phase except that the root will periodically jump from the `synchronized` state to the *discovered-neighbor* state. As in the synchronization phase, the root will soon re-enter the *synchronized* state and re-broadcast `time_sync`. We also let nodes in the `synchronized` state return to the `sync-ready` state on receiving `time_sync` or `synchronization_pulse`. Thus, another `time_sync` message will trigger another round of the synchronization phase, which is precisely the intent of resynchronization. The primary issue here is to set the resynchronization interval long enough to ensure synchronization has first completed.

**System** The system-level specification consists of $N$ nodes, $N$ channels, and the UPG processes:

```
n0 = node(0,1,2); n1 = node(1,0,3); n2 = node(2,0,4); n3 = node(3,1,4); n4 = node(4,2,3);
c0 = channel(0);  c1 = channel(1);  c2 = channel(2);  c3 = channel(3);  c4 = channel(4);
system n0, n1, n2, n3, n4, c0, c1, c2, c3, c4, universal_pulse_generator;
```

The choice of node-process parameters reflects the initial ring topology we are considering for the system. The system definition can be easily scaled using the process-parameter mechanism, as long as one carefully sets the neighbors of each node.

# 5 Simulation and Verification Results

We verified our model of the TPSN protocol given in Section 4, including the level-discovery, synchronization and periodic-resynchronization phases, using the UPPAAL model checker. We regard the root node as the node with the ideal clock (zero time drift) and all other nodes have (distinct and fixed) positive and negative TDRs. Since nodes are initially unsynchronized, each node is given a different initial clock value. Our model is also parameterized by the minimum and maximum node-response delay (20 and 40, respectively), channel delay (4), and minimum and maximum resynchronization interval (10 and 20, respectively), for which we have provided appropriate values.

**Simulation Results** We used the UPPAAL TRACER tool, and a resynchronization interval of 200 time units, to obtain simulation results for our model with periodic resynchronizations of the local clocks of all nodes. Figure 3 is obtained by extracting the local clock values of all nodes from a random simulation trace. This was accomplished by first saving the history of states to a trace file and then using the external program UPPAAL TRACER to convert the data in the trace file into to a readable form. Figure 2 of Section 4 was obtained similarly.

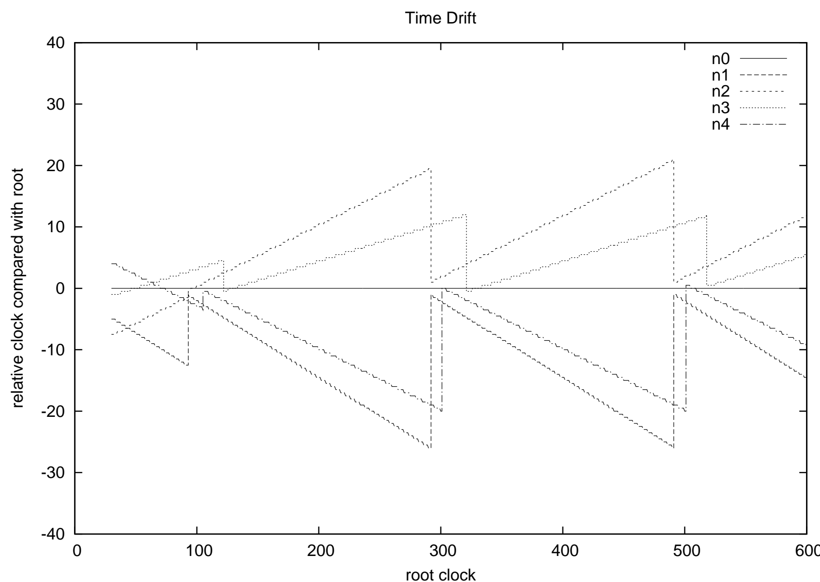



Figure 3: Integer Time Drift with a resynchronization interval of 200 time units

In Figure 3, plotting starts at time 30 since the root's starting clock value is set to 30. Between time 30 and 100, the network executes the level-discovery phase. The synchronization phase takes place

approximately between time 100 and 120. From there, all local clocks advance at their own rate, which, as can be seen, consists of four relative clocks each having a distinct TDR. At time 300, and then again at time 500, the system performs a resynchronization, which brings non-root local-clock values very close to 0. Notice that different nodes get synchronized at different times. Nodes 1 and 2 are synchronized at the same time because they are adjacent to the root. Nodes 3 and 4 are synchronized after them because of node response time and packet delay.

**Verification Results** We used UPPAAL to verify the following properties of our model. (Correctness properties can be specified in UPPAAL using a subset of the CTL temporal logic.)

**No Deadlock:** $A\,[\,]\,not\ \ deadlock$
Absence of deadlock.

**Synchronized:** $A\,[\,]\,(<>n_i.state == synchronized)$
All nodes enter a *synchronized* state at least once. This implies that the level-discovery and synchronization phases are correctly implemented.

**Relative Time Bounded:** $A\,[\,]\,abs(n_i.local_clock - n_0.local_clock) < X$
A node's local time relative to the root is bounded at all times ($X = 25$), which implies correct repeated resynchronization.

**Relative Time Close:** $A\,[\,]\,(<>abs(n_i.local_clock - n_0.local_clock) < Y)$
A node's relative time will always eventually get very close to 0 ($Y = 5$), another implication of correct repeated resynchronization.

We used UPPAAL to perform model checking of these properties on networks having $N = 3-5$ nodes. The corresponding verification results (CPU time and memory usage) are given in Table 1. In all cases, UPPAAL reported that the four properties are true. (The second and fourth properties, which are "infintiely often" properties, are verified only up to the time bound determined by the maximum UPPAAL integer value of 32,767.) All results are obtained on a Linux PC with a 2GHz Intel Pentium CPU and 2GB memory. UPPAAL's default settings were used.

| $N$ | No Deadlock | Synchronized | Relative Time Bounded | Relative Time Close |
|---|---|---|---|---|
| 3 | $0.61sec/21MB$ | $2.06sec/24MB$ | $0.62sec/21MB$ | $2.11sec/24MB$ |
| 4 | $6.5sec/22MB$ | $68.0sec/31MB$ | $6.7sec/22MB$ | $70.2sec/31MB$ |
| 5 | $6.1min/126MB$ | $214.9min/181MB$ | $6.3min/126MB$ | $236.4min/181MB$ |

Table 1: Verification Results: Time and Memory Usage

## 6 Related Work

In [6], the FTSP flooding-time synchronization protocol) for sensor networks has been verified using the SPIN model checker. Properties considered include *synchronization of all nodes to a common root node*, and *eventually, all nodes get synchronized*, for networks consisting of 2-4 nodes. In [4], UPPAAL has been used to verify a time-synchronization protocol for a time-division-multiple-access (TDMA) communication model. The aim of this protocol is to ensure that the clocks of nearby (neighboring) nodes are synchronized. UPPAAL has also been used to demonstrate the correctness of the LMAC medium access control protocol for sensor networks [11], and the the minimum-cost forwarding (MCF) routing protocol [12] for sensor networks [5].

*Time-triggered systems* are distributed systems in which the nodes are independently-clocked but maintain synchrony with one another. In [9], real-time verification of time-triggered protocols has been performed using a combination of mechanical theorem proving, bounded model-checking and SMT (satisfiability modulo theories) solving. Calendar automata, which use sparse-time constraints, are used in [2] for the modeling and verification of the fault-tolerant, real-time startup protocol used in the Timed Triggered Architecture [10].

# 7 Conclusions

We used the UPPAAL model checker for Timed Automata to obtain a number of critical verification results for the TPSN time-synchronization protocol for sensor networks. Clock-synchronization algorithms for sensor networks such as TPSN must be able to perform arithmetic on clock values to calculate clock drift and network propagation delays. They must be able to read the value of a local clock and assign it to another local clock. The introduction of the *integer clock* (with time drift) derived UPPAAL type greatly facilitated the encoding of these operations, as they are not directly supported by the theory of Timed Automata. We also used the UPPAAL TRACER tool to illustrate how integer clocks can be used to capture clock drift and resynchronization during protocol execution.

# References


[1] R. Alur and D. L. Dill. The theory of timed automata. *TCS*, 126(2), 1994.

[2] B. Dutertre and M. Sorea. Modeling and verification of a fault-tolerant real-time startup protocol using calendar automata. In *In FORMATS/FTRTFT*, pages 199–214, 2004.

[3] S. Ganeriwal, R. Kumar, and M. B. Srivastava. Timing-sync protocol for sensor networks. In *Proceedings of the 1st international conference on Embedded networked sensor systems (SenSys)*, pages 138–149. ACM, 2003.

[4] F. Heidarian, J. Schmaltz, and F. W. Vaandrager. Analysis of a clock synchronization protocol for wireless sensor networks. In *FM*, pages 516–531, 2009.

[5] W. Henderson and S. Tron. Verification of the minimum cost forwarding protocol for wireless sensor networks. In *IEEE Conference on Emerging Technologies and Factory Automation*, pages 194–201. IEEE Computer Society, 2006.

[6] B. Kusy and S. Abdelwahed. FTSP protocol verification using SPIN. Technical Report ISIS-06-704, Institute for Software Integrated Systems, May 2006.

[7] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1-2):134–152, 1997.

[8] M. Maróti, B. Kusy, G. Simon, and A. Lédeczi. The flooding time synchronization protocol. In *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems (SenSys)*, pages 39–49. ACM, 2004.

[9] L. Pike. Modeling time-triggered protocols and verifying their real-time schedules. In *Proceedings of Formal Methods in Computer Aided Design (FMCAD'07)*, pages 231–238. IEEE, 2007.

[10] W. Steiner and M. Paulitsch. The transition from asynchronous to synchronous system operation: An approach for distributed fault-tolerant systems. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS'02)*, page 329. IEEE Computer Society, 2002.

[11] L. van Hoesel and P. Havinga. A lightweight medium access protocol (LMAC) for wireless sensor networks: Reducing preamble transmissions and transceiver state switches. In *1st International Workshop on Networked Sensing Systems (INSS)*, pages 205–208, 2004.

[12] F. Ye, A. Chen, S. Lu, L. Zhang, and F. Y. A. Chen. A scalable solution to minimum cost forwarding in large sensor networks. In *IEEE 10th International Conference on Computer Communications and Networks*, pages 304–309, 2001.

# Can regulatory bodies expect efficient help from formal methods?

Eduardo R. López Ruiz
Onera
Toulouse, France
eduardo.lopez-ruiz@onera.fr

Michel Lemoine
Onera
Toulouse, France
michel.lemoine@onera.fr

**Abstract**

In the context of EDEMOI -a French national project that proposed the use of semiformal and formal methods to infer the consistency and robustness of aeronautical regulations through the analysis of faithfully representative models- a methodology had been suggested (and applied) to different (safety and security-related) aeronautical regulations. This paper summarizes the preliminary results of this experience by stating which were the methodology's expected benefits, from a scientific point of view, and which are its useful benefits, from a regulatory body's point of view.

## 1 Introduction

In order to safeguard civil aviation against accidental events (which is the concern of aviation safety) and against intentionally detrimental acts (which is the concern of aviation security), governments worldwide have imposed regulatory requirements upon the different aviation participants or, as they will be refer to in this paper, entity-classes[1].

Under the vigilant eyes of regulatory bodies, safety/security requirements are imposed on the entity-classes in order to (1) prevent the body of causes that may directly or indirectly lead these entity-classes into a hazardous operating/behavioral state, and/or (2) mitigate the consequences associated to such states.

However, in order to be effectual, the regulatory requirements need to be robust, consistent and pertinent. Indeed, their robustness ensures that the requirements exhaustively cover all the safety/security relevant scenarios within the regulation's domain of competence, or purview. Their consistency ensures that they will not be mutually contradictory or incompatible. And their pertinency ensures that they are relevant to enhancing aviation safety/security. Yet these three qualities can only be achieved through a complete understanding of the regulatory domain, and of the concerned (or participating) entity-classes, including their mutual interactions.

For this reason, regulatory bodies seek to fully identify all of the safety/security-concerned entity-classes that exist within their purview, including their relevant (universe of) states[2]. This information, along with their extensive practical and theoretical expertise, enables the regulatory bodies to adequately determine the preventative or mitigative measures that they should impose onto the entity-classes, in order to reduce their associated safety/security risks.

[1] This paper uses the term *entity-classes* to refer globally to the types of aviation participants upon which the regulatory requirement are imposed. The nature of these participants can range from the persons involved in civil aviation operations, to the objects that they use and the infrastructures that they employ. Some examples of an entity-class are: PASSENGER, FLIGHT CREW MEMBER, AIRCRAFT, AIRPORT, COCKPIT.

[2] An entity-class' *relevant universe of state* is the grouping of all of its pertinent possible states. In other words, the grouping of all the states with: *(a)* a direct or indirect effect on the overall safety and/or security level and *(b)* a possibility of occurrence greater than zero. An example of a pertinent and possible state for the FLIGHT CREW MEMBER entity-class is the incapacitated state. Indeed, safety regulations have identified that the in-flight pilot incapacitation scenario is a remote (4.5x10-7 per flight hour) but possible scenario with consequential impacts on safety. Therefore, regulators ensure that this scenario is taken into account by their regulations

This way, but rather unwittingly, regulatory bodies created a composite, abstract and subjective description of their purview, corresponding to their *Conceptual View of the Real World*. This conceptual view is in fact implicitly found within their regulations, and it affords them a manageable (although approximate) model of their regulated domain that helps them better grasp their regulatory domain's structure, correlations and interactions. It also serves them as an arbitrarily well-defined categorization that groups individual entities (*e.g.* passenger Alice and passenger Bob) into a uniquely defined sets of entities (*e.g.* the entity-class PASSENGER), upon which the regulatory requirements can then be affixed[3]. Indeed, through well-defined categorizations such as this one, the regulatory bodies can specifically denote the sets/subsets of entity-classes that are concerned with a specific regulatory requirement.

This means that the regulator's *Conceptual View of the Real World* is intimately tied to a regulation's innate quality because: *what can be expected of a regulation whose regulator has a distorted view of the Real World? Or, if this view is missing relevant elements and/or relations?*

Therefore, the regulator's *Conceptual View of the Real World* needs to be checked to ensure the validity of its assumptions and statements, but also the robustness (comprehensiveness) of their concern.

The figure shown below (Figure 1) elucidates on the notion of the *Conceptual View of the Real World* (Figure 1, ACV) and on how it influences the pragmatic aspects of the regulatory framework.

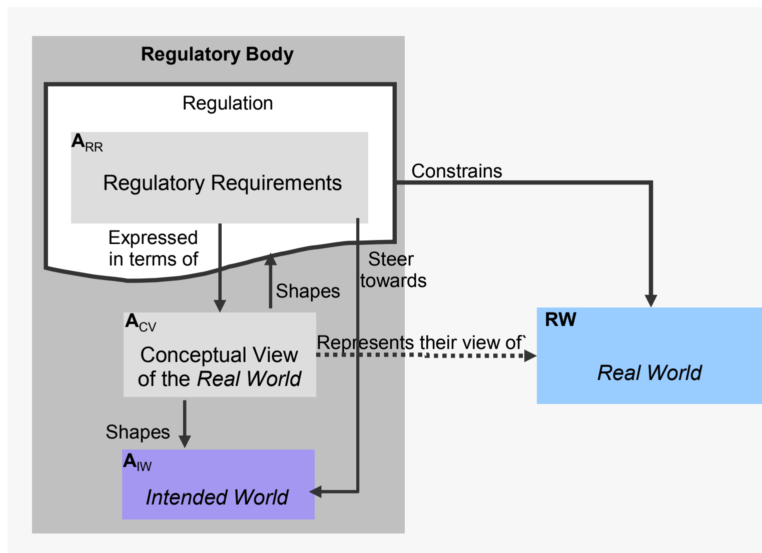



Figure 1: Conceptual Representation on the Operational description of the Regulatory Framework.

However, due to the (ever-growing) complexity of civil aviation and to its ever-changing state of affairs -brought on by *adjusting factors*[4]- regulators often find themselves overwhelmed by the challenge of continuously safeguarding such a dynamical and complexly interrelated industry.

Indeed, the process of 'developing/updating' the regulations undeniably entails the need to define and modify many of the fixed and context-dependent axioms and assumptions[5] that are the basis for the regulator's *Conceptual View*. This can result in the possible invalidation of some previously valid axioms and assumptions, as they change to reflect the new reality of the system.

[3]In accord with the principle that law must be general (*i.e.* impersonal).

[4]An *adjusting factor* is any operational, ideological and/or technological change whose introduction, into the civil aviation system, obliges a change in the contemporary regulations to preserve the appropriate overall functioning of the system.

[5]The axioms and assumptions are, respectively, the necessary truths and the generalized results that serve as the basis for the argumentation and/or inference of the regulatory requirements. They represent the *Domain Knowledge* shown in Figure 2

The *Conceptual View of the Real World* is therefore a crucial element in understanding the regulations, ensuring their pertinence and verifying their actuality. It is in this context that EDEMOI advocated the use of semiformal and formal methods and tools, to enhance the analysis (and consequently improve the inherent quality) of aeronautical safety and security regulations. In other words, EDEMOI sought to design an employable methodology[6] that would facilitate the assessment of regulatory requirements.

## 2 The Proposed Methodology

The underlying approach for the EDEMOI methodology was to determine the analogies that could be made between the domains where 'Requirements Analysis and Design in Systems and Software engineering'is successfully used, and 'civil aviation safety/security'. This was done with the objective of identifying the methods and tools used for these domains, to determine if they could be successfully implemented in the assessment of aviation safety/security requirements by way of a tailored methodology.

The EDEMOI methodology proposed enhancements to the rulemaking procedure currently used by civil aviation authorities. These enhancements included the incorporation of simulation and counterexample checking tools into the regulation's already established validation phase. This, with the objective of better ensuring the requirements' innate quality without any fundamental changes to the established rulemaking procedure.

The methodology is centered on a two-step approach (see Figure 2) involving two kinds of stakeholders: the *Aviation Authorities*, which establish regulations concerning civil aviation safety/security, and the *Model Engineers*, who translate these natural language documents into semiformal and formal models.

Given that regulations are rarely built from scratch, the methodology focused on studying/comparing the evolutions of existing regulations in search of possible regressions.

In the first step of this approach, a *Model Engineer* extracts the security goals and the imposed requirements from the original regulatory texts, and translates them into a semiformal (graphical) model that faithfully represents their structure and relations (while reducing the use of inherently ambiguous terms). Indeed, this model embodies the *Conceptual View of the Real World* that is partly implicit in the regulation (as discussed in Section 1). This *Graphical Model*, understandable by both kinds of stakeholders, is later revised and validated by the *Aviation Authority*, giving way to the methodology's second step, in which the *Model Engineer* performs a systematic translation of the semiformal model to produce a *Formal Model* that can be further analyzed.

This methodology was used in the formalization of various international and supranational aeronautical regulations[7] [7], [3] with discerning purviews and objectives. This allowed us to test the methodology, and conclude that it provides some interesting benefits. However, not all of them can be fully exploited by the regulatory bodies. The following sections provides a brief overview of the useful advantages of the methodology (Section 3) and of its shortcomings (Section 4).

[6]The EDEMOI methodology was not designed as a substitute for the practices currently employed in the assessment of regulatory requirements. On the contrary, it was designed to complement existing safety/security managements tools. Indeed, the traditional assessment methods such as the regulation's preliminary impact assessments, as well as the open (or closed) consultation periods before their enactment, are sufficiently effective in identifying the more common errors. However, and this was the motivation behind the EDEMOI project, new assessment techniques can allow the detection of the more elusive shortcomings and errors.

[7]Parts of the following regulations were formalized : ICAO's Annex 17 *(International Airport security)*, ICAO's Annex II *(Rules of Air)*, Regulation (EC) No. 2320/2002 *(European Airport Security)*, Directive 2008/114/EC *(Security of European Critical Infrastructure)* and Regulation (EC) No. 2096/2005 *(regarding the provision of Air Navigation Services)*.

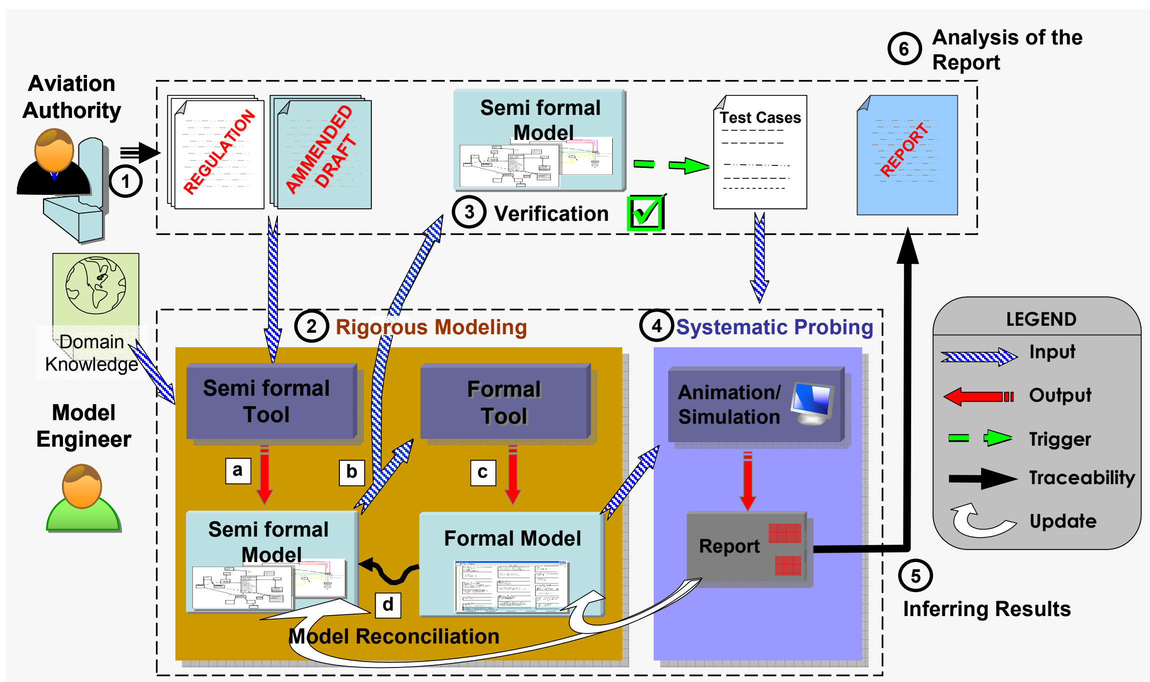



Figure 2: The Methodology proposed relies on the use of semiformal and formal tools to analyze regulatory requirements and enhance their innate quality.

# 3 The Methodology's useful benefits

## 3.1 Introduction

As was mentioned previously, aeronautical regulations are natural language documents that impose requirements onto real world entity-classes through a tacit abstract view of these entity-classes and of their environment. This was said to be the aviation authority's *Conceptual View of the Real World*. Formally specifying this *Conceptual View* yields a detailed documentation of its underlying assumptions and axiomatic base. Furthermore this formal specification can be accomplished while preserving a relatively high fidelity between the *Conceptual View* and the resulting models. Because, the *Conceptual View* is already an abstract and simplified model of the real world. This frees up the model engineer from the burden/responsibility of creating the models from zero.

Furthermore, formal tools can genuinely provide a sound basis for the comprehensive comparison of the abstract view and the real world it is supposed to embody. This, in order to detect diverging conceptions since a flawed view of the real world will suggest ineffectual or futile requirements. Also, using formal tools can facilitate the detailed understanding and analysis of the regulations' predicted implementation.

Figure 3 shows a side-by-side qualitative summary that synthesizes our experience and feedback with regards to the communicative aspects of semiformal and formal models for safety/security experts working within regulatory bodies. These results are part of the **A**ssessment of **L**egislative and **R**egulatory **R**equirements for **ATM** **S**ecurity (ALRRAS) project, a feasibility study into the use of computer science methods and tools to improve the assessment of pan-European aeronautical requirements. The eight criteria were selected based on the available literature [6], [4], [5], [2] and because practical knowledge of aviation safety/security identified them as facilitators of a regulation's quality.

With respect to these criteria, semiformal models (whose performance is shown in red diagonals)

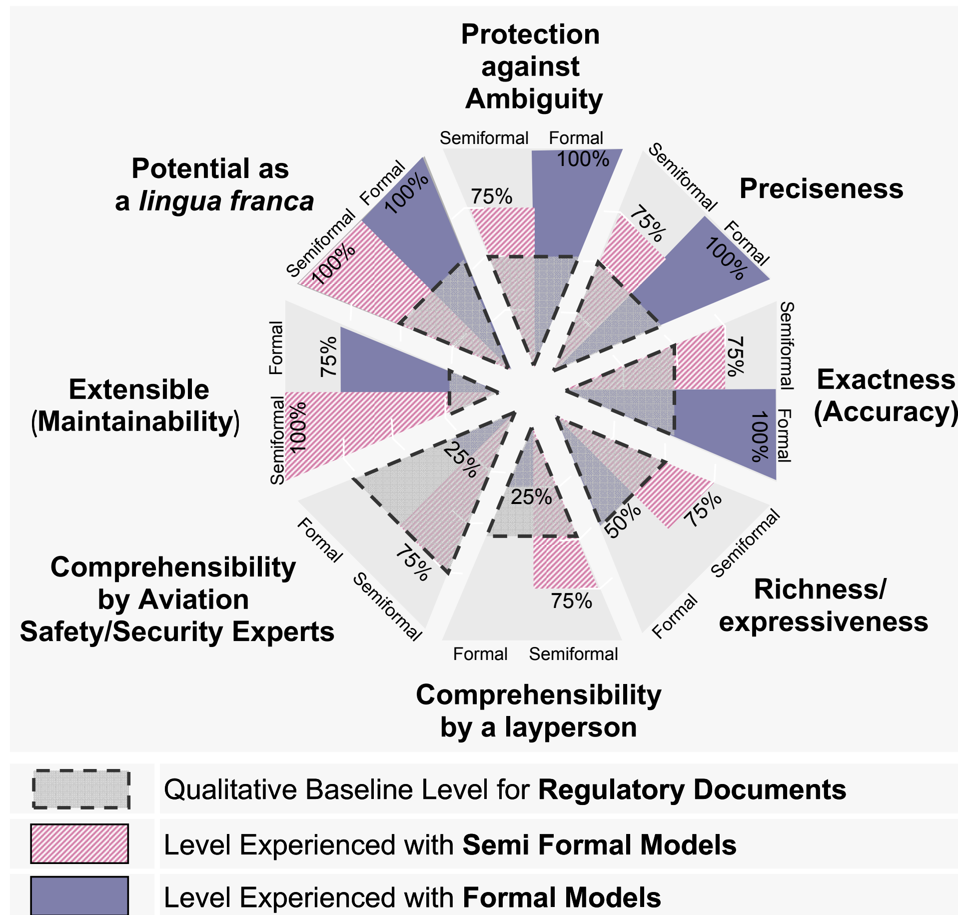



Figure 3: Graphical representation of a qualitative assessment of the communicative aspects of Aeronautical Regulatory Documents. Also shown, how these aspects are influenced by the complementary use of semiformal (in red diagonals) and formal models (in solid blue).

were found to be, overall, more consistent in their communicating capacities. Also, they have proven to be an enhancement for laypersons in terms of complementing their understanding of the aeronautical regulations.

Formal models, on the other hand (whose performance is shown in solid blue) stood out for their preciseness, exactness and their protection against ambiguity. However, as the qualitative values in the diagram need to be pondered for each modeling-type (to take into account the weight given to each criterion by regulatory bodies), the formal model's excellent protection against ambiguity was quickly overshadowed by its very poor comprehensibility by both laypersons and aviation safety/security experts.

Understandably, regulatory bodies, being less familiar with formal notations, need to be extremely cautious when validating/invalidating formal models, as they may be less able to detect specification errors[8]. This is especially true for regulators with a background in legal-studies, as they have been less exposed to such notations than their colleagues with an engineering background.

For this reason, the methodology foresaw the complementary use of semiformal and formal models (see Figure 2). Indeed, the process called for a semiformal model to be built directly from the regulatory text. This semiformal model was enriched with Object Constraint Language (OCL) expressions and -

[8]For the most part, regulators needed few instructions to be able to understand/interpret the semiformal notations

after having been analyzed and not invalidated[9] by the corresponding authority- used as the basis for a formal model.

A direct consequence of this methodology is that it provides a common semantic/understanding of the regulatory requirements that is (almost[10]) independent of any natural language. This is true for both the semiformal and formal models.

Also -and this is a byproduct of any specification process- the methodology can help identify imprecision, ambiguities and inconsistencies within the regulatory requirements.

More explicitly, we can say that formally specifying a regulation affords us:

- The ability to check where there are some holes in the regulations (a situation which is of a particular importance for security regulations!), and
- The ability to detect whether any regulatory amendments will introduce safety/security regressions.

### 3.2 Benefits of using Semiformal Models

The benefits provided by the semiformal models were, among others:

- Developing a common and understandable abstraction of the regulated domain and the participating entity-classes.
- Making the regulator's *Conceptual View of the Real World* explicit, enhancing the manipulation of their regulatory requirements.
- Providing a deeper linkage (traceability) between the different elements that comprise the regulatory framework.

The mixed semiformal/formal approach was indeed necessary, as was justified in figure 3. Through the use of UML-like notations, model engineers with a double competence in law and computer sciences can create semiformal models of the regulation's addressees. which convey their static (using class diagrams) and dynamic (using state-transition diagrams) properties. The utility of these models is that they can be used to represent, in a less ambiguous manner, how the regulatory requirements impact the entity-classes' structural and behavioral aspects. That is, the models can be used to show how the regulations reshape their static and dynamic properties.

Also, static models provide a deeper traceability between the different entity-classes and regulatory requirements. This allows a holistic view of normally separate (yet interrelated) regulations, and reconciles their *domain-oriented* structuring with their *class-entity oriented* implementation (*e.g.* all rules pertaining to the flight of an aircraft *vs.* all rules applicable to the FLIGHT CREW MEMBER entity-class). All of this helps facilitate the impact analysis of *adjusting factors*, as well as the regression analysis of the subsequent regulatory amendments.

Coupled with this, semiformal methods can help identify where the regulations are open for diverging interpretations. Moreover, these types of models can be used to create very rigorous specification of certain aspects of the entity-classes, particularly the dynamic diagrams (such as a state-transition diagram) where the use of guards can be very formal (more or less: IF ... THEN ... ELSE .. the presence of ELSE being mandatory for completeness reasons!). The advantage of this type of model is that the gain in formalism is not coupled with a loss in comprehensibility.

[9]When verifying the models, regulatory bodies are better positioned to detect errors within a specification -and therefore to 'invalidate' a model- rather than 'validating' them as being free of errors.

[10]The final models keep only remnants of the original regulatory text, preserving entity-class names, attribute and states descriptors.

### 3.3 Benefits of using Formal Models

Among the benefits provided by the use of formal models we can mention:

- It allows the regulation's meaning to be specified with more precision and exactness, helping the model engineer identify the more tricky areas (*i.e.* where the regulation can be interpreted differently).
- It affords the ability to automatically derive testing material.

Formal methods can be used to perform a comparative analysis between the Real World (Figure 1, RW) and the legislator's *Conceptual View of the Real World* (Figure 1, ACV). Using a formal models and tools for this analysis entails two benefits. Firstly, the act of formally specifying both the Real World and the regulator's *Conceptual View of the Real World* can be a preliminary way of identifying their differences/incongruities. Secondly, the formal specifications can be put through a comprehensive comparative analysis that is not possible by other means.

Much like the semiformal models, formal models also help identify some areas where the regulation may be interpreted differently, but since they allow the regulation's meaning to be better specified, they undoubtedly help the model engineer identify more clearly those areas where the regulation can be interpreted differently but also help them make sense of these tricky parts. Indeed, revisiting the semiformal model after having developed the formal one allowed us to make significant improvements in the semiformal model, particularly in terms of simplifying the model, but also by helping identify specification errors.

This was the case during the formal specification of a European airport security regulation, where a subtle language lapse in one of its articles[11] was discovered only after it had been formally specified. It had gone undetected in the original regulation, its eleven different language translations, and in its first semi-formal model. The article establishes the conditions (and limits) by which an airport can be labeled as 'small', and therefore be derogated from applying the stringent (and expensive) security standards enforced at larger airports. But, as shown in figure 4, the original text version alleviated only a fraction of the small airports it was supposed to exempt.

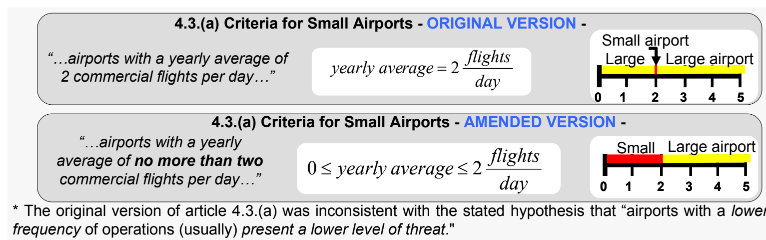



Figure 4: During the formal specification of Regulation (EC) 2320/2002, a subtle language lapse was identified. Although this wording error did not have a negative effect in terms of security -as it made the regulatory requirements more restrictive than originally intended-, it did have an economical impact on those small airports that where technically considered as large. The figure above transcribes the original regulatory requirement (on top) and its amended version (at the bottom). Also, the requirement is represented in two additional ways. In the center, as a mathematical expression. At the far right, as a one-dimensional graph.

[11] Regulation (EC) No. 2320/2002. Article 4.3.(a) Criteria for Small Airports

It is undoubtedly clear that formalization helps regulatory bodies to better understand and check their regulations from a technical point of view. But, what are the real uses and advantages that will result from this methodology?

# 4 The Methodology's shortcomings

As mentioned previously (Section 2), the methodology has both positive and negative aspects that need to be weighed, in order to define its utility as a tool to enhance the analysis of aeronautical safety and security regulations. This section will discuss its most consequential 'faults' or shortcomings.

From the previous section (Section 3), it is undoubtedly clear that formalizing regulatory requirements helps (from a technical point of view) regulatory bodies better understand their regulations by providing them with supplementary insight of their regulated domain and concerned entity-classes. However, it is not so clear from a practical standpoint.

Indeed lawyers are experts in law but they have a hard time understanding or, even more, developing such formal models [8]. Therefore, there will always be a need for a model engineer to develop the models. But, even when a model engineer develops a formal model of the regulations, the lawyers are unable to directly validate it, as they have a hard time understanding the notation. This leaves little use for such models. However there is a work-around to this problem. An alternative 'validation' solution is to animate/simulate the formal model in order to indirectly validate it. This indirect validation is done by comparing the results of the scenarios animatied/simulated, with the expected results of their actual implementation. The disparities between both results becoming the focus area for a detailed revision. However, this alternative solution also entails many difficulties, as regulations are very abstract texts, impossible to animate/simulate 'as is'. As shown in the following figure (Figure 5), regulatory texts need to be complemented by various other sources in order to have a model capable of being animated/simulated. Indeed, regulations need to be more or less stable through time -to ensure stakeholder's awareness of their obligations-, and the best way for ensuring this stability is for their regulatory requirements to be written using broad and general statements. Nevertheless other non-mandatory documents such as guidance material, industrial best practices, and standard procedures can help fill in the gaps between the regulation's abstract text and its detailed description, thereby enabling its animation/simulation.

For instance, the following regulatory requirement[12] concisely imposes that each country shall screen their originating passengers: *4.4.1 Each Contracting State shall establish measures to ensure that originating passengers of commercial air transport operations and their cabin baggage are screened prior to boarding an aircraft departing from a security restricted area.*

This text could lead to a very simple binary animation/simulation of the passenger screening which would be interesting if this where the first time the regulation is being enacted, to test its basic logic. However, since this requirement has been around since 1975, the requirement has to be complemented by its associated guidance material and by integrating the domain knowledge and best practices, to produce a more complete animation/simulation of the same process, and try to find the more elusive errors.

The fact that lawyers cannot easily understand formal models entails another problem. Since the formal models cannot be directly validated by the regulatory bodies, there will never be a benchmark formal specification of the Real World! Any model-to-model comparison (such as the one between the Real World and the legislator's *Conceptual View of the Real World*) will only provide a **relative assertion** into their validity. In fact, since there is no single 'valid' model to which others can be compared, all that can be expected from a model-to-model comparison is a measure of compatibility among the compared models, without any clear reference into which one of the discerning models is preferable.

[12] ICAO - Annex 17. Eighth Edition, 11th Amendment. Measures relating to passengers and their cabin baggage.

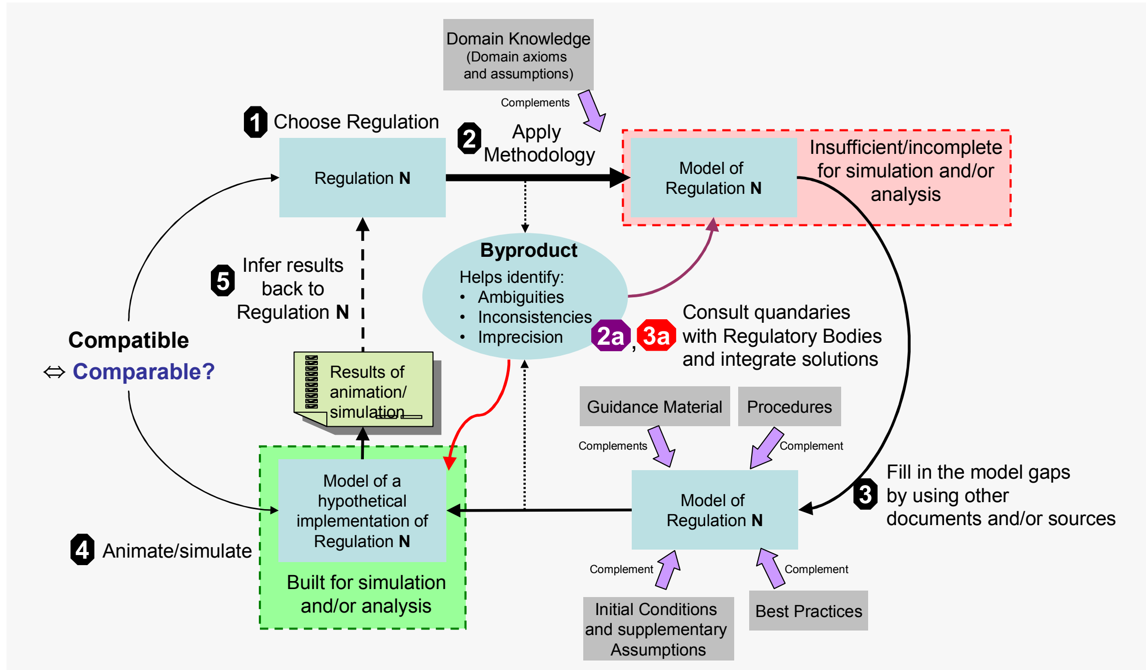



Figure 5: Regulations are not comprehensive sources of information so their modeling will not be able to produce an animatable/simulatable model. These models have to be completed through other sources in order to produce a 'runnable' model.

Nevertheless, one can look at the 'half-full' part of the glass and say that: even though the comparisons will only be relative, the disparities/incongruities between the compared models will help legislators by giving them a focus area or some starting points for their conventional validation process. And, in the end, this could lead to a more concrete and/or accelerated validation process (*i.e.* improve the assertiveness and reactiveness of the process).

Finally, the methodology is necessarily a collaborative modeling process, as it requires the regulatory body to validate the models produced by the model engineer. This can be done through a cross-reading of the semiformal models. Unfortunately, this does nothing to improve the quality of the validation process currently used[13]). This is because the validation process is still exposed to erroneous assessments -a false appreciation of the model that leads it to be validated as a faithful representation of the regulations. Nevertheless, an extensive traceability between the regulations and the produced models should strongly limit this situation.

# 5 Conclusion

The regulator's *Conceptual View of the Real World* is a crucial element for understanding the regulations, ensuring their pertinence and verifying their actuality. As such, it needs to be made explicit and checked to ensure the validity of its assumptions, of its statements and the robustness (comprehensiveness) of their concern.

Through the formalization of various international and supranational aeronautical regulations we have concluded that, in order to achieve this, it should be mandatory to integrate the use of semiformal methods into the current rulemaking process! Indeed, as mentioned in Section 3.2, semiformal methods

[13]For more information concerning the current rulemaking process see [1]

that have been enriched with OCL expressions allow regulatory bodies to 'master the complexity' of their regulations, by providing them with a comprehensible, structured and maintainable representation of their *Conceptual View of the Real World*, where the entity-classes and regulatory requirements are interlinked. This last point is very important as it promotes the holistic analysis/view of the regulations, and facilitates their regression analysis in case of amendments.

Granted, formal methods could also contribute to the accomplishment of these improvements, however their low comprehensibility by regulators (see Figure 3) means that they can only be used 'behind the scenes', to help disambiguate the most tricky elements/parts of the regulations before presenting them to the regulatory authorities via semiformal models. Otherwise, one must consider the costs associated to (and the time consumed in) training regulators in the use/utilization of formal notations. These costs, weighed against the foreseen benefits, have convinced us that, presently, this alternative is not a worthwhile enterprise.

Nevertheless, if one decides to undertake this course, and adopt formal methods as the primary tool for assessing civil aviation regulations, they should not undermine the importance of having the model validated by the aviation authority. For this, they should opt for a validation process involving a third trusted party. This party, external to the civil aviation authority and to the model engineers could be composed of engineers with a double-competency in civil aviation regulations and in formal methods. This double-competency would allow them to validate the formal models and help with the analysis of the regulations.

## References


[1] European Aviation Safety Agency (EASA) Management Board. Decision of the Management Board Amending and Replacing Decision 7-03 Concerning the Procedure to be Applied by the Agency for the Issuing of Opinions, Certification Specifications and Guidance Material ('RULEMAKING PROCEDURE'), 2007. http://www.easa.europa.eu/ws_prod/g/doc/About_EASA/Manag_Board/2007/MBDecision08-2007amendingrulemakingprocedure.pdf.

[2] Asaf Degani. On the Design of Flight-Deck Procedures. Technical Report NASA Contract NCC2-327 and NCC2-581, Georgia Institute of Technology, 1994. http://ti.arc.nasa.gov/m/profile/adegani/Flight-Deck_Procedures.pdf.

[3] Regine Laleau et al. Adopting a situational requirements engineering approach for the analysis of civil aviation security standards. *The Journal of Software Process: Improvement and Practice (S.P.I.P.)*, 11(5):487–503, July 2006. http://dx.doi.org/10.1002/spip.291.

[4] Federal Aviation Administration (FAA). *FAA Writing Standards. Order 1000.36*. Federal Aviation Administration (FAA), 2003. http://www.faa.gov/documentlibrary/media/order/branding_writing/order1000_36.pdf.

[5] United States Government Accountability Office (GAO). *System Safety Approach Needs Further Integration into FAAs Overisght of Airlines*. United States Government Accountability Office (GAO), 2005. http://www.gao.gov/cgi-bin/getrpt?GAO-05-726.

[6] Plain English Network (PEN). *Writing User-Friendly Documents: A Handbook for FAA Drafters*. Federal Aviation Administration (FAA), 2000.

[7] Eduardo Rafael López Ruiz. Formal Specification of Security Regulations: The Modeling of European Civil Aviation Security, 2006.

[8] Eduardo Rafael López Ruiz and Béatrice Trigeaud. La Modélisation Informatique des Règles de Droit Relatives à la Sûreté du Transport Aèrien International. *Annuaire Français de Droit International (A.F.D.I.)*, 53:672–696, 2007.

# Synthesis Of Greedy Algorithms Using Dominance Relations

Srinivas Nedunuri
Dept. of Computer Sciences
University of Texas at Austin
nedunuri@cs.utexas.edu

Douglas R. Smith
Kestrel Institute
smith@kestrel.edu

William R. Cook
Dept. of Computer Sciences
University of Texas at Austin
cook@cs.utexas.edu

**Abstract**

Greedy algorithms exploit problem structure and constraints to achieve linear-time performance. Yet there is still no completely satisfactory way of constructing greedy algorithms. For example, the Greedy Algorithm of Edmonds depends upon translating a problem into an algebraic structure called a matroid, but the existence of such a translation can be as hard to determine as the existence of a greedy algorithm itself. An alternative characterization of greedy algorithms is in terms of dominance relations, a well-known algorithmic technique used to prune search spaces. We demonstrate a process by which dominance relations can be methodically derived for a number of greedy algorithms, including activity selection, and prefix-free codes. By incorporating our approach into an existing framework for algorithm synthesis, we demonstrate that it could be the basis for an effective engineering method for greedy algorithms. We also compare our approach with other characterizations of greedy algorithms.

## 1 Introduction

A greedy algorithm repeatedly makes a locally optimal choice. For some problems this can efficiently lead to a globally optimal solution. Edmonds [Edm71] characterized greedy algorithms in terms of *matroids*. In 1981, Korte and Lovasz generalized matroids to define *greedoids* [KLS91]. The question of whether a greedy algorithm exists for a particular problem reduces to whether there exists a translation of the problem into a matroid/greedoid. However, the characterization does not provide any guidance on how to construct this translation. In addition, there are problems that have greedy solutions, such as Activity Selection and Prefix-free Codes, [CLRS01], that do not seem to fit within the matroid/greedoid model. A number of other attempts have been made to characterize greedy algorithms, [BM93, Cur03, Cha95, HMS93] but the challenge in all of these approaches is establishing the conditions required for a given problem to meet that characterization. Thus, there has been very little work in helping a developer actually construct greedy algorithms.

An alternative approach to constructing algorithms is to take a very general program schema and specialize it with problem-specific information. The result can be a very efficient algorithm for the given problem, [SPW95, SW08, NC09]. One such class of algorithms, Global Search with Optimality (GSO) [Smi88], operates by controlled search, where at each level in the search tree there are a number of choices to be explored. We have recently [NSC10] been working on axiomatically characterizing a class of algorithms, called Greedy Global Search (GGS), that specializes GSO, in which this collection of choices reduces to a single locally optimal choice, which is the essence of a greedy algorithm. Our characterization is based on dominance relations [BS74], a well-known technique for pruning search spaces. However, this still leaves open the issue of deriving a greedy dominance relation for a given problem, which is what we address in this paper. We start with a specialized form of the dominance relation in [NSC10] which is easier to work with. Our contribution is to introduce a tactic which enables the two forms of dominance to be combined and also show how the main greediness axiom of GGS theory can be constructively applied. We have used this approach to derive greedy solutions to a number of problems, a couple of which are shown in this paper.

Although our derivations are currently done by hand, we have expressed them calculationally as we hope to eventually provide mechanical assistance for carrying them out. In addition to providing a

**Algorithm 1** Program Schema for GGS Theory

```
--given x:D satisfying i returns optimal (wrt. cost fn c) z:R satisfying o(x,z)
function solve :: D -> {R}
solve x =
    if Φ(x,r̂0(x) ∧ i(x)) then (gsolve x r̂0(x) {}) else {}

function gsolve :: D -> {R̂} -> {R} -> {R}
gsolve x space soln =
    let gsubs = {s | s∈subspaces x space ∧ ∀ss ∈ subspaces x space,s δx ss}
        soln' = opt c (soln ∪{z | χ(z,space) ∧ o(x,z)})
    in if gsubs = {} then soln'
       else let greedy = arbPick gsubs in gsolve x greedy soln'

function opt :: ((D,R) -> C) -> {R̂}-> {R̂}
opt c {s} = {s}
opt c {s,t} = if c(x,s)>c(x,t) then {s} else {t}
function subspaces :: D -> R̂-> {R̂}
subspaces x r̂ = {ŝ: ŝ⋖x r̂∧Φ(x,ŝ)}
```

process for a developer to systematically construct greedy algorithms, we also believe that our approach has a potential pedagogical contribution. To that end, we show that, at least in the examples we consider, our derivations are not only more systematic but also more concise than is found in algorithms textbooks.

# 2 Background

## 2.1 Greedy Global Search (GGS) Theory

GSO, the parent class of GGS, is axiomatically characterized in [Smi88]. The definition contains a number of type and abstract operators, which must be instantiated with problem specific information. GGS [NSC10] specializes GSO with an additional operator, and axioms. The operators of GGS theory, which we informally describe here, are named $\widehat{r}_0, \chi, \in, \lessdot, \delta, \Phi$ along with an additional type $\widehat{R}$ ; they parametrize the program schema associated with the GGS class (Alg. 1). $D, R, C, o$ and $c$ come from the problem specification which is described in Section 2.2.

Given a *space* of candidate solutions (also called a *partial solution*) to a given problem (some of which may not be optimal or even correct), a GGS program partitions the space into *subspaces* (a process known as *splitting*) as determined by a subspace relation $\lessdot_x$. Of those subspaces that pass a *filter* (a predicate $\Phi$ which is some weakened efficiently evaluable form of the correctness condition, $o$ ) one subspace is greedily chosen, as determined by a dominance relation $\delta_x$, and recursively searched[1]. If a predicate $\chi$ on the space is satisfied, a solution is extracted from it. If that solution is correct it is compared with the best solution found so far, using the cost function $c$. The process terminates when no space can be further partitioned. The starting point is an initial space, computed by a function $\widehat{r}_0$, known to contain all possible solutions to the given problem. The result, if any, is an optimal solution to the problem. Because spaces can be very large, even infinite, they are rarely represented extensionally, but instead by a *descriptor* of some type $\widehat{R}$. A relation $\in$ determines whether a given solution is contained in a space.

The process of algorithm development using this theory consists of the following steps:

1. Formally specify the problem. Instantiate the types of GGS theory.
2. Develop a domain theory (usually associativity and distributivity laws) for the problem.

[1] Such an approach is also the basis of branch-and-bound algorithms, common in AI

3. Instantiate the abstract search-control operators in the program schema. This is often done by a mechanically-assisted constructive theorem proving process called *calculation* that draws on the domain theory formulated in the previous step

4. Apply further refinements to the program such as finite differencing, context simplification, partial evaluation, and datatype refinement to arrive at an efficient (functional) program.

Our focus in this paper is on Step 3, in particular how to derive the dominance relation, but we also illustrate Steps 1 and 2. Step 4 is not the subject of this paper. Specware [S], a tool from Kestrel Institute, provides support for carrying out such correctness preserving program transformations. Details can be found in [Smi90].

## 2.2 Specifications

A *problem specification* (Step 1) is a 6-tuple $\langle D,R,C,i,o,c\rangle$, where $D$ is an input type (the type of the problem instance data), $R$ an output type (the type of the result), $C$ a cost type, $i : D \rightarrow Boolean$ is precondition defining what constitutes a valid input, $o : D \times R \rightarrow Boolean$ is an *output* or *post condition* characterizing the relationship between valid inputs and valid outputs. The intent is that an algorithm for solving this problem will take any input $x : D$ that satisfies $i$ and return a *solution* $z : R$ that satisfies $o$ (making it a *feasible* solution) for the given $x$. Finally $c : D \times R \rightarrow C$ is a *cost criterion* that the result must minimize. When unspecified, $C$ defaults to $Nat$ and $i$ to $true$. A *constraint satisfaction problem* is one in which $D$ specifies a set of variables and a value set for each variable, $R$ a finite map from values to variables, and $o$ requires at least that each of the variables be assigned a value from its given value-set in a way that satisfies some constraint.

## 2.3 Dominance Relations

A dominance relation provides a way of comparing two spaces in order to show that one will always have a cheaper best solution than the second. The first one is said to *dominate* the second, and the second can be eliminated from the search. Dominance relations have a long history in operations research, [BS74, Iba77]. For our purposes, let $\widehat{z}$ be a partial solution in some type of partial solutions $\widehat{R}$, and let $\widehat{z} \oplus e$ be a partial solution obtained by "extending" the partial solution with some extension $e : t$ for some problem-specific type $t$ using an operator $\oplus : \widehat{R} \times t \rightarrow \widehat{R}$. The operator $\oplus$ has some problem-specific definition satisfying $\forall z \cdot z \in \widehat{z} \oplus e \Rightarrow z \in \widehat{z}$. Lift $o$ up to $\widehat{R}$ by defining $o(x,\widehat{z}) = \exists z \cdot \chi(z,\widehat{z}) \wedge o(x,z)$. Similarly, lift $c$ by defining $c(x,\widehat{z}) = c(x,z)$ exactly when $\exists! z \cdot \chi(z,\widehat{z})$. Then

**Definition 1.** *Dominance* is a relation $\delta \subseteq D \times \widehat{R}^2$ such that:

$$\forall x,\widehat{z},\widehat{z}' \cdot \delta(x,\widehat{z},\widehat{z}') \Rightarrow (\forall e' \cdot o(x,\widehat{z}' \oplus e') \Rightarrow \exists e \cdot o(x,\widehat{z} \oplus e) \wedge c(x,\widehat{z} \oplus e) \leq c(x,\widehat{z}' \oplus e'))$$

Thus, $\delta(x,\widehat{z},\widehat{z}')$ is sufficient to ensure that $\widehat{z}$ will always lead to at least one feasible solution cheaper than any feasible solution in $\widehat{z}'$. For readability, $\delta(x,\widehat{z},\widehat{z}')$ is often written $\widehat{z}\delta_x\widehat{z}'$. Because dominance in its most general form is difficult to demonstrate, we have defined a stronger form of dominance which is easier to derive. This stronger form of dominance is based on two additional concepts: Semi-congruence and extension dominance, which are now defined.

**Definition 2.** *Semi-Congruence* is a relation $\rightsquigarrow \subseteq D \times \widehat{R}^2$ such that

$$\forall x, \forall e, \widehat{z}, \widehat{z}' \cdot \rightsquigarrow (x,\widehat{z},\widehat{z}') \Rightarrow o(x,\widehat{z}' \oplus e) \Rightarrow o(x,\widehat{z} \oplus e)$$

That is, semi-congruence ensures that any feasible extension of $\widehat{z}'$ is also a feasible extension of $\widehat{z}$. For readability, $\rightsquigarrow (x,\widehat{z},\widehat{z}')$ is written $\widehat{z} \rightsquigarrow_x \widehat{z}'$.

And

**Definition 3.** *Extension Dominance* is a relation $\widehat{\delta} \subseteq D \times \widehat{R}^2$ such that

$$\forall x, e, \widehat{z}, \widehat{z}' \cdot \widehat{\delta}(x, \widehat{z}, \widehat{z}')' \Rightarrow o(x, \widehat{z} \oplus e) \wedge o(x, \widehat{z}' \oplus e) \Rightarrow c(x, \widehat{z} \oplus e) \leq c(x, \widehat{z}' \oplus e)$$

That is, extension dominance ensures that one feasible completion of a partial solution is no more expensive than the same feasible completion of another partial solution. For readability, $\widehat{\delta}(x, \widehat{z}, \widehat{z}')$ is written $\widehat{z}\,\widehat{\delta}_x\,\widehat{z}'$. Note that both $\rightsquigarrow_x$ and $\widehat{\delta}_x$ are pre-orders. The following theorem and proposition show how the two concepts are combined.

**Theorem 2.1.** *Let $c^*$ denote the cost of the best feasible solution in a space. If $\rightsquigarrow$ is a semi-congruence relation, and $\widehat{\delta}$ is an extension dominance relation, then*

$$\forall x, \widehat{z}, \widehat{z}' \cdot \widehat{z}\widehat{\delta}_x\widehat{z}' \wedge \widehat{z} \rightsquigarrow_x \widehat{z}' \Rightarrow c^*(x, \widehat{z}) \leq c^*(x, \widehat{z}')$$

*Proof.* See Appendix □

It is not difficult to see that $\widehat{\delta}_x \cap \rightsquigarrow_x$ is a dominance relation.The following proposition allows us to quickly get an extension dominance relation for many problems. We assume we can apply the cost function to partial solutions.

**Proposition 1.** *If the cost domain $C$ is a numeric domain (such as Integer or Real) and $c(x, \widehat{z} \oplus e)$ can be expressed as $\widehat{c}(x, \widehat{z}) + k(x, e)$ for some functions $\widehat{c}$ and $k$ then $\widehat{\delta}_x$ where $\widehat{z}\,\widehat{\delta}_x\,\widehat{z}' = \widehat{c}(x, \widehat{z}) \leq \widehat{c}(x, \widehat{z}')$ is an extension dominance relation.*

*Proof.* See Appendix □

In addition to the dominance requirement from Theorem 2.1, there is an additional condition on $\delta$, [NSC10]:

$$i(x) \wedge (\exists z \in \widehat{r} \cdot o(x, z)) \Rightarrow$$
$$(\exists z^* \cdot e(z^*, \widehat{r}) \wedge o(x, z^*) \wedge c(x, z^*) = c^*(\widehat{r})) \vee \exists \widehat{s}^* \lessdot_x \widehat{r}, \forall \widehat{s} \lessdot_x \widehat{r} \cdot \widehat{s}^*\, \delta_x\, \widehat{s} \quad (2.1)$$

This states that, assuming a valid input $x$, an optimal feasible solution $z^*$ in a space $\widehat{r}$ that contains feasible solutions must be immediately extractable or a subspace $\widehat{s}^*$ of $\widehat{r}$ must dominate all the subspaces of $\widehat{r}$.

## 2.4 Notation

The following notation is used throughout the paper: $\mapsto$ is to be read as "instantiates to". A type declaration of the form $\{a : T, b : U, \cdots\}$ where $T$ and $U$ are types denotes a product type in which the fields are accessed by $a, b, \ldots$ using a "dot" notation $o.a, o.b$, etc. An instance of this type can be constructed by $\{a = v, b = w, \ldots\}$ where $v, w, \cdots$ are values of type $T, U, \cdots$ resp. The notation $o\{a_i = v, a_j = w, \ldots\}$ denotes the object identical to $o$ except field $a_i$ has the value $v$, $a_j$ has $w$, etc. $[T]$ is the type of lists of elements of type $T$, $as_i$ accesses the $i$th element of a list $as$, $[a]$ constructs a singleton list with the element $a$, $[a \mid as]$ creates a list in which the element $a$ is prefixed onto the list $as$, and $as{+\!\!+}bs$ is the concatenation of lists $as$ and $bs$, $as - bs$ is the list resulting from removing from $as$ all elements that occur in $bs$. *first* and *last* are defined so that for a non-empty list $as = \mathit{first}(as){+\!\!+}[\mathit{last}(as)]$. Similarly, $\{T\}$ is the type of sets of elements of type $T$. $T \rightarrowtail U$ is the type of finite maps from $T$ to $U$.

# 3 A Process For Deriving Greedy Algorithms

We first illustrate the process of calculating $\rightsquigarrow$ and $\widehat{\delta}$ by reasoning backwards from their definitions on a simple example.

**Example 1.** Activity Selection Problem [CLRS01]

> Suppose we have a set $S = \{a_i, a_2, \ldots, a_n\}$ of $n$ proposed activities that wish to use a resource, such as a lecture hall, which can be used by only one activity at a time. Each activity $a_i$ has a start time $s_i$ and finish time $f_i$ where $0 \le s_i < f_i < \infty$. If selected, activity $a_i$ takes place in the half-open time interval $[s_i, f_i)$. Activities $a_i$ and $a_j$ are compatible if the intervals $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap. The activity selection problem is to select a maximum-size subset of mutually compatible activities.

Step 1 is to formally specify the problem. The input is a set of activities and a solution is subset of that set. Every activity is uniquely identified by an *id* and a start time ($s$) and finish time ($f$). The output condition requires that activities must be chosen from the input set, and that no two activities overlap. For convenience we define a precedence operator $\preceq$ :

$$
\begin{array}{rcl}
D & \mapsto & \{Activity\} \\
 & & Activity = \{id : Nat, s : Nat, f : Nat\} \\
R & \mapsto & \{Activity\} \\
o & \mapsto & \lambda(x,z) \cdot noOvp(x,z) \wedge z \subseteq x \\
 & & noOvp(x,z) \doteq \forall i, j \in z \cdot i \neq j \Rightarrow i \preceq j \vee j \preceq i \\
 & & i \preceq j = i.f \le j.s \\
c & \mapsto & \lambda(x,z) \cdot \|z\|
\end{array}
$$

In order to devise a greedy algorithm, the question is what should the basis be for an optimal local choice? Should we always pick the activity that starts first? Or the activity that starts last? Or the activity that overlaps the least number of other activities? Or something else? We now show how to systematically arrive at the answer.

Most of the types and operators of GGS theory are straightforward to instantiate. We will just set $\widehat{R}$ to be the same as $R$. The initial space is just the empty set. The subspace relation $\lessdot$ splits a space by selecting an unchosen activity if one exists and adding it to the existing partial solution. The extract predicate $\chi$ can extract a solution at any time:

$$
\begin{array}{rcl}
\widehat{R} & \mapsto & R \\
\widehat{r}_0 & \mapsto & \lambda x \cdot \emptyset \\
\lessdot & \mapsto & \lambda(x, \widehat{z}, \widehat{z}') \cdot \exists a \in x - \widehat{z} \cdot \widehat{z}' = \widehat{z} \cup \{a\} \\
\chi & \mapsto & \lambda(z, \widehat{z}) \cdot z = \widehat{z} \\
\Phi & \mapsto & ? \\
\delta & \mapsto & ?
\end{array}
$$

The tricky bit is finding bindings for $\delta$ and $\Phi$ to complete the instantiation, which we will do in step 3. First, in step 2 we explore the problem and try and formulate a domain theory. The composition operator $\oplus$ is just $\cup$. The $\preceq$ relation can be lifted up to sets of activities by defining the start and finish times of a set of activities, namely $(u \oplus v).f = \max\{u.f, v.f\}$ and $(u \oplus v).s = \min\{u.s, v.s\}$. The following theorem will come in handy:

**Theorem 3.1.** *$noOvp(s) \Leftrightarrow \exists s_1, \ldots s_n \subseteq s \cdot s = \bigcup_{1 \le i \le n} s_i \wedge (\forall i \cdot 1 \le i < n \Rightarrow s_i \preceq s_{i+1} \wedge noOvp(s_i)) \wedge noOvp(s_n)$*

$o(x,\widehat{y}\oplus\{a\}\oplus e)$
$=\{\text{defn}\}$
$noOvp(\widehat{y}\cup\{a\}\cup e)\wedge\widehat{y}\cup\{a\}\cup e\subseteq x$
$\Leftarrow\{\text{Theorem 3.1 above}\}$
$\widehat{y}\preceq\{a\}\preceq e\wedge noOvp(\widehat{y})\wedge noOvp(e)\wedge\widehat{y}\cup\{a\}\cup e\subseteq x$
$\Leftarrow\{noOvp(\widehat{y})\wedge noOvp(e)\wedge\widehat{y}\cup e\subseteq x \text{ by assumption}\}$
$\widehat{y}\preceq\{a\}\preceq e\wedge a\in x$
$\Leftarrow\{\widehat{y}\preceq\{a'\}\preceq e,\ \text{apply transitivity}\}$
$\widehat{y}\preceq\{a\}\wedge a.f\leq a'.f\wedge a\in x$

Figure 3.1: Derivation of semi-congruence relation for Activity Selection

This says that any set of non-overlapping activities can be partitioned into internally non-overlapping subsets that follow each other serially.

For Step 3, we instantiate Definition 2 and reason backwards from its consequent, while assuming the antecedent. Each step of the derivation is accompanied by a hint (in $\{\}$) that justifies the step. Additional assumptions made along the way form the required semi-congruence condition. First note that a solution $z'\neq\emptyset$ can be expressed as $\widehat{y}\cup\{a'\}\cup e$ or alternatively $\widehat{y}\oplus\{a'\}\oplus e$, for some $\widehat{y},a',e$, such that, by Theorem 3.1, $\widehat{y}\preceq\{a'\}\preceq e$. Now consider the feasibility of a solution $\widehat{y}\oplus\{a\}\oplus e$, obtained by switching out $a'$ for some $a$, assuming $o(x,\widehat{y}\oplus\{a'\}\oplus e)$ as shown in Fig. 3.1

That is, $\widehat{y}\oplus a$ can be feasibly extended with the same feasible set of activities as $\widehat{y}\oplus a'$ provided $\widehat{y}$ finishes before $a$ starts and $a$ finishes before $a'$ finishes and $a$ is legal. By letting $\widehat{y}\oplus a$ and $\widehat{y}\oplus a'$ be subspaces following a split of $\widehat{y}$ this forms a semi-congruence condition between $\widehat{y}\oplus a$ and $\widehat{y}\oplus a'$. Since $c$ is a distributive cost function, and all subspaces of a given space are the same size, the dominance relation equals the semi-congruence relation, by Proposition 1. Next, instantiating condition 2.1, we need to show that if $\widehat{y}$ contains feasible solutions, then in the case that any solution immediately extracted from $\widehat{y}$ is not optimal, (ie. the optimal lies in a subspace of $\widehat{y}$) there is *always* a subspace $\widehat{y}\oplus a$ that dominates every extension of $\widehat{y}$. Unfortunately, the dominance condition derived is too strong to be able to establish the instantiation of 2.1. Logically, what we established is a *sufficient* dominance test. That is $\widehat{y}\preceq\{a\}\wedge a.f\leq a'.f\wedge a\in x\Rightarrow\widehat{y}\oplus\{a\}\,\delta_x\,\widehat{y}\oplus\{a'\}$. How can we weaken it? This is where the filter $\Phi$ comes in. The following theorem shows how to construct a simple dominance relation from a filter:

**Theorem 3.2.** *Given a filter* $\Phi$ *satisfying* $\forall\widehat{z}'\cdot(\exists z\in\widehat{z}'\cdot o(x,z))\Rightarrow\Phi(x,\widehat{z}')$, $\neg\Phi(x,\widehat{z}')\Rightarrow\forall\widehat{z}\cdot\widehat{z}\,\delta_x\,\widehat{z}'$.

The theorem says that a space that does not pass the necessary filter is dominated by any space.On a subspace $\widehat{y}\oplus a'$, one such filter (that can be mechanically derived by a tool such as KIDS [Smi90]) is $\widehat{y}\preceq\{a'\}$. Now, we can combine both dominance tests with the 1st order variant of the rule $(p\Rightarrow r\wedge q\Rightarrow r)\Rightarrow(p\vee q\Rightarrow r)$ , reasoning backwards as we did above as shown in Fig. 3.2.

The binding for $m$ above shows that 2.1 is satisfied by picking an activity in $x-\widehat{y}$ with the earliest finish time, after overlapping activities have been filtered out. Note how in verifying 2.1 we have extracted a witness which is the greedy choice. This pattern of witness finding is common across many examples. The program schema in Alg. 1 can now be instantiated into a greedy solution to the Activity Selection Problem.

In contrast to our derivation, the solution presented in [CLRS01] starts off by assuming the tasks are sorted in order of finishing time. Only after reading the pseudocode and a proof is the reason for this clear (though how to have thought of it a priori is still not!). For us, the condition falls out of the process of investigating a possible dominance relation. Note that had we partitioned the solution $\widehat{y}\oplus\{a'\}\oplus e$

$$\exists a \in x-\widehat{y}, \forall a' \in x-\widehat{y} \cdot \widehat{y} \oplus a\, \delta_x \widehat{y} \oplus a'$$
$$\Leftarrow \{\widehat{y} \preceq \{a\} \wedge a.f \leq a'.f \wedge a \in x \Rightarrow \widehat{y} \oplus \{a\}\, \delta_x \widehat{y} \oplus \{a'\}\ \&\widehat{y} \not\preceq \{a'\} \Rightarrow \widehat{y} \oplus \{a\}\, \delta_x \widehat{y} \oplus \{a'\}\}$$
$$\exists a \in x-\widehat{y}, \forall a' \in x-\widehat{y} \cdot \widehat{y} \not\preceq \{a'\} \vee (\widehat{y} \preceq \{a\} \wedge a.f \leq a'.f)$$
$$= \{\text{logic}\}$$
$$\exists a \in x-\widehat{y} \cdot \widehat{y} \preceq \{a\} \wedge \forall a' \in x-\widehat{y} \cdot \widehat{y} \preceq \{a'\} \Rightarrow a.f \leq a'.f$$
$$= \{\text{logic}\}$$
$$\exists a \in x-\widehat{y} \cdot \widehat{y} \preceq \{a\} \wedge \forall a' \in x-\widehat{y} \cap \{b \mid \widehat{y} \preceq \{b\}\} \cdot a.f \leq a'.f$$
$$= \{\text{define } a \leq b = a.f \leq b.f\}$$
$$\exists a \in x-\widehat{y} \cdot \widehat{y} \preceq \{a\} \wedge \forall a' \in x-\widehat{y} \cap \{b \mid \widehat{y} \preceq \{b\}\} \cdot m \leq a' \Rightarrow a.f \leq a'.f$$
$$\text{where } m = \min_{\leq} x-\widehat{y} \cap \{b \mid \widehat{y} \preceq \{b\}\}$$
$$= \{\text{law for monotone } p: \ (\forall x \in S \cdot m \leq x \Rightarrow p(x)) \equiv p(m)\}$$
$$\exists a \in x-\widehat{y} \cdot \widehat{y} \preceq \{a\} \wedge a.f \leq m.f \text{ where } m = \min_{\leq} x-\widehat{y} \cap \{b \mid \widehat{y} \preceq \{b\}\}$$
$$= \{\text{law for anti-monotone } p: \ (\exists x \in S \cdot m \leq x \wedge p(x)) \equiv p(m)\}$$
$$m.f \leq m.f \text{ where } m = \min_{\leq} x-\widehat{y} \cap \{b \mid \widehat{y} \preceq \{b\}\}$$
$$=$$
$$true$$

Figure 3.2:

differently as $e \preceq \{a'\} \preceq \widehat{y}$, we would have arrived at an another algorithm that grows the result going backwards rather than forwards, which is an alternative to the solution described in [CLRS01].

Next we apply our process to the derivation of a solution to a fairly non-trivial problem, that of determining optimum prefix-free codes.

**Example 2.** Prefix-Free Codes

> Devise an encoding, as a binary string, for each of the characters in a given text file so as to minimize the overall size of the file. For ease of decoding, the code is required to be *prefix-free*, that is no encoding of a character is the prefix of the encoding of another character (e.g. assigning “0” to ’a’ and “01” to ’b’ would not be allowed).

D.A. Huffman devised an efficient greedy algorithm for this in 1952. We show how it can be systematically derived. Step 1 is to specify the problem. The input is a table of character frequencies, and the result is a table of bit strings, one for each character in the input, satisfying the prefix free property.

$$\begin{array}{rcl} D & \mapsto & Char \rightarrowtail Frequency \\ & & Char = Frequency = Nat \\ R & \mapsto & Char \rightarrowtail [Boolean] \\ o & \mapsto & \lambda x,z.\ dom(z) = dom(x) \wedge \forall c \neq c' \in dom(z) \cdot \neg prefixOf(z(c), z(c')) \\ & & prefixOf(s,t) = \exists u \cdot t = s{+}{+}u \vee s = t{+}{+}u \\ c & \mapsto & \lambda x,z.\ \sum_{c \in dom(z)} \|z(c)\| \times x(c) \end{array}$$

Often a good way of ensuring a condition is to fold it into a data structure. Then the rules for constructing that data structure ensure that the condition is automatically satisfied. It turns out that a binary tree in which the leaves are the letters, and the path from the root to the leaf provides the code for that letter, ensures that the resulting codes are automatically prefix-free[2]. One obvious way to construct

[2]It is possible to systematically derive this datatype by starting with an obvious datatype such as a set of binary strings and folding in a required property such as prefix-freeness. However, we do not pursue that here

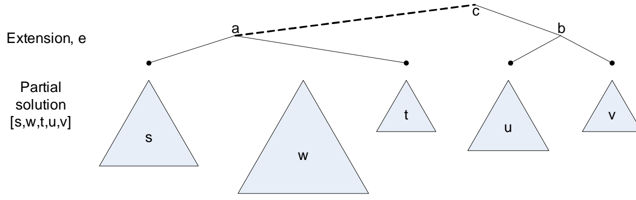



Figure 3.3: An extension applied to a partial solution with 5 trees

such a tree is by merging together smaller trees. This will be the basis of the $\lessdot$ relation: the different subspaces arise from the $t(t-1)/2$ choices of pairs of trees to merge, where $t$ is the number of trees at any point in the search. The starting point $\widehat{r}_0$ is the ordered collection of leaves representing the letters. The extract predicate $\chi$ checks there is only one tree left and generates the path to each leaf (character) of the tree, which becomes the code for that character. With this information, we can instantiate the remaining terms in GGS, except for $\delta$ ($\langle\rangle$ is the non-elementary$BinTree$ constructor)

$$
\begin{array}{lll}
\widehat{R} & \mapsto & [BinTree] \\
\widehat{r}_0 & \mapsto & \lambda x \cdot asList(dom(x)) \\
\lessdot & \mapsto & \lambda(x,\widehat{z},\widehat{z}') \cdot \exists s,t \in \widehat{z}.\ \widehat{z}' = [\langle s,t\rangle \mid (\widehat{z}-s-t)] \\
\chi & \mapsto & \lambda(z,\widehat{z}) \cdot \|\widehat{z}\| = 1 \wedge \forall p \in paths(\widehat{z}) \cdot z(last(p)) = first(p) \\
 & & paths(\langle s,t\rangle) = (map\, prefix0\, paths(s)) +\!\!+ (map\, prefix1\, paths(t)) \\
 & & paths(l) = [l] \\
 & & prefix0(p) = [0 \mid p],\ prefix1(p) = [1 \mid p] \\
\Phi & \mapsto & true \\
\delta & \mapsto & ?
\end{array}
$$

One interesting difference between this problem and the Activity Selection problem is that every subspace leads to a feasible solution. For that reason, $\Phi$ is just $true$. It is not obvious what the criterion should be for a greedy choice. Should it be to combine the trees with the least number of leaves, or those with the least height, or something else? To proceed with our process and apply Defs. 2 and 3, we will define $\oplus$ as a left-associative binary operator whose net effect is to merge two $BinTree$s from its left argument together into another $BinTree$. The right argument specifies which trees are to be merged. That is, $\widehat{z} \oplus (i,j) = [\langle \widehat{z}_i, \widehat{z}_j \rangle \mid \widehat{z} - \widehat{z}_i - \widehat{z}_j]$. For example, Fig 3.3 shows the merger of trees $s$ and $t$ and the merger of trees $u$ and $v$ in a partial solution $\widehat{z}$ to form two subtrees with roots $a$ and $b$. This is described by the expression $(\widehat{z} \oplus (1,3)) \oplus (3,4) = \widehat{z} \oplus (1,3) \oplus (3,4)$. The extension in this case is $(1,3) \oplus (3,4)$. A semi-congruence condition is shown in Fig. 3.4. Assuming $o(x, \widehat{z}' \oplus e)$ and a definition of $lvs(\widehat{z}) = map\, last\,(paths(\widehat{z}))$. (Note we omit the remainder of the output condition since it is implied by the Binary Tree construction) This says that any two partial solutions of the same size are extensible with the same feasible extension. This semi-congruence condition is trivially satisfied by any two subspaces of a split. For the extension dominance condition, it is easy to show that $c(x, \widehat{z} \oplus e)$ can be expressed as $\widehat{c}(x,\widehat{z}) + k(x,e)$ for some $k$ where $\widehat{c}(x,\widehat{z}) = \sum_{i=1}^{|lvs(\widehat{z})|} d(\widehat{z})(i) \cdot x(lvs(\widehat{z})_i)$ where $d(\widehat{z})$ is a function returning the depth of its argument leaf within the tree $\widehat{z}$, and therefore by Prop. 1, it is sufficient to determine conditions under which $\widehat{c}(x,\widehat{z}) \le \widehat{c}(x,\widehat{z}')$. However, if we try to calculate such a condition as we have done for semi-congruence we will end up with an expression that involves the depths of individual leaves in the trees. Is there a simpler form? We have been investigating ways to provide a developer with hints about how to proceed. We call these *tactics*. In earlier work [NSC09] we introduced tactics for the derivation of operators for non-optimization problems. We now introduce a tactic for dominance relations. We have used this tactic to derive greedy solutions for a number of problems, including Machine Scheduling, several variations on the Maximum Segment Sum Problem,[NC09], Minimum Spanning

$o(x, \widehat{z} \oplus e)$
$= \{\text{defn of } o \text{ on } \widehat{R}\}$
$\exists z \cdot \chi(z, \widehat{z} \oplus e) \wedge o(x,z)$
$= \{\text{defn of } \chi, o\}$
$\exists z \cdot \|\widehat{z} \oplus e\| = 1 \wedge \forall p \in paths(\widehat{z} \oplus e) \cdot z(last(p)) = first(p) \wedge dom(z) = x \wedge \cdots$
$= \{\text{intro defn}\}$
$\|\widehat{z} \oplus e\| = 1 \wedge dom(z) = x \wedge \cdots$
where $z = \{last(p) \mapsto first(p) \mid p \in paths(\widehat{z} \oplus e)\}$
$\Leftarrow \{o(x, \widehat{z}' \oplus e) \Rightarrow dom(z') = x \text{ where } z = \{last(p) \mapsto first(p) \mid p \in paths(\widehat{z}' \oplus e)\}\}$
$\|\widehat{z} \oplus e\| = 1 \wedge asSet(lvs(\widehat{z})) = asSet(lvs(\widehat{z}'))$
$= \{\text{split does not alter set of leaves}\}$
$\|\widehat{z} \oplus e\| = 1$
$= \{\|\widehat{z} \oplus e\| = \|\widehat{z}\| - \|e\|, \|\widehat{z}' \oplus e\| = 1\}$
$\|\widehat{z}\| = \|\widehat{z}'\|$

Figure 3.4: Derivation of extension dominance relation for Huffman problem

$$
\begin{array}{ll}
& c(\widehat{z}) \le c(\widehat{z}') \\
= & \{\text{unfold defn of } c\} \\
& \sum_{i=1}^{|lvs(s)|} (d(s)(i)+h) \cdot x(lvs(s)_i) + \sum_{i=1}^{|lvs(t)|} (d(t)(i)+h) \cdot x(lvs(t)_i) \\
& + \sum_{i=1}^{|lvs(u)|} (d(u)(i)+2) \cdot x(lvs(u)_i) + \sum_{i=1}^{|lvs(v)|} (d(v)(i)+2) \cdot x(lvs(v)_i) \\
& \qquad\qquad \le \\
& \sum_{i=1}^{|lvs(u)|} (d(u)(i)+h) \cdot x(lvs(u)_i) + \sum_{i=1}^{|lvs(v)|} (d(v)(i)+h) \cdot x(lvs(v)_i) \\
& + \sum_{i=1}^{|lvs(s)|} (d(s)(i)+2) \cdot x(lvs(s)_i) + \sum_{i=1}^{|lvs(t)|} (d(t)(i)+2) \cdot x(lvs(t)_i) \\
= & \{\text{algebra}\} \\
& (h-2) \cdot \sum_{i=1}^{|lvs(s)|} x(lvs(s)_i) + (h-2) \cdot \sum_{i=1}^{|lvs(t)|} x(lvs(t)_i) \\
& \le (h-2) \cdot \sum_{i=1}^{|lvs(u)|} x(lvs(u)_i) + (h-2) \cdot \sum_{i=1}^{|lvs(v)|} x(lvs(v)_i) \\
\Leftarrow & \{\text{algebra}\} \\
& \sum_{i=1}^{|lvs(s)|} x(lvs(s)_i) + \sum_{i=1}^{|lvs(t)|} x(lvs(t)_i) \le \sum_{i=1}^{|lvs(u)|} x(lvs(u)) + \sum_{i=1}^{|lvs(v)|} x(lvs(v)_i) \wedge h > 2
\end{array}
$$

Figure 3.5:

Tree, and Professor Midas' Driving Problem.

> Exchange Tactic: *Try to derive a dominance relation by comparing a partial solution $\widehat{y} \oplus a \oplus \alpha \oplus b$ (assuming some appropriate parenthesization of the expression) to a variant obtained by exchanging a pair of terms, that is, $\widehat{y} \oplus b \oplus \alpha \oplus a$, with the same parenthesization*

Given a partial solution $\widehat{y}$, suppose trees $s$ and $t$ are merged first, and at some later point the tree containing $s$ and $t$ is merged with a tree formed from merging $u$ and $v$ (tree *satcubv* in Fig. 3.3), forming a partial solution $\widehat{z}$. Applying the exchange tactic, when is this better than a partial solution $\widehat{z}'$ resulting from swapping the mergers in $\widehat{z}$, ie merging $u$ and $v$ first and then $s$ and $t$? Let $d(T)_i$ be the depth of leaf $i$ in a tree $T$, and let $h$ be the depth of $s$ (resp. $u$) from the grandparent of $u$ (resp. $s$) in $\widehat{z}$ (resp $\widehat{z}'$), (the distance from $c$ to the root of $s$ in Fig. 3.3). The derivation of the extension dominance relation is shown in Fig. 3.5.

That is, if the sum of the frequencies of the leaves of $s$ and $t$ is no greater than the sum of the frequencies of leaves of $u$ and $v$ then $s$ and $t$ should be merged before $u$ and $v$. (The condition $h > 2$

simply means that no dominance relation holds when $\langle u,v\rangle$ is immediately merged with $\langle s,t\rangle$. It is clear in that case that the tree is balanced). How does this help us derive a dominance relation between two subspaces after a split? The following theorem shows that the above condition serves as a dominance relation between the two subspaces $[\langle s,t\rangle \mid (\widehat{y}-s-t)]$ and $[\langle u,v\rangle \mid (\widehat{y}-u-v)]$:

**Theorem 3.3.** *Given a GGS theory for a constraint satisfaction problem,* $(\exists\alpha\cdot(\widehat{y}\oplus a\oplus\alpha\oplus b)\,\delta_x\,(\widehat{y}\oplus b\oplus\alpha\oplus a))\Rightarrow\widehat{y}\oplus a\;\delta_x\;\widehat{y}\oplus b$

By using a witness finding technique to verify condition 2.1 as we did for Activity Selection, we will find that the greedy choice is just the pair of trees whose sums of letter frequencies is the least. This is the same criterion used by Huffman's algorithm. Of course, for efficiency, in the standard algorithm, a stronger dominance test is used: $\sum_{i=1}^{|lvs(s)|}x(lvs(s)_i)\le\sum_{i=1}^{|lvs(u)|}x(lvs(u)_i)\wedge\sum_{i=1}^{|lvs(t)|}x(lvs(t)_i)\le\sum_{i=1}^{|lvs(v)|}x(lvs(v)_i)$ and the sums are maintained at the roots of the trees as the algorithm progresses. We would automatically arrive at a similar procedure after applying finite differencing transforms, [Smi90, NC09]. In contrast to our stepwise derivation, in most presentations of Huffman's algorithm, (e.g. [CLRS01]) the solution is presented first, followed by an explanation of the pseudocode, and then several pages of lemmas and theorems justifying the correctness of the algorithm. The drawback of the conventional approach is that the insights that went into the original algorithm development are lost, and have to be reconstructed when variants of the problem arise. A process for greedy algorithm development, such the one we have proposed here, is intended to remedy that problem.

# 4 Related Work

Curtis [Cur03] has a classification scheme for greedy algorithms. Each class has a some conditions that must be met for a given algorithm to belong to that class. The greedy algorithm is then automatically correct and optimal. Unlike Curtis, we are not attempting a classification scheme. Our goal is to simplify the process of creating greedy algorithms. For that reason, we present derivations in a calculational style whenever the exposition is clear. In contrast, Curtis derives the "meta-level" proofs, namely that the conditions attached to a given algorithm class in the hierarchy are indeed correct, calculationally but the "object-level" proofs, namely those showing a given problem formulation does indeed meet those conditions, are done informally. We believe that this should be the other way around. The meta-level proofs are (hopefully) carried out only a few times and are checked by many, but the object level proofs are carried out by individual developers, and are therefore the ones which ought to be done calculationally, not only to keep the developer from making mistakes but also with a view to providing mechanical assistance (as was done in KIDS, a predecessor of Specware). Another difference between our work and Curtis is that while Curtis's work is targeted specifically at greedy algorithms, for us greedy algorithms are just a special case of a more general problem of deriving effective global search algorithms. In the case that the dominance relation really does not lead to a singleton choice at each split, it can still prove to be highly effective. This was recently demonstrated on some Segment Sum problems we looked at, [NC09]. Although the dominance relation we derived for those problem did not reduce to a greedy choice, it was nonetheless key to reducing the complexity of the search (the width of the search tree was kept constant) and led to a very efficient breadth-first solution that was much faster than comparable solutions derived by program transformation.

Another approach has been taken by Bird and de Moor [BM93] who show that under certain conditions a dynamic programming algorithm simplifies into a greedy algorithm. Our characterization in [NSC10] can be considered an analogous specialization of (a form of) branch-and-bound. The difference is that we do not require calculation of the entire program, but specific operators, which is a less onerous task.

Helman [Hel89] devised a framework that unified branch-and-bound and dynamic programming. The framework also incorporated dominance relations. However, Helman's goal was the unification of the two paradigms, and not the process by which algorithms can be calculated. In fact the unification, though providing a very important insight that the two paradigms are related at a higher level, arguably makes the derivation of particular algorithms harder.

Charlier [Cha95], also building on Smith's work, proposed a new algorithm class for greedy algorithms that embodied the matroid axioms. Using this class, he was able to synthesize Kruskal's MST algorithm and a solution to the $1/1/\sum T_i$ scheduling problem. However he reported difficulty with the equivalent of the Augmentation (also called Exchange) axiom. The difficulty with a new algorithm class is often the lack of a repeatable process for synthesizing algorithms in that class, and this would appear to be what Charlier ran up against. In contrast, we build on top of the GSO class, adding only what is necessary for our purposes. As a result we can handle a wider class of algorithms than would belong in Charlier's Greedy class, such as Prim's and Huffman's.

## References


[BM93] R. S. Bird and O. De Moor. From dynamic programming to greedy algorithms. In *Formal Program Development, volume 755 of Lecture Notes in Computer Science*, pages 43–61. Springer-Verlag, 1993.

[BS74] K.R. Baker and Z-S. Su. Sequencing with due-dates and early start times to minimize maximum tardiness. *Naval Research Logistics*, 21(1):171–176, 1974.

[Cha95] B. Charlier. The greedy algorithms class: formalization, synthesis and generalization. Technical report, 1995.

[CLRS01] T Cormen, C Leiserson, R Rivest, and C Stein. *Introduction to Algorithms*. MIT Press, 2nd edition, 2001.

[Cur03] S. A. Curtis. The classification of greedy algorithms. *Sci. Comput. Program.*, 49(1-3):125–157, 2003.

[Edm71] J. Edmonds. Matroids and the greedy algorithm. *Math. Programming*, 1(1):127–136, 1971.

[Hel89] P. Helman. A common schema for dynamic programming and branch and bound algorithms. *J. ACM*, 36(1):97–128, 1989.

[HMS93] P. Helman, B. M. E. Moret, and H. D. Shapiro. An exact characterization of greedy structures. *SIAM J. on Discrete Math.*, 6:274–283, 1993.

[Iba77] T. Ibaraki. The power of dominance relations in branch-and-bound algorithms. *J. ACM*, 24(2):264–279, 1977.

[KLS91] B. Korte, L. Lovasz, and R. Schrader. *Greedoids*. Springer-Verlag, 1991.

[NC09] S. Nedunuri and W.R. Cook. Synthesis of fast programs for maximum segment sum problems. In *Intl. Conf. on Generative Programming and Component Engineering (GPCE)*, Oct. 2009.

[NSC09] S. Nedunuri, D. R. Smith, and W. R. Cook. Tactical synthesis of efficient global search algorithms. In *Proc. NASA Symposium on Formal Methods*, April 2009.

[NSC10] S. Nedunuri, D. R. Smith, and W. R. Cook. A class of greedy algorithms and its relation to greedoids. *Submitted to: Intl. Colloq. on Theoretical Aspects of Computing (ICTAC)*, 2010.

[S] Specware. http://www.specware.org.

[Smi88] D. R. Smith. Structure and design of global search algorithms. Tech. Rep. Kes.U.87.12, Kestrel Institute, 1988.

[Smi90] D. R. Smith. Kids: A semi-automatic program development system. *IEEE Trans. on Soft. Eng., Spec. Issue on Formal Methods*, 16(9):1024–1043, September 1990.

[SPW95] D. R. Smith, E. A. Parra, and S. J. Westfold. Synthesis of high-performance transportation schedulers. Technical report, Kestrel Institute, 1995.

[SW08] D. R. Smith and S. Westfold. Synthesis of propositional satisfiability solvers. Final proj. report, Kestrel Institute, 2008.

# 5 Appendix: Proofs of Theorems

## 5.1 Proofs of Theorem 2.1 and Proposition 1:

**Theorem** 2.1: If $\rightsquigarrow$ is a semi-congruence relation, and $\widehat{\delta}$ is a extension dominance relation, then

$$\forall x, \forall \widehat{z}, \widehat{z}' \cdot \widehat{z}\widehat{\delta}_x\widehat{z}' \wedge \widehat{z} \rightsquigarrow_x \widehat{z}' \Rightarrow c^*(x,\widehat{z}) \leq c^*(x,\widehat{z}')$$

*Proof.* By contradiction. (input argument $x$ dropped for readability). Suppose that $\widehat{z}\widehat{\delta}_x\widehat{z}' \wedge \widehat{z} \rightsquigarrow_x \widehat{z}'$ but $\exists z'^* \in \widehat{z}', O(z'^*) \wedge c(z'^*) < c^*(\widehat{z})$, that is $c(z'^*) < c(z)$ for any feasible $z \in \widehat{z}$. We can write $z'^*$ as $\widehat{z}' \oplus e$ for some $e$. Since $z'^*$ is cheaper than any feasible $z \in \widehat{z}$, specifically it is cheaper than $z = \widehat{z} \oplus e$, which by the semi-congruence assumption and Definition 2, is feasible. But by the extension dominance assumption, and Definition 3, this means $c(z) \leq c(z'^*)$, contradicting the initial assumption. □

**Proposition** 1: If the cost domain $C$ is a numeric domain (such as *Integer* or *Real*) and $c(x,\widehat{z} \oplus e)$ can be expressed as $\widehat{c}(x,\widehat{z}) + k(x,e)$ for some functions $\widehat{c}$ and $k$ then $\widehat{\delta}_x$ where $\widehat{z}\widehat{\delta}_x\widehat{z}' = \widehat{c}(x,\widehat{z}) \leq \widehat{c}(x,\widehat{z}')$ is an extension dominance relation

*Proof.* By showing that Definition 3 is satisfied. $c(\widehat{z} \oplus e) \leq c(\widehat{z}' \oplus e) = \widehat{c}(\widehat{z}) + k(e) \leq \widehat{c}(\widehat{z}') + k(e)$ by distributivity of $c$ which is just $c(\widehat{z}) \leq c(\widehat{z}')$ after arithmetic. □

# A new Method for Incremental Testing of Finite State Machines

Lehilton Lelis Chaves Pedrosa *
University of Campinas, Brazil
lehilton.pedrosa@students.ic.unicamp.br

Arnaldo Vieira Moura †
University of Campinas, Brazil
arnaldo@ic.unicamp.br

**Abstract**

The automatic generation of test cases is an important issue for conformance testing of several critical systems. We present a new method for the derivation of test suites when the specification is modeled as a combined Finite State Machine (FSM). A combined FSM is obtained conjoining previously tested submachines with newly added states. This new concept is used to describe a fault model suitable for incremental testing of new systems, or for retesting modified implementations. For this fault model, only the newly added or modified states need to be tested, thereby considerably reducing the size of the test suites. The new method is a generalization of the well-known W-method [4] and the G-method [2], but is scalable, and so it can be used to test FSMs with an arbitrarily large number of states.

## 1 Introduction

Test case generation for reactive and critical systems using formal methods has been widely studied [1, 2, 4, 6, 8, 11, 12, 14]. In such methods, system requirements are described by means of mathematical models and formally specified functionalities. When using formal specification models, the automatic generation of adequate test cases rises as an important problem. Methods to automate the generation of test suites must be *efficient*, in terms of test suites size, and *accurate*, in terms of fault detection [3, 11]. When test suites are applied, the notion of conformance [5] can be used, so that if an implementation passes a test suite, its behavior is said to conform to the behavior extracted from the specification.

Finite State Machines (FSMs) are the basic formalism in many methods that automate the generation of conformance test case suites. For surveys, see [1, 11, 14]. Among such methods, the W-method [4] is based on the notion of characterization sets, and provides full fault coverage for minimal, completely specified and deterministic FSMs. Several derivations have been proposed around it. In particular, the G-method [2] is a generalization of the W-method that does not depend on characterization sets.

These methods assume that the system specification is treated in a monolithic way. However, in many situations, systems are modular, with their specifications being formed by several subsystems. If one such subsystem is also modeled by a FSM, we call it a submachine. Then, the full FSM model is a combination of several submachines, with the aid of a few new states and transitions. In this article, we propose a new approach to test combined FSMs when submachine implementations are assumed correct.

Testing using the combined FSM abstraction is useful in at least two situations. If a new system is modeled as a combination of several submachines, then we can implement and test each submachine independently. Later, we can then test the combined machine using a smaller test suite. In this incremental testing approach, if an implementation does not pass a test suite, only a few states need to be retested, avoiding reapplying large test suites, as in the W-method. On the other hand, suppose that a given specification is changed, then only the corresponding part of a former implementation gets modified. If we use methods like the W-method or the G-method, we would have to test the entire system again. However, if the specification is a combined machine, only the affected submachines need to be retested.

There are related works on retesting modified implementations. But they are restricted to certain types of errors and modifications, and require that implementations maintain the same number of states

*Work supported by FAPESP grant 08/07969-9.
†Work supported by FAPESP grant 02/07473-7.

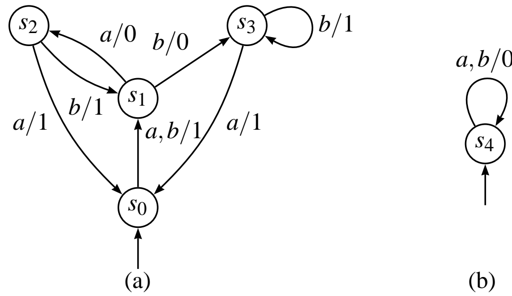



Figure 1: Finite State Machines.

as in the specification [7, 10]. In this paper, we do not restrict the types of errors in an implementation, neither how it is modified, and we allow implementations with more states than in the specification.

In Section 2, we review the FSM model and related conventions. In Section 3, we describe equivalence relations of FSMs and introduce the concept of separators, a powerful tool to test FSMs. In Section 4, we formalize the notion of combined FSMs. In Section 5, we present the new test case generation method, here named the C-method. In Section 6, we compare our method with the W-method, and discuss that the C-method is scalable, that is, it can be used to test FSMs with a large number of states.

# 2 Basic definitions

Let $A$ be an alphabet. Then $A^*$ is the set of all finite sequences of symbols, or words, over $A$. The length of a word $\rho \in A^*$ will be denoted by $|\rho|$, and $\varepsilon$ will denote the empty word. So, $|\varepsilon| = 0$. The concatenation, or juxtaposition, of two words $\alpha$, $\beta \in A^*$ will be indicated by $\alpha\beta$.

## 2.1 Finite State Machines

A FSM is a tuple $M = (X,Y,S,s_0,\delta,\lambda)$, where: (i) $X$ is a finite input alphabet, (ii) $Y$ is a finite output alphabet, (iii) $S$ is the set of states, (iv) $s_0 \in S$ is the initial state, (v) $\delta : X \times S \to S$ is the transition function, and (vi) $\lambda : X \times S \to Y$ is the output function.

From now on we fix the notation $M = (X,Y,S,s_0,\delta,\lambda)$ and $M' = (X,Y',S',s'_0,\delta',\lambda')$. Sequences of input symbols will be represented by words $\rho \in X^*$, and sequences of output symbols will be represented by words $\sigma \in Y^*$. The end state after the successive application of each input is given by the extended function $\widehat{\delta} : X^* \times S \to S$, and the output extended function is $\widehat{\lambda} : X^* \times S \to Y^*$, defined by

$$\widehat{\delta}(\varepsilon,s) = s, \qquad \widehat{\delta}(a\rho,s) = \widehat{\delta}(\rho,\delta(a,s)),$$
$$\widehat{\lambda}(\varepsilon,s) = \varepsilon, \qquad \widehat{\lambda}(a\rho,s) = \lambda(a,s)\widehat{\lambda}(\rho,\delta(a,s)).$$

for all $a \in X$, $\rho \in X^*$ and $s \in S$.

Usually, a FSM is represented by a state diagram. Figure 1 illustrates two FSMs with initial states $s_0$ and $s_4$, respectively. We will refer to this figure through the paper.

## 2.2 Concatenation of words and relative concatenation

We adopt the usual notation of concatenation of two sets of words and denote by $X_n$ the set of all input words with length at most $n$. For the sake of completeness, we give a definition below.

**Definition 1.** *Let $A,B \subseteq X^*$ and let $n$ be a non-negative integer. Then, (i) $AB = \{\alpha\beta | \alpha \in A, \beta \in B\}$, (ii) $X^n = \{\rho \in X^* | \, |\rho| = n\}$, and (iii) $X_n = \bigcup_{k=0}^{n} X^k$.* ∎

Suppose that a set of input words, $Z$, must be applied to a set of states, $S$. To accomplish this, we generate test cases, by selecting a set of words, $Q$, to reach the states in $S$, and concatenating $Q$ with $Z$. For example, if $Z = \{a\}$, $S = \{s_1, s_2\}$ and we may reach $s_1$ and $s_2$ applying $a$ and $ab$ to $s_0$, respectively, then we may select $Q = \{a, ab\}$, and generate test cases $QZ = \{aa, aba\}$. Now, suppose that specific

sets are applied to distinct states, that is, $Z_1 = \{a\}$ is applied to $s_1$, and $Z_2 = \{b\}$ is applied to $s_2$. In this case, the conventional concatenation is not useful. To address this problem, the relative concatenation was introduced [8]. First, we need the following, where $\mathscr{P}(A)$ stands for the power set of a set $A$.

**Definition 2.** *Let $M$ be a FSM and let $\Pi$ a partition of $S$. A state attribution is a function $\mathcal{B} : S \to \mathscr{P}(X^*)$. A class attribution is a function $\mathcal{B} : \Pi \to \mathscr{P}(X^*)$.* ∎

A class attribution induces a state attribution a in a natural way. If $\mathcal{B}$ is a class attribution over a partition $\Pi$, then the induced state attribution, $\overline{\mathcal{B}}$, is defined by $\overline{\mathcal{B}}(s) = \mathcal{B}(C)$, for all $s \in C$ and all $C \in \Pi$.

**Definition 3.** *Let $M$ be a FSM, $A \subseteq X^*$, and $\mathcal{B}$ be a state attribution of $M$. Given a state $s$, we define the $s$-relative concatenation of $A$ and $\mathcal{B}$ as $A \underset{s}{\otimes} \mathcal{B} = \{\alpha\beta | \alpha \in A, \beta \in \mathcal{B}(\widehat{\delta}(\alpha, s))\}$.* ∎

Whenever $s = s_0$, we may drop the state index and simply write $A \otimes \mathcal{B}$. If $\mathcal{B}$ is a class attribution, then we may also write $A \underset{s}{\otimes} \mathcal{B}$ to mean $A \underset{s}{\otimes} \overline{\mathcal{B}}$. The usual concatenation may be thought of as a particular case of the relative concatenation, as observed below.

**Observation 4.** *Let $M$ be a FSM and $A, B$ be sets of input word. Let also $\mathcal{B}$ be a state attribution such that $\mathcal{B}(s) = B$ for all $s \in S$. Then $A \underset{s}{\otimes} \mathcal{B} = AB$.* ∎

### 2.3 State Reachability

**Definition 5.** *Let $M$ be a FSM. A state $s$ is reachable if and only if there exists $\rho \in X^*$ such that $s = \widehat{\delta}(\rho, s_0)$. $M$ is connected if and only if every state is reachable.* ∎

When applying an input word $\rho$ to a start state $s$, if all we know is that the length of $\rho$ is at most $n$, then the output fetched when applying $\rho$ starting at $s$ will depend only on states that are at a distance of at most $n$ from $s$. Such states around $s$ form a *neighborhood*, defined as follows.

**Definition 6.** *Let $M$ be a FSM.*

1. *The $k$-*radius *of a state $s$, denoted by* $\text{rad}(s,k)$*, is the set of states that can be reached starting at $s$ and using input words of length at most $k$. That is, $r \in \text{rad}(s,k)$ if and only if there exist an input word $\rho \in X^*$ such that $r = \widehat{\delta}(\rho, s)$ and $|\rho| \leq k$.*
2. *The $k$-*neighborhood *of a set of states $C$, denoted by* $\text{nbh}(C,k)$*, is formed by the $k$-radiuses of states in $C$. That is,* $\text{nbh}(C,k) = \bigcup_{s \in C} \text{rad}(s,k)$. ∎

### 2.4 Cover sets

Cover sets are used in many FSM test methods in order to guarantee that every state is reached from the initial state and that every transition in the model is exercised at least once. But, if we know that some states have already been tested, then we do not need to reach them or exercise their corresponding transitions. In this situation, only untested states must be covered, and partial cover sets are sufficient.

**Definition 7.** *Let $M$ be a FSM and $C$ be a set of states. A set $P \subseteq X^*$ is a partial transition cover set for $C$ if, for every state $s \in C$ and every symbol $a \in X$, there exist $\rho, \rho a \in P$ such that $s = \widehat{\delta}(\rho, s_0)$.* ∎

Whenever $C$ is the set of all states, $P$ is, in fact, a transition cover set as defined in [2, 4, 8]. A transition cover set may be obtained from a labeled tree for $M$ [4]. A procedure to construct the labeled tree is given in [2]. Although that is intended to cover the entire set of states, one can modify this procedure in a straightforward way in order to obtain a partial cover set.

## 3 State equivalence and state separators

In this section, we define state equivalences and introduce the essential notion of a separator.

### 3.1 State equivalence

**Definition 8.** *Let $M$ and $M'$ be two FSMs over the same input alphabet, $X$, and let $s$ and $s'$ be states of $M$ and $M'$, respectively.*

1. *Let $\rho \in X^*$. We say that $s$ is $\rho$-equivalent to $s'$ if $\widehat{\lambda}(\rho,s) = \widehat{\lambda}'(\rho,s')$. In this case, we write $s \approx_\rho s'$. Otherwise, $s$ is $\rho$-distinguishable from $s'$, and we write $s \not\approx_\rho s'$.*
2. *Let $K \subseteq X^*$. We say that $s$ is $K$-equivalent to $s'$ if $s$ is $\rho$-equivalent to $s'$, for every $\rho \in K$. In this case, we write $s \approx_K s'$. Otherwise, $s$ is $K$-distinguishable from $s'$, and we write $s \not\approx_K s'$.*
3. *State $s$ is equivalent to $s'$ if $s$ is $\rho$-equivalent to $s'$ for every $\rho \in X^*$. In this case, we write $s \approx s'$. Otherwise, $s$ is distinguishable from $s'$, and we write $s \not\approx s'$.* ∎

In Figure 1, state $s_1$ in (a) is $bb$-distinguishable from state $s_4$ in (b), so we write $s_1 \not\approx_{bb} s_4$.

We say that two FSMs, $M$ and $M'$, are equivalent, if $s_0 \approx s'_0$. So, we say that a FSM correctly implements another FSM if the initial states of the corresponding machines are equivalent.

If $M$ and $M'$ are the same machine, the definition above can be taken as specifying equivalence relations over sets of states $C \subseteq S$. In this case, for a set of input words $R \subseteq X^*$, the relation $\approx_R$ induces a partition of the states in $C$. We denote such partition by $[C/R]$. For example, in Figure 1(a), with $C = \{s_0,s_1,s_2,s_3\}$, $R = \{aaaa\}$ induces the partition $[C/R] = \{\{s_0\},\{s_1\},\{s_2,s_3\}\}$.

The number of pairwise distinguishable states of a FSM is called its index, as defined below.

**Definition 9.** *Let $M$ be a FSM and $C$ be a set of states. The number of equivalence classes induced by the $\approx$ relation over $C$ is denoted by $\iota(C)$. The index of $M$ is $\iota(S)$. If $\iota(S) = |S|$, then the machine is said to be minimal.* ∎

### 3.2 State separators

From Definition 8, two states $s$ and $r$ are distinguishable if and only if there exists a word $\gamma$ with $s \not\approx_\gamma r$. Whenever this happens, we say that $\gamma$ separates $s$ and $r$. We extend this notion, so that we can *separate* any two sets of states. In this case, we use a collection of input sequences instead of just one sequence.

**Definition 10.** *Let $M$ be a FSM, let $A,B$ be two subsets of states, not necessarily disjoint, and let $R \subseteq X^*$ be a set of input words. $R$ is a $(A,B)$-separator if and only if for each pair of distinguishable states $s$ and $r$, such that $s \in A$ and $r \in B$, we have $s \not\approx_R r$.* ∎

To exemplify this new concept, consider machine (a) in Figure 1, and let $A = \{s_0,s_1\}$, $B = \{s_0,s_2\}$ and $C = \{s_0,s_3\}$. The set of input sequences $R = \{ab\}$ is a $(A,B)$-separator, but, since $s_2 \approx_R s_3$, and $s_2 \in B, s_3 \in C$, $R$ is not a $(B,C)$-separator. Note that state $s_0$ is a common element of $A$ and $B$.

Notice that, in this paper, we adopt a more flexible definition of characterization sets than that found in [9]. In the latter, the FSM being minimal is a necessary condition for the existence of a characterization set, while in our definition, any FSM has a characterization set. The same happens with respect to identification sets as defined in [8]. We don't even require a characterization set or an identification set to be minimal. Also, note that, in Definition 10, the sets $A$ and $B$ may have a nonempty intersection. This often happens in the case of characterization sets, which are used to separate any pair of distinguishable states of the machine. Actually, we impose no restriction on what sets of states we may select.

A number of special cases are worth noticing: (i) A $(S,S)$-separator is a *characterization set* for $M$. (ii) An *identification set* for a state $s$ is any $(\{s\},S)$-separator. (iii) For a given set $C \subseteq S$, a $(C,C)$-separator is also called a *partial characterization set for $C$*. (iv) If $R \subseteq X^*$ is a $(A,B)$-separator such that $r \not\approx_R s$, for every pair $r \in A$, $s \in B$, then $R$ is also called a *strict* $(A,B)$-separator.

In Section 5, $R$ is a separator that exemplifies a number of situations: it will be an identification set for a state $s$, a partial characterization set for a set of states $C$, and a strict separator for sets of states $A,B$.

Next, we point out some useful separator properties.

**Observation 11.** *Consider a FSM, $M$. Let $A,B,C$ and $D$ be subsets of states, not necessarily disjoint, and let $T$ and $U$ be sets of input sequences. Let also $r$ and $s$ be states of $M$. Then,*

1. *$T$ is a $(A,B)$-separator if and only if $T$ is a $(B,A)$-separator;*
2. *If $T$ is a $(A,B)$-separator and $U$ is a $(C,D)$-separator, then $T \cup U$ is a $(A \cup C, B \cap D)$-separator;*
3. *If $T$ is a strict $(A,B)$-separator, $r \in A$ and $r \approx_T s$, then $s \notin B$;*
4. *If $T$ is a $(A,B)$-separator, $r \in A$, $s \in B$ and $r \approx_T s$, then $r \approx s$;*
5. *If $T$ is a $(A,B)$-separator, $C \subseteq A$ and $D \subseteq B$, then $T$ is a $(C,D)$-separator.* ∎

We can use a separator to construct another one. With the next lemmas and corollary, we obtain a partial characterization set from a weaker separator. The proofs are available in [13].

**Lemma 12.** *Let $M$ be a FSM. Let $C \subseteq S$ be a set of states, let $B = \mathrm{nbh}(C,1)$ be its close neighborhood and let $R$ be a $(B, B\backslash C)$-separator such that $R$ partitions $C$ into at least $n$ classes, that is, $|[C/R]| \geq n$. If there exist two distinguishable states $r,s \in C$ such that $r \approx_R s$, then $XR \cup R$ separates $C$ in at least $n+1$ classes, that is, $|[C/(XR \cup R)]| \geq n+1$.* ∎

Suppose that we applied the last lemma and obtained a new separator $X_1R$. If there exist two distinguishable states in $C$ that are $X_1R$-equivalent, then we may use the lemma again to obtain a stronger separator, $X_2R$. In fact, the lemma may be used several times successively. We do this in the following.

**Lemma 13.** *Let $M$ be a FSM. Let $C \subseteq S$ be a set of states, let $B = \mathrm{nbh}(C,1)$ be its close neighborhood and let $R$ be a $(B, B\backslash C)$-separator such that $R$ partitions $C$ into at least $n$ classes, that is, $|[C/R]| \geq n$. If $m$ is an upper bound on the number of $\approx$-equivalence classes in $C$, and $l$ is an integer such that $n \leq l \leq m$, then $X_{l-n}R$ separates $C$ in at least $l$ classes, that is, $|[C/X_{l-n}R]| \geq l$.* ∎

**Corollary 14.** *$X_{m-n}R$ is a $(C,C)$-separator.* ∎

This corollary can be used to obtain a partial characterization set for $C$. It generalizes a known result from Chow [4], demonstrated in [2], that gives us the ability to generate characterization sets. The latter result is, in fact, a particular case of Corollary 14, when $C = S$.

A separator for two sets of states $A$ and $B$ can be obtained by selecting a minimal subset of a characterization set that is also a $(A,B)$-separator. Standard methods to reduce a FSM and to obtain a characterization set for it are known [9]. Although this can be used for any FSM, we may obtain shorter separators if we take into consideration the specificities of the FSM being tested.

# 4 Combined Finite State Machines

Many systems are actually aggregations of other, smaller, subsystems. When modeling such systems, it is usual to adopt the *building block strategy* for the development cycle, in which each subsystem is designed, implemented and tested separately. Though each individual part of the system is tested and deemed correct, we have no guarantee that the integrated final implementation is also correct. In order to test such systems efficiently, we formalize below the concepts of combined FSMs.

**Definition 15.** *Let $M = (X,Y,S,s_0,\delta,\lambda)$ be a FSM. A FSM $\dot{N} = (\dot{X},\dot{Y},\dot{S},\dot{s_0},\dot{\delta},\dot{\lambda})$ is called a submachine of $M$ if and only if $\dot{X} = X$, $\dot{Y} \subseteq Y$, $\dot{S} \subseteq S$ and, for every $a \in X$ and $s \in \dot{S}$, we have $\dot{\delta}(a,s) = \delta(a,s)$ and $\dot{\lambda}(a,s) = \lambda(a,s)$.* ∎

The definition ensures that a state of a subsystem behaves exactly in the same way, regardless of whether it is considered a state of a submachine or as new state of the combined machine. A combined FSM is formed by conjoining one or more submachines. That is, a FSM may be constructed by adding new states and new transitions to connect a set of submachines. Since each subsystem has only one entry point, every transition that enters a submachine should end in that submachine initial state. If, for specific needs, a submachine has more then one entry point, then we may consider several submachines, with the same set of states and the same transitions, but with different initial states.

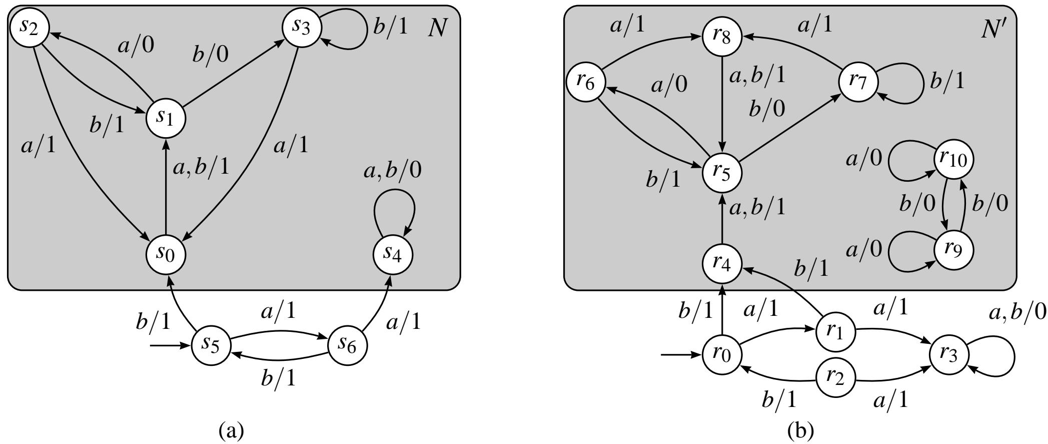



Figure 2: A Combined Finite State Machine and a candidate implementation.

**Definition 16.** *Let $M$ be a FSM and $N$ be a set of submachines of $M$. Define $S_N = \{s \in \dot{S} | \dot{N} \in N\}$ as the set of all submachine states, and $S_M = S \setminus S_N$ as the set of additional states. Also, define $I_N = \{\dot{s_0} | \dot{N} \in N\}$ as the set of all submachines initial states. Then, $M$ is $N$-combined if and only if $s_0 \in S_M$ and, for every pair of states $s$ and $r$ such that $s \in S_M$ and $r \in S_N$, if there exists $a \in X$ such that $r = \delta(a,s)$, then $r \in I_N$.* ∎

In Figure 2(a), we illustrate a combined FSM. The set of submachines, $N$, is formed by the machines defined in Figure 1. For this machine, we have $S_N = \{s_0, s_1, s_2, s_3, s_4\}$, $I_N = \{s_0, s_4\}$ and $S_M = \{s_5, s_6\}$. The initial state is $s_5 \in S_M$. We notice that, in fact, this machine satisfies the properties of Definition 16. For example, for states $s_5 \in S_M$ and $s_0 \in S_N$, since $s_0 = \delta(b, s_5)$, $s_0 \in I_N$.

We shall use the notation introduced in Definitions 15 and 16. So, given a machine $M$ and a set of submachines $N$, we have the sets $S_N$, $S_M$, $I_N$ and submachines $\dot{N}$ in $N$. Moreover, decorations carry over uniformly, *e.g.*, from a FSM $M'$ and a set of submachines $N'$, we have the sets $S'_M$, $S'_N$, and so forth.

# 5 The C-method

We present a new method, named the C-method, to test combined FSM specifications. We assume that an implementation is also a combined FSM in which each submachine is already tested and deemed correct. Also, the number of additional states is limited to a fixed upper bound. If these conditions are satisfied, then the C-method automatically yields a test case suite with full fault coverage.

A submachine can be itself a combined FSM. It can, of course, also be tested using the C-method, giving rise to a recursive testing approach. Notice that the set of submachines may be empty, so one can always use the C-method to directly test FSM specifications. In this particular case, using the C-method is equivalent to using the G-method [2]. Also, notice that it is necessary to test a submachine only once, and then implementations that use it can be tested several times at a reduced cost. Further, retesting is possible, so that, if the specification is changed, only the affected submachines need to be retested. Next, we formalize our fault model. Then, we describe the construction of the test suite.

## 5.1 The fault model

The system specification $M$ is a combined FSM, obtained from a set $N$ of submachines. We assume that $M$ is connected, and that for every pair of states, $s \in S_M$ and $r \in S_N$, we have $s \not\approx r$. Such assumptions are reasonable, because there is no meaning in having unreachable states in the specification, or in reimplementing the behavior of an already available submachine state. We also assume that each submachine $\dot{N} \in N$ has a correct implementation $\dot{N}'$, and denote the set of submachine implementations by $N'$. A system implementation $M'$ is a combination of submachines from $N'$ with up to $m$ new states. The goal is to test $M'$ against $M$. But, first, we need to describe the fault model.

**Definition 17.** *Let $M$ be a FSM specification and let $N$ be a set of submachines of $M$ such that $M$ is $N$-combined. Let $N'$ be a set of FSMs and $m$ be a positive integer. A FSM candidate implementation $M'$ is $(N',m)$-combined if: (i) $M'$ is $N'$-combined; (ii) $\iota(S_M) \leq |S'_M| \leq m$; (iii) for every $\dot{N} \in N$, there exists $\dot{N}' \in N'$ such that $\dot{s_0} \approx \dot{s_0}'$; and (iv) for every $\dot{N}' \in N'$, there exists $\dot{N} \in N$ such that $\dot{s_0}' \approx \dot{s_0}$.* ∎

Figure 2(b) illustrates a candidate implementation for the combined machine depicted in Figure 2(a). We claim that the former obeys Definition 17 with $m = 4$. Clearly, $2 = \iota(S_M) \leq |S'_M| \leq m = 4$. Also, each state in $S_N$ has a corresponding state in $S'_N$. For instance, we have $s_0 \in S_N$ and $r_4 \in S'_N$ such that $s_0 \approx r_4$. Notice that each submachine implementation need not be minimal. For example, we have $r_9 \approx r_{10}$.

## 5.2 Test suite generation

The C-method is now presented. We first obtain some intermediate sets of input words, namely, a partial transition cover set $P$, and two separators $R$ and $T$. We use $R$ to determine a parameter $n$, while $R$ and $T$ are used to define a state attribution $\mathcal{Z}$. Then, we use the relative concatenation operator to connect $P$ and $\mathcal{Z}$, thus obtaining the final test suite. Procedure 1 summarizes all steps. We expand on each step.

THE COVER SET $P$: It is a partial transition cover set for $S_M$ with $\varepsilon \in P$. This set is used to reach every additional state in the specification, so that one can exercise the corresponding transitions. Since states in $S_N$ are known to be already correctly implemented, there is no need to cover them.

THE SEPARATOR $R$: We select $R$ as any $(I_N \cup S_M, S_N)$-separator. This set assumes several different roles, depending on the states we are testing. For example, as a strict $(S_M, S_N)$-separator, $R$ is used to distinguish submachine states from additional states. As a $(I_N, S_N)$-separator, $R$ may be used to identify initial states of submachines, and so on.

THE PARAMETER $n$: The relation $\approx_R$ induces partitions on $M$. Based on this, we define a parameter $l$ by letting $l = |[S/R]| - |[S_N/R]|$. Similarly, $\approx_R$ induces a partition on the states of $M'$. In this case, we have to *choose* a parameter $l'$ with the proviso that $l' \leq |[S'/R]| - |[S'_N/R]|$. If no information about $M'$ is available, we can always choose $l' = 0$. Then, we set $n = \max\{l, l'\}$. This parameter influences the size of the test suite, that is, the larger $n$ is, the smaller the test suite will be. As suggested in the G-method [3], if knowledge is available about the implementation, then $l'$ may be set to larger values, thus giving rise to more succinct test suites. We notice that we always have $m \geq n$, otherwise no correct candidate implementation would be possible.

THE SEPARATOR $T$: It is used to complement $R$, whenever there is a need to identify states in neighborhoods of $I_N$. We define $A = \text{nbh}(I_N, m-n-1)$ and select $T$ to be any $(A, S_N)$-separator. Notice that in the case $m = n$, $A$ contains no element, so we may define $T$ as the empty set.

THE STATE ATTRIBUTION $\mathcal{Z}$: We use $T$ only for input words that reach states in $S_N$. Then, to avoid generating unnecessary test sequences, we use a class attribution $\mathcal{R}$, given by $\mathcal{R}(S_N) = T \cup R$ and $\mathcal{R}(S_M) = R$. We then define a state attribution $\mathcal{Z}$ by letting $\mathcal{Z}(s) = X_{m-n} \underset{s}{\otimes} \mathcal{R}$, for all $s \in S$.

THE TEST SUITE $\pi$: The test suite generated by C-method is computed as $\pi = P \otimes \mathcal{Z}$.

The correctness of C-method is guaranteed by the following theorem.

**Theorem 18.** *Let $M$ be a FSM specification and $M'$ be a FSM candidate implementation, as described in Subsection 5.1. Obtain a test suite $\pi$ using Procedure 1. Then, $s_0 \approx s'_0$ if and only if $s_0 \approx_\pi s'_0$.*

*Proof sketch.* The proof relies on Corollary 14. We use a weaker separator $R$ to obtain a partial characterization set, $Z = X_{m-n}R$, for the set of additional states $S'_M$. We then use $T$ to separate implementation states that are distinguishable only by sequences that reach the initial states of submachines. Once we have a partial characterization set for $S'_M$, we use arguments similar to those appearing in proofs involving the the G-method. We give a complete and detailed proof of the C-method correctness in [13]. ∎

**Procedure 1:** Test suite construction for C-method

**Input:** $M, m$ **Output:** $\pi$
**begin**
Obtain a partial transition cover set $P$ for $S_M$ such that $\varepsilon \in P$ ;
Obtain a $(S_M \cup I_N, S_N)$-separator $R$ ;
Define $l \longleftarrow |[S/R]| - |[S_N/R]|$ ;
Choose $l' \leq |[S'/R]| - |[S'_N/R]|$ ;
Define $n \longleftarrow \max\{l', l\}$ ;
Define $A \leftarrow \text{nbh}(I_N, m-n-1)$ ;
Obtain a $(A, S_N)$-separator $T$ ;
Define $\mathcal{R}(S_M) \longleftarrow R$ and $\mathcal{R}(S_N) \longleftarrow R \cup T$ ;
**foreach** $s \in S$ **do**
Define $\mathcal{Z}(s) \longleftarrow X_{m-n} \underset{s}{\otimes} \mathcal{R}$ ;
**return** $\pi \longleftarrow P \otimes \mathcal{Z}$ ;

# 6 Comparison and Discussion

In this section, we briefly review the W-method and generate test suites for an example specification using W-method and C-method. For this example, we limit the number of sequences generated by each method and give the number of unique and prefix-free test cases [3]. Then, we discuss the general case.

## 6.1 The W-method

The W-method applies to minimal, completely specified and deterministic FSMs. The set of implementation candidates comprehends all faulty machines with up to $m_W$ states. Test suites for this fault model are called $m_W$-complete. The method depends on two sets, $P_W$ and $W$. $P_W$ is a transition cover set for $S$, and $W$ is a characterization set for $S$. Then, an intermediate set $Z_W$ is defined as $X_{m_W-n_W}W$, where $n_W$ is the number of specification states. The final test suite is given by $\pi_W = P_W Z_W$.

## 6.2 Example

We will generate test suites for the specification depicted in Figure 2(a). The test suites are intended for $(N', m)$-combined candidate implementations, with $m = 4$ and where $N'$ is illustrated in Figure 2(b).

USING THE W-METHOD: The specification in Figure 2(a) has $n_W = 7$ states and the candidate implementation has $|S'_N| = 7$ submachines states and up to $m = 4$ additional states. So the minimum value for $m_W$ we may choose is $m_W = 7 + 4 = 11$. Next, we select a transition cover set, $P_W$, and then we choose a minimal characterization set, $W$. Finally the $Z_W$ set is computed.

- $P_W = \{\varepsilon, a, b, aa, ab, aaa, aab, ba, bb, baa, bab, baaa, baab, baba, babb\}$;
- $W = \{aaaa, bb\}$;
- $Z_W = X_{m_W-n_W}W = X_4W$.

So, $\pi_W = P_W Z_W$ and $|\pi_W| \leq |P_W||X_4||W| = 930$. In fact, $\pi_W$ has 256 prefix-free words.

USING THE C-METHOD: We select $P$ as a minimal subset of $P_W$ that is a partial transition cover set for $S_M$. Then a $(S_M \cup I_N, S_N)$-separator $R$ is extracted from $W$. Notice that $R$ is a weaker separator than $W$, since, for example, $s_2 \approx_R s_3$, but $s_2 \not\approx_W s_3$. Next, we first partition the states of $M$ and obtain the value $l$. Since no specific information is available about $M'$ we choose $l' = 0$. From those two values we obtain the parameter $n$. Proceeding, we define $A$ as the $(m-n-1)$-neighborhood of $I_N$, and then we select a $(A, S_N)$-separator $T$ from $W$. Finally, we calculate the state attribution $\mathcal{Z}$:

- $P = \{\varepsilon, a, b, aa, ab, aaa, aab, ba, bb\}$;
- $R = \{aaaa\}$;
- $[S/R] = \{\{s_0\}, \{s_1\}, \{s_2, s_3\}, \{s_4\}, \{s_5\}, \{s_6\}\}$;

- $[S_N/R] = \{\{s_0\},\{s_1\},\{s_2,s_3\},\{s_4\}\}$;
- $l' = 0$, $l = |[S/R]| - |[S_N/R]| = 2$ and so $n = \max\{l',l\} = 2$;
- $A = \text{nbh}(I_N, m-n-1) = \text{nbh}(\{s_0,s_4\},1) = \{s_0,s_1,s_4\}$;
- $T = \{aaaa\}$;
- $\mathcal{R}(S_N) = T \cup R = R$ and $\mathcal{R}(S_M) = R$;
- $\mathcal{Z}(s) = X_{m-n} \underset{s}{\otimes} \mathcal{R} = X_{m-n}R = X_2R$ for every $s \in S$.

So, $\pi = P \otimes \mathcal{Z} = PX_2R$ and $|\pi| \leq |P||X_2||R| = 63$. In fact, $\pi$ has 20 prefix-free test cases. Also, the submachines of Figure 2(a) may have been tested previously using 24 test cases. So, one can use the C-method to test the entire specification using only 44 words.

Comparing the results, the gains afforded by the C-method are evident.

### 6.3 Discussion

The difference between the two test suites obtained in the previous example is mainly due to two factors. First, in the C-method, we use a partial cover set, and so we can use a subset of the cover set used by W-method. Second, since $m - n \leq m_W - n_W$, the set $X_{m-n}$ used by C-method may have exponentially less sequences than the set $X_{m_W - n_W}$ used by W-method. This can be seen by the following theorem. The proof is available in [13].

**Theorem 19.** *Let M be a minimal connected N-combined FSM. Assume that $|X| \geq 2$. Consider the fault model defined by all implementations $M'$ such that $M'$ is $(N',m)$-combined, $|S'_M| = m$ and $|S'_N| = k$. Let $\pi_W$ be the test suite generated by the W-method, with $P_W$ the set used as a transition cover set. Then, we can use the C-method and obtain a test suite $\pi$, associated with a partial transition cover set P obtained from $P_W$, in such a way that $\pi \subseteq \pi_W$, and satisfying*

$$(i)\ \frac{|P_W|}{|P|} \geq 1 + \frac{|X|}{|X|+1}\frac{j}{l}, \quad (ii)\ |\pi| \in O(l(j+l)^2|X|^{m-l+1}), \quad and \quad (iii)\ |\pi_W| \in O((j+l)^3|X|^{m-l+k-j+1}),$$

*where $l = |S_M|$ and $j = |S_N|$.* ∎

This result allows us to compare the test suites generated by both the W-method and the C-method. Clearly, both test suites depend on the cover sets that are used. The first claim in Theorem 19, estimates the ratio between the sizes of the cover sets. It indicates that the larger is the number of submachines states, $j$, compared to the number of additional states, $l$, the greater is the advantage of using C-method. This is expected, since, when using C-method, we do not need to test submachine states. In Theorem 19, the second claim gives a bound to the size of the test suites generated by the C-method. The factor $l|X|$ corresponds to the cover set $P$, the factor $(j+l)^2$ is a rough limit on the size of separator $R$, and the factor $|X|^{m-l}$ comes from the set $X_{m-l}$. This set is used to allow the test of implementations with more states than the specification. Claim (iii) at Theorem 19 concerns the W-method. We have a similar bound, except that there is an extra factor of $|X|^{k-j}$, which corresponds to the difference between the number of submachine states in the implementation, $k$, and in specification, $j$. Since submachines are known to be correctly implemented, we do not need to deal with these states when using the C-method. This indicates that the C-method may generate test suites with exponentially less test cases than W-method.

We also argue that the C-method is scalable. That is, unlike the W-method, which requires that specifications have a small number of states, with the C-method we can test systems with a high number of states, provided that the number of additional states is kept low. This is due the fact that, in spite of the specification being arbitrarily large, the size of the generated test suite is only polynomial on the number of submachines states. Compare this to the bound obtained for W-method. We conclude that scalability is a major advantage of the C-method when testing systems designed with a building block strategy.

# 7 Conclusions

The W-method [4] is widely used to test critical systems modeled as FSMs. However, if the number of states is large, this method, and derivations from it, become impractical. Moreover, in several common situations, using the W-method is inefficient. Such cases include testing modified implementations with minor specifications changes and testing new systems modeled by the building block strategy.

To address these issues, we introduced the concept of combined FSMs, thus capturing the situation when the specification is given by a composition of other, previously tested, submachines. With this new concept, we were able to represent the above situations in a specific fault model. This resulted in the C-method, which can generate smaller test suites for combined FSMs.

We also introduced separators, generalizing the notion of characterization sets. Separators showed to be useful tools to prescribe the generation of test suites. By using separators, instead of characterization set, as in the W-method, we can distinguish only as few states as we need and, therefore, we may use smaller sets of distinguishing sequences, thereby reducing the size of the test suites beg generated.

Further, although the C-method can always obtain test suites that are subsets of those generated using W-method, our method may, in fact, generate exponentially less sequences than W-method.

Finally, we showed that C-method is scalable, provided that the number of additional states is kept small. This means that we can test FSMs with an arbitrarily large number of states if we apply a building block strategy during system development and maintenance.

# References

[1] G. V. Bochmann and A. Petrenko. Protocol testing: review of methods and relevance for software testing. In *ISSTA '94: Proc. of the 1994 ACM SIGSOFT Inter. Sym. on Soft. Testing and Analysis*, pages 109–124, 1994.

[2] A. L. Bonifácio, A. V. Moura, and A. S. Simão. A generalized model-based test generation method. In *6th IEEE Inter. Conferences on Software Engineering and Formal Methods*, pages 139–148, 2008.

[3] A. L. Bonifácio, A. V. Moura, and A. S. Simão. Exponentially more succinct test suites. Technical Report IC-09-07, Institute of Computing, University of Campinas, 2009.

[4] T. S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, 4(3):178–187, 1978.

[5] E. M. Clarke and J. M. Wing. Formal methods: state of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996.

[6] R. Dorofeeva, K. El-Fakih, and N. Yevtushenko. An improved conformance testing method. In *IFIP 25th Inter. Conference on Formal Techniques for Networked and Distributed Systems*, pages 204–218, 2005.

[7] K. El-Fakih, N. Yevtushenko, and G. V. Bochmann. Fsm-based incremental conformance testing methods. *IEEE Transactions on Software Engineering*, 30(7):425–436, 2004.

[8] S. Fujiwara, G. V. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test selection based on finite state models. *IEEE Transactions on Software Engineering*, 17(6):591–603, 1991.

[9] A. Gill. *Introduction to the theory of finite state machines*. McGraw-Hill, 1962.

[10] I. Koufareva, A. Petrenko, and N. Yevtushenko. Test generation driven by user-defined fault models. In *Proc. of the IFIP TC6 12th Inter. Workshop on Testing Communicating Systems*, pages 215–236, 1999.

[11] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - a survey. In *Proc. of the IEEE*, pages 1090–1123, 1996.

[12] G. Luo, A. Petrenko, and G. V. Bochmann. Selecting test sequences for partially-specified nondeterministic finite state machines. In *IFIP 7th Inter. Workshop on Protocol Test Systems*, pages 91–106, 1994.

[13] L. L. C. Pedrosa and A. V. Moura. Testing combined finite state machines. Technical Report IC-10-01, Institute of Computing, University of Campinas, 2010. Available in `http://www.ic.unicamp.br/~reltech/2010/abstracts.html`.

[14] D.P. Sidhu and T.-K. Leung. Formal methods for protocol testing: a detailed study. *IEEE Transactions on Software Engineering*, 15(4):413–426, 1989.

# Verification of Faulty Message Passing Systems with Continuous State Space in PVS

Concetta Pilotto
Computer Science Department
California Institute of Technology
pilotto@cs.caltech.edu

Jerome White
Computer Science Department
California Institute of Technology
jerome@cs.caltech.edu

**Abstract**

We present a library of PVS meta-theories that verifies a class of distributed systems in which agent communication is through message-passing. The theoretic work, outlined in [4], consists of iterative schemes for solving systems of linear equations, such as message-passing extensions of the Gauss and Gauss-Seidel methods. We briefly review that work and discuss the challenges in formally verifying it.

## 1 Introduction

We present a framework for verifying a class of distributed message passing systems with faulty communication. In this class processes (*agents*) communicate by exchanging messages that can be lost, delayed or reordered. In earlier work, we introduced and analyzed the theoretic basis for this protocol by presenting a class of iterative message-passing schemes for solving systems of linear equations [4]. This class of problems has practical applications in areas such as robot pattern formation protocols [15, 5]. For example, decentralized schemes where groups of robots form a fence around a dangerous area can be expressed as message-passing algorithms where agents solve a system of linear equations. Our previous work formalized this class of systems and proved their convergence to the solution, even in the presence of faulty communication. This paper is an addendum to that work, providing a formal model of the problem in PVS [16], and presenting its verification using the tool.

A challenge that arises in verification of distributed systems in general, and this problem in particular, is that systems exhibit a dense state space operating over continuous time. These systems are highly nondeterministic and contain uncountable data-types. Such properties make its verification using implementation based tools, such as model checkers, difficult, and in some cases impossible. As is outlined in this paper, a theorem prover is an appropriate tool to address these issues.

Our model consists of a library of PVS meta-theories [17] built on top of I/O automata [11, 2] with extensions for timed and hybrid systems [10, 14, 13]. The library offers a solution to the verification of distributed linear system solvers in general. By exposing various assumptions, it is applicable to instances of this problem in particular. Our work follows a very large body of literature where theorem provers have been used for modeling [7, 8, 1, 3], and verification [9, 12, 6]

This paper is organized as follows. Section 2, summarizes the class of distributed systems we focus on. Section 3 describes the architecture of our framework, showing PVS code fragments where necessary. Section 4 discusses our verification procedure and presents an application of our framework to an existing pattern formation protocol. We conclude with Section 5.

## 2 Faulty Message Passing Distributed System

In this section, we discuss a class of distributed systems where agents interact by sending messages. Messages may be lost, delayed or arrive out of order. This class consists of message-passing iterative schemes for solving systems of linear equations of the form $Ax = b$, where $A$ is a real-valued matrix of

size $N \times N$, and $x, b$ are real-valued vectors of length $N$, with $N > 0$. The goal of these systems is to iteratively compute the vector $x$ starting from an initial guess vector $x0$. The message passing version of the Gauss, Jacobi and Gauss-Seidel algorithms are examples of iterative schemes belonging to this class.

We model this as a distributed system of $N$ agents that communicate via a faulty broadcast channel. Each agent is responsible for solving a specific variable using a specific equation of the system of linear equations. For example, agent $i$ is responsible for solving the variable $x[i]$ using the $i$-th equation. We consider schemes where each agent repeatedly broadcasts a message containing the current value of its variable, $x[i]$. We assume agents broadcast their value infinitely often, where the number of messages sent within a finite time interval is assumed to be finite. Upon receiving a message, an agent computes a value for its variable using its equation. It assumes that its variable is the only unknown of the equation and sets the values of the other variables to the last message received from the corresponding agents. For example, agent $i$ sets the value $x[i]$ as follows:

$$x[i] := b[i] - \sum_{j \neq i} A[i,j]x[j] \tag{1}$$

where $x[j]$ stores the last message received from agent $j$. We assume that the matrix $A$ (D1) is invertible, (D2) has diagonal entries are all equal to 1, (D3) is weakly diagonally dominant ($\forall i : \sum_{j \neq i} A[i,j] \leq A[i,i]$), and (D4) is strictly diagonally dominant in at least one row ($\exists k : \sum_{j \neq k} A[k,j] < A[k,k]$).

Agents within our model use two variables, $x$ and $z$, to represent the solution of their equation. Given agent $i$, the value of $x[i]$ is the current solution to Eq. (1), while the value of $z[i]$ is the agents current estimate of the solution. Initially, both $x[i]$ and $z[i]$ are set to $x0[i]$. When an agent receives a message, it updates the solution $x[i]$ using Eq. (1). The variable $z[i]$ is updated when the agent "moves," at which point it evolves $z[i]$ according to some predefined dynamics towards $x[i]$.

Using two variables to store the solution vector gives us a better means to represent continuous dynamics. In real-world applications, it is unrealistic to assume instantaneous updates of the solution vector $x$; thus, we maintain the current estimate vector $z$. In robotics, for example, systems of equations are used in pattern formation protocols, where $x$ can be thought of as agent positions. When a robot receives a message, the move from its current location to its newly computed one is not instantaneous. Instead, the robot moves according to some time-related dynamics. In robotics application, $z$ models the current robot positions while $x$ represents their destination locations.

In earlier work, we have provided sufficient conditions for proving correctness of this class of distributed systems [4]. Specifically, these schemes are correct if $x$ and $z$ converge to the solution of the system $A^{-1}b$ as time approaches infinity. More formally,

**Theorem 1** ([4]). *If A satisfies D1–4, then the system converges to* $A^{-1}b$.

In the proof, it is shown that the maximum error of the system at each point of the computation eventually decreases by a factor of $\alpha$, with $\alpha \in [0,1)$. Convergence is then proven using induction over the agent set. As a base case, we show that eventually the maximum error of the agents satisfying D4 is reduced by $\alpha$. Then, assuming that this is the case for agents at distance $k$ from some agent satisfying D4, we show that the property holds for all agents at distance $k+1$. This induction proof can be represented as a forest of trees of size $N$, rooted at agents satisfying D4. Iterating the proof, it is possible to show that starting from some initial maximum error of the system, $E_0$, the error eventually decreases to $\alpha E_0$, then to $\alpha^2 E_0$, then $\alpha^3 E_0$, ..., converging to 0 as time goes to infinity.

## 3 PVS Verification Framework

In this section we describe the tool for verifying this class of distributed systems. Our framework [17], summarized in Figure 1, has been built on the top of the PVS formalization of I/O automata [11, 2] and

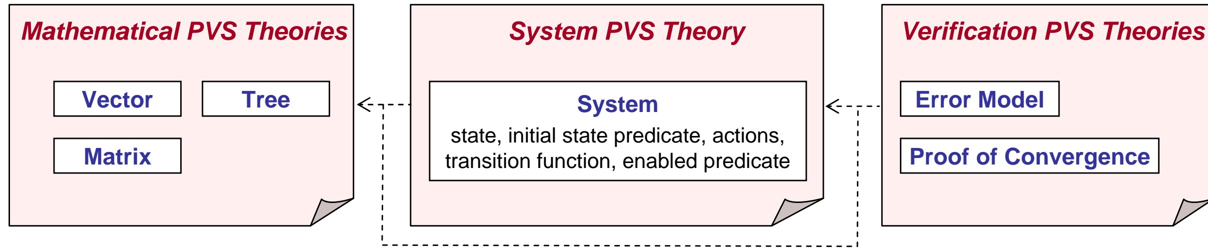



Figure 1: The structure of our PVS framework. A set of mathematical theories describe generic datatypes (Section 3.1). These are used by the theory that models a distributed message passing system (Section 3.1). Finally, we verify that the system converges to the solution using properties of the mathematical structures (Section 3.2).

its extensions to timed and hybrid I/O automata [10, 14, 13]. Our library uses the parts of the PVS NASA library [1]. First, we present the formalization of the system, then discuss the proof of correctness.

### 3.1 Modeling a Distributed Message-Passing System

The system is modeled using the automaton meta-theory [14] that we have specialized for matrix algebra. The automaton consists of system state (Figure 2), system action (Figure 3(b)), enabling condition for those actions (Figure 4(b)), and transition functions (Figure 4(a)). The enabling condition is a means of defining which subset of actions are valid in a given state. Our action set handles system communication and timing, along with subsequent agent state changes. In this section, we describe the system's overall operation from the standpoint of our action set.

We have added the appropriate linear algebra types for the matrices and vectors to the automaton theory:

```
Vector: TYPE = [Index -> real]
Matrix: TYPE = [Index, Index -> real]
```

A vector is modeled as a function from type `Index` to real numbers, and a matrix as a function that maps an `Index` pair to a real number. We have defined these types functionally to facilitate manipulation in PVS. Users supply our library with instances of these types, which we refer to throughout with the variables `A`, of type `Matrix`, and `b` and `x0`, of type `Vector`. The type `Index` is an interval of natural numbers bounded by `N`, with `N` being the number of agents in the system

```
Index: TYPE = below(N)
```

That is, `Index` $= \{0, \ldots, \mathtt{N}-1\}$. The type `Vector` and `Index` belong to the PVS NASA library [1], while the type `Matrix` is our extension to this library.

#### 3.1.1 System state

The state of the system is made up of the state of the agents, along with the state of the communication channel. An agent is responsible for its current and target values, along with the set of messages in the communication channel for which it is the recipient. We define the system state in PVS with the record type `S`, outlined in Figure 2. The fields `target`, and `lastmsg` describe the state of the agents, while `buffer` describes the state of the channel. The field `now` corresponds to the clock of the system, storing the current time. The field `next` is a vector containing, for each agent, the future time that that agent is allowed to execute a `send` action. The `target` field stores the target value of each agent. The `lastmsg`

```
S: TYPE = [# target:  Vector,                      % vector of agents
             lastmsg: Matrix,                      % matrix of values
             buffer: [Index, Index -> Pset],       % state of the channel
             now:     nonneg_real,                 % system clock
             next:   [Index -> nonneg_real] #]     % agent send deadlines
```

Figure 2: System state. Refer to Figure 3(a) for the definition of Pset.

```
Msg  :TYPE = [# loc: real,
                 id: Index #]
Pkt  :TYPE = [# msg: Msg,
                ddl: posreal #]
Pset:TYPE = set[Pkt]
b    :posreal
d    :posreal
```

(a) Channel types

```
ACS: DATATYPE BEGIN
 send(p:Pkt, i:Index, d1:posreal): send?
 receive(p:Pkt, i:Index): receive?
 move(i:Index, delta_t:posreal): move?
 msgloss(p:Pkt, i:Index): msgloss?
 nu_traj(delta_t:posreal): nu_traj?
END ACS
```

(b) Actions of the system

Figure 3: Components of the system automaton

field is a matrix in which the diagonal entries hold the current value of each agent; the non-diagonal entries store the last message that agent $i$ has received from agent $j$. The target and the diagonal entries of lastmsg correspond to the variables $x$ and $z$, respectively, from the mathematical model outlined in Section 2. The field buffer models the communication between agents; each pair of agents $(i, j)$ has a directed communication channel, with $i$ being the sender of the messages in the channel and $j$ being the receiver. The entry buffer(i)(j) of type Pset contains the set of packets sent by $i$ and not yet received by $j$. The type Pset is defined in Figure 3(a).

The initial condition of the system is described using the predicate *starts*?. It holds in the state s if the global clock of this state is set to 0, and the target set to the initial guess, x0:

```
start?(s: S): bool =
 now(s) = 0 AND (FORALL(i: Index): next(s)(i) <= d) AND
 target(s) = x0 AND (FORALL(i, j: Index): lastmsg(s)(i,j) = x0(i))
```

The condition on next vector is necessary because in our model agents send messages infinitely often. This condition ensures that all agents will execute a send action in bounded time. Parameter d, defined formally in Section 3.1.2, is the upper bound on an agents sending rate. Note that the communication channels are not necessarily empty initially.

### 3.1.2 Communication channel

The communication layer is a broadcast channel allowing for lost, delayed, or out-of-order messages. Our model assumes that (C1) messages are either lost or delivered in finite but unknown time and (C2) for each pair of communicating agents the receiver receives infinitely many messages. The first assumption models a communication medium with bounded, but unknown, delay. The second assumption ensures that there are no permanent partitions between communicating agents.

To model such faulty communication, we consider not only packets and channels, but timing and ordering properties as well. Figure 3(a) outlines the PVS data types used for this purpose. Messages

```
trans(a: ACS, s: S): S =
CASES a OF
 nu_traj(delta_t): % ...
 move(i,delta_t): % ...
 send(p,i,d1): % ...
 receive(p,i): % ...
 msgloss(p,i): % ...
ENDCASES
```

(a) Transition function of the system describing the behavior of each action of the system.

```
enabled(a: ACS, s: S): bool =
CASES a OF
 nu_traj(delta_t): % ...
 move(i,delta_t): % ...
 send(p,i,d1): % ...
 receive(p,i): % ...
 msgloss(p,i): % ...
ENDCASES
```

(b) Enabling predicate of the system describing the enabling condition of each action of the system.

Figure 4: Components of the system automaton. The body of the transition function and enabling predicate are defined in Section 3.1.3

sent between agents (`Msg`) are represented as a record, consisting of the agent's location and identifier. Messages, along with their delivery deadline, are contained within packets (`Pkt`). Sets of packets (`Pset`) make up a dedicated, directed, channel between two agents. Because a set lacks ordering, it makes it an appropriate type for a communication channel that allows for out-of-order messages. Timing within the channel is handled by the constants `b` and `d`. The former is an upper bound on packet deadlines—at most `b` units of time—and models maximum message delay. In our model, each packet can be received at any time in the interval $[\texttt{now}, \texttt{now}+\texttt{b}]$. The constant `d` is an upper bound on the interval between consecutive `send` actions. Combined with `next`, it ensures that the `send` action is executed infinitely often. The future time that an agent is allowed to execute a `send` action is set to a value in the interval $[\texttt{now}, \texttt{now}+\texttt{d}]$.

#### 3.1.3 System actions

Actions within our system consist of agent movement and message transmission, channel manipulation, and system clock maintenance. Their corresponding PVS definitions are outlined in Figure 3(b) and 4. The `send`, `receive`, and `move` actions are executed by agents, while the `msgloss` action is used by the communication channel to simulate packet loss. Finally, the `nu_traj` action updates the system time. This section describes the behavior of each action, implemented by the function `trans` (see Figure 4(a)), and when they are enabled, implemented by the predicate `enabled` (see Figure 4(b)).

**New trajectory.** The `nu_traj` action advances the time variable of the system, `now`, by `delta_t` units, where `delta_t` is the input parameter of the action;

```
| nu_traj(delta_t: posreal): s WITH [now := now(s) + delta_t]
```

It is enabled when the new value of the global clock does not violate a packet deadline:

```
| nu_traj(delta_t): FORALL(p: Pkt): ddl(p) >= now(s) + delta_t
```

**Agent move.** The `move` action models the movement of an agent from its current value to one based on its locally computed solution to the equation of the system. The parameters of the action are the agent

that moves and the time interval. In our implementation, agent $i$ sets $z[i]$ (stored in `lastmsg(i,i)`) to $x[i]$ (stored in `target(i)`) and advances the global clock of `delta_t` units. In PVS

```
move(i: Index, delta_t: posreal): s WITH
  [ lastmsg := lastmsg(s) WITH [ (i,i) := target(s)(i) ],
        now := now(s) + delta_t]
```

This action can be executed only if packet deadlines are not violated by the new time of the system:

```
move(i, delta_t): FORALL(p: Pkt): ddl(p) >= now(s) + delta_t
```

**Agent send.** When executing a `send` action, agent `i` broadcasts its packet `p` to all agents in the system and schedules its next `send` action:

```
send(p: Pkt, i: Index, d1: posreal): s WITH [
  buffer := LAMBDA (k, j: Index):
   IF ((k = i) AND (j /= i)) THEN union(p, buffer(s)(k,j))
   ELSE buffer(s)(k,j)
   ENDIF,
  next := next(s) WITH [(i) := next(s)(i) + d1 ]]
```

In updating the buffer, the agent is adding its packet, `p`, to all of its outgoing channels. Notice that an agent does not send a message to itself. The `send` action is executed only if the time when the agent is allowed to send is equal to the global time of the system. Furthermore, the sent packet must contain the identifier of the agent, its current target location, and correct packet deadline. The detailed PVS code follows

```
send(p, i, d1): next(s)(i) =
  now(s) AND d1 <= d AND id(msg(p)) = i AND
  loc(msg(p)) = target(s)(i) AND ddl(p) = now(s) + b
```

**Agent receive.** When agent `i` receives packet `p`, it updates the `lastmsg` variable, computes a new value for its `target`, and removes the packet from the channel:

```
receive(p: Pkt, i: Index):
 LET m: Msg = msg(p), j: Index = id(m), l: real = loc(m),
     Ci: vector = update(row(lastmsg(s), i), j, l) IN s WITH
  [ buffer := buffer(s) WITH
    [ (j,i)   := remove(p, buffer(s)(j,i)) ],
      lastmsg := lastmsg(s) WITH [ (i,j) := l ],
      target  := target(s)  WITH [   (i) := gauss(Ci,i) ]]
```

The `gauss` function implements Eq. (1). The action can be executed if the `p` is in the channel from `msg(p)` to `i`, and its deadline does not violate the global time of the system,

```
receive(p,i): buffer(s)(id(msg(p)), i)(p) AND ddl(p) >= now(s)
```

**Message loss.** Message loss is modeled by removing a given packet from a directed channel:

```
msgloss(p: Pkt, i: Index): LET m: Msg = msg(p), j: Index = id(m) IN s
  WITH [ buffer := buffer(s)
    WITH [ (j,i) := remove(p, buffer(s)(j,i)) ]]
```

It is enabled only if the packet belongs to this channel:

```
msgloss(p,i): buffer(s)(id(msg(p)), i)(p)
```

## 3.2 Verification Meta-Libraries

This subsection focuses on proving system correctness. We discuss how the error of the system is represented, and how it is used to prove convergence in PVS. As outlined in [4], and briefly discussed in Section 2, we prove that the error of each agent converges to 0 as time tends to infinity.

### 3.2.1 Error Model

The error of an agent is defined as the distance between its current value and its desired value. We define the vector of desired values, `xstar`, axiomatically, using the standard definition:

```
xstar_def_ax: AXIOM FORALL(i: Index): xstar(i) = gauss(xstar,i)
```

During system execution, the value of an agent is represented in three places: within its state, within the set of packets in transit on its outgoing channels, and within the received message field of other agents. Although these are all values of the same agent, agent dynamics, message delay and reordering do not guarantee their equality. In the proof of correctness, it is enough to consider only the maximum error of these values. We define the error of the system axiomatically

```
me_all_error: AXIOM FORALL(i: Index): mes(s,i) <= me(s)
me_ex_error:  AXIOM EXISTS(i: Index): mes(s,i)  = me(s)
```

where `mes` is the maximum error of agent `i` within in the system, and `me` is the error of the system, defined as maximum of all agents errors within the system.

### 3.2.2 Proof of Correctness

As discussed in Section 2, our proof of correctness relies on certain assumptions about the given matrix `A` (see D1–4). Invertibility, along with diagonal and strict-diagonal dominance, were modeled, respectively, as

```
inv?(m): bool = EXISTS(n: Matrix):
                   prod(m,n) = prod(n,m) AND diag?(prod(m,n))
dd?(m):  bool = FORALL(r: Index): sum(row(abs(m),r),r) <= abs(m(r,r))
sdd?(m): bool = EXISTS(r: Index): sum(row(abs(m),r),r) <  abs(m(r,r))
```

where `m` is of type `Matrix`. Using these definitions, we can describe the assumptions on `A` made by the model. We use the PVS assumption facility to access these properties within the meta-theory and obligate users of our library to discharge them;

```
ASSUMING
  inverse_exist: ASSUMPTION inv?(A)                            % D1
  diag_entry:    ASSUMPTION FORALL(i: Index): A(i,i) = 1       % D2
  diag_dominant: ASSUMPTION dd?(A)                             % D3
  strictly_diag_dominant: ASSUMPTION sdd?(A)                   % D4
ENDASSUMING.
```

Reasoning about system convergence requires the analysis of the system along an arbitrary execution. Our responsibility is to show that (E1) the error of the system does not increase, and that (E2) it

eventually decreases by a lower-bounded amount. Using the diagonally dominant assumption on A, we can prove E1:

```
not_incr_error: LEMMA enabled(a,s) IMPLIES me(s) >= me(trans(a,s))
```

To prove E2, as discussed in Section 2, we built a representation of the forest structure in PVS. The tree is represented by functions ancs, root_t? and parent using the PVS list structure defined in the prelude.

```
ancs: FUNCTION[Matrix, Index -> ListIndex]
root_t?(m: Matrix, i: Index): bool = sdd?(m,i) AND null?(ancs(m,i))
parent(m: Matrix, i: Index): Index =
   IF root_t?(m,i) THEN i ELSE car(ancs(m,i)) ENDIF
```

where ancs is a function which, given a matrix and agent identifier, returns the complete path of the agent to a rooted node in the tree. The path is defined as a list of agent identifiers having type ListIndex. Note that rooted agents satisfy D4.

Next, we define a factor $\alpha$ by which the system will eventually decrease. Given the rooted forest, for each node of the forest we recursively define the quantity p_value. We prove that this value is positive and (strictly) upper bounded by 1. The factor $\alpha$ is the maximum of these quantities. In PVS,

```
alpha_all: AXIOM FORALL(i: Index): p_value(i) <= alpha
alpha_ex:  AXIOM EXISTS(j: Index): alpha = p_value(j)
```

Using induction on the forest, we prove that the maximum error of the system eventually decrease by $\alpha$. Assuming that the error of the system is upper-bounded by $W$, the base case is to prove that the error of the roots of the tree eventually decreases by $\alpha$. From there, we prove that eventually the error of the node is upper bounded by $W \cdot \alpha$, assuming that the error of all ancestors of a node is upper-bounded by the same quantity. To handle this within the theorem prover, we defined our own induction scheme:

```
p: VAR [Index -> bool]
induct_ancs: THEOREM
  (null?(ancs(m,i)) IMPLIES p(i)) AND
  (cons?(ancs(m,i)) AND p(car(ancs(m,i))) IMPLIES p(i))
    IMPLIES FORALL(j: I | member(j, ancs(m,i)) OR j = i): p(j)
```

This theorem ensures that if we prove that some property holds at the root of the tree, and, given a node, we prove that it holds at the node, given that it holds for its parent, then we can safely derive that the property holds for all nodes along a path of the tree. In this case, that property is the factor of $\alpha$ decrease.

# 4 Framework Discussion

In this section, we offer commentary on our experience using PVS and present an application of our framework.

## 4.1 PVS Implementation

Our library consists of over 30 lemmas, almost 2000 proof steps. We took advantage of PVS pre- and user-defined types for modeling agent and channel states. Implementation of the system infrastructure consumed about 30% of our effort. Vectors, matrices and trees were used extensively throughout our library. The PVS NASA libraries provided some relief, but modeling diagonally dominant matrices and proving lemmas on products of matrices and vectors forced us to extend them. Although NASA does provide a representation of trees, their recursive implementation made proving properties we required

very difficult. Unlike the NASA implementation, where they visit the tree starting from the leaves, we were more interested in both proving properties on the tree starting from the root and inducting over the structure as well. Further, the NASA library has limited support for weighted directed graphs, something critical for our model. For these reasons, we preferred to give an implicit definition of trees and ensure the needed properties axiomatically.

While proving the convergence property of the system, we gave implicit definitions to many nonlinear functions for convenience within the proofs. For example, we did not define the maximum error of the system recursively. We preferred, instead, to use two axioms and define the maximum as the upper bound on the error of the agents such that the value is reached by some agent (see `me_all_error` and `me_ex_error` in Section 3.2.1). This choice reduced the number of proofs in the library, as we did not derive these facts from their recursive definitions. Proving these facts in PVS is possible, but beyond the scope of our work.

We managed the proof of Theorem 1 by breaking it into smaller lemmas. This allowed us to tackle small proofs where the goal was to show that eventually a specific property held. For example: proving that the error of the target position of an agent eventually decreases by the constant factor $\alpha$; then proving that its error in the outgoing channels eventually decreases by this factor; and finally showing that its error stored in the state of the remaining agents eventually decreases by this factor. We showed these lemmas using the two assumptions on the communication medium (see C1–2). These assumptions simplified our proofs since we did not consider traces of the automaton in the proof of convergence. Using this collection of sub-lemmas, we were able to prove, directly, that after $d+b$ time units the maximum error of the agent decreases by the factor $\alpha$.

### 4.2 Applications

In earlier work, we discussed a specific pattern formation protocol where the goal of the system was for agents to form a straight line [5]. The system consisted of $N$ robots, with robot 0 and $N-1$ fixed throughout the execution. Agent $i$ ($0 < i < N-1$) only communicated with its immediate neighbors, robots $i-1$ and $i+1$. Based on this communication, $i$ computed its new position as the average of the left and right received values.

This protocol can be modeled as a system of linear equations, and, as such, can be verified using our library. We instantiate the matrix $A$ with a tri-diagonal matrix having the value 1 along the main diagonal, and $-0.5$ on the secondary diagonals (with the exception of rows 0 and $N-1$, which have 0 on the secondary diagonals). Given this structure, we are obligated to discharge the assumptions of the library outlined in Section 3.2.2: that the input matrix $A$ satisfies D1–4.

## 5 Conclusions

We have presented a verification framework for distributed message-passing systems that takes into account a faulty communication medium. This framework consists of a library built within the PVS theorem prover that allows for the verification of distributed algorithms for solving systems of linear equations. We have also detailed the implementation of the framework, outlining our use of a specialized automaton theory and induction scheme. We have discussed challenges in the implementation and presented an application where the framework is used to verify a specific robot pattern formation protocol. Future work includes extending these results to a richer class of systems, such as non linear convex systems of equations, and providing their corresponding verification using a theorem prover.

## 6 Acknowledgments

The authors would like to thank the referees for their very helpful and constructive comments. This work was supported in part by the Multidisciplinary Research Initiative (MURI) from the Air Force Office of Scientific Research.

## References

[1] Nasa langley pvs libraries. http://shemesh.larc.nasa.gov/fm/ftp/larc/PVS-library/pvslib.html, 2010.

[2] M. Archer, C. Heitmeyer, and S. Sims. TAME: A PVS interface to simplify proofs for automata models. In *Proceedings of the 1st International Workshop on User Interfaces for Theorem Provers (UITP '98)*, July 1998.

[3] L. Bulwahn, A. Krauss, and T. Nipkow. Finding lexicographic orders for termination proofs in isabelle/hol. In *Proceedings of the 20th International Conference on Theorem Proving in Higher Order Logics (TPHOLs '07)*, volume 4732 of *Lecture Notes in Computer Science*, pages 38–53, Berlin, Heidelberg', September 2007. Springer-Verlag.

[4] K. M. Chandy, B. Go, S. Mitra, C. Pilotto, and J. White. Verification of distributed systems with local-global predicates. *To Appear: Formal Aspect of Computing*.

[5] K.M. Chandy, S. Mitra, and C. Pilotto. Convergence verification: From shared memory to partially synchronous systems. In *Proceedings of 5th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS '08)*, volume 5215 of *Lecture Notes in Computer Science*, pages 217–231. Springer Verlag, September 2008.

[6] P. Frey, R. Radhakrishnan, H.W. Carter, P.A. Wilsey, and P. Alexander. A formal specification and verification framework for time warp-based parallel simulation. *IEEE Transition on Software Engineering*, 28(1):58–78, 2002.

[7] H. Gottliebsen. Transcendental functions and continuity checking in pvs. In *Proceedings of the 13th International Conference on Theorem Proving in Higher Order Logics (TPHOLs '00)*, volume 1869 of *Lecture Notes in Computer Science*, pages 197–214, Berlin, Heidelberg, August 2000. Springer-Verlag.

[8] J. Harrison. *Theorem Proving with the Real Numbers*. Springer-Verlag, 1998.

[9] P.B. Jackson. Total-correctness refinement for sequential reactive systems. In *Proceedings of the 13th International Conference on Theorem Proving in Higher Order Logics (TPHOLs '00)*, volume 1869 of *Lecture Notes in Computer Science*, pages 320–337, Berlin, Heidelberg, August 2000. Springer-Verlag.

[10] D.K. Kaynar, N.A. Lynch, R. Segala, and F. Vaandrager. *The Theory of Timed I/O Automata (Synthesis Lectures in Computer Science)*. Morgan & Claypool Publishers, 2006.

[11] N.A. Lynch and M.R. Tuttle. An introduction to Input/Output automata. *CWI-Quarterly*, 2(3):219–246, September 1989.

[12] S. Maharaj and J. Bicarregui. On the verification of VDM specification and refinement with PVS. In *Proceedings of the 12th International Conference on Automated Software Engineering (ASE '97)*, page 280, Washington, DC, USA, November 1997. IEEE Computer Society.

[13] S. Mitra. *A Verification Framework for Hybrid Systems*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2007.

[14] S. Mitra and M. Archer. PVS strategies for proving abstraction properties of automata. *Electronic Notes in Theoretical Computer Science*, 125(2):45–65, 2005.

[15] R. Olfati-Saber, J.A. Fax, and R.M. Murray. Consensus and cooperation in networked multi-agent systems. *Proceedings of the IEEE*, 95(1):215–233, January 2007.

[16] S. Owre, J.M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *Proceedings of 11th International Conference on Automated Deduction (CADE '92)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

[17] C. Pilotto and J. White. Infospheres project. http://www.infospheres.caltech.edu/nfm, January 2010.

# Phase Two Feasibility Study for Software Safety Requirements Analysis Using Model Checking

Gregory Turgeon
PPT Solutions, Inc.
Huntsville, Alabama, USA
greg.turgeon@pptsinc.com

Petra Price
PPT Solutions, Inc.
Huntsville, Alabama, USA
petra.price@pptsinc.com

Keywords: Formal verification, Model checking

**Abstract**

A feasibility study was performed on a representative aerospace system to determine the following: (1) the benefits and limitations to using SCADE®, a commercially available tool for model checking, in comparison to using a proprietary tool that was studied previously [1] and (2) metrics for performing the model checking and for assessing the findings. This study was performed independently of the development task by a group unfamiliar with the system, providing a fresh, external perspective free from development bias.

## 1 Introduction

Reviewing software requirements for a system is an important task, as during the requirements phase approximately 70% of errors are introduced [2]. The earlier those errors are detected, the less costly they are to fix, making the requirements phase an opportune time to find and correct errors. However, typically only 3.5% of errors are removed during the requirements phase [2]. Model checking makes requirements reviews more efficient than manual techniques, as it can identify more defects in less time.

Historically, major drawbacks to the application of formal methods to industry included the reluctance to introduce more time and expense into the software lifecycle process and the fear that formal methods is too proof-heavy and therefore requires expert knowledge of mathematics. However, studies have shown that using a process to formally verify software actually can save time and money during a project [3]. Furthermore, while some knowledge of mathematics is needed, expertise is certainly not necessary because formal methods have evolved from being solely an academic pursuit and is now gaining acceptance in the industrial realm. With this evolution comes improved ease of use and applicability.

A previous feasibility study was performed on a representative aerospace system to assess the viability of a particular set of techniques and tools to perform formal verification of the software requirements [1]. The study explored the types of potential safety issues that could be detected using these tools and determined the level of knowledge required, the appropriateness and the limitations of the tools for real systems, and the labor costs associated with the technique.

The conclusions for the previous study were:

1) The model checking tool set was useful in detecting potential safety issues in the software requirements specification.
2) The particular tool set was not able to model the entire system at one time. The model had to be partitioned, and then assumptions had to be verified about the interfaces between the sections.
3) With basic training in Matlab®/Simulink® and specific training on the tool, engineers were able to become productive with this method in a reasonable time frame.
4) The costs to perform the analysis were commensurate with the system being modeled.

The purpose of this, the subsequent study, was to:

1) Determine if another tool set offers increased scalability and is able to model and verify the entire system with no partitioning.
2) Determine if another tool set is more user-friendly, including better commercially available training, fewer bugs and crashes, and increased ease of use.
3) Develop metrics for the number of requirements that can be modeled and verified per labor hour and the number of safety and other defects found per requirement.
4) Prototype methods for modeling and verifying design aspects.
5) Document training requirements and guidelines.

## 2 Overview of System

In this phase of the study, the same aircraft vehicle management system (VMS) was analyzed as in the previous phase. The VMS determines the overall health status of the aircraft to allow for maximum availability of critical systems. The VMS consists of two redundant computers that are both safety- and mission-critical. The VMS interfaces with the aircraft avionics and the radio communication systems. A data link between the redundant computers provides channel state data and synchronization. The VMS is subdivided into 18 individual managers such as a Communications Manager and Electrical Systems Manager. Due to the proprietary nature of the system, Figure 1 is purposefully general. Elements are encapsulations of functionality either described in the SRS or included by the modelers for clarity and organization.

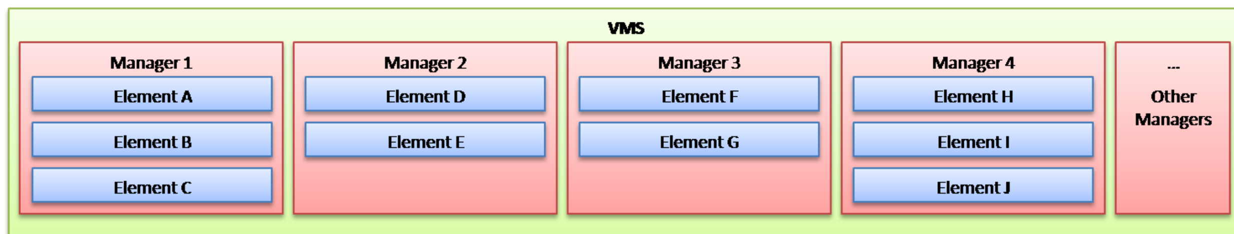



Figure 1: Overview of the System

The VMS performs vehicle mode, flight planning, navigation, and vehicle health functions. There are redundant VMS computers, with one always acting as a master. If the master is determined to be bad, then the slave VMS will assume control of the vehicle. The VMS interfaces with sensors, avionics, and actuators that perform the low-level communications, position, and control surface functions.

The model consisted of 104 block diagrams and 11 state machines, all organized hierarchically. Some state machines were depicted in the requirements document, while the modelers added others when the requirements nicely fit into one.

## 3 Methodology

### 3.1 Stages of Analysis

Analysis begins with a system safety engineer generating system safety requirements based on the system requirements definition and system design. These system safety requirements form the basis of safety properties, which are translated into SCADE® design verification operators. Concurrently, a model of the software requirements is built in SCADE®. Then the SCADE® Design Verifier™ tool is used to determine if the safety properties hold over the entire software requirements model.

For example, the VMS can have one of two designations: master or slave. Having exactly one VMS designated as master is very important for the correct functioning of the entire system, and therefore having one and only one VMS declared as master was a safety property. To determine if this safety property always held, we ran verifiers such as the following: both VMSs are simultaneously master, both VMSs are simultaneously slave, and one VMS is master while the other VMS is slave. All verifiers returned true, meaning that it is possible for the system to have one master, two masters, or no masters; the latter two situations can lead to hazardous outcomes.

Additional algorithm detail is then modeled in Stage 2 using the software design. The safety properties are modified as necessary and formally verified against the design model.

After performing Stages 1 and 2 of the safety analysis, the analyst prepares a safety property report, which lists assumptions made while verifying the system, requirements and design defects found, potential safety hazards, and any other pertinent findings. Based on the safety property report, refinements to the system and software requirements and designs can be made. Changes in requirements or design trigger a new round of analysis. The cycle can continue until no more defects are detected.

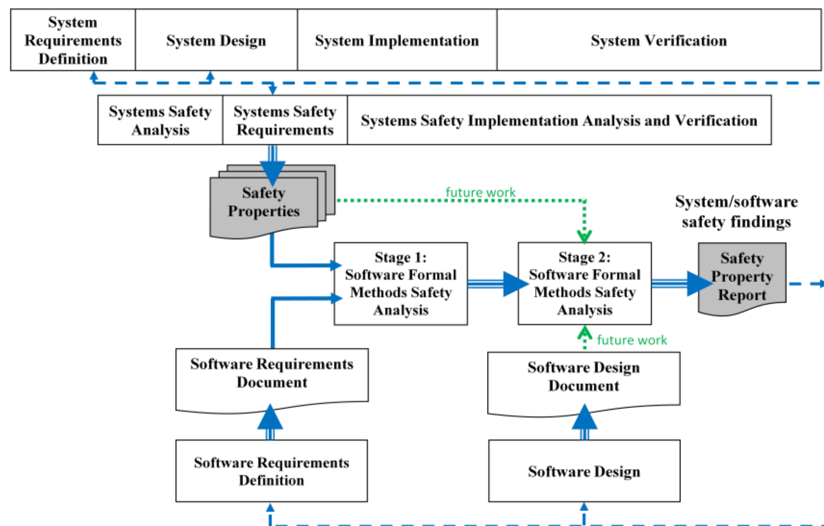



Figure 2: Safety Analysis Stages

### 3.2 Toolset

The tool set consisted of SCADE® version 6.1 including SCADE® Simulator and Design Verifier™. After creating the requirements model in SCADE® Simulator, the model is automatically loaded into the Design Verifier™. The safety properties are also automatically loaded into Design Verifier™. From there, Design Verifier™ determines the status of each property and returns either valid or invalid. Those findings are summarized in the safety property report.

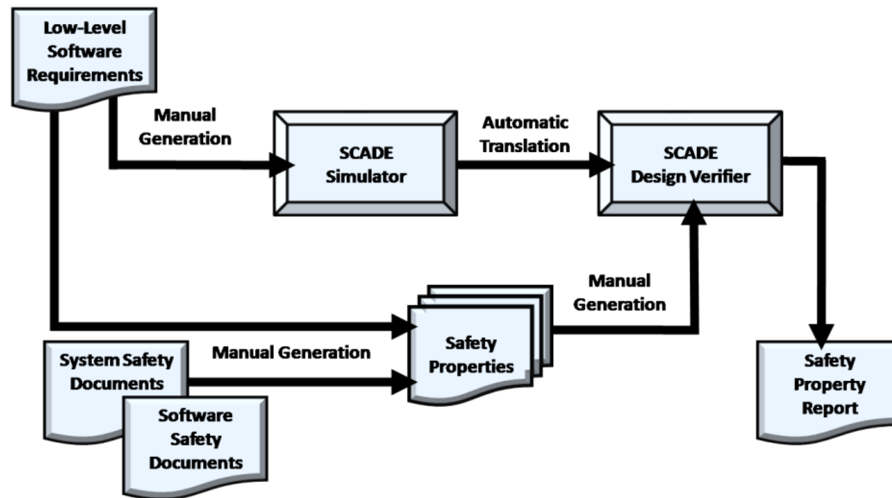



Figure 3: Safety Analysis Toolset

### 3.3 Detailed Methodology

The first step is to create a SCADE® model based on a set of requirements. When creating the model, it is important to keep the model as true to the requirements as possible. Any differences with the requirements from either introducing or eliminating defects weaken the overall effectiveness of the analysis. Any assumptions made in creating the model should be recorded in the safety property report. These requirements assumptions must then be dispositioned by the software requirements engineering team.

After the model is made, the properties to be checked are modeled in SCADE® alongside the requirements model. Four main categories of properties are considered: those based on critical safety conditions, those based on inputs, those based on requirements, and those based on system safety.

The properties based on safety conditions are modeled upon the safety conditions as they are written. Sometimes it is necessary to add constraints to the properties because the safety conditions are vague or unclear. Such constraints are noted and kept to a minimum.

A second method is to model properties based on inputs: each input to the SCADE® model is assessed for its range of values and the possible system repercussions of each of those values. For example, what should occur if both VMSs report a health status of unhealthy? What should occur if a VMS receives an invalid address? These types of properties are written to show that the system behaves as expected based on varying inputs.

A third method is to model properties based on requirements to verify the SCADE® implementation. These properties mainly test functionality, like "If both VMSs are not healthy, then the software in VMS 1 shall assume the responsibility as master." These properties check that the SRS was modeled correctly and behaves as expected. These properties are not very interesting, in that they are meant to show that the modelers faithfully modeled the requirements.

The system safety properties are modeled from a standpoint overlooking the various systems, their overall behavior, and the interactions among them. These properties included ones like "Subsystem A must send signal B to subsystem C before action D takes place."

The next step is to assess the validity of the properties using the SCADE® Design Verifier™, a companion to SCADE® Simulator. The Design Verifier™ is backed by Prover® Plug-In, a commercially available proof engine. The analyst creates a proof objective for each property. He then executes each proof in Design Verifier™, with the results being either true (the property always holds), false (the Verifier™ found a counterexample), or undetermined (meaning either the model has errors or certain settings are preventing the Verifier™ from completing its run).

If a property returns false, then three points of error should be considered. First, the error can come from incorrect implementation of the property. Second, the error can come from incorrect implementation of the model. If both the model and property are correct, then the error can come from an incorrect requirement in the SRS. These sources of error are usually checked in the stated order and the fix to the appropriate area is made, or in the case of an incorrect requirement, the error is documented.

If the error is a result of an incompatible SRS requirement, then the property can be edited to include additional constraints or invariants not specified in the software requirements. The addition of these constraints and invariants can lead to a true property, meaning that the original software requirements lacked specificity, were poorly worded, or otherwise were incompatible with the system. It is important to note that when adding constraints to a property, adding the fewest number of constraints leads to the strongest property. That is, too many constraints can narrow the focus of a property to the point it is no longer useful or meaningful.

This process of modeling properties, checking properties, and modifying properties continues. From this process, certain areas of the requirements specification usually emerge as vulnerable. The design and implementation of these at-risk areas should be reconsidered, as they might affect the safety of the system under analysis.

The entire analysis process can be repeated, incorporating more critical software requirements and design components.

### 3.4 Process Recommendations

These actions were found to save time, make the model checking process easier, and solve several bookkeeping issues. The first three actions are SCADE®-specific. A note about operators: In SCADE®, an operator is a computing unit or block with inputs and outputs. Operators can be pre-defined, like a Boolean *not* operator, or they can be user-defined, such as an operator that computes a custom equation or performs a specific function.

- Hierarchically organize the model using packages. (Packages are like folders.)
- Use structures to compose an operator's inputs and outputs. That way, when an operator gains an input, only the structure changes.
- An operator's contents can be spread across multiple layers so each layer is confined to the dimensions of one screen.
- Use comments to identify requirements and safety properties.
- The organization of proof objectives should match the organization of the model. That is, each operator should have its own set of proof objectives.

## 4 Results

### 4.1 Portion of System Modeled

Out of 1124 requirements in the SRS, we modeled 925 requirements, or 82%. We modeled requirements from every system described in the SRS. Of the 20 systems, we modeled 100% of 17 of them and more than 90% of two additional ones. The remaining 18% of requirements either did not add value to the model or were not related to software. The remaining requirements included ones that described hardware-software interaction and design.

### 4.2 Assumptions

We classified assumptions as any educated guess, clarification, or specification we had to make that we felt was missing from the SRS. We limited assumptions to those clarifications that we felt the requirements authors themselves assumed but did not include in the documented requirements specification. Making the assumptions allowed more of the model to function as expected, which let us investigate deeper and uncover more complex defects. Assumptions were needed to either refine a requirement or to disambiguate among multiple interpretations of a requirement. For example, we had to assume a precedence for every transition in the state machines because no precedence was stated. This kind of assumption was common. An example of a more severe assumption would be assuming that the VMSs are communicating in order for some requirements to always be true. We tracked assumptions both in a spreadsheet and as a note in the model itself. SCADE® does have the capability to create assertions, which are assumptions embedded into the model. We did not use assertions.

In order to create the model and thereby properly formalize the SRS, we made 121 assumptions, which is about one assumption per every 7.6 requirements. The following table gives some examples of assumptions and their associated requirements.

| Requirement | Assumption |
|---|---|
| If the VMS is in State S and is the master, it shall send a synchronization message to the other VMS. | The VMSs are communicating. |
| Thirty-five seconds after power up, the VMS shall begin periodic communications with the other VMS. | "Periodic" communications occur on every cycle. |
| When counting the number of times Event E occurs, the software shall not log more than the limit specified in the spreadsheet. | The name of the spreadsheet and where to find it are not indicated. Assumed that the limit was a constant. |
| The software shall read the RT address of its terminal to determine its own identity. | The RT address signal is latched. |

Table 5: Examples of Assumptions

The following example illustrates how formalizing requirements can lead to early defect detection. A requirement like the following, whose combinatorial logic is ambiguous, can be interpreted in more than one way: "When the following is true, the software shall clear variable A: variable B has been below 88% for at least 2 seconds and either variable C is true or variable D is true and variable E is false." It is unclear which combination should be implemented, as both of the following formal interpretations fit the English specification.

$$(B < 0.88 \text{ for} >= 2 \text{ seconds}) \ \& \ (C \mid D) \ \& \ (!E) \tag{1}$$

$$(B < 0.88 \text{ for} >= 2 \text{ seconds}) \ \& \ (C \mid (D \ \& \ !E)) \tag{2}$$

As the expression (2) was the one intended, simply separating the requirements as shown below resolves the ambiguity.

$$(B < 0.88 \text{ for } >= 2 \text{ seconds}) \ \& \ C \quad (3)$$

$$(B < 0.88 \text{ for } >= 2 \text{ seconds}) \ \& \ D \ \& \ !E \quad (4)$$

### 4.3 Defects Detected

Whereas requirements that needed further refinements warranted assumptions because we felt comfortable making a decision about the intention of the requirement, other requirements problems were not as easy to solve. We called these problems "defects" because we did not feel capable of resolving them ourselves, even with assumptions. For example, several inconsistencies among requirements were found and documented. These include duplicated requirements and conflicting requirements.

Out of the 925 requirements we modeled, we found 198 requirements defects, or about one defect per every 4.7 requirements. Fifty-four (27%) of the defects were found through traditional IV&V methods. Sixty-seven (34%) were found while building the model, and 77 (39%) were found using Design Verifier™. Some representative defects are in the following table.

| Method of Detection | Requirement and Defect |
|---|---|
| Manual IV&V | *If in State A and Input I is true, go to State S.*<br>*If in State A and Input I is true, go to State T.*<br>Requirements conflict because states S and T are mutually exclusive. |
| Model Creation | *If the VMS is Master and the other VMS fails, then it shall remain Master.*<br>What if the VMS is Slave and the other VMS fails? |
| Model Checking | *If the Launch Abort Command input is true, then the launch abort sequence should begin.*<br>Modeled property shows a counterexample where receiving a "Launch Abort Command" does not result in the software signaling to abort the launch sequence. |

Table 6: Example Defects

We classified each requirement that returned a counterexample in one of four categories: catastrophic, critical, marginal, or negligible. Analysis of the system through model verification found 62.5% of all potentially catastrophic defects we found.

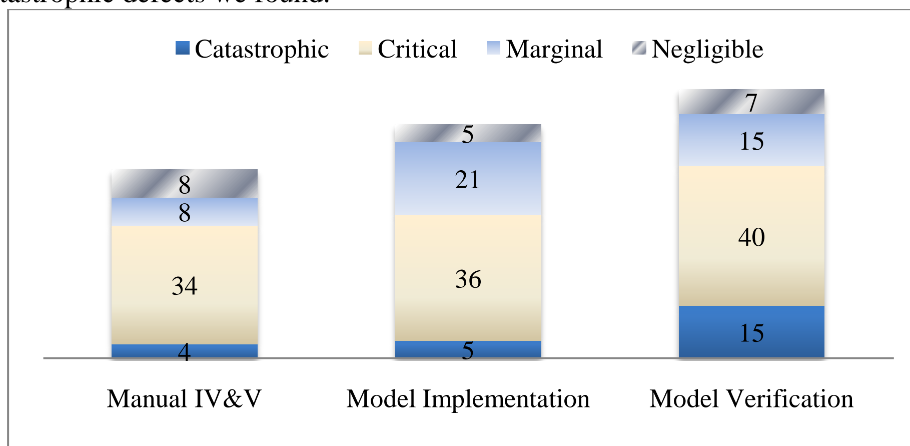



Figure 4: Categorization of Defect Origin

Some counterexamples were helpful in determining the exact reason that a property failed. These counterexamples tended to be short (demonstrated over only a few cycles) and involve few inputs. Other

counterexamples were more complex and difficult to follow. Where possible, we documented a description of the counterexamples as an explanation for a failed property.

We also found problems by verifying some safety properties. The source documents for most of the safety properties were other configured documents specific to the system, such as the Safety Hazard Analysis and Software Safety Analysis. Some safety properties were generated by analyzing the SRS.

We modeled 215 safety properties. Of these 215 properties, 68 or 32% evaluated to false and returned a counterexample. Some examples of safety properties are in the following table.

| Safety Property | Verification Operator Implemented As | Returned |
|---|---|---|
| If the datalink is broken, can both VMSs be master? | After the first cycle, if VMS_A is not communicating and VMS_B is not communicating, then VMS_A is not master or VMS_B is not master. | false |
| An invalid VMS is declared halted. | If a VMS is not valid, then it is in the halted state. | false |
| If the engine oil temperature is out of range a fault will be set. | If the oil temp is less than -50 or greater than 110, then oilTempFault will be true. | false |
| Before the engine started message is sent, the engine start request message must be received. | Is it true that the signal engineStartComAccepted has never been true and the signal engineRunning is true? | false |

Table 7: Example Safety Properties

### 4.4 Return on Investment

We spent approximately 934 hours modeling and verifying properties. Given that we modeled 925 requirements, our modeling and verification rate was about one requirement per hour. We spent 498 hours building the model, 436 hours using Design Verifier™, and 63 hours in training.

Training consisted of three days of formal instruction by representatives of Esterel Technologies. During training, we learned the basic functionalities of SCADE Suite® and practiced making operators. SCADE® is not difficult to learn, especially if one has knowledge of another graphic modeling tool like Simulink®.

Of the 498 hours spent building the model, the initial model creation took 424 hours. There were another 74 hours of editing the model after running Design Verifier™ to fix errors we had introduced to the model and to improve the model's organization.

The time spent in Design Verifier™ accounts for creating the safety properties, running the Verifier™, and editing the properties as needed.

This method is easy to learn and integrates well into the typical software lifecycle. In fact, by using well established rates of productivity based on the number of lines of code [4], we calculated that reviewers would spend approximately 40% less time in requirements review using this method over traditional IV&V methods. We also calculated a total project cost savings of 5% for the system we studied. On top of that savings, the model can be reused to automatically generate code and tests. The time required for training is minimal, though additional maturation with the tool occurs as the modelers gain experience. Considering a post-learning execution time and the number and significance of our findings, this process is not only a feasible inclusion to the software development lifecycle, but is also a valuable asset to the

lifecycle. It can lead to cost savings plus the detection of safety-critical defects that may prevent a catastrophic failure during flight test or operation.

# 5 Conclusions

## 5.1 Training Recommendations

The training provided by Esterel was adequate to begin creating the model. As with most tools, as we gained experience with SCADE®, we found certain methods and shortcuts for improved efficiency. Perhaps a follow-up training session a few weeks after the initial training would have been useful for instruction in some finer details and higher-level usage of the tool's more advanced capabilities. The graphical modeling environment of SCADE® is very similar to that of Simulink®. Engineers familiar with Simulink® should have a smooth transition to SCADE®.

The specific concepts that should be understood prior to training include knowledge of logical operators (and, or, not, implies), Boolean algebra, and state transition diagrams. In general, an engineer who has had college-level math or logic is capable of benefitting from the training.

## 5.2 Objectives Revisited

We had five objectives for this phase of the study. Our conclusions based on those objectives are as follows:

1) *Determine if another tool set is able to model and verify the entire system at one time.* We were able to model 82% of a software requirements document in one SCADE® model. Scalability was not an issue.
2) *Determine if another tool set is more user-friendly, including better commercially available training, fewer bugs and crashes, and increased ease of use.* We found SCADE® to be easy to learn, and the training provided was adequate to begin modeling. However, the first release of SCADE® that we were given was unreliable. We had to back up our work several times a day to prevent losing it because SCADE® crashed often. It was not unusual for SCADE® to crash once or even twice a day for every engineer using it. The second release of SCADE® that we were given was much more reliable and the number of times that this version crashed was few.
3) *Develop metrics for the number of requirements that can be modeled and verified per labor hour and the number of safety and other defects found per requirement.* We modeled approximately one requirement every hour. There was approximately one defect for every 4.7 requirements and one assumption per every 7.6 requirements.
4) *Prototype methods for modeling and verifying design aspects.* We were not able to complete this objective.
5) *Document training requirements and guidelines.* We were not able to complete this objective.

## 5.3 Limitations

There are two main constraints in using this method. One is the faithfulness of the model to the requirements. Is the model behaving as the software behaves? Is the model introducing or eliminating bugs? As the number and complexity of requirements increases, more assumptions are introduced to the model. This limitation applies to any model-checking method, not just SCADE's method. The other limitation is the difficulty of merging work among multiple modelers. We were not able to introduce a satisfactory method of combining the work of multiple modelers. The best we did was to divide the tasks each modeler worked on and manually combine models once a week. The difficultly lies in merging operators that more than one modeler has edited and in connecting and adding inputs and outputs.

SCADE® does not have the capability to "drop in" different versions of operators, even if their inputs and outputs are the same.

### 5.4 Efficacy of Model Checking Methods

This technique has value as a method for checking models. It can be used to determine software functions that contribute to potentially unsafe conditions. It is also a cost-effective means to ensure safety, as it can identify potential software safety defects early in the software's lifecycle, thereby reducing cost. Additionally, this technique can indicate critical software functions that need further testing. This technique also identifies requirements ambiguities. Clarifying these ambiguities helps improve safety and reduces software development and testing costs by minimizing re-work.

We found a total of 198 requirements defects during our analysis, and only 54 of those were found through traditional IV&V methods. The additional 144 defects were discovered while building the model and while running Design Verifier™. Thus traditional IV&V methods missed 73% of the total defects we found.

### 5.5 Recommendations for the Future

The main deficit we recognize is the need for a systematic way to manage and merge versions. We suspect that a free version control tool like CVS would work for managing versions. An efficient way to merge versions proves more elusive.

## References

[1] Greg Turgeon and Petra Price. "Feasibility Study for Software Safety Requirements Analysis Using Formal Methods." Presented at the 27th International System Safety Conference, Huntsville, Alabama, August 2009. Available for purchase at http://www.system-safety.org/products/.

[2] NIST Planning Report 02-3. "The Economic Impacts of Inadequate Infrastructure for Software Testing," May 2002. http://www.nist.gov/director/prog-ofc/report02-3.pdf, last viewed December 2009.

[3] Jonathan P. Bowen and Michael G. Hinchey. "The Ten Commandments of Formal Methods." *Computer*, vol. 28, no. 4, pp. 56-63, April 1995. http://www.is.pku.edu.cn/~qzy/fm/lits/ten_commandmentsFM0.pdf, last viewed December 2009.

[4] Watts S. Humphrey. *Introduction to the Team Software Process*. Addison Wesley, 2000.

Simulink® and MATLAB® are registered trademarks of The MathWorks™.
SCADE® and SCADE Suite® are registered trademarks of Esterel Technologies.
Design Verifier™ is a trademark of Esterel Technologies.

# A Prototype Embedding of Bluespec SystemVerilog in the PVS Theorem Prover

Dominic Richards and David Lester
The University of Manchester
Manchester, U.K.
richardd@cs.man.ac.uk, dlester@cs.man.ac.uk

**Abstract**

Bluespec SystemVerilog (BSV) is a Hardware Description Language based on the guarded action model of concurrency. It has an elegant semantics, which makes it well suited for formal reasoning. To date, a number of BSV designs have been verified with hand proofs, but little work has been conducted on the application of automated reasoning. We present a prototype shallow embedding of BSV in the PVS theorem prover. Our embedding is compatible with the PVS model checker, which can automatically prove an important class of theorems, and can also be used in conjunction with the powerful proof strategies of PVS to verify a broader class of properties than can be achieved with model checking alone.

## 1 Introduction

Bluespec SystemVerilog (BSV) [Nik04] is a language for high-level hardware design, and produces hardware that's competitive with hand-written Register Transfer Level designs in terms of time and area for many applications [WNRD04, Nik04]. It developed from research using Term Rewriting Systems (TRS) to produce hardware specifications that could be synthesised and formally verified [AS99]; in common with TRS, BSV is semantically elegant, which makes it well suited for formal reasoning [ADK08]. However, only one previous investigation has been made into the mechanised verification of BSV designs [SS08], which concentrated on direct model checking of BSV designs in the SPIN model checker [Hol03], without the capacity for interactive reasoning or abstraction.

In this paper we present a prototype shallow embedding of BSV in the PVS theorem prover [ORS92]. Our embedding is compatible with the PVS model checker [ORR+96], which can automatically prove an important class of theorems. Furthermore, the PVS model checker can be used in combination with the powerful proof strategies of PVS to verify a far broader class of properties than can be achieved with model checking alone.

We use a novel application of monads to capture the complex state-encapsulation mechanisms of BSV in a concise and readable way. Our embedding supports several advanced language features of BSV, including module instantiation, side-effecting methods and rule composition from side-effecting methods. In this work we present an embedding strategy: we have yet to build a BSV-to-PVS compiler, which will be developed in conjunction with our related research into automated abstraction of BSV designs.

The material in this paper is presented at length in the principal author's PhD thesis [Ric10a], and the associated code is available online [Ric10b]. Throughout the paper we present code fragments in BSV and PVS; to make the difference clear, all BSV code is presented in Courier font and surrounded by frames labeled 'BSV'. All PVS code is presented in Times New Roman, without surrounding frames.

## 2 Bluespec SystemVerilog

BSV is a language for high-level hardware design. For an in-depth introduction, see the online lecture series at [Blu]. In BSV, hardware is specified in terms of 'modules' that associate elements of state with

'rules' (guarded actions) that transform the state and 'methods' that can be called by other modules to return values from the state, and possibly transform it in the process.

A simple example of a module is Reg, which specifies a register with one element of state, no internal rules and two methods, _read and _write. Other modules can create instances of Reg, and use the methods _read and _write in their own rules and methods. For example, the following rule uses two registers, request and acknowledge, both of which hold elements of type Bool[1]:

```
rule request_rl (!(request._read() || acknowledge._read()));
  request._write(True);
endrule
```
BSV

The rule has a guard, which is a predicate on the state of the registers, and an 'action', which transforms the state of the request register. In general, a rule has the form:

```
rule my_rule (rl_guard);
  statement_1;
  statement_2;
  ...
endrule
```
BSV

The statements in the rule body are individual actions that transform the state in some way. The set of statements are applied *in parallel* to the state; each statement is applied to the state as it was immediately before the rule was activated, so that the changes made by statement_1 aren't seen by statement_2, or any other statement. The BSV compiler ensures that the statements in a rule don't conflict with each other by simultaneously attempting to write to the same elements of state.

# 3 The Challenges of Embedding BSV in PVS

BSV uses guarded actions to express concurrency, which makes it similar to the guarded action languages that were developed for the formal study of concurrency; these include UNITY [CM88], Promela [Hol03], the SAL language [dMOR+04] and the model checkable subset of the PVS language [ORR+96]. There is a rich body of literature on the use of model checkers and, to a lesser extent, theorem provers for verifying systems expressed in these languages. However, BVS is a more complex language in some respects, being intended for hardware design rather than abstract specification:

1. **Complex encapsulation of state.** As seen in §2, BSV has a 'module' construct that allows elements of state to be associated with 'rules' and 'methods'. Rules can be composed from the methods provided by other modules; in this way, the execution of a rule in one module can alter the state in another.

2. **Widespread presence of data paths.** Model checking is a useful tool for efficiently verifying finite state concurrent systems, but can be confounded by the presence of data paths (circuits that hold elements of data). Data paths can have very large state spaces, which can cause state space explosions in model checkers; for this reason, model checking has been more widely used to verify control-based systems. When abstract specification languages such as UNITY are used for hardware verification, a model can be constructed that only specifies the control-based components

[1] BSV users will notice that we've de-sugared the method calls for Regs; we do this throughout the paper to simplify the semantics, and also to emphasize the use of methods inside rules.

of a design (when the data path is irrelevant to the properties being verified), or specifies some abstract interpretation of the data path. With an embedding of BSV, however, the hardware design *is* the specification; we can't chose to omit data paths from our formal model. Because of this, we must find a tractable way to abstract away from data path complexity within our proof environment.

So, in order to produce a usable general purpose embedding of BSV in a formal guarded action language, we must first bridge the *semantic gap* by expressing the constructs of BSV with the more limited constructs of our target specification language, preferably in such a way that equivalence between the two can easily be established [CDH+00]. When we have achieved this, we must also bridge the *abstraction gap*, to obtain abstract specifications that can be efficiently verified [SS99, CDH+00].

In this paper we concentrate on the first of these two steps. We offer an embedding strategy that bridges the semantic gap between BSV and the model checkable subset of PVS with a novel use of monads [Bir98]. In further work, we hope to employ automatic abstraction [SS99] to our PVS specifications in order to bridge the abstraction gap.

We actually introduce two embedding strategies with different strengths and weaknesses, and combine them in a hybrid technique that preserves the strengths of both. In §5 we introduce an embedding strategy that we call 'primitive embedding', where we specify BSV modules with 'primitive transition relations'. Primitive transition relations can be efficiently model checked, but bear little resemblance to the BSV modules they represent, and we discuss the problems that this creates. In §6 we introduce an embedding strategy that we call 'monadic embedding', where we specify BSV modules with 'monadic transition relations'. Monadic transition relations are syntactically similar to the BSV modules they represent, which cleanly bridges the semantic gap. However, verification is demanding in terms of CPU time. We go on to introduce a hybrid technique that allows us to perform proofs over monadic transition relations with the efficiency normally associated with proofs over primitive transition relations. We demonstrate the practical applicability of our approach in §7 with a hand embedding of a BSV fair arbiter, for which we verify deadlock freedom, mutual exclusion and fairness properties.

# 4 The Semantics of a BSV Module

The behavior of a BSV module can be understood in terms of a simple semantics called Term Rewriting System (TRS) semantics, also called 'one-rule-at-a-time' semantics. According to TRS semantics, a module evolves from a given state by choosing *one* rule for which the guard is true and applying the associated action to transform the state. If more than one guard is true, a nondeterministic choice is made. When actual hardware is generated from BSV designs, a number of optimisations are applied (for example, a clock is introduced and multiple rules are executed per clock cycle) but the behavior is guaranteed to comply with the TRS semantics.

The TRS semantics gives the designer a simple, high-level way of understanding his design, and we use it in our embedding. In PVS, we describe a rule as a predicate over pairs of states:

$$\text{my_rule (pre, post: Module_State): bool = rl_guard (pre)} \wedge \text{post} = \text{rl_action (pre)}$$

Here, rl_guard and rl_action are the PVS representations of the rule's guard and action, and Module_State is the PVS representation of the module's state. We can then express the TRS semantics of a module by composing its rules together:

$$\text{TRS_transition_relation (pre, post: Module_State): bool = rule}_1\text{ (pre, post)} \vee \text{rule}_2\text{ (pre, post)} \vee \ \ldots$$

TRS_transition_relation is a binary relation on 'pre' and 'post'. It relates 'pre' to 'post' if any of its constituent rules relates them when applied in isolation: we have a nondeterministic, one-rule-at-a-time semantics. We consider the possible forms of Module_State and $\text{rule}_1$ etc. in the following sections.

# 5 A Primitive Embedding of BSV in PVS

PVS has a set of proof strategies for using model checking to verify temporal logic assertions about guarded action systems [ORR+96, SS99]. The state of a system can be defined inductively from boolean and scalar types, using tuples, records or arrays over subranges, and the transition relation is defined as a binary relation over pairs of states (as with TRS_transition_relation in §4).

Consider the following rule, which comes from the arbiter example of §7:

```
rule ack1_with_tok (token1._read() && req1._read()
                   && !(ack1._read() || ack2._read() || ack3._read()));
  ack1._write (True);
  move_token;
endrule
```
BSV

The guard of this rule is a predicate over the state of five registers and the action changes the state of register `ack1`, and also calls the 'local action' `move_token`:

```
Action move_token = (action token1._write(token3._read());
                            token2._write(token1._read());
                            token3._write(token2._read()); endaction);
```
BSV

We can specify the state of the `Reg` module as a PVS record with one field, and use this to specify the state of our arbiter system (which has nine boolean registers):

$$\text{Reg}: \text{TYPE} = \left[\#\ \text{val}: T\ \#\right]$$

$$\text{Arbiter}: \text{TYPE} = \left[\#\ \text{req1},\ \text{req2},\ \text{req3},\ \text{ack1},\ \text{ack2},\ \text{ack3},\ \text{tok1},\ \text{tok2},\ \text{tok3}:\ \text{Reg}\left[\text{bool}\right]\#\right]$$

The state of the `Reg` module could be expressed with a variable of type T, rather than a single-element record. In general, however, modules have more complex states (for example, even a simple two element 'First In, First Out' buffer would need at least 3 variables) so we factor the complexity of nested module states into records. This also helps to simplify our monadic embedding in §6.

We can now express the rule `ack1_with_tok` as a predicate over pairs of Arbiters:

$$\begin{aligned}
&\text{ack1_with_tok_concrete}\ (\text{pre},\ \text{post}: \text{Arbiter}): \text{bool} = \\
&\quad \text{pre`tok1`val} \land \text{pre`req1`val} \land \neg\ (\text{pre`ack1`val} \lor \text{pre`ack2`val} \lor \text{pre`ack3`val}) \\
&\quad \land\ \text{post} = \text{pre WITH } \big[(\text{ack1}) := (\#\ \text{val} := \text{TRUE}\ \#), \\
&\qquad\qquad (\text{tok1}) := (\#\ \text{val} := \text{pre`tok3`val}\ \#), \\
&\qquad\qquad (\text{tok2}) := (\#\ \text{val} := \text{pre`tok1`val}\ \#), \\
&\qquad\qquad (\text{tok3}) := (\#\ \text{val} := \text{pre`tok2`val}\ \#)\big]
\end{aligned}$$

This is a straightforward way to express rules, and it's compatible with the PVS model checker. The state of the module is expressed as a record; rule guards are then expressed as predicates over the fields of this record, and rule actions are expressed as record updates. When a statement calls a local action, the action can be expanded in-place to a series of record updates (as we did here with the local action `move_token`). Methods, which generally return some value and perform an action to transform the state, can be expanded in-place by expressing the value they return as a pure function on the state, and the action as a series of record updates. When methods appear in the guard, they will be side-effect free (this is guaranteed by the BSV compiler) and can be expanded to pure functions on the state. This

is our 'primitive' embedding strategy. We can represent a BSV module by specifying all of its rules in this way, and combining them using the TRS_transition_relation function of §4; we then refer to this as a 'primitive transition relation'.

This approach seems quite simple, but it has a drawback. If we express a rule by fully expanding all of the actions and method calls, we expose its full complexity; BSV provides the module and method constructs to avoid just this. If we specify a more complex module in this way (for example, one where rules and methods call methods that themselves call methods, all returning values and producing side-effects) we end up with a long-winded specification that bears little resemblance to the BSV module it represents. If we assume that the process of translating from BSV to PVS is not formally verified, it becomes difficult to provide assurance that the translation is accurate; we have a large and unverified semantic gap. This makes it difficult to rule out false positives when a property is proven, or conversely false negatives when a property is disproved. We would like to have a PVS representation that's compatible with the PVS model checker, but also relates back to the BSV code in a simple and transparent way. Our solution is to use a variation of the state monad from functional programming [Bir98].

# 6 A Monadic Embedding

Monads are a technique for representing state in pure functional languages. A monad is simply a function that takes an element of state as input and returns a value and a new state. Monads can by composed sequentially in the same way as statements in a procedural language; this is achieved with the infix operator $\gg$ (pronounced 'seq'), which is equivalent to the semi-colon in procedural languages, and also a related function $\gg=$ ('bind').

In this section we show that monads allow us to embed BSV rules at a higher level of abstraction than our primitive embedding of §5. If this is your first exposure to monads, the first half of the section should be fairly accessible (up to the definitions of $\gg=$ and $\gg$). The rest of the section may require some background reading; for example, [Bir98] has an excellent chapter on monads. Also, the principal author's PhD thesis [Ric10a] gives an introduction to monads in the context of the work presented here.

We develop a new embedding strategy for rules, where actions and method calls are expressed as monads, rather than being expanded in-place. Before getting into the details, let's take a look at the result. This is our monadic embedding of the rule used in §5:

$$
\begin{aligned}
\text{ack1_with_tok} = \text{rule}\ & (\text{tok1`read} \wedge \text{req1`read} \wedge \neg\ (\text{ack1`read} \vee \text{ack2`read} \vee \text{ack3`read}))\\
& (\text{ack1`write}\ (\text{TRUE}) \gg\\
& \ \text{move_token})
\end{aligned}
$$

At the level of syntax, this is very similar the original rule. In contrast to the primitive embedding strategy of §5, the complexity of methods and actions is factored out into monads. This yields rule specifications that are syntactically similar to the BSV rules that they represent, so that errors in the BSV-to-PVS translation process will be discernible by inspection; with a far smaller semantic gap, false positives and false negatives can be more easily ruled out. Because of the syntactic similarity, the BSV-to-PVS translator will be simpler; translation will essentially be a task of converting between two very similar concrete syntaxes. This is important because we expect that the BSV-to-PVS translator won't be formally verified, so we must keep it as simple as possible. Furthermore, we hope that the structured, compositional nature of our monadic specifications will simplify deductive reasoning when it's necessary.

ack1_with_tok is a function. It has the type $\left[\left[\text{Arbiter, Arbiter}\right] \rightarrow \text{bool}\right]$ and is, in fact, extensionally equivalent to ack1_with_tok_primitive from §5; it's just a more concise way of writing the same function. If we fully expand all of the functions in the definition of ack1_with_tok, we end up with the definition of ack1_with_tok_primitive; a simple function involving record updates and functions over

record fields (we can do this in the PVS proof environment with the (expand*) proof strategy). We can specify a BSV module by forming monadic specifications of its rules, and combining them using the TRS_transition_relation function of §4; we then refer to this as a 'monadic transition relation'.

The PVS model checker fails when it's called directly on a monadic transition relation; this is possibly because of the extensive use of higher order functions. One solution is to expand the monadic transition relation to the equivalent primitive transition relation with the (expand*) proof strategy, then call the (model-check) strategy, and finally discharge a small number of subgoals with the (grind) strategy. In our experience, this approach is fine for small examples but the expansion step is quite demanding in terms of CPU time, which might become problematic for larger examples.

We can avoid expanding the monadic transition relation *during every proof* if we compile a PVS theory containing the primitive transition relation. We can state in a lemma that the primitive transition relation is extensionally equivalent to the monadic transition relation, and prove it with a single application of the proof strategy (grind-with-ext). The PVS prover then treats the lemma as a rewrite rule, so that we can call the (model-check) proof strategy directly on the monadic transition relation, and the prover automatically rewrites it to the equivalent primitive transition relation, which can be efficiently model checked. This approach makes proofs faster in CPU time because we don't need to expand the monadic transition relation during every proof. There are additional overheads incurred because we must prove the equivalence lemma and compile the PVS theory containing the primitive transition relation, but these are only done once and can be re-used for all proofs. We evaluate the two verification approaches in §7. However, we can't yet evaluate the overhead incurred by compilation of the primitive transition relation because we have yet to build a BSV-to-PVS translator; we currently compile by hand.

### 6.1 A State Monad in PVS

So, how can we use monads to express actions and methods without expanding their full complexity in-place? Consider the body of action move_token:

```
token1._write(token3._read());
token2._write(token1._read());
token3._write(token2._read());
```
BSV

We have three statements, each composed of two method calls. The meaning we want to capture for the whole statement block is that an initial state is transformed independently by the three statements, and the changes made by each are combined to give a new state. We can actually achieve the same effect by applying the statements sequentially; we can apply the first statement to get a partially updated state, then apply the second statement to update this new state, and apply the third statement to update the result. This is possible because the statements are conflict-free; no two statements will update the same element of state, so we don't need to worry about later statements over-writing the updates made by earlier statements. However, each statement needs access to the initial state, as earlier statements might update elements of state that later statements need to read. This suggests that we specify statements as instances of the type:

$$\text{Monad: TYPE} = \big[[S,\ S] \rightarrow [A,\ S]\big]$$

'S' is the type of the module's state (in the case of move_token, it's Arbiter from §5). 'A' is the type of some return value; for the statements under consideration, it's Null. Instances of Monad take two copies of the module state (representing the initial state, and a partially updated state) and return a value and a new instance of the state, with any additional updates added to those of the partially updated state. We can compose statements to form rule bodies with the standard monad connectors (see [Bir98] for a good introduction) with $\gg=$ adapted to accept a pair of input states rather than a single input state:

$$\gg= (m\colon \text{Monad}[S,A],\ k\colon [A \to \text{Monad}[S,\ B]])\colon \text{Monad}[S,\ B] =$$
$$\lambda\,(\text{init},\ \text{updates}\colon S)\colon \text{LET (val, new_updates)} = m\,(\text{init},\ \text{updates})\ \text{IN}\ k\,(\text{val})\,(\text{init},\ \text{new_updates})$$

$$\gg (m\colon \text{Monad}[S,\ A],\ n\colon \text{Monad}[S,\ B])\colon \text{Monad}[S,\ B] = m \gg= \lambda\,(\text{val} : A)\colon\ n$$

Shortly, we will introduce monads that specify register methods `token1._read`, `token1._write` etc. and call them tok1‘read, tok1‘write etc. We can use these to compose a specification of `move_token`:

$$\begin{aligned}\text{move_token}\colon \text{Monad}[\text{Arbiter},\ \text{Null}] = {} & \text{tok1`read} \gg= \text{tok2`write} \gg \\ & \text{tok2`read} \gg= \text{tok3`write} \gg \\ & \text{tok3`read} \gg= \text{tok1`write}\end{aligned}$$

### 6.2 Monad Transformers

What form will the monads tok1‘read, tok1‘write etc. have? Given that move_tok has type Monad[Arbiter, Null], they are constrained to have the type Monad[Arbiter, T] for some T; that is to say, they must operate on instances of the Arbiter state, despite the fact that they only influence one register within that state. We can achieve this by specifying the two generic register methods (read and write) as monads that act on type Reg$[T]$, and lifting them to monads that act on type Arbiter with monad transformers:

$$\text{read} : \text{Monad}[\text{Reg}[T],\ T] = \lambda\ (\text{init},\ \text{updates}\colon\ \text{Reg}[T])\colon\ (\text{init`val},\ \text{updates})$$

$$\text{write}\,(d\colon T) : \text{Monad}[\text{Reg}[T],\ \text{Null}] = \lambda\ (\text{init},\ \text{updates}\colon\ \text{Reg}[T])\colon\ (\text{null},\ (\#\ \text{val} := d\ \#))$$

$$\text{Transformer} : \text{TYPE} = [\text{Monad}[R,\ A] \to \text{Monad}[S,\ A]]$$

$$\text{transform}\,(\text{get_R}\colon\ [S \to R],\ \text{update_R}\colon\ [[S,\ R] \to S])\colon\ \text{Transformer} =$$
$$\begin{aligned}\lambda\,(m\colon \text{Monad}[R,A])\,(\text{init, updates: } S) : {} & \text{LET (val, new_updates)} = m\,(\text{get_R (init), get_R (updates)}) \\ & \text{IN}\ \ (\text{val},\ \text{update_R}\,(\text{updates},\ \text{new_updates}))\end{aligned}$$

A function of type Transformer takes a monad over state R and lifts it to become a monad over state S. We can use the ‘transform’ function to produce a Transformer that lifts the generic register functions (read and write) to operate on our Arbiter state. For example, we can do this for the tok1 register:

$$\text{tok1T}\colon\ \text{Transformer}[\text{Reg}[\text{bool}],\ \text{Arbiter},\ T] = \text{transform(get_tok1},\ \text{update_tok1)}$$

where get_tok1 and update_tok1 access and update the tok1 field of an Arbiter record. We can then define our lifted monads tok1‘read and tok1‘write:

$$\begin{aligned}\text{tok1}\colon\ & [\#\ \text{read} :\ \text{Monad}[\text{Arbiter},\ \text{bool}],\ \text{write} :\ [\text{bool} \to \text{Monad}[\text{Arbiter},\ \text{Null}]]\ \#] \\ & = (\#\ \text{read} := \text{tok1T}\,[\text{bool}]\,(\text{read}),\quad \text{write} := \lambda\,(x : \text{bool}) : \text{tok1T}\,[\text{Null}]\,(\text{write}\,(x))\ \#)\end{aligned}$$

We can use these monads in the guard if we overload the standard Boolean and equality operators with functions over monads. For example:

$$\wedge\,(\text{m, n: Monad}[S,\ \text{bool}]) : \text{Monad}[S,\ \text{bool}]$$
$$\begin{aligned}= \lambda\,(\text{init},\ \text{updates} : \text{S}) : {} & \text{LET}\ b_1 = (\text{m (init, updates)})\text{`}1,\ \ b_2 = (\text{n (init, updates)})\text{`}1 \\ & \text{IN}\ \ (b_1 \wedge b_2,\ \text{updates})\end{aligned}$$

This allows us to construct guard predicates in a readable way, having a concrete syntax similar to guards in BSV. An example of this was seen earlier in the section, in the guard of rule ack1_with_tok.

Finally, when we have monadic specifications of a rule’s guard and body, we can form a ‘rule’ that is a predicate over pairs of states, using the function:

rule (guard: Monad [S, bool]) (action: Monad [S, Null]) (pre, post: S): bool =
  (guard (pre, pre))‘1 $\wedge$ post = (action (pre, pre))‘2

# 7 Experimental Results: Embedding a Fair Arbiter

To evaluate our embedding strategy, we produced a BSV design for the control circuitry of a 3-input fair arbiter, which we embedded in PVS and verified with the PVS model checker. Our design is a variation of the synchronous bus arbiter presented in [dM04], and amounts to just over 100 lines of BSV.

We represent the three inputs by Boolean ‘request’ registers, which are used to request access to the output. The arbiter also has a Boolean ‘acknowledge’ register for each input, which is used to inform the input client that its request has been acknowledged, and that it has exclusive access to the output until the request is removed. In order to guarantee fair access to the output, the arbiter also has a separate Boolean ‘token’ register for each input; of the three ‘token’ registers, only one is set to true at any time. When an input ‘has the token’ the arbiter gives it priority over all other incoming requests. Each time a request is granted, the token is cycled round to another input, so that all inputs are guaranteed to receive the token (and therefore access to the output) infinitely often.

We use the Computation Tree Logic of PVS [ORR+96] to specify deadlock freedom, mutual exclusion and fairness properties for our arbiter, using an initial state defined by mkArbiter. The deadlock freedom theorem declares that every state that can be reached from the initial state by repeated application of the transition relation can reach another, different state with one more application. The mutual exclusion theorem declares that no two inputs will ever be acknowledged (and hence have access to the output) at the same time. The fairness theorem declares that a request by any input will be acknowledged:

deadlock_freedom: THEOREM AG (transitions, $\lambda\, a$ : EX (transitions, $\lambda\, a_1 : a \neq a_1$) ($a$)) (mkArbiter)

mutex: THEOREM AG (transitions, mutex_pred) (mkArbiter)
  WHERE mutex_pred ($a$) =
    $\neg$ ($a$‘ack1‘val $\wedge$ $a$‘ack2‘val $\vee$ $a$‘ack2‘val $\wedge$ $a$‘ack3‘val $\vee$ $a$‘ack3‘val $\wedge$ $a$‘ack1‘val)

fairness: THEOREM AG (transitions, fairness_pred) (mkArbiter)
  WHERE fairness_pred ($a$) = ($a$‘req1‘val $\wedge$ $\neg$ $a$‘ack1‘val $\Rightarrow$ AF (transitions, $\lambda\, a_1$ : $a_1$‘ack1‘val) ($a$))
    $\wedge$ ($a$‘req2‘val $\wedge$ $\neg$ $a$‘ack2‘val $\Rightarrow$ AF (transitions, $\lambda\, a_1$ : $a_1$‘ack2‘val) ($a$))
    $\wedge$ ($a$‘req3‘val $\wedge$ $\neg$ $a$‘ack3‘val $\Rightarrow$ AF (transitions, $\lambda\, a_1$ : $a_1$‘ack3‘val) ($a$))

We proved all three theorems with the ‘expansion’ and ‘rewrite’ approaches introduced in §6. With both approaches, the proofs are relatively simple to carry out. For the ‘expansion’ approach, we call `(expand*)` followed by `(model-check)` and `(grind)`. The ‘rewrite’ approach requires a proof of equivalence between the monadic and primitive transition relations, which could be done with the strategy `(grind-with-ext)`, and thereafter the theorems of interest are verified with `(model-check)` and `(grind)`. In terms of CPU time, we found that the ‘rewrite’ approach was significantly faster:

| Theorem | Proof with Expansion | Proof with Rewrite | Speedup |
|---|---|---|---|
| deadlock_freedom | 39 secs | 0.54 secs | × 72 |
| mutex | 15 secs | 0.39 secs | × 38 |
| fairness | 79 secs | 2.0 secs | × 40 |
| equivalence of transition relations | – | 3.3 secs | – |

The figures were obtained on a MacBook Pro with an Intel Core 2 Duo 2.53 GHz processor and 2 GB

1067 MHz DDR3 memory. We could not quantify the extra CPU time required to compile the primitive transition relation in the 'rewrite' approach, as we currently compile by hand.

## 8 Related Work

Prior to our research, there was one investigation into the mechanised verification of BSV designs. In [SS08] Singh and Shulka present a translation schema from a subset of BSV into Promela, the specification language of the SPIN model checker. They use SPIN to verify that rule scheduling is a valid refinement, and also to verify LTL assertions. The subset of BSV that they consider is similar to ours, but doesn't include: instantiation of non-trivial nested modules; methods with arbitrary side-effects and return values; rule composition from methods with arbitrary side-effects and return values. They translate directly to Promela, in contrast to our approach, which uses monads to bridge the semantic gap.

In addition to the PVS embedding in this paper, we have also embedded the same subset of BSV in the SAL model-checker [RL10]. There are a number of advantages in compiling to a specialized model checker such as SAL. The PVS model checker is symbolic, but it's often useful to have access to explicit or bounded model checking; SAL provides both of these. Also, the PVS model checker doesn't provide counter-example traces when it fails to prove a property, whereas SAL does. However, we could not recreate our monadic embedding in SAL, because it requires interactive proof strategies. Instead, we produce a primitive embedding in SAL and use PVS to prove that instances of this embedding are equivalent to instances of a monadic embedding expressed in PVS. This is a good example of the benefits of interactive theorem proving, even for systems which at first inspection seem to fit well into fully automated proof tools.

In the wider literature, our work sits somewhere between theorem prover embeddings of formal guarded action languages and model checking of main-stream hardware and software languages. There are a number of theorem-prover embeddings of guarded action languages. In [Pau00], Paulson embeds UNITY in the Isabelle [Pau94] theorem prover. Rather than using model checking to discharge repetitive proofs, he uses a set-based formalism in his embedding that allows efficient use of Isabelle's proof tactics. In [CDLM08], Chaudhuri *et. al.* present a proof environment for TLA$^+$ [Lam02]; as with Paulson, they use automated deduction rather than model checking to lessen the proof burden.

Because languages like UNITY and TLA$^+$ were developed for specification rather than design, there is less emphasis on the use of abstraction for state-space reduction, which will be necessary in a general-purpose verification tool for BSV. There have been a number of applications of abstraction and model checking to verify programs expressed in main-stream hardware and software languages. Some of the larger projects include the Java model checkers Bandera [CDH$^+$00] and Java PathFinder [VHBP00], and the C model checkers FeaVer [HS00] and SLAM [BR02]. All of these tools employ some combination of static analysis, slicing, abstract interpretation and specialization to reduce the state space.

Monads have been used several times before to express state in theorem prover specification languages; notable examples include [Fil03, KM02, BKH$^+$08].

## 9 Conclusions and Further Work

We have presented a shallow embedding for a subset of Bluespec SystemVerilog in the PVS theorem prover. Our embedding uses monads to bridge the semantic gap, whilst allowing efficient use of the PVS model checker to discharge several important classes of proofs.

In further work, we plan to extend our approach with automated abstraction. PVS has a number of proof strategies for automatic predicate abstraction [SS99] that are closely integrated with the PVS model checker; this will be our first line of investigation. We hope to combine our work in PVS and SAL

by producing a translator that compiles BSV designs to both languages, giving users the benefits of both tools. We intend to road-test our combined system by verifying a BSV model of the communications network of the SpiNNaker super-computer, which we have previously verified using Haskell [RL09].

## References


[ADK08] Arvind, N. Dave, and M. Katelman. Getting formal verification into design flow. In *Proc. FM*, 2008.

[AS99] Arvind and X. Shen. Using term rewriting systems to design and verify processors. *IEEE Micro*, 19(3), 1999.

[Bir98] Richard Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall, 2 edition, 1998.

[BKH+08] L. Bulwahn, A. Krauss, F. Haftmann, L. Erkök, and J. Matthews. Imperative functional programming with Isabelle/HOL. In *Proc. TPHOLs '08*, 2008.

[Blu] Bluespec Inc. Bluespec SystemVerilog online training. `http://www.demosondemand.com/`.

[BR02] Thomas Ball and Sriram K. Rajamani. The SLAM project. In *Proc. POPL '02*, 2002.

[CDH+00] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and H. Zheng. Bandera: extracting finite-state models from Java source code. In *ICSE '00*, 2000.

[CDLM08] K. Chaudhuri, D. Doligez, L. Lamport, and S. Merz. A TLA+ Proof System. In *Proc. KEAPPA*, 2008.

[CM88] K. Mani Chandy and Jayadev Misra. *Parallel Program Design*. Addison Wesley, 1988.

[dM04] Leonardo de Moura. SAL: Tutorial. Technical report, SRI International, November 2004.

[dMOR+04] L. de Moura, S. Owre, H. Rueß, J. Rushby, N. Shankar, M. Sorea, and A. Tiwari. SAL 2. In *Proc. CAV'04*, 2004.

[Fil03] Jean-Christophe Filliâtre. Verification of non-functional programs using interpretations in type theory. *JFP*, 13(4), 2003.

[Hol03] Gerard Holzmann. *The SPIN model checker: primer and reference manual*. Addison-Wesley, 2003.

[HS00] G. J. Holzmann and M. H. Smith. Automating software feature verification. *Bell Labs Technical Journal*, 5(2), 2000.

[KM02] S. Krstic and J. Matthews. Verifying BDD algorithms through monadic interpretation. In *Proc. VMCAI '02*, 2002.

[Lam02] Leslie Lamport. *Specifying Systems*. Addison-Wesley, 2002.

[Nik04] R. Nikhil. Bluespec SystemVerilog. In *Proc. MEMOCODE '04*, 2004.

[ORR+96] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. K. Srivas. PVS: Combining specification, proof checking, and model checking. In *Proc. CAV '96*, 1996.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS. In *Proc. CADE-11*, 1992.

[Pau94] Lawrence C. Paulson. *Isabelle: a Generic Theorem Prover*. LNCS. 1994.

[Pau00] Lawrence C. Paulson. Mechanizing UNITY in Isabelle. *ACM Trans. Comput. Logic*, 2000.

[Ric10a] Dominic Richards. *Automated Reasoning for Hardware Description Languages*. PhD in Preparation, The University of Manchester, UK. 2010.

[Ric10b] Dominic Richards. Source code. `https://sourceforge.net/projects/ar4bluespec`, 2010.

[RL09] Dominic Richards and David Lester. Concurrent functions: A system for the verification of networks-on-chip. *Proc. HFL'09*, 2009.

[RL10] Dominic Richards and David Lester. A prototype embedding of Bluespec SystemVerilog in the SAL model checker. *Proc. DCC'10*, 2010.

[SS99] Hassen Saïdi and Natarajan Shankar. Abstract and model check while you prove. In *CAV '99*, 1999.

[SS08] Gaurav Singh and Sandeep K. Shukla. Verifying compiler based refinement of Bluespec specifications using the SPIN model checker. In *Proc. SPIN '08*, Berlin, Heidelberg, 2008.

[VHBP00] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Proc. ASE '00*, 2000.

[WNRD04] W-F Wong, R.S. Nikhil, D.L. Rosenband, and N. Dave. High-level synthesis. In *Proc. CAD04*, 2004.

# SimCheck: An Expressive Type System for Simulink*

Pritam Roy
University of California Santa Cruz
Santa Cruz, CA, USA
pritam.roy@gmail.com

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

**Abstract**

MATLAB Simulink is a member of a class of visual languages that are used for modeling and simulating physical and cyber-physical system. A Simulink model consists of blocks with input and output ports connected using links that carry signals. We extend the type system of Simulink with annotations and dimensions/units associated with ports and links. These types can capture invariants on signals as well as relations between signals. We define a type-checker that checks the well-formedness of Simulink blocks with respect to these type annotations. The type checker generates proof obligations that are solved by SRI's *Yices* solver for satisfiability modulo theories (SMT). This translation can be used to detect type errors, demonstrate counterexamples, generate test cases, or prove the absence of type errors. Our work is an initial step toward the symbolic analysis of MATLAB Simulink models.

## 1 Introduction

The MATLAB Simulink framework models physical and computational systems using hierarchical block diagrams. Simulink provides the framework for building and simulating systems from a set of basic building blocks and libraries. Blocks have input and output ports and the ports of different blocks are connected via links. Simulink has a basic type system for annotating ports and links, along with a limited capability for checking type correctness. There is, however, no systematic design-by-contract capability for Simulink. We have developed a SimCheck capability that allows contracts [Mey97, ORSvH95] to be specified by means of type annotations associated with the ports and links. *SimCheck* can be used to carry out type inference for basic types, annotate links with expressive types, unit types, generate test cases, capture interfaces and behaviors, monitor type conformance, and verify type correctness relative to such expressive types. The SimCheck type system can be used to specify and verify high-level requirements including safety objectives.

Unlike prior work [RdMH04, AC05, KAI+09, AKRS08, BHSO07, WCM+07] in the verification and testing of Simulink/Stateflow designs, we pursue an approach that is closer to the *correct-by-construction* paradigm. Our SimCheck tool is similar to the type-checker of the strongly-typed functional languages (e.g. OCaml, Haskell etc.). The SimCheck tool is written in Matlab language and is integrated with MATLAB Simulink. The designer can annotate the Simulink blocks with a type written in a simple annotation language. SimCheck tool can type-check the model with respect to the type and provide feedback to the designer whether the design behaves in the intended manner. If it does not, then the tool generates the inputs that are responsible for the design to fail the type-checker. The designer can simulate the generated test inputs on the design via dynamic monitoring of type constraints during simulation. The designer can repeatedly fix the design and type-check the modified design in the design-cycle until the SimCheck type-checker passes the design. Thus the integration of the SimCheck tool with the Simulink diagrams provides a powerful, interactive tool to the designer.

There is a long tradition of work on adding units and dimensions to type systems [ACL+05, Ken97, Ken96]. Many system errors occur because of incompatibilities between units. For example, in 1985,

*This research was supported by NSF Grant CSR-EHCS(CPS)-0834810 and NASA Cooperative Agreement NNX08AY53A.

a strategic defense initiative (SDI) experiment went awry because the space shuttle pointed its mirror at a point 10,023 nautical miles, instead of feet, above the earth. In 1999, the Mars Climate Orbiter entered an incorrect orbit and was destroyed because of confusion between metric and imperial units of measurement. Though Simulink deals primarily with physical quantities, we know of no prior work on developing a type system with dimensions and units for it.

Each Simulink model consists of network of blocks. Each block has some common parameters: the block label, the function, default initial values for state variables, and input and output ports. Blocks can be hierarchical so that they are composed from sub-blocks connected by links. Simulink can be used to model both continuous and synchronous systems. Each output of a Simulink block is a function of its inputs. In a continuous-time model, the output of a block varies continuously with the input, whereas in a discrete-time block, the signals are update synchronously in each time step. A block can also contain state variables so that the output can be a function, in both the discrete and continuous-time models, of the input and current state. We develop four analyzers for Simulink models:

1. An expressive type system that can capture range information on signals as well as relationships between signals. Type correctness can be proved using the Yices solver.
2. A test case generator that produces inputs that conform to the specified type constraints, or that illustrate type violations.
3. A unit analyzer that looks at the actual unit dimensions (length, mass, time, etc.) associated with a numeric type.
4. A verifier for properties using bounded model checking and $k$-induction.

While we make heavy use of Yices to solve the constraints involved in type inference, analysis, and test case generation, it is quite easy to plug in a different back-end solver. The target applications for SimCheck are in various fields including hardware/protocol verification, Integrated Vehicle Health Management (IVHM) and in cyber-physical systems (CPS). In these cases, the type system is used to capture the safety goals of the system to generate monitors for the primary software in order to generate alerts.

## 2 Types and Simulink

In Simulink, parameters, ports and signals have types. The *basic* types range over `bool`, `int8`, `int16`, `int32`, `uint8`, `uint16`, `uint32`, and `single` and `double` floating point numbers. Complex numbers are represented as a pair of numbers (integer or floating point). We ignore enumerated types which contain string values. Fixed point real numbers are supported in an extension, but we do not consider them here. The datatype for a port can be specified explicitly or it can be inherited from that of a parameter or another port. The type of an output port can be inherited from a constant, an input port, or even from the target port of its outgoing link. Simulink has structured types which are vectors or matrices of the basic Simulink types. Simulink also supports objects and classes which are ignored here. We do not address basic typechecking for Simulink since this is already handled by MATLAB. We add a stronger type-checking to the built-in type-checker. In this paper, however, we focus on more expressive type annotations for Simulink covering dimensions and units, predicate subtypes and dependent types, and bounded model checking. A solver for *Satisfiability Modulo Theories (SMT)* can determine if a given formula has a model relative to background theories that include linear arithmetic, arrays, data types, and bit vectors. *Yices*[DdM06] is an SMT Solver developed at SRI International. Our annotations are written in the constraint language of the Yices SMT solver. We use Yices as our main tool in checking contracts

and finding test cases and counterexamples. We provide a basic introduction to the Yices solver in the Appendix.

## 3 Translation of Designs to the *Yices* Language

The translation scheme is illustrated with the example of the trajectory of a projectile. Figure 1 shows the MATLAB model and it takes three inputs: the firing angle $\theta$ with respect to the horizontal, the initial velocity $v$, and the height $h_0$ above the ground. The model computes the horizontal and vertical distance of the projectile at time $t$ from the origin (0,0) via $dx$ and $dy$ respectively. The quantity $vy$ denotes the vertical velocity of the projectile.

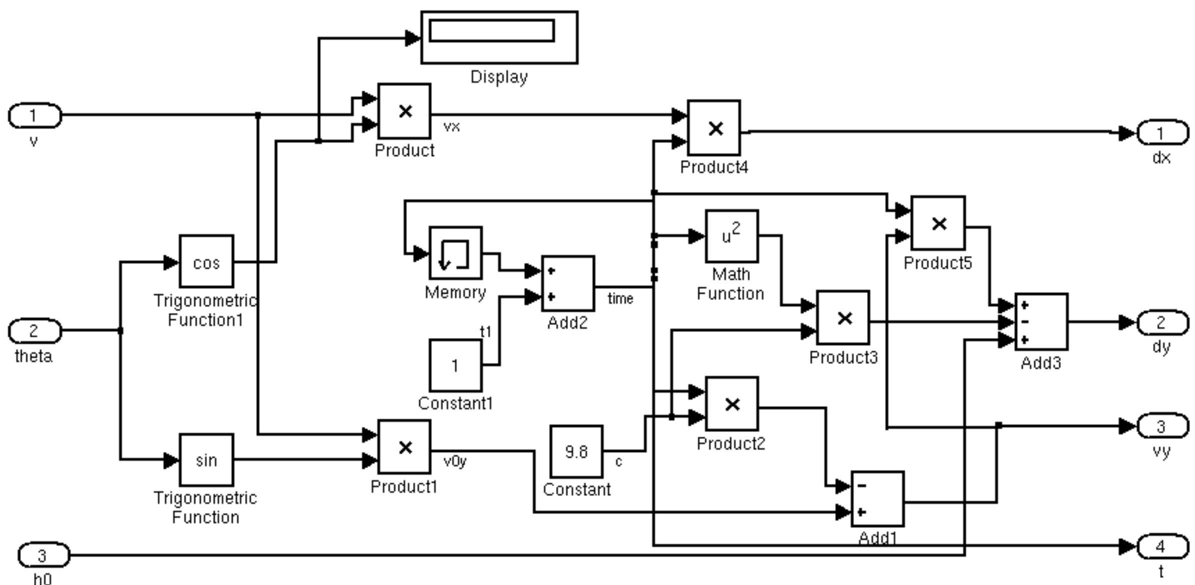



Figure 1: Projectile Subsystem

**Parsing MDL Files :** Simulink diagrams are textually represented as *mdl* files. We have used the built-in Simulink functions to create an *abstract syntax tree (AST)*. Algorithm 1 resolves the types of every signal by block and connector information. Matlab blocks generally have polymorphic types. All the blocks in Figure 1 can take different types. However *Yices* is a strongly-typed language; so the algorithm needs to resolve the types of each signal before the translation process. The algorithm traverses the Simulink blocks from the source blocks (e.g. *Constant* blocks) and obtains the type information for the ports /signals gradually. For some signals and ports the concrete type information is not available and the tool creates an equivalence type-map for these variables. For example, type of $v$ is equivalent to the type of first input of *Product* block in Figure 1. The algorithm traverses the blocks in the design in a *Breadth-First Search (BFS)* manner. The only interesting cases are when the algorithm dequeues a *subsystem* block or an outport of a subsystem block. In the first case, the algorithm appends all the inports of the subsystem block. The outport of a subsystem block appends the successor blocks of the parent subsystem.

**Block Translation:** The function *dump2yices* (Algorithm 2) translates every visited block and connector transition into the Yices language. The procedure *printBlockInfo(block,d)* dumps the type information

**Algorithm 1** MDL2YICES (model)

$visitedBlocks = \emptyset$
1. create the block transitions by parsing the connectors
2. push source blocks of the design into a queue
3. **while** queue is non-empty
4. block $x$ = dequeue()
5. parse block $x$ for description, type information and block-transitions
6. $visitedBlocks = visitedBlocks \cup \{x\}$
7. **if** block is subsystem **then** append inports of $x$
8. **else if** block is an outport **then** append the successors of its parent subsystem
9. **else** append its successor of $x$
10. **end if**
11. **end while**
12. dump2yices(model,visitedBlocks,1)

of the ports to the *Yices* file. The procedure also provides the transition-function of the block from the input ports to the output ports.

**Example 1.** *The type information and the block transitions of the* Divide *block are as follows:*

```
(define  Divide ::(-> real real real ))
(define  DivideIn1 :: real )
(define  DivideIn2 ::( subtype (n :: real ) (/= n 0)))
(define  DivideOut :: real )
(assert (=  DivideOut ( Divide  DivideIn1 DivideIn2 )))
```

The algorithm *printConnectors* translates the connector assignments. Each connector signal obtains the value and type from its source block. Similarly, the signal variables propagate the type and value to its destination block inputs. We provide translation schemes for all other basic blocks in the Appendix. The algorithm *parseDescription* parses the user-given annotations for the type-checking and verification step. We describe the annotation grammar in the next section.

**Algorithm 2** dump2yices($model, visitedBlocks, depth$)

1. $descr$ = ' '
2. **for** d $\in \{1,2,\ldots,depth+1\}$
3. **for** block $\in visitedBlocks$
4. $descr = descr + parseDescription(block, d)$
5. printBlockInfo(block,d)
6. **end for**
7. printConnectors(d)
8. **end for**

# 4 An Expressive Type System

In the *SimCheck* tool, the user can provide the constraints and properties for type-checking as well as verification via annotations. These annotations can be written as block descriptions or simulink annotation blocks. The simple grammar of the annotation language has the following syntax:

```
blockannotation = def | constraints
def = scopedef | typedef | unitdef
scopedef = ( input | output | local) <varname>
typedef = type <varname> :: <vartype>
unitdef = unit <varname> :: <varunitvalue>
constraints= ( prop | check | iinv | oinv | bmcprop | kind ) <expressions>
```

where, the terminal tokens are shown in bold. The start symbol of the grammar *blockannotation* consists of two parts : definitions and properties. The definitions can provide the details about the scope, datatype or units of the given signal. The tokens $<$ *varname* $>$, $<$ *vartype* $>$, $<$ *varunitvalue* $>$ denote the scope, type, units of the signals respectively. The constraints are in either input-output invariant (tokens *iinv* and *oinv* respectively) style or the assume-guarantee (tokens *check* and *prop* respectively) style. We use the tokens *bmcprop* and *kind* for bounded-model-checking and k-induction purposes. The token $<$ *expressions* $>$ denotes a prefix notation of the expression over the signal variables (syntactically the same as Yices expressions). Figure 2(a) shows an example of user-given block description.

**Compatibility Checking** The user can provide various constraints over different Simulink blocks. The tool checks the compatibility of the design with user-provided constraints. Let $C_{model}$ and $C_{user}$ denote the set of assertions from the model and user-provided constraints respectively. If the user-given constraints are not satisfiable with respect to the Simulink design, then the tool declares that the design blocks are not compatible with the constraints. In other words, the Yices solver returns *unsat* result for the query $\bigcap_{cs \in (C_{model} \cup C_{user})} cs$. There is no environment (i.e test input) that can satisfy the given user-constraints and the design.

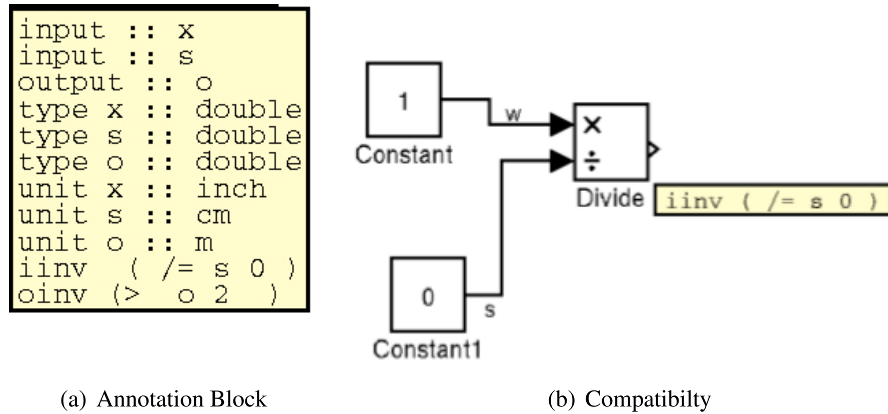
**Example 2.** *For example, Figure 2(b) illustrates that the divider block only accepts non-zero inputs. If the design connects constant zero to the second input of the divider block the design fails the compatibility checking. The translation algorithm adds the following lines to the codes of Example 1:*

```
(assert (= DivideIn2 0))
(check)
```

$\rightarrow$ *unsat*

**Dependent Types and Refinement Types:** The expressive type-systems of the SimCheck tool are similar to the PVS type-system and our tool supports predicate subtypes and dependent subtypes. Predicate subtyping can be used to capture constraints on the inputs to a function, such as, the divisor for a division operation must be non-zero. They can also be used to capture properties on the outputs, so that, for example, the output of the absolute value function is non-negative. Finally, dependent typing can be used to relate the type of an argument or a result to the value of an input. For example, we can constrain the contents array of a stack data type to have a size equal to the size slot. We can also ensure that the length of the result of appending two lists is the sum of the lengths of these two lists.

**Example 3.** *Figure 3 shows the expressive power of the type-systems via thermostat example. The output signal* on *can only take two integer values* 0 *and* 1 *(*predicate subtype*). The input signals* ref *and* houseTemp *are dependent in such a way that the value of* houseTemp *should be always higher or equal to 5 degrees below of* ref *(*dependent types*).*




(a) Annotation Block (b) Compatibilty

Figure 2: Expressive Types

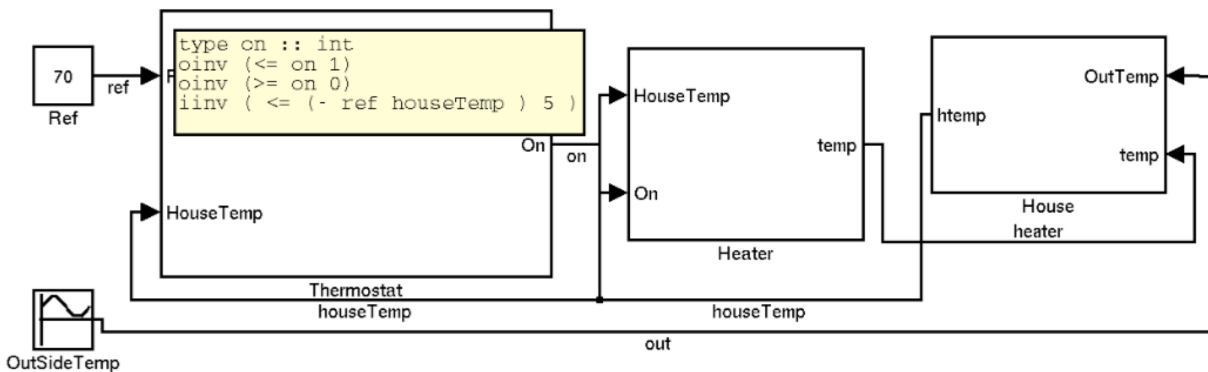



Figure 3: Thermostat Example with Dependent and RefinementTypes

With this kind of expressive typing, we can generate test cases that conform to input constraints. The test criterion can itself be expressed as a type constraint. When a type constraint does not hold, we can construct test cases that illustrate the violation. If no such violations are found, this validates type correctness relative to the expressive type annotations. Of course, this assumes that we have a sound and complete decision procedure for the proof obligations arising from such type constraints, but this is not always the case. Many Simulink models are finite-state systems and expressive type checking can be decidable, even if it is not practical. However, for the purpose of expressive type checking, we assume that the signal types are taken from mathematical representations such as integers and real numbers.

**Test Case Generation** A test-case generator produces inputs that conform to the specified type constraints, or illustrate type violations. Let $S$ and $C$ denote the set of signals and user-constraints in a design respectively. For every signal $s \in S$ and for every constraint $cs \in C$ on signal $s$, the tool asks for the satisfiability of the $(\neg cs) \cap (C \setminus cs)$. The Yices solver returns either a negative answer (*unsat*) or a positive answer with a satisfying variable assignment. In the former case, we learn that there is no input that can satisfy $(\neg cs) \cap (C \setminus cs)$. In the latter case, the variable assignment can provide a test-case that fails exactly one of the user-given constraints and satisfies the other constraints.

**Example 4.** *Figure 4 shows a design with various user-given input assumptions* $C = \{(/= \ s\ 0), (<= \ x\ 8), (>= \ z\ 5)\}$ *of a design. Figure 2(a) shows the constraints of the* Add *block in the Figure 4. We can verify that each of the three test-cases represents one case where all except one constraint is satisfied.*

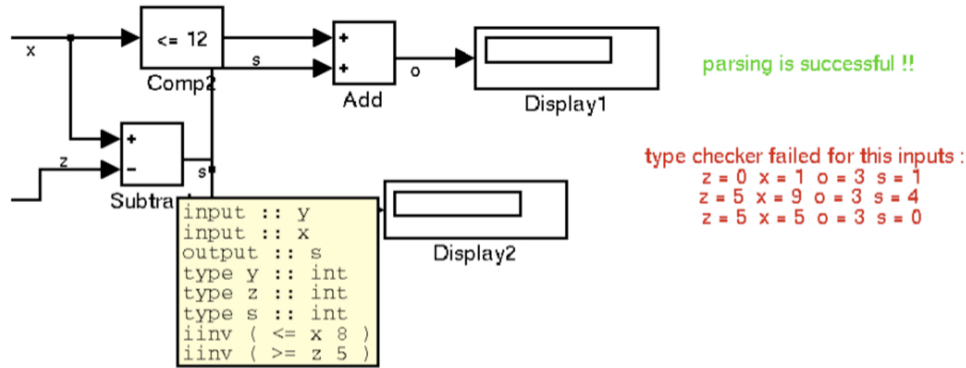



Figure 4: Test Case Generation

## 5 Property Verification via BMC and k-induction

Most practical designs contain memory elements and state variables; thus the effect of inputs spans multiple clock-cycles. Hence, for a design with state-variables, the errors can only be detected with a sequence of test inputs. The type checking for the sequential Simulink models can be extended to the verification of the systems by bounded model-checking and k-induction algorithms. Algorithm 3 and Algorithm 4 provide the pseudo-code for the bounded model checking and k-induction respectively for a property $\phi$ and a user-given depth $k$.

**Algorithm 3** BMC($k, vBlocks, \phi, model$)

1. $transition(1,k+1) = dump2yices(model, vBlocks, k)$
2. $\psi = \vee_{d \in \{1,2,\ldots,k+1\}} \neg \phi_d$
3. check(transition,$\psi$)

**Example 5.** *Figure 5 shows a design of a modulo-3 counter using 2 memory bits and an invariant property that both bits cannot become* 1 *together. We have*

$$transition(i,i+1) = (m_0(i+1) = \neg(m_0(i) \vee m_1(i))) \wedge (m_1(i+1) = (m_0(i) \wedge \neg(m_1(i))))$$

*and* $\phi_i = \neg(m_0(i) \wedge m_1(i))$*. If the design is correct then the counter will never reach* 3 *and the given property will hold for any k. For any user-given depth k, the bounded model checking (BMC) tool asks the following query to the Yices solver :* $\wedge_{i \in \{1,2,\ldots,k\}} transition \wedge (\vee_{i \in \{1,2,\ldots,k\}} \neg \phi_i)$*. For a correct design, the negated invariant property will never get satisfied and the BMC will return the* unsat *result .*

**Algorithm 4** K-Induction($k, vBlocks, \phi, model$)

1. $transition(1,k+1) = dump2yices(model, vBlocks, k)$
2. $\psi = (\wedge_{d \in \{1,2,\ldots,k\}} \phi_d) \wedge \neg \phi_{k+1}$
3. check (transition,$\psi$)

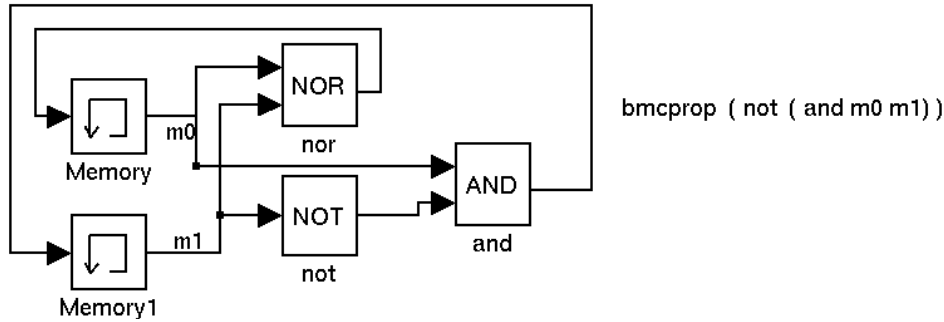



Figure 5: Bounded Model Checking

## 6 Specifying and Checking Dimensions

Most types in Simulink are numeric. Dimensions add interpretation to such types. A signal may have a numeric type without indicating whether the numeric quantity represents force or volume. Dimension checking can also highlight errors. For instance, the time derivative of distance should yield velocity, and the time derivative of velocity should yield acceleration. Multiplying a mass quantity with an acceleration quantity should yield a force. Each of these, distance, velocity, acceleration, mass, and force can also be represented in different units. Confusion about units has led to some software-related failures. In September 1999, NASA's Mars Climate Orbiter was placed into a 57km orbit around Mars instead of a 140-150km orbit because some calculations used pound force instead of Newtons. We use Novak's classification [Nov95] of units into a 7-tuple consisting of *distance*, *mass*, *time*, *temperature*, *current*, *substance*, and *luminosity*. We have other dimensions like *angle* and *currency*, but these are distinct from the basic ones considered above. The dimension of a quantity is represented as a 7-tuple of integers. For example, $\langle 1,0,-1,0,0,0,0\rangle$ represents velocity since it is of the form $\frac{distance}{time}$. A dimension of the form $\langle 0,0,0,0,0,0,0\rangle$ represents a dimensionless scalar quantity. Figure 6 shows the projectile design and the output of the dimension checker tool. The projectile subsystem block already shown in the Figure 1 We infer dimensions for Simulink signals from the annotations for the input signals and the outputs of constant blocks. When a summation operation is applied to a set of input signals, the dimensions of these signals must match, and the resulting summation has the same dimension. When two or more inputs are multiplied, the dimension of the corresponding output is given by the pointwise summation of the dimensions.

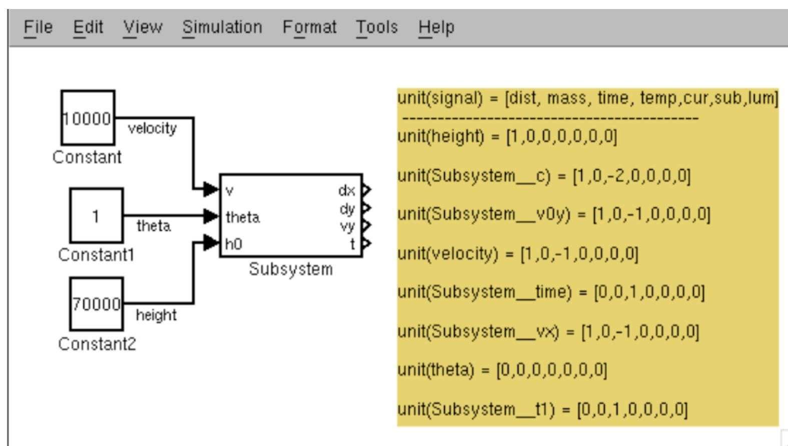



Figure 6: Unit and Dimensions

# 7 Related Work

Our work covers the verification and validation of MATLAB Simulink models through type annotations. We use the Yices SMT solver to both verify type correctness as well as to generate test cases. We focus in particular on expressive types that go beyond signal datatypes to capture dynamic ranges and relationships between signals. We also employ types to describe and check dimension and unit information. Finally, we employ bounded model checking and $k$-induction to verify invariant properties and generate counterexamples.

There is a substantial body of work [RdMH04, AC05, KAI$^+$09, AKRS08, CK03, Sys, TSCC05, WCM$^+$07] in all of the above areas. The Reactis tool [Sys] generates and simulates test cases for Simulink/Stateflow models including embedded C code. The Mathworks Simulink Design Verifier also performs test generation. The Simulink Design Verifier is also capable of generating test cases and for proving and refuting functional properties of Simulink blocks. The CheckMate tool [CK03] analyzes properties of hybrid systems represented by Simulink/Stateflow models using a finite state abstraction of the dynamics. The Honeywell Integrated Lifecycle Tools and Environment (HiLiTE) [BHSO07] from Honeywell Research uses static analysis to constrain the ranges of inputs in order to generate test cases, detect ambiguities and divide-by-zero errors, and unreachable code. The HiLiTE tool bases its analysis on numeric techniques on a floating-point representation, whereas our approach is based on symbolic constraint solving over the real numbers. The Simulink-to-Lustre translator defined by Tripakis, *et al* [TSCC05], performs type inference on Simulink models and the resulting Lustre output can be analyzed using model checkers. The Gryphon tool suite [WCM$^+$07] integrates a number of tools, including model checkers, for analyzing properties of Simulink/Stateflow models.

There is a long tradition of work on adding units and dimensions to type systems. These are surveyed in Kennedy's dissertation [Ken96] where he presents a polymorphic dimension inference procedure. Kennedy [Ken97] also proves that these programs with polymorphic dimension types are parametric so that the program behavior is independent of the specific dimension, e.g., whether it is weight or length, or its unit, e.g., inches or centimetres. The resulting type system, in which the numeric types are tagged by dimension, has been incorporated into the language F#. The recently designed Fortress programming language for parallel numeric computing also features dimension types [ACL$^+$05]. We use an approach to dimensions that is based on a fixed set of seven basic dimensions. Operations are parametric with respect to this set of dimensions. The type is itself orthogonal to the dimension, so that it could be an `int32`, `float`, or `double`, and still have a dimension corresponding to the weight. We use an SMT solver to perform type inference with dimensions, and use scaling to bridge mismatched units.

# 8 Future Directions

This work is our first step toward type-based verification of physical and cyber-physical models described in Simulink. Our work can be extended in a number of directions. Many models also contain Simulink blocks to represent discrete control in the form of hierarchical state machines. For static type checking and bounded model checking, we can extract the relationships between inputs and outputs of the Stateflow descriptions. We can also associate interface types that capture the temporal input-output behavior of Stateflow state machines. Our current translation to Yices maps the Simulink bounded integer types to the unbounded integer types of Yices, and the floating point types of Simulink are mapped to the real numbers in Yices. This makes sense for checking the idealized behavior of the models. We can also map these types to their bounded representations or even directly to bit vectors. The latter translations are more useful when checking the execution behavior of the models or in validating the code generated from the models. The SimCheck type system and Yices translation can also be extended to capture

more extensive checking of Simulink models. We would like to extend our checks to cover Simulink S-functions which are primitive blocks defined in the M language. We also plan to cover other properties such as the robustness of the model with respect to input changes, error bounds for floating point computations, and the verification of model/code correspondence through the use of test cases [BHSO07]. The type system can also be extended to handle interface types [dAH01, TLHL09] that specify behavioral constraints on the input that ensure the absence of type errors within a block. Finally, we plan to translate Simulink models to SAL and HybridSAL representations for the purpose of symbolic analysis.

## 9 Conclusions

We have outlined an approach to the partial verification of Simulink models through the use of an expressive type system. This type system can capture constraints on signals, the dimensions and units of signals, and the relationships between signals. Our annotations are already expressed in the constraint language of Yices. The resulting proof obligations are translated to the Yices constraint solver which is then used to check compatibility with respect to dimensions, generate counterexamples and test cases, and to prove type correctness. We also use Yices to perform bounded model checking and $k$-induction to refute or verify type invariants. SimCheck represents a preliminary step in exploiting the power of modern SAT and SMT solvers to analyze the Simulink models of physical and cyber-physical systems. Our eventual goal is to use this capability to certify the correctness of such systems.

## References

[AC05] M. M. Adams and Philip B. Clayton. ClawZ: Cost-effective formal verification for control systems. In Kung-Kiu Lau and Richard Banach, editors, *Formal Methods and Software Engineering, 7th International Conference on Formal Engineering Methods, ICFEM 2005, Manchester, UK, November 1-4, 2005, Proceedings*, volume 3785 of *Lecture Notes in Computer Science*, pages 465–479. Springer, 2005.

[ACL+05] Eric Allen, David Chase, Victor Luchango, Jan-Willem Maessen, Sukyoung Ryu, Jr. Guy L. Steele, and Sam Tobin-Hochstadt. The Fortress language specification, version 0.618. Technical report, Sun Microsystems, Inc., 2005.

[AKRS08] Rajeev Alur, Aditya Kanade, S. Ramesh, and K. C. Shashidhar. Symbolic analysis for improving simulation coverage of Simulink/Stateflow models. In *Proceedings of the 8th ACM & EMSOFT 2008, Atlanta, USA, October 19-24*, pages 89–98. ACM, 2008.

[BHSO07] D. Bhatt, S. Hickman, K. Schloegel, and D. Oglesby. An approach and tool for test generation from model-based functional requirements. In *Proc. 1st International Workshop on Aerospace Software Engineering*, 2007.

[CK03] A. Chutinan and B. Krogh. Computational techniques for hybrid system verification. *IEEE Transactions on Automatic Control*, 48(1):64–75, Jan. 2003.

[dAH01] Luca de Alfaro and Thomas A. Henzinger. Interface automata. *SIGSOFT Softw. Eng. Notes*, 26(5):109–120, 2001.

[DdM06] Bruno Dutertre and Leonardo de Moura. A fast linear-arithmetic solver for DPLL(T). In *18th CAV 2006, Seattle, WA, USA, August 17-20*, volume 4144 of *Lecture Notes in Computer Science*, pages 81–94. Springer, 2006.

[KAI+09] Aditya Kanade, Rajeev Alur, Franjo Ivancic, S. Ramesh, Sriram Sankaranarayanan, and K. C. Shashidhar. Generating and analyzing symbolic traces of Simulink/Stateflow models. In *21st CAV, Grenoble, France, June 26 - July 2, 2009*, volume 5643 of *LNCS*, pages 430–445. Springer, 2009.

[Ken96] A. J. Kennedy. *Programming Languages and Dimensions*. PhD thesis, University of Cambridge, 1996. Published as University of Cambridge Computer Laboratory Technical Report No. 391.

[Ken97] Andrew J. Kennedy. Relational parametricity and units of measure. In *The 24th ACM POPL '97)*, pages 442–455, January 1997.

[Mey97] Bertrand Meyer. Design by contract: Making object-oriented programs that work. In *TOOLS (25)*, page 360, 1997.

[Nov95] Gordon S. Novak. Conversion of units of measurement. *IEEE TSE*, 21(8):651–661, 1995.

[ORSvH95] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[RdMH04] John Rushby, Leonardo de Moura, and Grégoire Hamon. Generating efficient test sets for Matlab/Stateflow designs. Task 33b report for Cooperative Agreement NCC-1-377 with Honeywell Tucson and NASA Langley Research Center, Computer Science Laboratory, SRI International, Menlo Park, CA, May 2004.

[Sys] Reactive Systems. Model based testing and validation with Reactis, Reactive Systems inc., http://www.reactive-systems.com.

[TLHL09] Stavros Tripakis, Ben Lickly, Tom Henzinger, and Edward A. Lee. On relational interfaces. In *Embedded Software (EMSOFT'09)*, October 2009.

[TSCC05] Stavros Tripakis, Christos Sofronis, Paul Caspi, and Adrian Curic. Translating discrete-time simulink to lustre. *ACM Trans. Embed. Comput. Syst.*, 4(4):779–818, 2005.

[WCM+07] Michael W. Whalen, Darren D. Cofer, Steven P. Miller, Bruce H. Krogh, and Walter Storm. Integration of formal analysis into a model-based software development process. In Stefan Leue and Pedro Merino, editors, *FMICS*, volume 4916 of *Lecture Notes in Computer Science*, pages 68–84. Springer, 2007.

# A Appendix

## A.1 Satisfiability Modulo Theories and Yices

The formula language for Yices is similar to that of PVS and is quite expressive. It includes a higher-order logic with predicate subtypes and dependent types. Interaction with Yices occurs through some basic commands

1. `(define` $\langle identifier \rangle :: \langle type \rangle \langle expression \rangle$`)` : Defines a constant.
2. `(assert` $\langle formula \rangle$`)` : Asserts a formula to the context.
3. `(check)` : Checks if the context is satisfiable.

There are other commands for resetting the context, pushing and popping contexts, setting various flags, and invoking weighted satisfiability. As a simple example, Yices responds with `unsat` to the input

```
(define x::real)
(assert (< (+ x 1) x))
```

Yices also returns *unsat* on the following integer constraints.

```
(define i::int)
(assert (> i 0))
(assert (<= (* 2 i) 1))
```

On the other hand, if we relax the second constraint to a non-strict inequality, we get `sat` with an assignment of 0 for `i`.

```
(define i::nat)
(assert (<= (* 2 i) 1))
(check)
```

Yices can of course handle large problems with thousands of variables and constraints involving Booleans, arrays, bit vectors, and uninterpreted function symbols. We can run *Yices* in the batch mode : $./yices\ ex1.ys$. Yices also provides a basic typechecker.

## A.2 Block Translation Details

**Memory Blocks** One memory element with initial value 70 can be translated in two consecutive clock cycles as follows:

```
(define Memory__Out1__time1 :: real)
(define Memory__In1__time1 :: real)
(assert (= 70 Memory__Out1__time1))
(define Memory__Out1__time2 :: real)
(define Memory__In1__time2 :: real)
(assert (= Memory__Out1__time2 Memory__In1__time1))
```

**Basic Blocks** We exploit the YICES operators directly to translate basic arithmetic and logical operator blocks. For example, *Product* block in Figure 1 is translated as

```
(define  Product__In1__time1 :: real )
(define  Product__In2__time1  :: real )
(define  Product__Out1__time1  :: real )
(assert (=  Product__Out1__time1  (*  Product__In1__time1
                  Product__In2__time1 )))
```

**Constants** Simulink constant blocks can have either a scalar value, a vector or a matrix. In the translation process we assume the matrix of type $t$ as a function of type $int \rightarrow int \rightarrow t$ The translation process can catch the following problems via type checking rows and columns- (1) row-column mismatches between blocks, (2) calling out of indices (for vectors/arrays as well as matrices). For example, *Constant1* block in Figure 1 is translated as

```
(define  Constant1__Out1__time1  :: real )
(assert (=  Constant1__Out1__time1  2))
```

**Signal Connectors** For example, the signal $c$ in Figure 1 adds the following translations

```
(define  c__time1  :: real )
(assert (=  c__time1  Constant__Out1__time1 ))
(define  Product3__In2__time1  :: real )
(assert (=   Product3__In2__time1   c__time1 ))
(define  Product2__In2__time1  :: real )
(assert (=  Product2__In2__time1  c__time1 ))
```

# Coverage Metrics for Requirements-Based Testing: Evaluation of Effectiveness*

Matt Staats, Michael W. Whalen, and Mats P.E. Heimdahl
University of Minnesota
[staats,whalen,heimdahl]@cs.umn.edu

Ajitha Rajan
Laboratoire d'Informatique de Grenoble
arajan@cs.umn.edu

**Abstract**

In black-box testing, the tester creates a set of tests to exercise a system under test without regard to the internal structure of the system. Generally, no objective metric is used to measure the adequacy of black-box tests. In recent work, we have proposed three *requirements coverage metrics*, allowing testers to objectively measure the adequacy of a black-box test suite with respect to a set of requirements formalized as Linear Temporal Logic (LTL) properties. In this report, we evaluate the effectiveness of these coverage metrics with respect to fault finding. Specifically, we conduct an empirical study to investigate two questions: (1) *do test suites satisfying a requirements coverage metric provide better fault finding than randomly generated test suites of approximately the same size?*, and (2) *do test suites satisfying a more rigorous requirements coverage metric provide better fault finding than test suites satisfying a less rigorous requirements coverage metric?*

Our results indicate (1) only one coverage metric proposed—Unique First Cause (UFC) coverage—is sufficiently rigorous to ensure test suites satisfying the metric outperform randomly generated test suites of similar size and (2) that test suites satisfying more rigorous coverage metrics provide better fault finding than test suites satisfying less rigorous coverage metrics.

## 1 Introduction

When validating a system under test (SUT), the creation of a black box test suite—a set of tests that exercise the behavior of the model without regard to the internal structure of the model under test—is often desirable. Generally, no objective standard is used for determining the adequacy of test suite with respect to a set of requirements. Instead, the adequacy of such suites is inferred by examining different coverage metrics on the executable artifact defining the SUT, such as source code. However, given formalized software requirements it is possible to define meaningful coverage metrics *directly on the structure of the requirements*. Of interest here is previous work in which we adopted structural code coverage metrics to define three increasingly rigorous requirements coverage metrics over Linear Temporal Logic (LTL) requirements: *requirements coverage, antecedent coverage*, and *Unique First Cause (UFC) coverage* [19]. The relationship between the criteria forms a *linear subsumption hierarchy*, as test suites satisfying more rigorous coverage metrics are guaranteed to satisfy less rigorous coverage metrics.

In this work we empirically evaluate the fault finding ability of test suites satisfying these requirements coverage metrics. Specifically, we investigate if (1) in general a test suite satisfying a requirements coverage metric provides greater fault finding than a randomly generated test suite of the same size and (2) the linear subsumption hierarchy between metrics reflects the relative fault finding of test suites satisfying these requirements coverage metrics. Our study is conducted using four commercial examples drawn from the civil avionics domain. For each case example, we generate 600 mutants, a test suite

*This work has been partially supported by NASA Ames Research Center Cooperative Agreement NNA06CB21A, NASA IV&V Facility Contract NNG-05CB16C, and the L-3 Titan Group.

satisfying each requirements coverage metric, and 1,000 random tests. We resample these sets to create 25 sets of 200 mutants and 25 reduced test suites for each coverage metric. For each reduced test suite, we create a random test suite of approximately the same size by resampling from the set of 1,000 random tests. We explore fault finding using each set of mutants and each test suite. In total, we examine the fault finding of 15,000 combinations of case examples, test suites, and mutant sets.

From our results we draw the following observations. First, the relative rigor of the requirements coverage metrics reflects the fault finding ability of test suites satisfying them. Second, the structure of requirements significantly impacts the fault finding of test suites satisfying the coverage metrics, an observation previously made in relation to other coverage metrics [16]. Finally, test suites satisfying the UFC coverage criterion often provide greater fault finding than randomly generated test suites of equal size, while test suites satisfying the other coverage metrics do not. This indicates that of the three coverage metrics evaluated, only UFC coverage is a consistent measure of the adequacy of test suites—the intuition behind the other metrics may be useful, but they are not rigorous enough to be used as an objective measure of test suite adequacy. Our results highlight how this lack of rigor creates the potential for test suites to "cheat" a coverage criterion by *technically* satisfying the criterion, but doing so in an uninteresting or non-representative manner. This is particularly a problem when using automatic test generation tools, as these tools generally have no concept of realistic tests and tend to generate the simplest possible tests satisfying a coverage criterion.

## 2 Related Work

In [19], we formally defined the requirements coverage metrics studied in this paper and outlined our rationale for each. As detailed in [19], these metrics are related to work by Beer et al., Kupferman et al., and Chockler et al. on *vacuity checking* and *coverage metrics* for temporal logics [2, 4, 10]. More recently, we examined the fault finding effectiveness of test suites satisfying UFC coverage of the requirements on a system and MC/DC (Modified Condition/Decision Coverage) over the system itself [16]. We determined that test suites satisfying MC/DC generally outperform test suites satisfying UFC coverage (though fault finding was often close). This study, however, did not investigate the fault finding effectiveness of requirements and antecedent coverage as compared to UFC.

Fraser and Gargantini [5] and Fraser and Wotawa [6] explored automatic test generation using NuSMV for a number of coverage criteria, including UFC and mutation coverage. Neither of these works compare the effectiveness of test suites satisfying UFC coverage to randomly generated tests suites of equal size or perform any analysis that considering the size of the test suite.

## 3 Requirements Coverage Metrics

Previously, we defined three requirements coverage metrics: *requirements coverage*, *antecedent coverage*, and *Unique First Cause (UFC)* coverage [19]. Each requirements coverage metric is defined over formalized requirements expressed as Linear Temporal Logic (LTL) properties [15]. These coverage metrics provide different levels of rigor for measuring test adequacy and form a linear subsumption hierarchy, with requirements coverage as the weakest and UFC coverage as the strongest.

**Requirements Coverage:** To satisfy *requirements coverage*, a test suite must contain at least one test per requirement that—when executed—causes the requirement to be met. Consider the following natural language requirement: *"The onside Flight Director cues shall be displayed when the Auto-Pilot is engaged."* A test derived from this requirement might examine the following scenario: (1) Engage the Auto-Pilot, and (2) Verify that the Onside Flight Director comes on. Alternatively, the test might simply leave the Auto-Pilot turned off. Technically, this test meets the requirement, albeit in a way that is not particularly illuminating.

**Antecedent Coverage:** Requirements, such as the previous example, are often implications, formally $G(A \rightarrow B)$. $A$ on the left hand side of $\rightarrow$ is referred to as the *antecedent*, and $B$ on the right hand side as the *consequent*. To satisfy antecedent coverage, a test suite must contain at least one test case per requirement that, when executed, causes the requirement to be met and causes the antecedent to evaluate to *true*.

**Unique First Cause (UFC) Coverage:** Requirements are often more complex than the simple example given previously. For such complex requirements (and often simple ones), it is desirable to have a rigorous coverage metric that requires tests to demonstrate the effect of each atomic condition in the requirement; this ensures that every atomic condition is necessary and can affect the outcome of the property. Requirements and antecedent coverage cannot do this. Therefore, we defined a coverage metric called *Unique First Cause (UFC)* coverage over LTL requirements. It is adapted from the MC/DC criterion [3, 9], a structural coverage metric designed to demonstrate the independent effect of basic Boolean conditions (i.e., subexpressions with no logical operators) on each Boolean decision (expression) within source code.

A test suite satisfies UFC coverage over a set of LTL requirements if executing the test cases in the test suite will guarantee that (1) every basic condition in each formula has taken on all possible outcomes at least once and (2) each basic condition in each formula has been shown to independently affect the formula's outcome.

# 4 Study

Based on the subsumption hierarchy between these metrics outlined in the previous section we expect that test suites satisfying UFC coverage will provide better fault finding than test suites satisfying the other requirements coverage metrics, and we expect that test suites satisfying antecedent coverage will provide better fault finding than test suites satisfying requirements coverage. Furthermore, we expect that a test suite satisfying a coverage metric will provide better fault finding than a randomly generated test suite of equal size since a test suite satisfying a requirements coverage metric is designed to systematically exercise the system.

We conducted an experiment to evaluate the following hypotheses:

> **Hypothesis 1 ($H_1$):** *A test suite satisfying a requirements coverage metric provides greater fault finding than a randomly generated test suite of approximately equal size.*
> **Hypothesis 2 ($H_2$):** *A test suite satisfying a requirements coverage metric provides greater fault finding than a test suite satisfying a less rigorous requirements coverage metric.*

We used four industrial systems in our experiment. For each case example, we performed the following steps:

1. **Generated a test suite for each coverage metric:** We generated three suites providing UFC, antecedent, and requirements coverage. (Section 4.2.)
2. **Generated random tests:** We generated random inputs for 1,000 random tests. (Section 4.3.)
3. **Generated mutants:** We generated 600 single-fault mutants (Section 4.4.)
4. **Ran test suites with mutants and case example:** We ran each mutant and the original case example using the three generated test suites and the random test suite.
5. **Generated reduced requirements coverage test suites:** Each requirements coverage test suite was naïvely generated and thus highly redundant. We generated 25 reduced test suites that maintain the original coverage from each full test suite. (Section 4.2.)
6. **Generated random test suites:** For each reduced requirements coverage test suite, we randomly generated a random test suite approximately the same size as the reduced requirements coverage test suite by sampling the 1,000 random tests previously generated. (Section 4.3.)

7. **Generated mutants sets:** We randomly generated 25 sets of 200 mutants by resampling from the 600 mutants. (Section 4.4.)
8. **Assessed fault finding ability of each reduced test suite:** For each test suite and each mutant set, we determined how many mutants were detected by the test suite.

In the remainder of this section, we describe in detail our experimental approach.

## 4.1 Case Examples

In our experiment, we use four industrial avionics applications from displays and flight guidance systems. All four systems were modeled using the Simulink notation from Mathworks Inc. [12] and were translated to the Lustre synchronous programming language [8] to take advantage of existing automation [13]. For more information on these systems, see [16].

## 4.2 Requirement Test Suite Generation and Reduction

We have used an automated approach to generate test cases from a model of the system behavior. This model represents the knowledge of the desired system behavior a domain expert might possess. Using this technique, we can generate the large number of tests required for our experiments. Furthermore, the approach tends to generate the shortest, simplest tests that satisfy obligations, and, thus, the tests generated are unlikely to be unusually effective (and may in fact be unusually poor), a phenomenon previous observed in [16].

Several research efforts have developed automatic test generation techniques based on formal models and model checkers [7, 18]. The technique we use is outlined in [18] and operates by generating NuSMV counterexamples through trap properties. Using this technique, we can create test suites achieving the maximum achievable coverage for a specified coverage metric.

A test suite generated using this approach will be highly redundant, as a single test case will often satisfy several obligations. Such a test suite is not representative of the coverage metric, as it may contain far more tests than are needed and may therefore bias the test suite's fault finding ability. We therefore reduce each test suite using a greedy algorithm. First, we randomly select a test case from the generated test suite, determine which obligations are satisfied by the test and add it to a reduced test suite. We continue by randomly choosing another test case from the generated test suite, determining if any obligations not satisfied by the reduced test suite are satisfied by the test case selected, and if so adding the test case to the reduced test suite. This process yields a randomly reduced test suite achieving the same requirements coverage as the the original test suite. We generated 25 reduced test suites for each coverage metric and each case example to avoid threats to validity due to a small sample size.

## 4.3 Random Test Suite Generation and Reduction

We generated a single set of 1,000 random tests for each case example. Each individual test contains between 2-10 steps with the number of tests of each test length distributed evenly in each set of tests. For each reduced requirements coverage test suite we create a random test suite with the same number of steps (or slightly more) by randomly sampling from the set of 1,000 random tests generated. As a result, each set of reduced requirements coverage test suites satisfying a coverage metric has a corresponding set of random test suites of approximately the same size. The number of steps was used as a measurement of size rather than the number of tests as this avoids creating a random test suite with significantly longer or shorter tests on average than the corresponding reduced requirements coverage test suite.

## 4.4 Mutant Generation

We created 600 *mutants* (faulty implementations) for each case example by introducing a single fault into the correct implementation. Each fault was introduced by either inserting a new operator into the system or by replacing an operator or variable with a different operator or variable. The faults seeded include

| | RC | RC-RAN | AC | AC-RAN | UFC | UFC-RAN |
|---|---|---|---|---|---|---|
| **DWM_1** | 77.8% | 78.7% | 78.6% | 78.7% | 88.0% | 81.6% |
| **DWM_2** | 0.0% | 1.60% | 0.0% | 1.29% | 5.41% | 9.42% |
| **Latctl_Batch** | 17.1% | 49.8% | 59.0% | 62.1% | 85.2% | 82.7% |
| **Vertmax_Batch** | 22.4% | 26.6% | 46.9% | 42.5% | 81.5% | 72.2% |

| | AC:RC Imp | UFC:RC Imp | UFC:AC Imp | RC:RC-RAN Imp | AC:AC-RAN Imp | UFC:UFC-RAN Imp |
|---|---|---|---|---|---|---|
| **DWM_1** | 1.01% | 13.1% | 12.0% | -1.0% | -0.1% | 7.8% |
| **DWM_2** | 0.0% | ∞ | ∞ | -100% | -100% | -42.0% |
| **Latctl_Batch** | 244.1% | 396.9% | 44.4% | -65.0% | -5.1% | 3.0% |
| **Vertmax_Batch** | 108.9% | 262.4% | 73.5% | -15.0% | 10.3% | 12.9% |

Table 1: Average and Relative Improvement in Fault Finding
Column Header ($X : Y$) Denotes Relative Fault Finding Imp. Using Test Suite $X$ over Test Suite $Y$
RC = Requirements Coverage, AC = Antecedent Coverage
$X$-RAN = Random Test Suite w/ Size ≈ Size of a Reduced Test Suite Satisfying $X$ Coverage

| | UFC ≤ AC | UFC ≤ RC | AC ≤ RC | UFC ≤ UFC-RAN | AC ≤ AC-RAN | RC ≤ RC-RAN |
|---|---|---|---|---|---|---|
| **DWM_1** | <0.001 | <0.001 | <0.001 | <0.001 | 0.62 | 1.0 |
| **DWM_2** | <0.001 | <0.001 | 1.0 | 1.0 | 1.0 | 1.0 |
| **Latctl_Batch** | <0.001 | <0.001 | <0.001 | <0.001 | 1.0 | 1.0 |
| **Vertmax_Batch** | <0.001 | <0.001 | <0.001 | <0.001 | 1.0 | 1.0 |

Table 2: Statistical Analysis for $H0_1$ and $H0_2$
RC = Requirements Coverage, AC = Antecedent Coverage
$X$-RAN = Random Test Suite w/ Size ≈ Size of a Reduced Test Suite Satisfying $X$ Coverage

arithmetic, relational, boolean, negation, delay introduction, constant replacement, variable replacement and parameter replacement and are described in more detail in [16].

We generated 600 mutants for each case example. From these 600 mutants, we generated 25 sets of 200 mutants. We generated multiple sets of mutants to avoid threats to validity due to a small sample size. Note that we did not check that generated mutants are semantically different from the original implementation. This weakness in mutant generation does not affect our results since we are interested in the relative fault finding ability between test suites.

# 5 Results

To determine the fault finding of a test suite $T$ and a mutant set $M$ for a case example we simply compare the output values produced by the original case example against every mutant $m \in M$ using every test case $t \in T$. The fault finding effectiveness of the test suite for the mutant set is computed as the number of mutants killed divided by the number of mutants in the set. We perform this analysis for each test suite and mutant set for every case example yielding 7,500 fault finding measurements per case example. We use the information produced by this analysis to test our hypotheses and infer the relationship between coverage criteria and fault finding.

The complete analysis is too large to include in this report[1]. For each case example, we present in Table 1 the average and relative improvement in fault finding for each coverage metric and for the random test suites corresponding to each coverage metric. We limit the discussion in this section to statistical analysis, a discussion of these results and their implications follows in Section 6.

## 5.1 Statistical Analysis

To evaluate our hypotheses (from Section 4), we first evaluate each hypothesis for each combination of a case example and requirements coverage metric (for $H_1$) or pairing of requirements coverage metrics (for $H_2$). Using these results, we determine what conclusions can be generalized across all systems.

[1] Our data can be retrieved at `http://crisys.cs.umn.edu/public_datasets.html` and is available to the research community for additional analysis.

| UFC > AC | UFC > RC | AC > RC | UFC > UFC-RAN | AC > AC-RAN | RC > RC-RAN |
|---|---|---|---|---|---|
| Supported | Supported | Unsupported | Unsupported | Unsupported | Unsupported |

Table 3: Conclusions for Hypotheses Across All 4 Case Examples
RC = Requirements Coverage, AC = Antecedent Coverage
$X$-RAN = Random Test Suite w/ Size $\approx$ Size of a Reduced Test Suite Satisfying $X$ Coverage
Each column ($X > Y$) Denotes Hypothesis of "Test Suites Satisfying $X$ Have Greater Fault Finding Than Test Suites Satisfying $Y$"

Each of the individual hypotheses states that one set of fault finding measurements should be in general higher than another set of fault finding measurements. For example, we hypothesize that fault finding measurements for reduced test suites satisfying UFC coverage are better than fault finding measurements for the set of comparably sized random test suites (for each oracle and case example). To evaluate our hypotheses, we use a bootstrapped permutation test, a non-parametric statistical test that determines the probability that two sets of data belong to the same population [11], and explore 1,000,000 permutations when calculating each $p$-value. From a practical standpoint, we are only interested in scenarios where test suites satisfying a requirements coverage metric outperform random test suites and where test suites satisfying a more rigorous coverage metric outperform a less rigorous coverage metric. Consequently, we formulate our null hypotheses as *one-tailed* tests, restating $H_1$ and $H_2$ as the null hypotheses $H0_1$ and $H0_2$, respectively:

***H0*$_1$*:*** *For case example CE, the data points for percentage of mutants caught by test suites satisfying coverage metric C are less than or equal to the data points for percentage of mutants caught by random test suites of approximately the same size.*

***H0*$_2$*:*** *For case example CE, the data points for percentage of mutants caught by test suites satisfying coverage metric $C_1$ are less than or equal to the data points for percentage of mutants caught by test suites satisfying a less rigorous coverage metric $C_2$.*

We evaluate $H0_1$ and $H0_2$ using 4 case examples and 3 coverage metrics or 3 pairings of coverage metrics, respectively. We therefore produce 12 $p$-values each for $H0_1$ and $H0_2$, listed in Table 2. These $p$-values are used to generalize across our results.

We use the Bonferonni correction, a conservative method for guarding against erroneously rejecting the null hypothesis when generalizing results. The Bonferonni correction sets the alpha required for each null hypothesis at $1/n$ times the alpha desired for the entire set, where $n$ is the number of results generalized; in this case, the alpha is set at $1/4 * 0.05 = 0.0125$ for each hypothesis. We present the results in Table 3.

Given these results, we can evaluate our original hypotheses:

> **Hypothesis 1 ($H_1$):** *A test suite satisfying a requirements coverage metric provides greater fault finding than a randomly generated test suite of approximately equal size.*
> **Hypothesis 2 ($H_2$):** *A test suite satisfying a requirements coverage metric provides greater fault finding than a test suite satisfying a less rigorous requirements coverage metric.*

The majority of test suites satisfying a coverage metric provide worse fault finding than random test suites of similar size, and thus $H_1$ is not statistically supported. Furthermore, test suites satisfying antecedent and requirements coverage provide similar levels of fault finding for the *DWM_2* case example, and thus $H_2$ is not statistically supported.

However, we note here several interesting results from Tables 2 and 3, and explore them in Section 6. First, $H_2$ fails only when test suites satisfying antecedent and requirements coverage find no faults. Second, test suites satisfying UFC coverage outperform random test suites 75% of the time (3 of 4 case examples). Finally, random test suites always outperform test suites satisfying requirements and

antecedent coverage *with statistical significance* (derivation of this significance is not shown), indicating that the random tests are not simply equivalent to the tests generated to satisfy a coverage metric, but preferable. These results and our recommendations are explored in Section 6.

### 5.2 Threats to Validity

**External Validity:** We only used four synchronous reactive critical systems in the study. We believe, however, that these systems are representative of the critical systems in which we are interested and that our results can therefore be generalized to other related systems.

As our implementation language we used the synchronous programming language Lustre [8] rather than a more common language such as C or C++. Systems written in Lustre are similar in style to traditional imperative code produced by code generators. Therefore, testing Lustre code is sufficiently similar to testing reactive systems written in C or C++ to generalize the results to such systems.

We have used automatically seeded faults to evaluate the fault finding ability of tests suites. It is possible the faults seeded are not representative. Nevertheless, previous work indicates fault seeding methods similar to our own are representative of real faults encountered in software development [1].

We only report results using test oracles based on output variables. In pilot studies, we observed that test oracles considering internal variables in addition to output variables produced similar results.

Finally, we used automatic test generation rather than a domain expert when creating test suites. Clearly, the tests produced by an automated tool differ from tests likely to be produced by a domain expert, particularly in the case of requirements coverage. Nevertheless, we are interested in evaluating the *coverage metrics*, not domain experts, and thus view the worst-case test generation provided by the tools to be preferable—these tools tend to highlight deficiencies in coverage metrics, and thus our evaluation is not influenced by the skill of a domain expert.

**Conclusion Validity:** For each case example, we have performed resampling using a random test suite containing 1,000 tests and a set of 600 mutants. These values are chosen to yield a reasonable cost for the study. It is possible that sampling from larger random test suites may yield different results, or that the number of mutants is too low. Based on past experience, however, we have found results using 200 mutants to be representative [17, 16], and thus believe 600 mutants to sufficient.

For each case example, we have generated 25 reduced test suites for each coverage metric, a corresponding set of 25 random test suites for each coverage metric, and 25 sets of 200 mutants. These numbers were chosen to keep the cost of resampling reasonable. It is possible that the fault finding measurements that result do not accurately represent the set of possible results. Nevertheless, the low variance in fault finding measurements observed in the sets of test suites and mutants—coupled with the consistency of our results across case examples—indicates our conclusions are accurate.

## 6 Discussion

In Section 5, we showed that neither hypothesis was statistically supported. Nevertheless, the results lead to several worthwhile observations. First, despite the lack of statistical significance, the subsumption hierarchy between coverage metrics generally reflects the relative fault finding of test suites satisfying a coverage metric; the more rigorous the coverage metric, the more effective the corresponding test suite will be. This indicates a useful tradeoff in test suite rigor and test suite size exists.

This observation, however, is rendered largely moot by the effectiveness of random testing. For both requirements and antecedent coverage, random test suites provide greater fault finding than test suites satisfying these coverage metrics, often with statistical significance. Furthermore, for the *DWM_2* system, test suites satisfying UFC coverage do not outperform randomly generated test suites. The former result highlights problems with using automatic test generation tools to generate tests satisfying a coverage criterion. The latter result highlights how the structure of requirements can affect the usefulness of a coverage metric.

In the remainder of this section, we will discuss the implications of these results and recommendations concerning requirements coverage metrics.

## 6.1 Pitfalls of Automatic Test Generation

As mentioned in Section 4.2, automatic test generation tends to produce tests that satisfy coverage obligations using minimal effort. Tests generated tend to frequently use the default inputs (e.g., all signals set to false), and may in that way technically satisfy obligations while not exercising useful or representative scenarios. Consequently, a test suite generated to meet a coverage criterion might do mostly nothing, while a random test suite of equal size will generally do something (albeit something arbitrary). This accounts for the relatively good fault finding achieved by random test suites.

The degree to which this is a problem depends on the complexity of the coverage obligations. For example, the coverage obligations produced using requirements coverage for the *DWM_2* system are very simple, and consequently every test generated is exactly the same; therefore, one test satisfies every obligation. Conversely, the coverage obligations produced using UFC coverage for the *Vertmax_Batch* system are quite complex, requiring over 10 times as many tests as needed to satisfy requirements or antecedent coverage for the same system. When using complex obligations, automatic test generation must generate tests meeting a wide variety of constraints, including specific sequences of events, and consequently the test cases produced tend to be more realistic and more effective.

## 6.2 Pitfalls of Requirements Structure

For the *DWM_2* system, the randomly generated test suites outperform the test suites satisfying UFC coverage. We have observed similar issues in previous work [16] where we noted the requirements for this particular system are structured such that the UFC criterion is rendered ineffective as compared to our other case examples[2].

```
LTLSPEC G(var_a > (
    case
        foo : 0 ;
        bar : 1 ;
    esac +
    case
        baz : 2 ;
        bpr : 3 ;
    esac
));
```

Figure 1: Original LTL Requirement

```
LTLSPEC G(var_a > (
    case
        foo & baz : 0 + 2 ;
        foo & bpr : 0 + 3 ;
        bar & baz : 1 + 2 ;
        bar & bpr : 1 + 3 ;
    esac
));
```

Figure 2: Restructured LTL Requirement

We note that many of the requirements for the *DWM_2* system were of the form (formalized as SMV [14]) in Figure 1. Although this idiom may seem unfamiliar, this use of case expressions is not uncommon when specifying properties in NuSMV. Informally, the sample requirement states that *var_a* is always greater than the sum of the outcomes of the two case expressions. Generating UFC obligations for the above requirement yields very simple test obligations since there are no complex decisions – we simply need to find tests where the top-level relational expression and each atomic condition has taken on the values true and false.

This requirement can be restructured without changing the meaning as shown in Figure 2. Achieving UFC coverage over this restructured requirement will require more obligations than before since the boolean conditions in the case expression are more complex and we must demonstrate independence of each condition in the more complex decisions. Thus, the structure of the requirements has a significant impact on the number and complexity of UFC obligations required. For this experiment, we did not restructure any requirements. Consequently, the UFC obligations for the *DWM_2* do not have as complex

[2] This discussion is partially adopted from [16] as we encountered the same phenomenon in that investigation.

of a structure as those for the other three systems, and automatic test generation (as with requirements and antecedent coverage) produces poor tests.

### 6.3 Recommendations

The goal of using a coverage criterion is to infer the efficacy of a test suite. By verifying that a test suite satisfies a coverage criterion, one hopes to provide evidence that the test suite is "good" and is likely to uncover faults. While no objective standard for "good" fault finding exists, at a minimum a test suite satisfying a useful coverage criterion should provide better fault finding than a randomly generated test suite of similar size; otherwise, little confidence in the quality of a test suite is gained by showing it satisfies the coverage criterion.

The test suites generated by automatic test generation to satisfy requirements and antecedent coverage are in some sense "worst-case" test cases, and it is likely a domain expert using these metrics as a guide would produce far better tests. We are, however, not evaluating our coverage metrics as guidelines for developing tests, but as objective measure of black-box test suite adequacy, and our results clearly demonstrate that it is easy to satisfy requirements or antecedent coverage using inadequate test suites. We therefore conclude that *requirements and antecedent coverage are not sufficiently rigorous* to ensure a test suite satisfying one of these metrics is better than a random test suite of equal size—the intuition behind the metrics may be useful, but merely satisfying the metric does not provide convincing evidence.

Conversely, the test suites generated to provide UFC coverage, despite also being "worst-case" test cases, generally outperform random test suites of approximately the same size. The caveat to this is requirements structure—by breaking up complex requirements into several less complex requirements, the benefits of UFC coverage can be diminished. Nevertheless, our results indicate that the adequacy of black-box tests can be inferred using UFC coverage. We therefore conclude that UFC coverage is sufficiently rigorous to ensure a test suite is better than a random test suite of equal size provided the requirements are not structured to mask complexity. Admittedly, this is a rather weak conclusion and more research is needed to identify coverage criteria that are more effective and more robust with respect to the structure of the requirements.

Thus, in practice, testers who wish to objectively infer the adequacy of a black-box test suite with respect to a set of requirements formalized as LTL properties should use UFC coverage while noting the pitfalls outlined above.

## 7 Conclusion

In [19], we defined coverage metrics over the structure of requirements formalized as Linear Temporal Logic (LTL) properties. Such metrics are desirable because they provide objective, model-independent measures of the adequacy of black-box testing activities. In this paper, we empirically demonstrated that of the three coverage metrics explored, only the UFC coverage criterion is rigorous enough to be a useful measurement of test suite adequacy. We also noted that the usefulness of the UFC coverage criterion is influenced by the structure of the requirements. Consequently, we conclude that testers who wish to measure the adequacy of a test suite with respect to a set of LTL requirements should use a coverage criterion at least as strong as UFC coverage, but should be mindful of the structure of their requirements.

Furthermore, our work highlights the difference between using coverage metrics as guidelines for developing test suites, and using coverage metrics as objective measurements of test suite adequacy. We demonstrate how using an insufficiently rigorous coverage criterion for inferring test suite adequacy can lead to incorrectly concluding a test suite that technically meets the criterion is adequate, when the test suite is no better than a randomly generated test suite of approximately the same size. We believe this has implications in domains such as avionics and critical systems, where test coverage metrics are used by regulatory agencies to infer the efficacy of a testing process and thus the quality of a software system. We hope to investigate this problem in depth in future work.

# References


[1] J.H. Andrews, L.C. Briand, and Y. Labiche. Is Mutation an Appropriate Tool for Testing Experiments? *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, pages 402–411, 2005.

[2] I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh. Efficient detection of vacuity in ACTL formulas. In *Formal Methods in System Design*, pages 141–162, 2001.

[3] J. J. Chilenski and S. P. Miller. Applicability of Modified Condition/Decision Coverage to Software Testing. *Software Engineering Journal*, pages 193–200, September 1994.

[4] H. Chockler, O. Kupferman, and M. Y. Vardi. Coverage metrics for formal verification. In *12th Advanced Research Working Conference on Correct Hardware Design and Verification Methods, volume 2860 of Lecture Notes in Computer Science*, pages 111–125. Springer-Verlag, October 2003.

[5] G. Fraser, A. Gargantini, and V. Marconi. An evaluation of model checkers for specification based test case generation. In *Proceedings of the 2009 International Conference on Software Testing Verification and Validation*, pages 41–50. IEEE Computer Society Washington, DC, USA, 2009.

[6] G. Fraser and F. Wotawa. Complementary criteria for testing temporal logic properties. In *Proceedings of the 3rd International Conference on Tests and Proofs*, page 73. Springer, 2009.

[7] Angelo Gargantini and Constance Heitmeyer. Using model checking to generate tests from requirements specifications. *Software Engineering Notes*, 24(6):146–162, November 1999.

[8] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The Synchronous Dataflow Programming Language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[9] K.J. Hayhurst, D.S. Veerhusen, and L.K. Rierson. A practical tutorial on modified condition/decision coverage. Technical Report TM-2001-210876, NASA, 2001.

[10] O. Kupferman and M. Y. Vardi. Vacuity detection in temporal model checking. *Journal on Software Tools for Technology Transfer*, 4(2), February 2003.

[11] P.H. Kvam and B. Vidakovic. *Nonparametric Statistics with Applications to Science and Engineering*. Wiley-Interscience, 2007.

[12] Mathworks Inc. Simulink product web site. http://www.mathworks.com/products/simulink.

[13] S. Miller, E. Anderson, L. Wagner, M. Whalen, and M. Heimdahl. Formal verification of flight critical software. In *Proceedings of the AIAA Guidance, Navigation and Control Conference and Exhibit*, August 2005.

[14] The NuSMV Toolset, 2005. Available at http://nusmv.irst.itc.it/.

[15] A. Pnueli. Applications of temporal logic to specification and verification of reactive systems: A survey of current trends. *Lecture Notes in Computer Science Number 224*, pages 510–584, 1986.

[16] A. Rajan, M. Whalen, M. Staats, and M.P. Heimdahl. Requirements coverage as an adequacy measure for conformance testing. In *Proceedings of the 10th International Conference on Formal Methods and Software Engineering*, pages 86–104. Springer, 2008.

[17] A. Rajan, M.W. Whalen, and M.P.E. Heimdahl. The effect of program and model structure on MC/DC test adequacy coverage. In *Proc. of the 30th Int'l Conference on Software engineering*, pages 161–170. ACM New York, NY, USA, 2008.

[18] Sanjai Rayadurgam and Mats P.E. Heimdahl. Coverage based test-case generation using model checkers. In *Proc. of the 8th IEEE Int'l. Conf. and Workshop on the Engineering of Computer Based Systems*, pages 83–91. IEEE Computer Society, April 2001.

[19] M.W Whalen, A. Rajan, and M.P.E. Heimdahl. Coverage metrics for requirements-based testing. In *Proceedings of International Symposium on Software Testing and Analysis*, pages 25–36, July 2006.

# Software Model Checking of ARINC-653 Flight Code with MCP

Sarah J. Thompson
SGT Inc., NASA Ames Research Center
MS269-1, Moffett Field, California
`sarah.j.thompson@nasa.gov`

Guillaume Brat
CMU, NASA Ames Research Center
MS-269-1, Moffett Field, California
`guillaume.p.brat@nasa.gov`

Arnaud Venet
SGT Inc., NASA Ames Research Center
MS269-1, Moffett Field, California
`arnaud.j.venet@nasa.gov`

**Abstract**

The ARINC-653 standard defines a common interface for Integrated Modular Avionics (IMA) code. In particular, ARINC-653 Part 1 specifies a process- and partition-management API that is analogous to POSIX threads, but with certain extensions and restrictions intended to support the implementation of high reliability flight code.

MCP is a software model checker, developed at NASA Ames, that provides capabilities for model checking C and C++ source code. In this paper, we present recent work aimed at implementing extensions to MCP that support ARINC-653, and we discuss the challenges and opportunities that consequentially arise. Providing support for ARINC-653's time and space partitioning is nontrivial, though there are implicit benefits for partial order reduction possible as a consequence of the API's strict interprocess communication policy.

## 1 Introduction

NASA missions are becoming increasingly complex, and, more and more of this complexity is implemented in software. In 1977, the flight software for the Voyager mission amounted to only 3000 lines. Twenty years later, the software for Cassini had grown by a factor of ten, and more strikingly, the software for the Mars Path Finder mission amounted to 160 KLOCs (thousands of lines of code). Nowadays, the software for the Space Shuttle has reached half a million lines of code (0.5 MLOCs). Moreover the software for the International Space Station has exploded to 3 MLOCs and it is still increasing. Some experts have estimated that the software required by the Constellation project will reach at least 50 MLOCs. Yet, NASA is still relying on traditional (and expensive) testing and simulation to verify its software.

Naturally, NASA is now exploring new ways to speed up the testing process, reduce its cost, and increase its efficiency. Model checking is seen as a way of helping verifying software, especially for multi-threaded code. Our colleagues at JPL developed the well-known SPIN model checker [12]. Unfortunately, it requires translating code into a modeling language called Promela, which is not always feasible or practical. However Promela models are compiled into C programs that perform the model checking activity. JPL therefore developed *model-driven verification* [13]; it consists of embedding C code fragments into the compiled Promela models. This technique allows them to partially check C programs. NASA Ames has taken a different approach called software model checking. For example, the JPF (Java PathFinder) model checker can check Java programs without any translation [11].

Experience from the JPF project has demonstrated the utility of *software model checking* (i.e. model checking that acts directly on a program, rather than on a model that has been manually extracted from it). However, current flight software is mostly implemented in C, not Java, and in the future it seems increasingly likely that C++ will become the platform of choice. The MCP model checker is being developed to fill the requirement for an explicit-state software model checker, in the style of JPF and

SPIN, that fully supports the C++ language. Moreover, we envision designing a continuum of tools for the verification of C and C++ programs and test case generation. The tools will span techniques ranging from static analysis to model checking.

## 1.1 ARINC-653

The ARINC-653 standard [1, 3, 2] specifies the software interface to be used by developers of Integrated Modular Avionics flight software. It is comprised of three parts:

**Part 1:** *Avionics software standard interface (APEX).* This part defines the operating system's interface to avionics code, with specific focus on partition management, process management and error handling.

**Part 2:** *APEX extensions.* Since Part 1 is insufficient on its own to support many kinds of practical code, Part 2 extends the standard to provide interfaces for file handling, communications and a few other commonly used operating system services.

**Part 3:** *Compliance testing specification.* Part 3 does not define extra functionality in and of itself – rather, it specifies how compliance testing should be carried out for implementations of parts 1 and 2.

In this paper we concentrate on ARINC-653 Part 1. In a very real sense, Part 1 can be thought of as occupying the same part of the software food chain as POSIX threads [15], though its execution model is somewhat different.

### 1.1.1 Partitions

The key defining feature of ARINC-653 is its inclusion of *partitions.* A partition is analogous to a Unix process, in the sense that it runs in its own, separate memory space that is not shared with that of other partitions. Partitions also have strictly protected time slice allocations that also may not affect the time slices of other partitions – the standard's aim is to ensure that if a partition crashes, other correctly functioning partitions are unaffected. It is not possible, in standards-compliant code, to define areas of shared memory between partitions – all interpartition communication must be mediated via the APEX API.

One area where partitions differ considerably from Unix processes is in their strict adherence to a well-defined start up and shutdown mechanism, with strict prohibition of dynamic allocation and reconfiguration. On cold- or warm-boot[1], only the partition's primary process may execute. It then starts any other processes, creates and initializes interprocess and interpartition communications channels, allocates memory and performs any other necessary initialization. The primary process then sets the partition's state to NORMAL, at which point no further dynamic initialization (including memory allocation) is allowed. Setting the partition's state to IDLE causes all threads to cease execution.

### 1.1.2 Processes

ARINC-653 processes are analogous to POSIX threads[2]. A partition may include one or more processes that share time and space resources. Processes have strictly applied priorities – if a process of higher priority than the current process is blocked and becomes able to run, it will preempt the running process

[1] The exact meaning of cold and warm is left to the implementer.

[2] Arguably, the usage of the word 'process' in the standard is unconventional, and can be confusing.

immediately. Processes that have equal priority are round-robin scheduled. Memory is shared between all processes within a partition.

### 1.1.3 Error handling

Each partition has a special, user-supplied, error handling process that runs at a higher priority than all other processes in the partition. Normally it sits idle, consuming no time resources, until it is invoked as a consequence of an error detected by the operating system or explicitly raised by the running code. It then defines how the partition should respond, and can (for example) cause a partition to be restarted if necessary.

It is possible to define watchdog timeouts for processes that cause the error handler to be invoked automatically if time limits are exceeded.

### 1.1.4 Events

ARINC-653 Part 1 events are similar, though somewhat simpler than, the event synchronization facilities provided in most other threading APIs. Events may be explicitly in a *set* or *reset* state – when set, they allow all processes waiting on the event to continue. When reset, all processes waiting on the event are blocked. No support for self-resetting events, or events that allow only a single process to proceed are supported, nor is there explicit support for handling priority inversion.

### 1.1.5 Semaphores

Semaphores in Part 1 behave in the traditional way – they are typically used to protect one or more resources from concurrent access. A semaphore created with an initial resource count of 0 and a resource count limit of 1 behaves exactly like the mutex facilities found in other threading APIs.

### 1.1.6 Critical Sections

APEX defines a single, global, critical section that, when locked, prevents scheduling. This is a little different, and more extreme in effect, in comparison with the critical section facilities in POSIX threads and in the Windows threading API – rather, it is analogous to turning off interrupt handling.

### 1.1.7 Buffers and Queuing Ports

ARINC-653 *buffers* are actually thread-safe queues intended for interprocess, intrapartition communication. They are constructed with a preset maximum length and maximum message length which may not be varied at run time. Messages consist of blocks of arbitrarily formatted binary data. Processes attempting to read from an empty buffer block until another process inserts one or more messages. Similarly, attempting to write to an already-full buffer will cause the sending process to block until space becomes available.

*Queuing ports* are the interpartition equivalent to buffers, and provide queued communication between partitions.

### 1.1.8 Blackboards and Sampling Ports

*Blackboards,* ARINC-653's other interprocess/intrapartition communications mechanism, are analogous to buffers, except that they store exactly zero or one messages. Processes may write to blackboards at any time without blocking – the message that they send replaces the existing message, if any. Reading

from an empty blackboard causes the reading process to block until another process writes a message to the blackboard.

*Sampling ports* are the interpartition counterpart to blackboards. Their semantics are slightly different, in that neither sending or receiving processes ever block. They also add a timestamping facility that makes it easy for a receiving process to check whether the data it has retrieved has become stale.

## 1.2 The MCP Model Checker

The MCP (Model Checker for C++) project began as an attempt to reimplement the JPF architecture for C++: JPF implements an extended virtual machine interpreting Java bytecode with backtracking, and the first version of MCP had a similar architecture, substituting LLVM [16] for Java. MCP's current architecture is closer to that of SPIN than of JPF, however – rather than running code in an instrumented virtual machine with backtracking, we use program transformation techniques to instrument the program itself, then run it natively in an environment whose run-time system implements backtracking.

### 1.2.1 LLVM: Low Level Virtual Machine

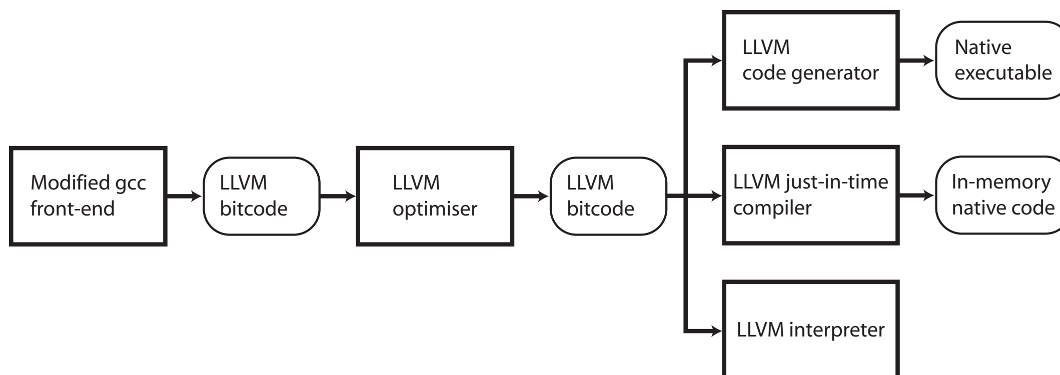



Figure 1: Simplified LLVM architecture

Fig. 1 shows a simplified version of the LLVM flow[3]. A modified version of the `gcc` front-end is used to parse the C++ source code and to lower most of the language's constructs to a level closer to that of a typical C program. The original `gcc` back-end is discarded in favour of emitting *LLVM bitcode,* which is then optimised and passed on to various alternative back-ends.

The LLVM bitcode format was specifically designed to support program analysis, transformation and optimization – a Static Single Assignment (SSA) representation [8] is adopted, making many analyses and transformations (including ours) far more straightforward than they might otherwise be.

### 1.2.2 The MCP Architecture

Fig. 2 shows an outline of the MCP architecture. Functionality is split across several subsystems:

**Transformation Passes** Several MCP-specific transformations that instrument the code under test are implemented as an extension module for LLVM's `opt` program transformation framework.

[3]Many LLVM tools have been omitted here for clarity – LLVM is a large, rich toolset, so we concentrate on the subsystems that are specifically relevant to MCP.

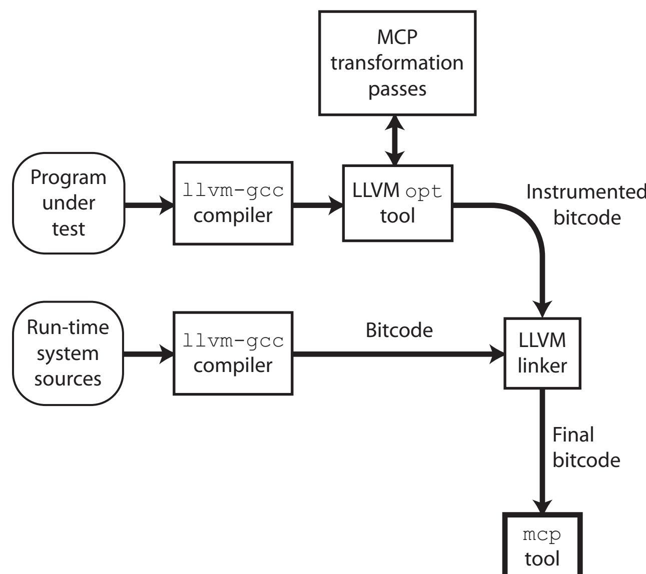



Figure 2: MCP Architecture

**Run-time System** A run-time system is implemented in C++, compiled with LLVM's `gcc` front-end, then linked with the program under test after it has been transformed. Its primary purpose is to intercept system calls that MCP needs to handle differently, *e.g.,* `printf`, `malloc`, `free`, `memset`, `memmove` and `memcpy`. This approach also provides a convenient place to implement compatibility wrappers that allow code written to specific operating system APIs to be handled without modification – the ARINC-653 subsystem is implemented as an extension to the run-time system.

**JIT Environment/User Interface** Model checking is initiated by users by running the `mcp` command-line application. The `mcp` tool comprises an instance of the LLVM just-in-time (JIT) compiler environment, as well as MCP's implementations of memory versioning, hashing, state space searching etc.

### 1.2.3 Emulating Threading

One of the trickier issues surrounding the model checking of C++ source code is the fact that the language standard specifically does not mandate any particular threading model. Real-time program semantics are therefore dependent upon the execution environment, so any attempt to analyse multithreaded code must inevitably make some kind of assumption about the underlying threading model.

MCP implements its own low-level API, on top of which arbitrary threading models may be constructed. This API is deliberately designed to, as far as possible, serve as a superset of all of the threading models that are of interest.

### 1.2.4 Memory Management, State Storage and Backtracking

Model checking C and C++ directly is significantly complicated by the languages' widespread use of weakly typed pointers and of pointer arithmetic. Legal programs can cast pointers to other types, including integers, or directly access the memory in which the pointer is stored – many pointer analyses

therefore, attempted statically, though feasible for a subset of programs become equivalent to the halting problem for the general case. As a consequence, in order to maintain the primary goal of being able to support the entire programming language, there was little alternative other than to maintain a very concrete memory model that is in practice identical to that of a normally executing program. Programs running within the MCP environment access memory in the same way that normal executables do (including pointer arithmetic), with the exception that the run-time system tracks and records all non-thread-local reads and writes. Context switches record changes to stack frames and CPU register contents. This information, along with source line and intermediate code tracing information is recorded in a tree structure that, by means of undoing or redoing recorded changes, makes it possible to recreate any previously recorded memory and CPU state. All state changes are stored compactly as deltas – complete states are never stored. State hashes are computed incrementally and collision checked, though it is also possible to hand-annotate code in order to allow states to be matched abstractly.

#### 1.2.5 Search Algorithms

MCP implements several built-in search strategies:[4].

**Breadth-first search** Paths through the search space are explored in breadth order.

**Depth-first search** Paths are executed to completion before backtracking continues.

**User-specified heuristic search** In this strategy, the test harness provides a ranking function that allows MCP to implement a best-first search strategy.

**Randomised search** Randomised search explores the search space in a random order. In practice, randomised search has a 'look and feel' somewhere between that of breadth- and depth-first search, but in practice it has less tendency to get stuck in local minima.

**Interpreted mode** In this mode, the program is executed normally, without backtracking, with threading implemented by a fair scheduler. Programs are still traced in just as much detail as in any other mode, however, so a backtrace is still generated on termination.

**Skew-First Search** In this mode, search concentrates on a fair schedule first, then works toward cases where threads have increasing relative *skew*[5] as the search progresses. In practice, this mode resembles depth-first search, though it is more aggressive in avoiding the favoring of any thread over any other. Used in an incomplete search, skew-first search essentially begins by running the program conventionally, then tries alternatives that are decreasingly close to a fair schedule until a time, memory or search depth limit is hit.

**Reverse Skew-First Search** This mode essentially negates the search metric used in skew-first search. As a consequence, the search explores corner cases with maximum relative skew first, working toward a fair schedule last.

[4]Since MCP's partial order reduction algorithm does not presuppose any particular execution order, MCP does not have a performance bias toward any particular search strategy.

[5]MCP counts the number of program steps executed by each thread, with the difference between the minimum and maximum step count across all running threads being regarded as the *skew*.

**Coverage-First Search** MCP has the ability to track source line coverage[6], primarily for the purpose of providing a code coverage report on termination. However, since coverage metrics are available dynamically, it was possible to feed this back as a search strategy, allowing the model checker to optimize its search strategy in order to execute as much of the program as possible as soon as possible.

**Per-thread independent search strategies** MCP has the capability of setting a search strategy independently for each thread, thereby giving the user a lot of control over search order without requiring user-specificied heuristics. For example, if a problem is suspected to exist between a pair of threads, it is possible to perform exhaustive depth- or breadth-first search on those threads, whilst merely interpreting other threads. Though used in this way this leads to incomplete analyses, it allows many problems that are too large or too complex for complete search to be addressed straightforwardly.

**Kill mode** In this mode, the program is not executed at all, and terminates immediately[7].

#### 1.2.6 Assertions and Backtrace Generation

Following practical experience in applying the JPF software model checker, a decision was made to move away from the more typical approach of specifying properties in terms of, for example, Linear Temporal Logic (LTL). Though LTL is convenient for many kinds of high-level properties, it does not lend itself well to low-level properties that require, for example, assertions about consistent pointer and memory usage – in such cases, expressing properties directly in C++ tends to be more natural, and of course C++, unlike LTL, is Turing-complete. Some experiments have been carried out on a C++ template library that implements LTL, which when more mature will allow users to specify properties in LTL if that is their preference, however.

When an assertion occurs or when MCP detects an error (*e.g.,* due to a segmentation fault, deadlock, assertion failure or some other problem in the code under test), it generates a backtrace from the beginning of execution of the current execution trace until the most recently executed instruction is reached. Backtrace logs may optionally include all executed LLVM instructions, executed source lines and all memory contents that are read or written by the program. Only the actual trace from the currently executing code fragment is generated – the (usually enormous and irrelevant) logging information from other traces that did not lead to errors are ignored.

## 2 Model checking ARINC-653 code

Since MCP supports the entire C++ programming language, the special considerations required for checking ARINC-653 code are specifically related to the implementation of its peculiar threading semantics. Generally, when dealing with library functionality in an explicit state model checking environment it is difficult to simply create a simplistic library implementation consisting only of stubs for each function – where such libraries affect program semantics by means of creation or manipulation of threading behavior, this is compounded significantly. Consequentially, our ARINC-653 implementation therefore needed in practice to become a fairly complete implementation of the standard, similar to that which might be included in an actual real-time operating system kernel. Though much of ARINC-653 can be mapped to existing approaches, there is sufficient oddity that a simplistic approach such as mapping APEX to an existing POSIX threads implementation is not sufficient.

[6] At the time of writing, other coverage metrics, particularly MCDC, are being considered for inclusion in a future version.

[7] Kill mode is not generally useful when applied to a whole program, but when applied just to one or more threads it can be used to dynamically search a program slice that excludes those threads.

**Time management** Like most threading APIs, ARINC-653 Part 1 provides facilities for dealing with time, ranging from sleeping for a particular interval, waiting until a specific time, timeouts on waiting on blocking synchronization objects, etc. It also supplies some facilities that are squarely aimed at avionics code, such as watchdog timeouts and automatically detecting stale data.

Ideally, it would be preferable to be able to accurately model program execution time. However, since actual flight code is likely to be targeted at an entirely different CPU architecture [14, 22] and compiled with a different toolchain [9], this is impractical, and indeed it could be potentially dangerous to extrapolate results from one platform to another. Therefore, code is assumed to run arbitrarily (though not infinitely) quickly, unless it explicitly waits via an API call. We therefore concentrate on modeling time at the level of explicit waits and timeouts rather than at the level of instructions.

Since MCP backtracks, its time implementation must also be able to backtrack. Consequentially, time is emulated rather than taken from a real-time clock. This has a number of benefits, not least of which being that arbitrarily long wait intervals can be emulated without needing to actually wait for the specified length of time – modeling, for example, Earth-Mars communications links with very long packet round-trip times, becomes feasible and efficient.

**Process management** ARINC-653 Part 1 processes map fairly directly to MCP's existing thread model, so could be accommodated relatively straightforwardly. Partitions were trickier to support, because they required a new memory protection system to be implemented that could segment memory access and check that it is being accessed with appropriate partitions. This turned out to be the single largest change necessary to MCP in order to support the APEX API.

**Partition start up/initialization control** Most of the requirements of partition start-up and initialization control involve parameter checking within API calls, so this is dealt with largely with assertions in the API implementation. MCP's memory manager was extended to support a flag that causes memory allocation to throw an error, making it possible to detect accidental memory allocations while a partition is in NORMAL mode.

**Sampling ports, queuing ports, buffers and blackboards** These communications APIs were implemented fairly straightforwardly, and did not require alteration to the MCP core. Synchronization was implemented purely in terms of MCP's native event mechanism.

**Events, semaphores, critical sections and error handling** These APEX API features mapped more or less directly to MCP's existing facilities. Some changes were necessary, but these were mostly consequential to the partitioned memory model support.

## 3 Partial order reduction under a partitioned memory model

Partial order reduction is carried out by model checkers in order to reduce the impact of the exponential time complexity inherent in backtracking. Given $N$ representing the size of the program fragment under test and $a$ representing the amount of possible nondeterminism at each decision point, time complexity for explicit state model checking has an upper bound proportional to $a^N$. It therefore behooves us to try to get $a$ down to as close to 1 as possible, because this has a dramatic effect on the size of program $N$ that can be practically analyzed. Partial order reduction techniques typically attack this in two ways: making the execution steps bigger by bundling thread-local operations together (thereby effectively reducing $N$), and where possible bundling together nondeterministic choices that have equivalent consequences (reducing $a$ where possible). MCP does both – it leverages LLVM's static analysis and optimization

capability in order to make the steps between yield points as large as possible, and second mechanism tracks reads and writes to shared memory, suspending evaluation lazily when only a subset of currently running threads have touched the relevant locations.

Though MCP's existing partial order reduction strategy can be applied to ARINC-653 code, there are some potential benefits available as a consequence of the partitioned execution scheme. In particular, the decision to only allow partitions to affect each other via API calls has profound consequences. A partition with a single process, or with multiple processes none of which having the same priority, is inherently deterministic. Therefore, nondeterminism may only arise as a consequence of timing relationships between such partitions. The MCP APEX implementation optionally treats partitions as executing atomically between API calls, offering a huge speedup with minimal time or memory overhead. Initial results are encouraging, though at the time of writing this functionality is too new for it to be possible to quote performance statistics.

# 4 Related work

Structurally, MCP probably bears closest resemblance to JPF (Java Pathfinder) [11], though at the time of writing it does not approach JPF's maturity. The most significant differences between JPF and MCP stem from the differences between Java and C++; for example, JPF takes advantage of reflection and the standard threading package in Java, which MCP can not since those features are not present in C or C++.

Since JPF is a explicit-state model checker "a la SPIN", MCP is also a close cousin to SPIN [12]. Besides the explicit-state model, MCP also shares with SPIN the concept of compiling the model checking problem into an executable program. SPIN starts with Promela models while MCP performs transformations on C/C++ programs to embed the model checking problem into the original program.

Another model checker directly addressing C++ is Verisoft [10], which takes a completely different approach. Verisoft follows a stateless approach to model checking while MCP follows a conventional explicit-state model similar to SPIN [12].

CBMC is a bounded Model Checker for C and C++ programs [5]. It can check properties such as buffer overflows, pointer safety, exceptions and user-specified assertions. CBMC does model checking by unwinding loops and transform instructions into equations that are passed to a SAT solver. Paths are explored only up to a certain depth.

There are, however, several model checkers that address C. SLAM [4] is really more of a static analyzer than a model checker. It relies heavily on abstractions, starting from a highly abstracted form and building up to a form that allows a complete analysis. CMC [17] uses an explicit-state approach, but it requires some manual adaptation when dealing with complex types (*pickle* and *unpickle* functions).

Finally, there have been some attempts within NASA to use the Valgrind tool [18, 20] as a model checker. Unfortunately, it implies using very crude steps between transitions.

Other approaches to model checking code involve a translation step, be it automatic or manual. For example, Bandera [7] provides model checking of Java programs by translating automatically the program into a PVS [19], Promela [21] or SMV [6] model.

# 5 Conclusions

The ARINC-653 Part 1 support in MCP is very new and still under development, so the purpose of this paper is to provide a first look at the capability within NASA's wider formal methods community.

Other than a requirement for detailed testing and attention to detail with respect to standards compliance, the Part 1 implementation is largely complete at the time of writing. We hope, over the next few

months, to put together a larger demonstration of the technology based on a moderate-sized, ARINC-653-based flight code model example, which will help us further tune our APEX implementation. If sufficient interest is shown, extending our implementation to encompass Part 2 would be feasible.

We hope also to take advantage of MCP's ability to set its search strategy independently on a per-thread basis in order to parallelize incomplete search. Since many threading problems exist only between pairs (or at least a small subset) of threads, enumerating all possible pairwise combinations and searching them in parallel, then proceeding to all 3-way combinations, then all 4-way combinations, etc., makes it possible to execute these searches in parallel on separate CPU cores or physically separate computers in a cluster.

At the time of writing, MCP is not publicly available. However, we are progressing an internal NASA release process that, if successful, will eventually allow MCP to be released as open source software.

## Acknowledgments

The work described in this paper was supported by the NASA ETDP (Exploration Technology Development Program) Intelligent Software Design Project

## References

[1] Aeronautical Radio Inc. *Avionics application software standard interface Part 1 – required services*, 12 2005. ARINC Specification 653P1-2.

[2] Aeronautical Radio Inc. *Avionics application software standard interface Part 3 – conformity test specification*, 10 2006. ARINC Specification 653P3.

[3] Aeronautical Radio Inc. *Avionics application software standard interface Part 2 – extended services*, 12 2009. ARINC Specification 653P2-1.

[4] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. Rajamani, and A. Ustuner. Thorough static analysis of device drivers, 2006.

[5] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Proc. TACAS*, pages 168–176, 2004.

[6] CMU. The SMV system. http://www.cs.cmu.edu/ modelcheck/smv.html.

[7] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering*, June 2000.

[8] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.

[9] GHS Inc. *Green Hills Software Inc. web site*. http://www.ghs.com/.

[10] P. Godefroid. Model checking for programming languages using Verisoft. In *Symposium on Principles of Programming Languages*, pages 174–186, 1997.

[11] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer (STTT)*, V2(4):366–381, March 2000.

[12] G. Holzmann. *The SPIN model checker: primer and reference manual*. Addison-Wesley, September 2003.

[13] G. Holzmann and R. Joshi. Model-driven software verification. In *Proceedings of the 11th SPIN Workshop*. ACM SIGSOFT, April 2004.

[14] IBM. *PowerPC 740, Power PC 750 RISC Microprocessor User's Manual*, 2 1999. GK21-0263-00.

[15] IEEE. *IEEE Standard for Information Technology – Portable operating system interface (POSIX). Shell and utilities*, 2004. 1003.1-2004.

[16] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[17] M. Musuvathi, A. Chou, D. L. Dill, and D. Engler. Model checking system software with CMC. In *EW10: Proceedings of the 10th workshop on ACM SIGOPS European workshop: beyond the PC*, pages 219–222, New York, NY, USA, 2002. ACM Press.

[18] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of PLDI 2007*, June 2007.

[19] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. K. Srivas. PVS: Combining specification, proof checking, and model checking. In *CAV '96: Proceedings of the 8th International Conference on Computer Aided Verification*, pages 411–414, London, UK, 1996. Springer-Verlag.

[20] J. Seward and N. Nethercote. Using Valgrind to detect undefined value errors with bit-precision. In *Proceedings of the USENIX'05 Annual Technical Conference*, April 2005.

[21] SPIN web site. Promela language reference. http://spinroot.com/spin/Man/promela.html.

[22] Synova Inc. *Mongoose-V MIPS R3000 Rad-Hard Processor web site*. http://www.synova.com/proc/mg5.html.

# Evaluation of a Guideline by Formal Modelling of Cruise Control System in Event-B


Sanaz Yeganefard
University of Southampton
United Kingdom
sanaz_yeganeh@yahoo.com

Michael Butler
University of Southampton
United Kingdom
mjb@ecs.soton.ac.uk

Abdolbaghi Rezazadeh
University of Southampton
United Kingdom
ra3@ecs.soton.ac.uk



**Abstract**

Recently a set of guidelines, or cookbook, has been developed for modelling and refinement of control problems in Event-B. The Event-B formal method is used for system-level modelling by defining states of a system and events which act on these states. It also supports refinement of models. This cookbook is intended to systematise the process of modelling and refining a control problem system by distinguishing environment, controller and command phenomena. Our main objective in this paper is to investigate and evaluate the usefulness and effectiveness of this cookbook by following it throughout the formal modelling of cruise control system found in cars. The outcomes are identifying the benefits of the cookbook and also giving guidance to its future users.


## 1 Introduction

Systems which consist of parts to interact with and react to the evolving environment continuously are known as embedded systems. They are complex and often used in life critical situations which means costs of their failure are usually high. Thus, reliability, safety and correctness in such systems are important [7, 11] and techniques such as formal methods can help to examine the behaviour of these systems in early development stages [12]. However the process of modelling itself can present considerable challenges and following modelling guidelines [8, 9] and patterns [16] can be helpful.

One of these is a set of guidelines, or cookbook, which has been developed recently for modelling and refinement of control problems in Event-B. Event-B formal method is used for system-level modelling. The main objective of this paper is to investigate how effective and useful having such a set of guidelines is by applying it to a real application. Cruise control system (CCS) is the chosen real application for the modelling, since the attempt of this cookbook is to outline the necessary steps of modelling an embedded system which consists of a controller, a plant and an operator. Also, in order to make this model a good example for the future users of the cookbook, some of the main points which helped us during the modelling are explained as tips.

This paper is organised into 5 sections. In Section 2 the background of this work is discussed. Here we look at Event-B and its tool Rodin. After that an outline of CCS and the cookbook are given. In Section 3 the modelling process is explained in more details. The most abstract level of the model is described in Section 3.1 and the refinement steps in 3.2 and 3.3. The refinement proofs related to these steps have been verified with Rodin tool. Finally in Section 4 and 5 we evaluate the cookbook and consider limitations and future work.

## 2 Background

### 2.1 Event-B and Refinement

Formal methods are mathematical based techniques which are used for describing the properties of a system. They provide a systematic approach for the specification, development and verification of software and hardware systems and because of the mathematical basis we can prove that a specification is satisfied

by an implementation [19]. The formal method used in our work is Event-B [5] which is extended from B-method [2]. It has simple concepts within which a complex and discrete system can be modelled. The main reasons for choosing Event-B are firstly that it is the language used in the cookbook and secondly, it has advantages such as simplicity of notations and extendibility and thirdly its tool support.

**Structure of Event-B** Event-B models consist of two constructs, *contexts* and *machines* [5, 13]. Contexts which specify the static part of a model and provide axiomatic properties, can contain the following elements: *carrier sets*, *constants* and *axioms*. Machines represent the dynamic part of a model and can contain *variables*, *invariants* and *events*. An event consists of two elements, *guards* which are predicates to describe the conditions need to hold for the occurrence of an event, and *actions* which determines how specific state variables change as a result of the occurrence of the event. A context may extend an existing context and a machine may refer to one or more contexts.

**Event-B Tool** Unlike programming languages, not all formal methods are supported by tools [19]. However, one of the advantages of Event-B is availability of an open source tool which is implemented on top of an extension of the Eclipse platform. This tool, known as RODIN, provides automatic proof and a wide range of plug-ins such as the ProB model checker and Camille Text Editor which were used in our work [1, 6, 13].

**Refinement** In some systems because of the size of states and the number of transitions, it is impossible to represent the whole system as one model. Here, refinement can help us to deal with the complexity in a stepwise manner by working in different abstract levels [19]. There are two forms of refinement; firstly, feature augmentation refinement (also known as horizontal refinement [10]) where in each step new features of the system are introduced. Secondly, data refinement (also known as vertical or structural refinement [10]) which is used to enrich the structure of a model to bring it closer to an implementation structure.

## 2.2 Cruise Control System

In order to have a better understanding of CCS an overview of its external observation is given in Figure 1 (based on [4]). This figure shows the relation between cruise control with the driver and the car. The role of the cruise control that we are interested in is maintaining the car speed as close as possible to the target speed which is set by the driver. The ways a driver and the car can interact with CCS are categorised as:

- Driver can switch CCS on or off, define a target speed for the system and increase or decrease this target speed. Also, the driver can regain the control of the car by using the accelerator, brake, or clutch.
- The car sends the actual speed as a feedback to the cruise control system.
- CCS signals the desired acceleration to the motor.

## 2.3 Overview of Cookbook

As mentioned the focus of the cookbook is on control systems which consist of plants, controllers and in some cases operators who can send commands to the controller (Figure 2[1]). The modelling steps suggested in the cookbook are based on the four-variable model of Parnas [18] and can be divided into

[1]The diagram uses Jackson's Problem Frame notation [15].

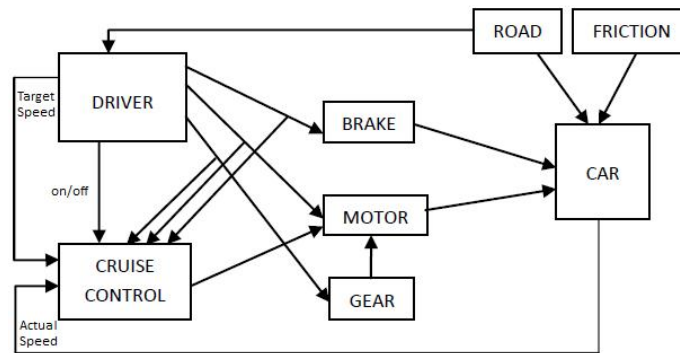



Figure 1: Interaction between driver, cruise control and car (based on [4]).

two major categories. Firstly, identifying the phenomenon in the environment and secondly representing the phenomena used for interaction between the controller and the environment (plant and operator) [8, 9].

Variables shared between a plant and a controller, labelled as 'A' in Figure 2, are known as environment variables [8] and are categorised into *monitored variables* whose values are determined by the plant and *controlled variables* whose values are set by the controller. There are also *environment events* and *control events* which update/modify monitored and controlled variables respectively. Also, in order to add design details the cookbook defines two steps of vertical refinement for introducing internal controller variables. Firstly adding *sensed variables* which defines how a controller receives the value of monitored variable. Secondly, adding *actuation variables* which sets the value of controlled variable [8, 9].

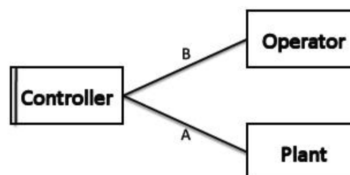



Figure 2: A control problem system [8, 9].

In addition to the variables introduced in four-variable model, the cookbook suggests the identification of the phenomena shared between controller and the operator in the cases where the system involves an operator, labelled as 'B' in Figure 2. These phenomena are represented by *command events* which are the commands from an operator and *commanded variables* whose values are determined by command events and can affect the way other events behave. The refinement step for defining when a controller receives a request from the operator is through introducing *buttons* [8, 9].

## 3 Cruise Control Model

The process of modelling CCS is divided into 3 main sections; First of all, in Section 3.1 M0 as the most abstract machine is defined. In order to do this we start with identifying the monitored, controlled and some of the commanded variables and also their corresponding events as suggested in the cookbook. In Section 3.2 we discuss horizontal refinements which represent the requirement specification of the system by introducing the remainder of the command events and commanded variables. Lastly, in Section 3.3

vertical refinements are defined to introduce the design steps through sensing, actuation and operator requests.

### 3.1 Initial Model (M0)

The process of modelling in the most abstract level starts with identifying the monitored, controlled and commanded variables. As was mentioned the role of the CCS which we are interested in is maintaining the actual speed of the car as close as possible to the target speed by controlling the acceleration of the car.

**Monitored Variable and Environment Event** Based on the role of the CCS, we identified the actual car speed as the monitored variable. This variable is represented by *sa*. Since the value of *sa* cannot be bigger than the maximum car speed, a constant named *n* was defined to represent the maximum car speed ($n \in \mathbb{N}$). Therefore, *sa* can be defined as $sa \in 0..n$. It is also necessary to add the environment event *UpdateActualSpeed* which can update or modify the value of the monitored variable *sa*.

Tip 1: The source which determines the value of a variable is important and a variable will be categorised as monitored, if the environment determines its value.

**Commanded Variables and Command Events** One of the identified commanded variables is *st* which represents the target speed determined by the driver. Based on the requirement, the target speed must be within a specific range. To model this the two *lb* and *ub* constants were defined to demonstrate the minimum and maximum of target speed. This results in having the invariant $stFlg = TRUE \Rightarrow st \in lb..ub$, showing that when *st* is defined it must be within the accepted range. The variable *stFlg* shown in the previous invariants is a boolean variable which turns to TRUE when *st* is set by the driver ($stFlg \in BOOL$). This flag is defined to represent whether the target speed has been defined or not. This is necessary as the CCS will control the car speed only when it has been switched on (*status* = *ON*) and the target speed has been defined (*stFlg* = *TRUE*).

The other commanded variable is named *status*. This variable is an element of a set called STATUS and shows the current status of CCS ($status \in STATUS$). The set STATUS represents the three possible statuses that CCS can be in at one moment of time. These three statuses are ON, OFF and SUSPEND which means CCS does not control the car speed as the result of the driver using one on the pedals. This is discussed in more details in Section 3.2.2.

Tip 2: Variables such as target speed can also be seen as monitored variables. However, we suggest defining them as commanded variables. This is because firstly these variables are internal to the controller and secondly their values are determined by an operator rather than environment.

Corresponding to these commanded variables the following command events were defined:

- *SetTargetSpeed*: when cruise control is on, driver can set *st* to the actual speed of the car.
- *ModifyTargetSpeed*: modifying *st* when CCS is on and *stFlg* is TRUE. Notice that at this level of abstraction we do not separate increment and decrement.
- *StatusOn, Suspend, and SwitchOff*: update *status* variable. In this level of abstraction we consider the relation between different values of *status* variable regardless of what causes changes (Figure 3 (a)). For instance, it is possible to get to ON from status OFF. This is shown in Figure 3 (b) where we added the guard *grd1* that *status* can be OFF and the action is status := ON.

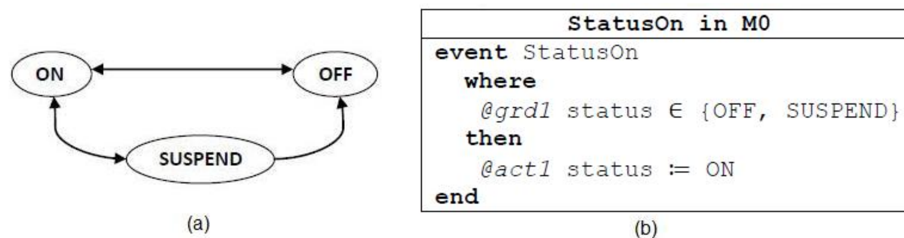



Figure 3: (a) The relation between different statuses of the cruise control system and (b) event StatusOn.

**Controlled Variable and Control Event** Based on the role of the CCS we identified *acceleration* as the only controlled variable ($acceleration \in \mathbb{Z}$). Also, in order to update its value, we added the control event *UpdateAcceleration*. To do this a function named *accFun*, which returns the value of *acceleration* based on actual speed and target speed, is defined. This is a total function since there must be a value defined for every tuple of *st* and *sa* ($accFun \in lb..ub \times 0..n \rightarrow \mathbb{Z}$). Therefore, the action which sets the value of *acceleration* is defined as $acceleration := accFun(st \mapsto sa)$.

Tip 3: The source which determines the value of a controlled variable is important and a variable will be categorised as controlled, if the controller determines its value.

## 3.2 Horizontal Refinements

In order to simplify the refinement steps, firstly all horizontal refinements were done and then vertical refinements. Horizontal refinements resulted in adding machines M1 to M3 to the model where every machine refines its previous machine. In machine M1 the action of increment and decrement of variable *st* are separated. According to the requirement document driver can interact with CCS through the pedals. The pedals are introduced to the model in machine M2. In the final step, the horizontal refinement gear is added.

Tip 4: Separation of horizontal and vertical refinement simplifies the process of modelling.

### 3.2.1 First Refinement (M1)

In this step of refinement the event *ModifyTargetSpeed* has been separated into two increase and decrease events In order to do this, we add the constant *unit* which defines the value added to/subtracted from *st* to increase/decrease it. It was possible to define these two events in the initial model. However we prefer to keep the initial model as abstract as possible to avoid any complexity.

### 3.2.2 Second Refinement (M2)

In this machine pedals which are a way for a driver to interact with CCS [4] are introduced to the model. Notice that all the variables introduced in this section are commanded variables and in the chain of horizontal refinements we only deal with commanded variables and command events.

Based on the requirement, pressing the accelerator, when CCS controls the car speed, will suspend the system temporarily and when the driver stops using this pedal, the CCS will regain the control of the car speed. However, using either brake or clutch causes permanent suspension and this suspension can be resumed by the driver. To distinguish the two types of permanent and temporary suspension, we defined a variable which shows sub-states of the super status SUSPEND. This variable, named *permanentSusp* is of type of boolean and is set to TRUE when CCS is permanently suspended. The other approach would be to introduce a new set consists of {on, off, permanentSusp, tempSusp} which refines the set STATUS.

However this can make the process of refinement proof slightly more complicated. Also, we defined two boolean variables *brkCltchPdl* and *accelerationPdl* to represent the relevant pedals. These variables are set to TRUE when their relevant pedals are pressed.

In the first modelling attempt we defined four events to represent actions of pressing and releasing of each accelerator pedal and brake/clutch pedals when CCS controls the car speed. In order to update the value of *status*, some of these events must refine one of the events *StatusOn* or *Suspend* from M0. Firstly, the event *PressAcceleratorPdl* refines *Suspend* and causes the value of *status* to change from ON to SUSPEND. Also, event *StopAcceleratorPdl* which represents the release of the accelerator refines *StatusOn* and returns the value of *status* to ON. In order to ensure this event can happen only when CCS is temporarily suspended, a guard showing *permanentSusp* must be FALSE is added to the event.

In a very similar way event *PressBrkCltchPdl* refines event *Suspend* and causes the value of *status* to change from ON or SUSPEND to SUSPEND. Notice that because pressing the brake/clutch is stronger than pressing the accelerator, using the brake or clutch while accelerator is pressed must cause permanent suspension. Finally, event *StopBrkCltchPdl* does not change variable *status* and only updates the variable *brkCltchPdl*. This is because CCS stays suspended when the driver releases brake/clutch. The other event which was defined to model the resume of the CCS from permanent suspension is *Resume*. This event allows the CCS to regain the control of the car speed by changing variable *status* to ON and variable *permanentSusp* to FALSE. Also this event can only happen when CCS is permanently suspended. Therefore, the flag *permanentSusp* must be TRUE in order for this event to be enabled.

#### 3.2.3 Third Refinement (M3)

The last machine of horizontal refinements is M3 where the gear is introduced, because CCS can be switched on only when the vehicle is in second or higher gear. This is modelled by adding variable *gear* whose type is a number from -1 to 5 where -1 represents reverse gear, 0 neutral and 1 to 5 represent first to fifth gear. Also, event *ChangeGear* is introduced to be able to change the value of *gear*.

### 3.3 Vertical Refinements

M3 was the last machine of feature augmentation (horizontal) refinements. The remainder of our model consists of machines M4 to M6 which represent the added design details to M3. In the same way as horizontal refinement every machine refines its previous one. These structural (vertical) refinements are based on the cookbook [8, 9].

The first vertical refinement suggested in the cookbook is to introduce sensors through which the controller receives the value of monitored variable. This is modelled in machine M4 by defining an internal variable which gets the value of *sa* by sensing it through an event. In the same way, CCS has an internal variable which sets the value of controlled variable. This internal variable acts as an actuator for the controlled variable. This design detail is introduced in machine M5. Finally, M6 represents the design of buttons through which the driver sends a request to the CCS. According to the cookbook every button is modelled as a boolean variable and the action of pressing that button is modelled as an event. We will discuss these refinements further in the remainder of this section.

Tip 5: Introduce each of the three steps of vertical refinement in a separate machine in order to have less complicated Event-B machines and consequently less proof obligations in each step.

#### 3.3.1 Forth Refinement (M4)

CCS receives the value of a monitored variable through sensors. The sensed/received value needs to be defined as an internal variable for the CCS in order to distinguish values of a sensed variable and a

monitored variable [8, 9]. We defined *sensedSa* as the sensed variable and an event called *SenseSa* which sets *sensedSa* to the current value of *sa*. Since the two monitored and sensed variables are not always equal, their equality is represented as a boolean flag [9]. This flag is called *sensFlg* and becomes TRUE when event *SenseSa* sets variable *sensedSa* to *sa*.

Tip 6: For every monitored variable, defining one sensed variable and one boolean flag is necessary. The sensed variable is of the same type as its corresponding monitored variable and usually is initialised to the same value as the monitored variable is initialised to.

Based on the cookbook, variable *sa* in the control event *UpdateAcceleration* was substituted with *sensedSa* (Figure 4 (a), @act1) in this refinement. Also, this event can only happen when *sensedSa* is equal to *sa* which is the reason for adding @grd2 in Figure 4 (a). The cookbook also suggests adding the invariant $sensFlg = TRUE \Rightarrow sensedSa = sa$ in order to ensure that when *sensFlg* is TRUE, *sensedSa* represents the value of *sa* [9]. This results in a proof problem in *UpdateActualSpeed*, since it can change the value of *sa* while *sensFlg* is TRUE. Therefore, it is necessary to add the guard *sensFlg = FALSE* to this event. Notice that we assumed in between the CCS sensing the value of monitored variable *sa* and setting the *acceleration*, the monitored variable does not change. This is an engineering simplification which helps us to reduce some of the complexity of the modelling of the system.

```
event UpdateAcceleration
refines
UpdateAcceleration
  where
   @grd1 status = ON
   @grd2 stFlg = TRUE
   @grd3 sensFlg = TRUE
  then
   @act1 acceleration ≔
     accFun(st↦sensedSa)
   @act2 sensFlg ≔ FALSE
end
```
(a)

```
event Update_actAcc

 where
   @grd1 status = ON
   @grd2 stFlg = TRUE
   @grd3 sensFlg = TRUE
  then
   @act1 actAcc ≔
  accFun(st ↦ sensedSa)
   @act2 actFlg ≔ TRUE

end
```
(b)

```
event
ActuatingAcceleration
refines
UpdateAcceleration
  where
   @grd1 actFlg = TRUE
  then
   @act1 acceleration ≔
         actAcc
   @act2 sensFlg ≔ FALSE
   @act3 actFlg ≔ FALSE
end
```
(c)

Figure 4: (a) Event UpdateAcceleration in M4, (b) events Update_actAcc in M5 and (c) event ActuatingAcceleration in M5.

### 3.3.2 Fifth Refinement (M5)

In this machine we discuss that CCS decides on the value of *acceleration* distinctly from actuating it [9]. Based on the cookbook, to do this we need to define an actuation variable and a boolean flag. These are named *actAcc* and *actFlg* respectively. The flag *actFlg* will be set to TRUE when the internal process of determining the value of a controlled variable is finished and this variable can be actuated.

Tip 7: For every controlled variable, defining one actuation variable and one boolean flag is necessary. Also, the actuation variable is of the same type as its corresponding controlled variable and usually is initialised to the same value as the controlled variable initialisation.

According to the cookbook, we also defined two events. Firstly, an internal event called *Update_actAcc*, to set the value of actuation variable *actAcc* and set the flag *actFlg* to TRUE (Figure 4 (b), @act1 and @act2). Secondly, *ActuatingAcceleration* which sets the value of *acceleration* to the value of the internal variable *actAcc* and turns *actFlg* to FALSE (Figure 4 (c), @act1 and @act3). This event refines the control event *UpdateAcceleration*, since it modifies the value of *acceleration*. Note that *ActuatingAcceleration* can happen only when the value of *actAcc* is decided by the controller, therefore the guard *actFlg = TRUE* is added to this event (Figure 4 (c), @grd1).

In addition to these variables and events, based on the cookbook it is necessary to define two invariants. The first invariant represents that in between control decision on the value of *actAcc* and actuation of *acceleration*, we assume *st* and *sensedSa* do not change. The second invariant shows that after actuation of *acceleration* the value of this variable and *actAcc* are equal. Although it is not mentioned in the cookbook, this invariant is only needed when the value of the controlled variable changes depending on its previous value as well as some other variables. Since this is not the case in this model, defining the second invariant is unnecessary.

Tip 8: Invariant actFlg = FALSE ⇒ actAcc = acceleration mentioned in the cookbook is unnecessary if the value of controlled variable is set independent of its previous value.

### 3.3.3 Sixth Refinement (M6)

In this section the operator's command request and CCS's response are distinguished by introducing buttons. According to the cookbook, a boolean variable representing a button, needs to be defined for every command event. Also the action of pressing a button should be introduced through an event which sets the button variable to TRUE [9]. We introduced the followings as buttons: *switchBtn, setBtn, incBtn, decBtn* and *rsmBtn*. Because CCS responds to a request only after the relevant button has been pressed, a guard which requires the relevant button to be TRUE is added to each command event. Also, CCS turns the relevant button back to FALSE when it responds to the request. Notice, there is no button defined for the command events related to pedals, since pedal variables *brkCltchPdl* and *acceleratPdl* in Section 3.2.2 count as buttons. Also, there are cases where CCS cannot respond to a coming request, for instance when CCS is OFF. These cases are not mentioned in the cookbook, but we prefer to model these situations as ignorance of CCS to the button's request.

Tip 9: For every button an event can be defined to represent cases where there should be no response to the pressed button. We add this event by defining its guards as the negation of the conjunction of all the guards in the command event corresponds to the button.

This is the last machine of our model and we have modelled the CCS based on the requirement document and the cookbook. In the remainder of this paper we reflect on the results of this work.

# 4 Results and Limitations

## 4.1 Evaluation of the Cookbook

The cookbook is mainly a guideline on vertical refinement. In addition to vertical refinement guidance, the cookbook suggests identifying monitored, controlled and commanded variables and their corresponding events at the most abstract level of a model. Once the variables are found, identifying events which modify and update them in horizontal refinements becomes straightforward. Also, the focus of the cookbook is mainly on the discrete aspects such as status, pedals and buttons and less on continuous, since many of the complexity of the requirements are related to discrete aspects.

One of the other advantages of using cookbook is that almost all the necessary variables, events and invariants for every step of vertical refinement are described. This can be helpful for the designers with not a lot of knowledge on formal modelling in Event-B. In addition, some proof problems caused by the invariants mentioned in the cookbook can help to identify errors of the model. In our work, the process of modelling machines M4 to M6, which was done based on the cookbook, was reasonably easy. In particular, M4 where the sensor was introduced had the most effortless refinement. However, the cookbook lacks a means of dealing with some issues which can raise during modelling process, such as modelling ignorance of a button, mentioned in Section 3.3.3.

Finally, based on the achieved results we believe the main advantage of following the cookbook is the structure that it gives to the process of modelling and refinement. While deciding on how to organise the refinement steps is known as a source of difficulty in the usage of refinement [3], modelling a control problem domain based on the cookbook can help to identify the required steps of refinement quicker and easier. In the case of this work the steps of vertical refinements in machines M4 to M6 were decided purely based on the cookbook.

### 4.2 Limitations

The limitations of this work can be categorised into two types. Firstly, limitations imposed by formal methods themselves, although we should consider that gained benefits may outweigh these limitations. A detailed discussion is beyond the scope of this paper but as an outline one of the limitations is that models can only cover some aspects of a system's behaviours, because mapping formal model and the real world in limited [12]. The second types of limitations are what we have not considered in the process of modelling. First of all we have not considered fault-tolerance and failure of the hardware and it is assumed that the components do not fail. Also, timing and time constraints are not discussed. It is important to notice that our intention was not to prepare a model which is ready to be implemented. Therefore, such limitations will be considered in future work.

## 5 Related work, Future work and Conclusion

### 5.1 Related Work

One of the other approaches for development of embedded systems is Problem Frames (PFs) developed by Michael Jackson. This approach focuses on the separation of problem (what the system will do) and solution (how it will do it) domain. PFs describe any system engineering problem through the concept of a *machine* which is going to be design by a software application, *problem world* and *requirement* [17]. As part of the Deploy project a cruise control system was modelled using PFs. Here, the concepts of PFs are as followings: Machine is the cruise control software; Problem world is anything that cruise control software interacts with, such as pedals and driver; Requirement is controlling phenomena which otherwise would be controlled by the driver, here controlling the car speed [17].

The other work which is related to the cookbook is SCR (Software Cost Reduction) [14]. SCR is a requirement method for real-time embedded systems which is, in the same way as the cookbook, based on the four-variable model of Parnas [18]. As well as identifying the four variables of the Parnas model, SCR defines the following four constructs [14]: *modes* which represent states of a monitored variable; *terms* which are auxiliary functions defined to make the specification more concise; *conditions* to represent predicates and *events* to show the changes of the values in the model.

### 5.2 Future Work

The model of cruise control system represented in this paper contains the platform, the environment and the software application. Separation of these concepts through decomposition in later steps of design allows us to derive a specification of the control system, since the software and the hardware are being separated. In addition, other aspects of an embedded system are usually analysed and modelled through different techniques to formal methods. In order to ensure that cruise control system and other models of the system such as a model of car engine are consistent, meta-modelling can be used.

### 5.3 Conclusion

This work has achieved its main objectives in evaluation of the cookbook and preparation of the best design model for the cruise control system. We showed how the cookbook can make the process of modelling simpler and how it can help to find modelling errors. Also, the model of cruise control system represented in this paper and the given tips can be used by future users of the cookbook. We believe the outcomes of this work have contributed to the research in refinement-based methods such as Event-B and have the potential of leading to improved patterns and guidelines.

**Acknowledgement :** This work is partly supported by the EU research project ICT 214158 DEPLOY (Industrial deployment of system engineering methods providing high dependability and productivity) www.deploy-project.eu.

## References

[1] Roadmap for rodin platform. `http://wiki.event-b.org/images/Roadmap.pdf`, 2009. cited: 2009 Sep.

[2] Jean-Raymond Abrial. *The B-book: assigning programs to meanings*. Cambridge University Press, 1996.

[3] Jean-Raymond Abrial. Formal methods in industry: achievements, problems, future. In *ICSE*, pages 761–768, 2006.

[4] Jean-Raymond Abrial. Cruise control requirement document. Technical report, Internal report of the Deploy project, 2009.

[5] Jean-Raymond Abrial. Modeling in Event-B. To be published, 2010.

[6] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, and Laurent Voisin. An open extensible tool environment for Event-B. In *ICFEM*, pages 588–605, 2006.

[7] Jean-Raymond Abrial and Stefan Hallerstede. Refinement, decomposition, and instantiation of discrete models: Application to event-b. *Fundam. Inform.*, 77(1-2):1–28, 2007.

[8] Michael Butler. Chapter 8 modelling guidelines for discrete control systems, Deploy deliverable d15, d6.1 advances in methods public document. `http://www.deploy-project.eu/pdf/D15-D6.1-Advances-in-Methodological-WPs..pdf`, 2009. cited 2009 7th July.

[9] Michael Butler. Towards a cookbook for modelling and refinement of control problems. in Working Paper. ECS, University of Southampton, 2009.

[10] Kriangsak Damchoom and Michael Butler. Applying event and machine decomposition to a flash-based filestore in event-b. In *SBMF*, pages 134–152, 2009.

[11] Stephen Edwards, Luciano Lavagno, Edward A. Lee, and Alberto Sangiovanni-Vincentelli. Design of embedded systems: Formal models, validation, and synthesis. In *Proc. of the IEEE*, pages 366–390, 1997.

[12] Anthony Hall. Seven myths of formal methods. *IEEE Software*, 7(5):11–19, 1990.

[13] Stefan Hallerstede. Justifications for the Event-B modelling notation. *Lecture Notes in Computer Science*, 4355:49–63, 2006.

[14] Constance L. Heitmeyer, Ralph D. Jeffords, and Bruce G. Labaw. Automated consistency checking of requirements specifications. *ACM Trans. Softw. Eng. Methodol.*, 5(3):231–261, 1996.

[15] Michael Jackson. *Problem Frames: Analyzing and Structuring Software Development Problems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[16] Xin B. Li and Feng X. Zhao. Formal development of a washing machine controller model based on formal design patterns. *WTOS*, 7(12):1463–1472, 2008.

[17] Felix Lösch. Chapter 6 problem frames, Deploy deliverable d15, d6.1 advances in methods public document. `http://www.deploy-project.eu/pdf/D15-D6.1-Advances-in-Methodological-WPs..pdf`, 2009. cited 2009 3rd Aug.

[18] David Lorge Parnas and Jan Madey. Functional documents for computer systems. *Sci. Comput. Program.*, 25(1):41–61, 1995.

[19] Jeannette M. Wing. A specifier's introduction to formal methods. *IEEE Computer*, 23(9):8–24, 1990.

# Formal Verification of Large Software Systems

Xiang Yin
University of Virginia
Charlottesville, VA, USA
xyin@cs.virginia.edu

John Knight
University of Virginia
Charlottesville, VA, USA
knight@cs.virginia.edu

## Abstract

We introduce a scalable proof structure to facilitate formal verification of large software systems. In our approach, we mechanically synthesize an abstract specification from the software implementation, match its static operational structure to that of the original specification, and organize the proof as the conjunction of a series of lemmas about the specification structure. By setting up a different lemma for each distinct element and proving each lemma independently, we obtain the important benefit that the proof scales easily for large systems. We present details of the approach and an illustration of its application on a challenge problem from the security domain.

## 1 Introduction

Formal verification of software continues to be a desirable goal, although such verification remains the exception rather than the rule. By formal verification, we mean verification of the functionality of the software in the sense of Floyd and Hoare [7]. Many challenges arise in formal verification [10], and these challenges are compounded by the complexity that arises with the increasing size of the software of interest.

In this paper, we present an approach to the formal verification of large software systems. The approach operates within a previously developed formal verification framework named Echo [13, 14, 15]. Our goal with the approach to proof was to develop a technique that would be: (a) *relevant*, i.e., applicable to programs that could benefit from formal verification; (b) *scalable*; i.e., be applicable to programs larger than those currently considered suitable for formal verification; (c) *accessible*, i.e., could be used by engineers who are competent but not necessarily expert in using formal methods; and (d) *efficient*, i.e, could be completed in an amount of time that would be acceptable in modern development environments.

We do not define "large" in a quantitative way. Rather, we claim that the approach we have developed can be applied successfully to software systems that are larger than those successfully verified in previous work, and we illustrate the approach in a case study of a system that is several thousand source lines long.

## 2 The Echo Approach

Details of Echo are available elsewhere [14, 15], and we present here only a brief summary. Echo verification is based on a process called *reverse synthesis* in which a high-level, abstract specification (referred to as the *extracted* specification) is synthesized mechanically from a combination of the software source code and a low-level, detailed specification of the software. Reverse synthesis includes verification refactoring in which semantics-preserving transformations are applied to the software to facilitate verification.

Formal verification in Echo involves two proofs: (1) the *implementation* proof, a proof that the source code implements the low-level specification correctly; and (2) the *implication* proof, a proof that the extracted specification implies the original system specification from which the software was built.

The basic Echo approach imposes no restrictions on how software is built except that development has

to start with a formal system specification, and developers have to create the low-level specification documenting the source code. Our current instantiation of Echo uses: (1) PVS [12] to document the system specification and the extracted specification; (2) the SPARK subset of Ada [3] for the source program; and (3) the SPARK Ada annotation language to document the low-level specification. The implementation proof is discharged using the SPARK Ada tools. The implication proof is the focus of this paper, and that proof is facilitated greatly by the implementation proof; most loop invariants, for example, are not present.

## 3 Structural Matching Hypothesis

We refer to our approach to formal verification of large software systems as *proof by parts*. The heart of proof by parts is the *structural matching hypothesis*. We hypothesize that many systems of interest have the property that the high-level structure of a specification is retained, at least partially, in the implementation. Ideally, a specification should be as free as possible of implementation detail. However, the more precise a specification becomes, the more design information it tends to include, especially structural design information. While an implementation need not mimic the specification structure, in practice an implementation will often be similar in structure to the specification from which it was built because: (a) repeating the structural design effort is a waste of resources; and (b) the implementation is more maintainable if it reflects the structure of the specification.

The hypothesis tends to hold for model-based specifications that specify desired system operations using pre- and post-conditions on a defined state. The operations reflect what the customer wants and the implementation structure would mostly retain those operations explicitly.

## 4 Proof Support

**Proof Structure.** The proof of interest is the implication proof. Given implementation $I$ and specification $S$, this proof has to establish the implication $I \Rightarrow S$. Specifically, we prove the implication weakens the precondition and decreases non-determinism from the specification:

$$(pre(S) \Rightarrow pre(I)\ ) \wedge\ (post(I) \Rightarrow post(S))$$

In order to establish this proof for large software systems, the proof structure involved must be scalable. In proof by parts, we rely upon the structural matching hypothesis, and we match the static operational structure of the extracted specification created by reverse synthesis to the original specification. We then organize the proof as a series of lemmas about the specification structure, i.e., proof by parts.

Each lemma is set up for declarative properties over a single distinctive element, e.g. type or operation, and is proved independently. The conjunction of all the lemmas then forms the whole implication theorem that the extracted specification implies the original specification. Since each lemma is over a different element of the system and is proved independently without reference to the whole system, proof by parts is expected to scale for large software systems.

Clearly, the construction of a proof in this way is only possible for a system for which the structural matching hypothesis holds for the *entire* implementation. Inevitably, this will rarely if ever be the case for real software systems. For systems in which the two structures do not match, we employ a technique within Echo referred to as *verification refactoring*, to restructure the implementation to match the specification structure.

**Verification Refactoring.** There are many reasons why the structural matching hypothesis will only apply partially to a particular system, e.g., optimizations introduced by programmers that complicate the program structure. Verification refactoring consists of selecting transformations that can be applied to the source program, proving they are semantics preserving, and applying them to the program before specification extraction. Details of verification refactoring are discussed in an earlier paper [15].

Echo's verification refactoring mechanism provides a set of semantics-preserving transformations that

can be applied to an implementation to facilitate verification in various ways. Several transformations help to reduce the length and complexity of the proof obligations to facilitate the implementation proof. Although the implementation proof is completed mostly by tools (the SPARK Ada tools in our current Echo instantiation), we have found that verification refactoring can be extremely beneficial for the implementation proof, in the limit making proof possible where the tools failed on the original program.

A second set of transformations restructures a program to align the structure of the extracted specification with the structure of the original specification, i.e., to make the matching hypothesis apply to more of the program.New transformations might be developed to accommodate a particular program. Even so, verification refactoring might not provide a match that will enable the implication proof in which case a different verification approach will be needed.

**Matching Metric.** We have defined a matching metric that summarizes the similarity of the structures of the original and the extracted specifications and thereby indicates the feasibility of proof by parts. The match ratio is defined as the percentage of key structural elements — data types, system states, tables, operations — in the original specification that have direct counterparts in the extracted specification. The match ratio does not necessarily imply the final difficulty of the proof, but the match ratio does provide an initial impression of the likelihood of successfully establishing the proof. In our Echo prototype, the metric is evaluated by visual inspection although significant tool support would be simple to implement.

Establishing the match ratio is fairly straightforward in many cases. Some of the matching can be determined from the symbols used, because the names used in the original specification are often carried through to the implementation and hence to the extracted specification.

## 5 Approach to Proof

The property that needs to be shown in the implication proof is implication not equivalence, hence the name. By showing that the extracted specification implies the original specification, but not the converse, we allow the original specification to be nondeterministic and allow more behaviors in the original specification than the implementation. The basic definition of implication we use for this is that set out by Liskov and Wing known as behavioral subtyping [11].

The way in which the extracted specification is created influences the difficulty of the later proof. In the case where the implementation retains the structural information from the original specification, a simple way to begin proof by parts is to also retain the structure by directly translating elements of the implementation language, such as packages, data types, state/operation representations, pre-conditions, post-conditions, and invariants, into corresponding elements in the specification language. For each pair of matching elements, we establish an implication lemma that the element in the extracted specification implies the matching element from the original specification. The final proof is organized as the conjunction of a series of such lemmas. There are three types of implication lemmas that we discuss in the next three subsections.

### 5.1 Type lemmas

For each pair of matching types, we define a retrieval function from the extracted type to the original type. When trying to prove the relation between each pair, two possibilities arise:

**Surjective Retrieval Function.** If the retrieval function can be proved to be surjective, the extracted type is a refinement of the original type, i.e., all properties contained in the types in the original specification are preserved. If the retrieval function can be proved to be a bijection, the two types are equivalent.

**Non-surjective Retrieval Function.** If the retrieval function is not surjective, then either: (a) there is a defect if the two types are intended to be matched; (b) certain values in the original specification can never arise; or (c) a design decision has been made to further limit the type in the implementation, i.e.,

to make the post-condition stronger. Upon review, if the user does not confirm that there is a defect or does not further refine the specification, we postpone the proof by transforming the types into subtype predicates on the same base type (e.g. integer). These extra predicates are added as conjuncts in function pre-conditions or post-conditions depending on where they appear, and they are checked when the later lemmas regarding those functions are established.

### 5.2 State lemmas

State is the set of system variables used to monitor or control the system. State is defined over types, thus type lemmas can be used to facilitate proofs of state lemmas. As with type lemmas, we set up a retrieval function from the extracted state to the original state. For each pair, we prove the following two lemmas:

**State Match.** As with the type lemmas, we prove that the retrieval function is surjective to show refinement (or equivalence in the bijection case). If it cannot be proved, indicating certain values of the original state cannot be expressed by the extracted state, we again present it for user review. It is either a defect or, by definition, certain values of original state cannot arise.

**State Initialization.** For states that require initialization, the extracted specification will contain an initialization function. We prove that whenever a state is initialized in the extracted specification, the corresponding retrieved original state also satisfies the initialization constraints in the original specification.

### 5.3 Operation lemmas

System operations are usually defined as functions or procedures over the system state. When matching pairs of operations in the extracted specification and the original specification, we set up an implication lemma for each pair. The operation extracted from the implementation should have weaker pre-condition and stronger post-condition than the operation defined in the original specification. Specifically, we prove:

**Applicability.** The extracted operation has a weaker pre-condition than the original operation. For any state `st` upon which the operation operates, given `R` as the retrieval function for `st`, we prove:

```
FORALL st: Pre_org(R(st)) => Pre_ext(st)
```

**Correctness.** The extracted operation has a stronger post-condition than the original operation if applicable. Given any `st1` and `st2` as input and output for an operation `f`, `R` as the retrieval function for state, we prove:

```
FORALL st1, st2 | st2 = f(st1):
   Post_ext(st2) AND pre_org(R(st1)) => post_org(R(st2))
```

By reasoning over predicates such as the pre-conditions and post-conditions in the low-level specification, we avoid implementation details as much as possible when proving these implication lemmas.

### 5.4 Implication theorem

The conjunction of all the lemmas forms the implication theorem. All the resulting proof obligations need to be discharged automatically or interactively in a mechanical proof system. Since the extracted specification is expected to have a structure similar to the original specification, the proof usually does not require a great deal of human effort. Also, by setting up the lemmas operation by operation rather than property by property and proving each operation independently, the proof structure easily scales.

## 6 Proof Process

Our process for applying the proof in practice is shown in Figure 1. In most situations, we choose to proceed with verification refactoring first to increase the match-ratio metric until it becomes stabilized through transformations. There are other types of refactoring and corresponding metrics we evaluate to

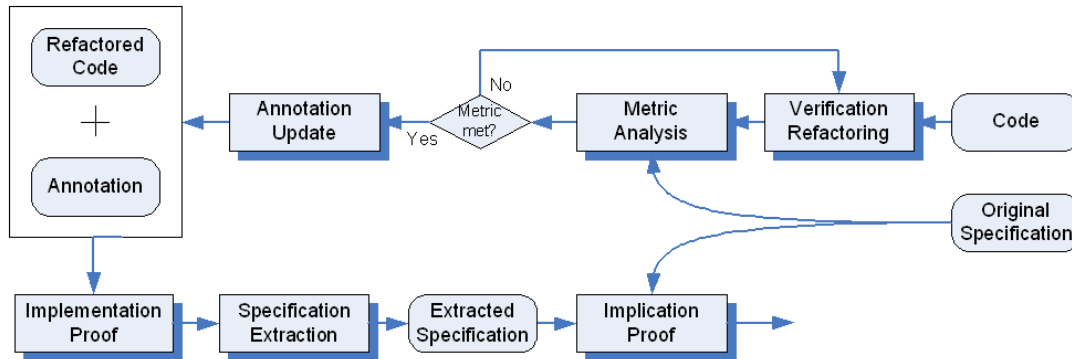



**Figure. 1.** The proof process.

facilitate the proof process, e.g., to reduce the size of the proof obligations generated. Details of this verification refactoring and metric analysis process and the benefits it brings to our proof have been discussed in our earlier paper [15].

After applying verification refactoring, we have a version of the implementation from which a specification can be extracted that shows structural similarity to the original specification. We then: (a) update and complete the low-level specification (documented as annotations) that might have become erroneous during the refactoring process (e.g. by splitting a procedure); and (b) prove that the code conforms to the low-level specification, i.e., create the implementation proof. This technology is well established, and we do not discuss it further here.

From the refactored code and the associated low-level specification, we extract a high-level specification, i.e., the extracted specification, using custom tools. We then establish the three types of implication lemma and the implication theorem following the approach discussed in section 5. Finally, we prove that the extracted specification implies the original specification.

### 6.1 Specification Extraction

In this section we demonstrate how we synthesize the extracted specification from the implementation.

**Extraction from annotation.** For functions that are annotated with proved pre- and post-condition annotations, the annotations provide a level of abstraction. This is helpful for programs that contain a lot of computation. For instance, when one specifies a function, the property that one cares about would be correctness of the output. The actual algorithm used is not important. If the function is annotated and the annotation is proved using code-level tools, we extract the specification from the annotations and leave out the unrelated implementation details. The proof of the annotations by the code-level tools can be introduced as a lemma that is proved outside the specification proof system. An example code fragment that is written in SPARK Ada and the extracted specification in PVS in which the details of the procedure body are omitted and an additional lemma introduced is shown in Figure 2. The additional lemma indicates that the post-condition will be met if the pre-condition is met. The lemma is marked as proved outside and can be used directly in subsequent proofs.

**Direct extraction from code.** The expressive power of the SPARK Ada annotations with which the low-level specification is documented is limited. Certain properties either: (a) cannot be expressed by the annotation language; (b) are expressible but not in a straightforward way; or (c) are expressible but are not helpful in abstracting out implementation details. In such situations we extract the specification directly from the source code. An example of both code and extracted specification fragments, again in SPARK Ada and PVS, is shown in Figure 3.

**Skeleton extraction.** A lightweight version of specification extraction is used to facilitate the metric analysis. When verification refactoring is used, we extract a skeleton specification from the transformed code.

```
type state is
   record
      a: Integer;
      b: Integer;
   end record;
procedure foo(st: in out state)
--# derives st from st;
--# pre   st.a = 0;
--# post  st = st~[a => 1];
is
begin
  -- procedure body
  …
end foo;
```

```
state: TYPE = [# a: int, b: int #]
foo_pre(st: state): bool = (st`a = 0)
foo_post(st_, st: state): bool =
   (st = st_ WITH [`a := 1])
foo(st: state): state
foo: LEMMA FORALL (st: state):
   foo_pre(st) => foo_post(st, foo(st))
```

**Figure. 2.** SPARK Ada code fragment and associated PVS fragment derived from annotations.

We refer to this specification as a skeleton because it is obtained using solely the type and function declarations and contains none of the detail from the annotations or the function definitions. The skeleton specification, however, does reflect the structure of the extracted specification. We can then compare the structure of the skeleton extracted specification with that of the original specification and evaluate the match-ratio metric to determine whether further refactoring is needed.

**Component Reuse & Model Synthesis.** Software reuse of both specification and code components is a common and growing practice. If a source-code component from a library is reused in a system to be verified and that component has a suitable formal specification, then that specification can be included easily in the extracted specification.

In some cases, specification extraction may fail for part of a system because the difference in abstraction used there between the high-level specification and the implementation is too large. In such circumstances, we use a process called model synthesis in which the human creates a high-level model of the portion of the implementation causing the difficulty. The model is verified by conventional means and then included in the extracted specification.

### 6.2 Implication Proof

The final step to complete the verification argument is the implication proof that the extracted specification implies the original specification. The implication argument is established by matching the structures and components of these two specifications and setting up and proving the implication theorem as discussed in section 5.

### 6.3 Justification of Correctness

The verification argument in Echo is as follows: (a) the implementation proof establishes that the code implements the low-level specification (the annotations); (b) the transformations involved in verification refactoring preserve semantics; (c) the specification extraction is automated or mechanically checked; (d) the implication theorem is proved; and (e) the combination of (a) through (d) provides a complete argument that the implementation behaves according to its specification.

Any defect in the code that could cause the implementation not to behave according to the specifica-

```
procedure foo(st: in out state)
is
begin
   foo1(st);
   st.a = 1;
   foo2(st);
end foo;
```

```
foo(st: state): state =
   LET st1 = foo1(st) IN
   LET st2 = st1 WITH [`a := 1] IN
   LET st3 = foo2(st2) IN
      st3
```

**Figure. 3.** SPARK Ada code fragment and associated PVS fragment derived from source code.

tion will be exposed in either the implementation proof or the implication proof. Any inconsistency between the code and the annotations will be detected by the code-level tools in the implementation proof. An inconsistency could arise because of a defect in either or both. If both are defective but the annotations match the defective code, it will not be detected by the implementation proof. However the annotations will not be consistent with the high-level specification in this case and so will be caught in the implication proof.

## 7 The Tokeneer Case Study

In order to evaluate our proof structure approach, we sought a non-trivial application that was built by others, was several thousand lines long, and was in a domain requiring high assurance. The target system we chose is part of a hypothetical system called *Tokeneer* that was defined by the National Security Agency (NSA) as a challenge problem for security researchers [4]. Our interest was in functional verification of the Tokeneer software, and so we applied the Echo proof approach to the core functions of Tokeneer.

### 7.1. Tokeneer Project Description

Tokeneer is a large system that provides protection to secure information held on a network of workstations situated in a physically secure enclave. The system has many components including an enrolment station, an authorization station, security resources such as certificate and authentication authorities, an administration console, and an ID station. We used the core part of the Tokeneer ID Station (TIS) in this research. TIS is a stand-alone entity responsible for performing biometric verification of the user and controlling human access to the enclave. To perform this task, the TIS asks the individual desiring access to the enclave to present an electronic token in the form of a card to a reader. The TIS then examines the biometric information contained in the user's token and a fingerprint scan read from the user. If a successful identification is made and the user has sufficient clearance, the TIS writes an authorization onto the user's token and releases the lock on the enclave door to allow the user access to the enclave.

Much of the complexity of the TIS derives from dealing with all eventualities. A wide variety of failures are possible that must be handled properly. Since Tokeneer is a security-critical system, crucial security properties such as unlocking only with a valid token and within an allowed time, and keeping consistent audit records need to be assured with high levels of confidence.

A fairly complete,high quality implementation of major parts of the TIS has been built by Praxis High Integrity Systems. The Praxis implementation includes a requirement analysis document, a formal specification written in Z (117 pages), a detailed design, a source program written in SPARK Ada (9939 lines of non-comment, non-annotation code), and associated proofs. Since our tools operate with PVS, we translated the Z specification of the TIS into PVS. The final specification in PVS is 2336 lines long.

### 7.2. Echo Proof of Tokeneer

The Praxis implementation of Tokeneer was developed using Correctness by Construction [6] with the goal of demonstrating rigorous and cost effective development. High-level security properties were established by documenting the properties using SPARK Ada annotations, including them in the code, and then proving them using the SPARK Ada tools. Our proof is of the functionality of the implementation as defined by the original, high-level specification. Given our proof of functionality, high-level security properties can be established by stating the properties as theorems and proving them against the high-level specification.

Turning now to the proof itself, upon review, we found that the TIS source program structure resembled the specification structure very closely, i.e., the structural matching hypothesis held. Almost all states and operations in the specification have direct counterparts in the source program, e.g., the `UnlockDoor` operation defined for system internal operations in the specification:

```
UnlockDoor(dla_i, dla_o: DoorLatchAlarm, c: Config): bool =
   dla_o`latchTimeout = dla_i`currentTime + c`latchUnlockDuration AND
   dla_o`alarmTimeout = dla_i`currentTime
                           + c`latchUnlockDuration + c`alarmSilentDuration AND
   dla_o`currentTime = dla_i`currentTime AND
   dla_o`currentDoor = dla_i`currentDoor
```

is implemented by a corresponding procedure with the same name in the Door package in the source code:

```
procedure UnlockDoor is
   LatchTimeout : Clock.TimeT;
begin
   LatchTimeout := Clock.AddDuration(
                       TheTime     => Clock.TheCurrentTime,
                       TheDuration => ConfigData.TheLatchUnlockDuration);
   Latch.SetTimeout(Time => LatchTimeout);
   AlarmTimeout := Clock.AddDuration(
                       TheTime     => LatchTimeout,
                       TheDuration => ConfigData.TheAlarmSilentDuration);
   Latch.UpdateInternalLatch;
   UpdateDoorAlarm;
end UnlockDoor;
```

We did not include the interface functions in our verification, and after removing them and performing a skeleton extraction, we found the match ratio to be 74.7%. Almost all states and operations in the specification have direct counterparts in the source program. The match ratio is not 100% (or at least close to it) because refinements carried out during the development added several operations that were not defined in the specification. We concluded that the structures of the specification and the source program were sufficiently similar that we could extract the necessary specification effectively and easily without performing verification refactoring. The final extracted PVS specification is 5622 lines long. As an example, the extracted specification using direct extraction from code for the above `UnlockDoor` procedure is:

```
UnlockDoor(st: State): State =
   LET LatchTimeout = Clock.AddDuration(TheCurrentTime(st),
                                  TheLatchUnlockDuration(st)) IN
   LET st1 = SetTimeout(LatchTimeout, st) IN
   LET st2 = st1 WITH [`AlarmTimeout := Clock.AddDuration(LatchTimeout,
                                  TheAlarmSilentDuration(st))] IN
   LET st3 = UpdateInternalLatch(st2) IN
   UpdateDoorAlarm(st3)
```

Following extraction of the specification, we performed both the implementation proof and the implication proof. For the implementation proof, we used the SPARK Ada toolset to prove functional behaviors of those subprograms that had been documented with pre- and post-condition annotations. The SPARK Examiner generates verification conditions (VCs) that must be proved true to demonstrate that the code does indeed meet its specified post-conditions, within the context of its pre-conditions. The Examiner also generates VCs that must be satisfied to ensure freedom from run-time exceptions. Altogether there were over 2600 VCs generated of which 95% were automatically discharged by the toolset itself. The remaining 5% required human intervention, and were covered in the documents from Praxis' proof.

The implication proof was established by matching the components of the extracted specification with those of the original specification. Identifying the matching in the case study was straightforward, and in most situations could be suggested automatically by pairing up types and functions with the same name as showed by the above example. For each matching pair, we created an implication lemma and altogether there were just over 300 such lemmas. Typechecking of the implication theorem resulted in 250 Type Correctness Conditions (TCCs) in the PVS theorem prover, a majority of which were discharged automatically by the theorem prover itself. In 90% of the cases, the PVS theorem prover could not prove the implication lemmas completely automatically. However, the human guidance required was straightforward due to the

tight correlation between the original specification and source code, typically including expansion of function definitions, introduction of type predicates, or application of extensionality. Each lemma that was not automatically discharged was interactively proved in the PVS theorem prover by a single person with moderate knowledge about PVS, and thus the implication theorem was discharged. The total typechecking and proof scripts running time in PVS is less than 30 minutes.

In section 1 we listed four goals for our proof approach. Drawing firm conclusions about our goals based on a single case study is impossible, but from our experience of the Tokeneer proof (and also earlier, smaller proofs [15]), we conclude the following general indications about our goals:

**Relevant.** The approach is relevant in so far as Tokeneer is typical of security-critical systems. Tokeneer was established by the NSA as a challenge problem because it is "typical" of the systems they require. The approach can be applied in other domains provided any necessary domain-specific axioms are available in PVS. They might have to be developed by domain experts. We anticipate that verification refactoring will expand the set of programs to which the technique can be applied.

**Scalable.** The approach scales to programs of several thousand lines provided the structural matching hypothesis holds. There is no obvious limit on the size of programs that could successfully be proved.

**Accessible.** The approach is mostly automated and relatively easy to understand, and should be accessible to practicing engineers who are generally familiar with formal methods. The Tokeneer proof was completed by a single individual with good knowledge of the technology in use but no special training.

**Efficient.** Detailed measurements could not be made of the resources used for the Tokeneer proof, but we observe that the resources required were well within the scope of a typical software development.

A valuable benefit of proof by parts is that location of implementation defects becomes easier. In general, a defect is usually located inside the component for which the proof fails for the corresponding lemma. During the proof of Tokeneer, we found several mismatches between the source program and the specification, but later found that they were changes documented by Praxis.

## 8 Related Work

Traditional Floyd-Hoare verification [7] requires generation and proof of significant amount of detailed lemmas and theorems. It is very hard to automate and requires significant time and skill to complete. Refinement based proof like the B method [1] intertwine code production and verification. Using the B method requires a B specification and then enforces a lock-step code production approach on developers.

Annotations and verification condition generation, such as that employed by the SPARK Ada toolset, is used in practice. However, the annotations used by SPARK Ada (and other similar techniques) are generally too close to the abstraction level of the program to encode higher-level specification properties. Thus, we use verification condition generation as an intermediate step in our approach.

Results in reverse engineering include retrieval of high-level specifications from an implementation by semantics-preserving transformations [5, 16]. These approaches are similar to our extraction and refactoring techniques, but the goal is to facilitate analysis of poorly-quality code, not to aid verification. Our approach captures the properties relevant to verification while still abstracting implementation details.

Andronick et al. developed an approach to verification of a smart card embedded operating system [2]. They proved a C source program against supplementary annotations and generated a high-level formal model of the annotated C program that was used to verify certain global security properties.

Heitmeyer et al. developed a similar approach to ours for verifying a system's high-level security properties [8]. Our approach is focused on general functionality rather than security properties.

Klein et al. demonstrated that full functional formal verification is practical for large systems by verifying the seL4 microkernel from an abstract specification down to its 8700 lines C implementation [9]. Proof by parts is more widely applicable and does not impose restrictions on the development process.

## 9 Conclusion

We have described proof by parts, an approach to the formal verification of large software systems, and demonstrated its use on a program that is several thousand lines long. The approach is largely automated and does not impose any significant restrictions on the original software development.

Proof by parts depends upon the specification and implementation sharing a common high-level structure. We hypothesize that many systems of interest have this property. Where the structures differ, we refactor the implementation to align the two structures if possible. If refactoring fails to align the structures, then proof by parts is infeasible.

We expect proof by parts to scale to programs larger than the one used in our case study with the resources required scaling roughly linearly with program size. For software for which our hypothesis holds, we are not aware of a limit on the size of programs that can be verified using our approach.

## 10 Acknowledgements

We are grateful to the NSA and to Praxis High Integrity Systems for making the Tokeneer artifacts available. Funded in part by NASA grants NAG-1-02103 & NAG-1-2290, and NSF grant CCR-0205447.

## References

[1] Abrial, J.R., *The B-Book: Assigning Programs to Meanings*, Cambridge University Press, 1996.

[2] Andronick, J., B. Chetali, and C. Paulin-Mohring, "Formal Verification of Security Properties of Smart Card Embedded Source Code", In: Fitzgerald, J., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 302-317. Springer-Verlag, 2005.

[3] Barnes, J., *High Integrity Software: The SPARK Approach to Safety and Security*, Addison-Wesley, 2003.

[4] Barnes, J., R. Chapman, R. Johnson, J. Widmaier, D. Cooper, and B. Everett, "Engineering the Tokeneer Enclave Protection Software", In: Proceedings of IEEE International Symposium on Secure Software Engineering, 2006.

[5] Chung, B. and G.C. Gannod, "Abstraction of Formal Specifications from Program Code", *IEEE 3rd Int. Conference on Tools for Artificial Intelligence*, 1991, pp. 125-128.

[6] Croxford, M. and R. Chapman, "Correctness by construction: A manifesto for high-integrity software", CrossTalk, *The Journal of Defense Soft. Engr*, 2005, pp. 5-8.

[7] Floyd, R., "Assigning meanings to programs", Schwartz, J.T. (ed.), Mathematical Aspects of Computer Science, *Proceedings of Symposia in Applied Mathematics* 19 (American Mathematical Society), Providence, pp.19-32, 1967.

[8] Heitmeyer, C., M. Archer, E. Leonard, and J. McLean, "Applying Formal Methods to a Certifiably Secure Software System", *IEEE Transaction on Software Engineering*, Vol.34, No.1, 2008.

[9] Klein, G., K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: formal verification of an OS kernel", Proceedings of the *ACM SIGOPS 22nd symposium on Operating systems principles*, Big Sky, Montana, USA, October, 2009.

[10] Knight, J.C., C.L. DeJong, M.S. Gibble, and L.G. Nakano, "Why Are Formal Methods Not Used More Widely?", *Fourth NASA Formal Methods Workshop*, Hampton, VA, 1997

[11] Liskov, B. and J. Wing, "A Behavioral Notion of Subtyping", *ACM Transactions on Programming Languages and Systems*, 16(6):1811--1841, 1994.

[12] Owre, S., N. Shankar, and J. Rushby, "PVS: A Prototype Verification System", Proceedings of *CADE 11*, Saratoga Springs, NY, 1992.

[13] Strunk, E., X. Yin, and J. Knight, "Echo: A Practical Approach to Formal Verification", Proceedings of FMICS05, *10th International Workshop on Formal Methods for Industrial Critical Systems*, Lisbon, Portugal, 2005.

[14] Yin, X., J. Knight, E. Nguyen, and W. Weimer, "Formal Verification By Reverse Synthesis", Proceedings of the *27th SAFECOMP, Newcastle*, UK, September 2008.

[15] Yin, X., J. Knight, and W. Weimer, "Exploiting Refactoring in Formal Verification", *DSN 2009: The International Symposium on Dependable Systems and Networks*, Lisbon, Portugal (June 2009).

[16] Ward, M., "Reverse Engineering through Formal Transformation", *The Computer Journal*, 37(9):795-813, 1994.

# Symbolic Computation of Strongly Connected Components Using Saturation *

Yang Zhao Gianfranco Ciardo
Department of Computer Science and Engineering, University of California, Riverside
zhaoy@cs.ucr.edu ciardo@cs.ucr.edu

**Abstract**

Finding strongly connected components (SCCs) in the state-space of discrete-state models is a critical task in formal verification of LTL and fair CTL properties, but the potentially huge number of reachable states and SCCs constitutes a formidable challenge. This paper is concerned with computing the sets of states in SCCs or terminal SCCs of asynchronous systems. Because of its advantages in many applications, we employ *saturation* on two previously proposed approaches: the Xie-Beerel algorithm and transitive closure. First, saturation speeds up state-space exploration when computing each SCC in the Xie-Beerel algorithm. Then, our main contribution is a novel algorithm to compute the transitive closure using saturation. Experimental results indicate that our improved algorithms achieve a clear speedup over previous algorithms in some cases. With the help of the new transitive closure computation algorithm, up to $10^{150}$ SCCs can be explored within a few seconds.

## 1 Introduction

Finding strongly connected components (SCCs) is a basic problem in graph theory. For discrete-state models, some interesting properties, such as LTL [8] and fair CTL, are related with the existence of SCCs in the state transition graph, and this is also the central problem in the language emptiness check for $\omega$-automata. For large discrete-state models (e.g., $10^{20}$ states), it is impractical to find SCCs using traditional depth-first search, motivating the study of symbolic computation of SCCs. In this paper, the objective is to build the set of states in non-trivial SCCs.

The structure of SCCs in a graph can be captured by its *SCC quotient graph*, obtained by collapsing each SCC into a single node. This resulting graph is acyclic, and thus defines a partial order on the SCCs. *Terminal SCCs* are leaf nodes in the SCC quotient graph. In the context of large scale Markov chain analysis, an interesting problem is to partition the state space into *recurrent* states, which belong to terminal SCCs, and *transient* states, which are not recurrent.

The main difficulties in SCC computation are: having to explore huge state spaces and, potentially, having to deal with a large number of (terminal) SCCs. The first problem is the primary obstacle to formal verification due to the obvious limitation of computational resources. Traditional BDD-based approaches employ *image* and *preimage* computations on state-space exploration and, while quite successful in fully synchronous systems, they do not work as well for asynchronous systems. The second problem constitutes a bottleneck for one class of previous work, which enumerates SCCs one by one. Section 2.3 discusses this problem in more detail.

This paper addresses the computation of states in SCCs and terminal SCCs. We propose two approaches based on two previous ideas: the *Xie-Beerel algorithm* and *transitive closure*. Saturation, which schedules the firing of events according to their locality, is employed to overcome the complexity of state-space exploration. Pointing to the second difficulty, our efforts are devoted to an algorithm based on the transitive closure, which does not suffer from a huge numbers of SCCs but, as previously proposed, often requires large amounts of runtime and memory. We then propose to use a saturation-based algorithm to compute the transitive closure, enabling it to be a practical method of SCC computation for complex systems. We also present an algorithm for computing recurrent states based on the transitive closure.

*Work supported in part by the National Science Foundation under grant CCF-0848463.

The remainder of this paper is organized as follows. Section 2 introduces the relevant background on data structure we use and the saturation algorithm. Section 3 introduces an improved Xie-Beerel algorithm using saturation. Section 4 introduces our transitive closure computation algorithm using saturation and the corresponding algorithms for SCC and terminal SCC computations. Section 6 compares the performance of our algorithms and that of Lockstep.

# 2 Preliminaries

Consider a discrete-state model $(\mathscr{S}, \mathscr{S}_{init}, \mathscr{E}, \mathscr{N})$ where the potential state space $\mathscr{S}$ is given by the product $\mathscr{S}_L \times \cdots \times \mathscr{S}_1$ of the local state spaces of $L$ submodels, thus each (global) state $\mathbf{i}$ is a tuple $(i_L, \ldots, i_1)$ where $i_k \in \mathscr{S}_k$, for $L \geq k \geq 1$; the set of initial states is $\mathscr{S}_{init} \subseteq \mathscr{S}$; the set of (asynchronous) events is $\mathscr{E}$; the next-state function $\mathscr{N} : \mathscr{S} \to 2^{\mathscr{S}}$ is described in disjunctively partitioned form as $\mathscr{N} = \bigcup_{\alpha \in \mathscr{E}} \mathscr{N}_\alpha$, where $\mathscr{N}_\alpha(\mathbf{i})$ is the set of states that can be reached in one step when $\alpha$ fires in state $\mathbf{i}$. We say that $\alpha$ is *enabled* in state $\mathbf{i}$ if $\mathscr{N}_\alpha(\mathbf{i}) \neq \emptyset$. Correspondingly, $\mathscr{N}^{-1}$ and $\mathscr{N}_\alpha^{-1}$ denote the inverse next-state functions, i.e., $\mathscr{N}_\alpha^{-1}(\mathbf{i})$ is the set of states that can reach $\mathbf{i}$ in one step by firing event $\alpha$.

State-space generation refers to computing the set of reachable states from $\mathscr{S}_{init}$, denoted with $\mathscr{S}_{rch}$. Section 2.2 introduces our state-space generation algorithm called *saturation*, which is executed prior to the SCC computation as a preprocessing step. Consequently, $\mathscr{S}_{rch}$, the sets $\mathscr{S}_k$, and their sizes $n_k$ are assumed known in the following discussion, and we let $\mathscr{S}_k = \{0, ..., n_k - 1\}$, without loss of generality.

## 2.1 Symbolic encoding of discrete-state systems

We employ *multi-way decision diagrams* (MDDs) [7] to encode discrete-state systems. MDDs extend binary decision diagrams (BDDs) by allowing integer-valued variables, thus are suitable for discrete-state models with bounded but non-boolean valued state variables, such as Petri nets [10]. There are two possible terminal nodes, **0** and **1**, for all MDDs. Each MDD has a single root node.

We encode a set of states with an $L$-level *quasi-reduced* MDD. Given a node $a$, its level is denoted with $a.lvl$ where $L \geq a.lvl \geq 0$. $a.lvl = 0$ if $a$ is **0** or **1** and $a.lvl = L$ if it is a root node. If $a$ is nonterminal and $a.lvl = k$, then $a$ has $n_k$ outgoing edges labeled with $\{0, ..., n_k - 1\}$, each of which corresponds to a local state in $\mathscr{S}_k$. The node pointed by the edge labeled with $i_k$ is denoted with $a[i_k]$. If $a[i_k] \neq \mathbf{0}$, it must be a node at level $k-1$. Finally, let $\mathscr{B}(a) \subseteq \mathscr{S}_k \times \cdots \times \mathscr{S}_1$ be the set of paths from node $a$ to **1**.

Turning to the encoding of the next-state functions, most asynchronous systems enjoy *locality*, which can be exploited to obtain a compact symbolic expression. An event $\alpha$ is *independent* of the $k^{\text{th}}$ submodel if its enabling does not depend on $i_k$ and its firing does not change the value of $i_k$. A level $k$ belongs to the *support* set of event $\alpha$, denoted $supp(\alpha)$, if $\alpha$ is not independent of $k$. We define $Top(\alpha)$ to be the highest-numbered level in $supp(\alpha)$, and $\mathscr{E}_k$ to be the set of events $\{\alpha \in \mathscr{E} : Top(\alpha) = k\}$. Also, we let $\mathscr{N}_k$ be the next-state function corresponding to all events in $\mathscr{E}_k$, i.e., $\mathscr{N}_k = \bigcup_{\alpha \in \mathscr{E}_k} \mathscr{N}_\alpha$.

We encode the next-state function using $2L$-level MDDs with level order $L, L', ..., 1, 1'$, where unprimed and primed levels correspond to "from" and "to" states, respectively, and we let $Unprimed(k) = Unprimed(k') = k$. We use the *quasi-identity-fully (QIF)* reduction rule [13] for MDDs encoding next-state functions. For an event $\alpha$ with $Top(\alpha) = k$, $\mathscr{N}_\alpha$ is encoded with a $2k$-level MDD since it does not affect state variables corresponding to nodes on levels $L, \ldots, k+1$; these levels are skipped in this MDD. The advantage of the QIF reduction rule is that the application of $\mathscr{N}_\alpha$ only needs to start at level $Top(\alpha)$, and not at level $L$. We refer interested readers to [13] for more details about this encoding.

## 2.2 State-space generation using saturation

All symbolic approaches to state-space generation use some variant of symbolic image computation. The simplest approach is the breadth-first iteration, directly implementing the definition of the state space

```
mdd Saturate({N_L,...,N_1}, mdd s)
 1 if InCache_Saturate(s,t) then return t;
 2 level k ← s.lvl;
 3 mdd t ← NewNode(k);  mdd r ← N_k;
 4 foreach i ∈ S_k s.t. s[i] ≠ 0 do
     • First saturate all its children
 5   t[i] ← Saturate({N_L,...,N_1}, s[i]);
 6 endfor
 7 repeat     • Get a local fixed point on the root
 8   foreach i,i' ∈ S_k s.t. r[i][i'] ≠ 0 do
 9     mdd u ←
       RelProdSat({N_L,...,N_1}, t[i], r[i][i']);
10     t[i'] ← Or(t[i'], u);
11   endfor
12 until t does not change;
13 t ← UniqueTablePut(t);
14 CacheAdd_Saturate(s,t);
15 return t;
```

```
mdd RelProdSat({N_L,...,N_1}, mdd s, mdd r)
 1 if s = 1 and r = 1 then return 1; endif
 2 if InCache_ConsRelProd(s,r,t) then return t; endif
 3 level k ← s.lvl;  mdd t ← 0;
 4 foreach i,i' ∈ S_k s.t. r[i][i'] ≠ 0 do
 5   mdd u ← RelProdSat({N_L,...,N_1}, s[i], r[i][i']);
 6   if u ≠ 0 then
 7     if t = 0 then t ← NewNode(k); endif
 8     t[i'] ← Or(t[i'], u);
 9   endif
10 endfor
11 t ← Saturate({N_L,...,N_1}, UniqueTablePut(t));
     • Return a saturated MDD
12 CacheAdd_RelProdSat(s,r,t);
13 return t;
     • UniqueTable guarantees the uniqueness of each node
     • Cache reduces complexity through dynamic programming
```

Figure 1: Saturation algorithms.

$\mathscr{S}_{rch}$ as the fixed point of $\mathscr{S}_{init} \cup \mathscr{N}(\mathscr{S}_{init}) \cup \mathscr{N}^2(\mathscr{S}_{init}) \cup \mathscr{N}^3(\mathscr{S}_{init}) \cup \cdots$. Given a set of states $\mathscr{X}$, their forward and backward reachable sets are $forward(\mathscr{X}) = \mathscr{X} \cup \mathscr{N}(\mathscr{X}) \cup \mathscr{N}^2(\mathscr{X}) \cup \mathscr{N}^3(\mathscr{X}) \cup \cdots$ and $backward(\mathscr{X}) = \mathscr{X} \cup \mathscr{N}^{-1}(\mathscr{X}) \cup (\mathscr{N}^{-1})^2(\mathscr{X}) \cup (\mathscr{N}^{-1})^3(\mathscr{X}) \cup \cdots$.

Locality and disjunctive partition of the next-state function form the basis of the saturation algorithm. The key idea is to apply the event firings in an order consistent with their *Top*. An event in $\mathscr{E}_k$ will not be fired until the events in $\mathscr{E}_h$ where $h < k$ do not further grow the explored state space. We say that a node $a$ at level $k$ is *saturated* if it is a fixed point with respect to firing any event that is independent of all levels above $k$: $\forall h, k \geq h \geq 1, \forall \alpha \in \mathscr{E}_h, \forall \mathbf{i} \in \mathscr{S}_L \times \ldots \times \mathscr{S}_{k+1},\ \{\mathbf{i}\} \times \mathscr{B}(a) \supseteq \mathscr{N}_\alpha(\{\mathbf{i}\} \times \mathscr{B}(a))$

Figure 1 shows the pseudocode of the saturation algorithm. In function *Saturate*, the nodes in MDD $s$ are saturated in order, from the bottom level to the top level. Different from the traditional relational product operation, *RelProdSat* always returns a saturated MDD. Saturation can also be applied to computing $backward(\mathscr{X})$ by using inverse next-state functions $\{\mathscr{N}_L^{-1}, \ldots, \mathscr{N}_1^{-1}\}$.

### 2.3 Previous work

Symbolic SCC analysis has been widely explored. Almost all of these algorithms employ BDD-based manipulation of sets of states. Many efforts have been made on computing the *SCC hull*. The SCC hull contains not only states in nontrivial SCCs, but also states on the paths between them. A family of SCC hull algorithms [12] with the same upper bound of complexity is available. We review two categories of previous work on the same problem as ours: *transitive closure* and the *Xie-Beerel algorithm*.

Hojati et al. [6] presented a symbolic algorithm for testing $\omega$-regular language containment by computing the transitive closure, namely, $\mathscr{N}^+ = \mathscr{N} \cup \mathscr{N}^2 \cup \mathscr{N}^3 \cup \cdots$. Matsunaga et al. [9] proposed a recursive procedure for computing the transitive closure. While it is a fully symbolic algorithm, due to the unacceptable complexity of computing the transitive closure, this approach has long been considered infeasible for complex systems.

Xie et al. [15] proposed an algorithm, referred as the Xie-Beerel algorithm in this paper, combining both explicit state enumeration and symbolic state-space exploration. This algorithm explicitly picks a state as a "seed", computes the forward and backward reachable states from the seed and finds the

```
mdd Lockstep(mdd P)
 1 if(P = ∅) then return ∅;
 2 mdd ans ← ∅;   mdd seed ← Pick(P);   mdd C ← 0;
 3 mdd F_front ← 𝒩(seed) ∩ 𝒫;   mdd B_front ← 𝒩^{-1}(seed) ∩ 𝒫;
 4 mdd F ← F_front;   mdd B ← B_front;
 5 while(F_front ≠ ∅ and B_front ≠ ∅)
 6   F_front ← 𝒩(F_front) ∩ 𝒫 \ F; B_front ← 𝒩(B_front) ∩ 𝒫 \ B;
 7   F ← F ∪ F_front;   B ← B ∪ B_front;
 8 endwhile
 9 if(F_front = ∅) then
10   mdd conv ← F;
11   while(B_front ∩ F ≠ ∅) do B_front ← 𝒩(B_front) ∩ 𝒫 \ B;  B ← B ∪ B_front; endwhile
12 else
13   mdd conv ← B;
14   while(F_front ∩ B ≠ ∅) do F_front ← 𝒩(F_front) ∩ 𝒫 \ F;  F ← F ∪ F_front; endwhile
15 endif
16 if(F ∩ B ≠ ∅ ) then C ← (F ∩ B) ∪ seed;   ans ← C; endif
17 ans ← ans ∪ Lockstep(conv \ C) ∪ Lockstep(P \ (conv ∪ seed));
18 return ans;
```

```
mdd XB_TSCC(mdd 𝒮)
 1 mdd ans ← ∅;   mdd P ← 𝒮;   mdd seed, F, B;
 2 while (P ≠ ∅)
 3   seed ← Pick(P);   F ← forward(seed) ∩ P;   B ← backward(seed) ∩ P;
 4   if F \ B = ∅ then ans ← ans ∪ F; endif                 • Find a terminal SCC
 5   P ← P \ B;
 6 endwhile
 7 return ans;
```

Figure 2: Lockstep for SCC computation and Xie-Beerel's algorithm for terminal SCC computation.

SCC containing this seed as the intersection of these two sets of states. Bloem et al. [2] presented a improved algorithm called *Lockstep*, shown in Figure 2. $Lockstep(\mathscr{S}_{rch})$ returns the set of states belonging to non-trivial SCCs. It has been proven that Lockstep requires in $O(n\log n)$ image and preimage computations (Theorem 2 in [2]), where $n$ is the number of reachable states. As shown in Figure 2, given a "seed" state, instead of computing sets of forward and backward reachable states separately, it uses the set which converges earlier to bound the other. This optimization constitutes the key point in achieving $O(n\log n)$ complexity. Ravi et al. [11] compared the SCC-hull algorithms and Lockstep. According to our experimental results, Lockstep often works very well for systems with few SCCs. However, as the number of SCCs grows, the exhaustive enumeration of SCCs becomes a problem. In this paper, we compare our algorithms to Lockstep.

Xie et al. [14] proposed a similar idea in computing recurrent states in large scale Markov chains. The pseudocode of that algorithm is shown as *XB_TSCC* in Figure 2. From a randomly picked seed state, if the forward reachable states ($F$) is a subset of backward reachable states ($B$), $F$ is a terminal SCC; otherwise ($F \nsubseteq B$), no terminal SCC exists in $B$, and $B$ can be eliminated from future exploration.

The main ideas of our two approaches belong to these two categories of previous work. In the Xie-Beerel algorithm, BFS-based state-space exploration can be replaced with saturation. For transitive closure computation, we propose a new algorithm using saturation.

```
mdd XBSaturation(mdd P)
 1  if(P = ∅) then return ∅;
 2  mdd ans ← ∅;  mdd seed ← Pick(P);
 3  mdd F_front ← N(seed) ∩ P;   mdd B_front ← N^{-1}(seed) ∩ P;
 4  mdd F ← Saturate({N_L ··· N_1}, F_front) ∩ P;
 5  mdd B ← Saturate({N_L^{-1} ··· N_1^{-1}}, B_front) ∩ P;
 -----------------------------------------------------------------
 6  mdd C ← F ∩ B;    if C ≠ ∅ then ans ← C; endif       • Line 6 – 8 are for computing SCCs
 7  ans ← ans ∪ XBSaturation(F \ C) ∪ XBSaturation(P \ F);
 8  return ans;
 -----------------------------------------------------------------
 6'  if F \ B = ∅ then ans ← ans ∪ F; endif           • Line 6'–8' are for computing terminal SCCs
 7'  ans ← ans ∪ XBSaturation(P \ B);
 8'  return ans;
```

Figure 3: Improved Xie-Beerel algorithm using saturation.

## 3 Improving the Xie-Beerel algorithm using saturation

A straightforward idea is to employ saturation on the state-space exploration in the Xie-Beerel algorithm. The pseudocode of our algorithms for computing SCCs and terminal SCCs is shown as *XBSaturation* in Figure 3. The merit of our algorithms comes from the higher efficiency of saturation in computing forward and backward reachable states ($B$ and $F$). However, our algorithms need to compute $B$ and $F$ separately, while Lockstep can use the set that converges first to bound the other, which may reduce the number of image computations (steps). Thus, there is a trade-off between the advantages of BFS and saturation. From a theoretical point of view, the complexity of our algorithm can hardly be compared directly with the result in [2], which measures the complexity by the number of steps. Since saturation executes a series of light-weight events firing instead of global image computations, its complexity cannot be captured as a number of steps. Furthermore, saturation results in more compact decision diagrams during state-space exploration, often greatly reducing runtime and memory. Performance is also affected by which seed is picked in each iteration. For a fair comparison, we pick the same seed in both algorithms at each iteration. The experimental results in Section 6 show that, for most models, the improved Xie-Beerel algorithm using saturation outperforms Lockstep, sometimes by orders of magnitude.

## 4 Applying saturation to computing transitive closure

We define the backward transitive closure ($TC^{-1}$) of a discrete-state model as follows:

**Definition 4.1.** *A pair of states* $(\mathbf{i},\mathbf{j}) \in TC^{-1}$ *iff there exists a non-trivial (i.e., positive length) path* $\pi$ *from* $\mathbf{j}$ *to* $\mathbf{i}$*, denoted by* $\mathbf{j} \to \mathbf{i}$*. Symmetrically, we can define* $TC$ *where* $(\mathbf{i},\mathbf{j}) \in TC$ *iff* $\mathbf{i} \to \mathbf{j}$.

As $TC$ and $TC^{-1}$ are symmetric to each other, we focus on the computation of $TC^{-1}$. $TC$ can then be obtained from $TC^{-1}$ by simply swapping the unprimed and primed levels. Our algorithm is based on the following observation:

$$(\mathbf{i},\mathbf{j}) \in TC^{-1} \text{ iff } \exists \mathbf{k} \in \mathcal{N}^{-1}(\mathbf{i}) \text{ and } \mathbf{j} \in Saturate(\{\mathcal{N}_L^{-1}, \cdots, \mathcal{N}_1^{-1}\}, \{\mathbf{k}\})$$

Instead of executing saturation on $\mathbf{j}$ for each pair of $(\mathbf{i},\mathbf{j})$, we propose an algorithm that executes on the $2L$-level MDD encoding $\mathcal{N}^{-1}$. In function $SCC_TC(\mathcal{N}^{-1})$ of Figure 4, $TC^{-1}$ is computed in line 1 using function *TransClosureSat*, which runs bottom-up recursively. Similar to the idea of saturation shown in Figure 1, this function runs node-wise on primed level and fires lower level events exhaustively until the local fixed point is obtained. This procedure guarantees the following Lemma.

**Lemma:** Given a $2k$-level MDD $n$, $TransClosureSat(n)$ returns an $2k$-level MDD $t$ that for any $(\mathbf{i},\mathbf{j}) \in \mathscr{B}(n)$, all $(\mathbf{i},\mathbf{k})$ where $\mathbf{k} \in (\mathscr{N}^{-1}_{\leq k})^*(\mathbf{j})$ belong to $\mathscr{B}(t)$.

**Theorem:** $TransClosureSat(\mathscr{N}^{-1})$ returns $TC^{-1}$.

This theorem can be proved directly from Lemma and the definition of $TC^{-1}$. The pseudocode of the SCC computation using $TC^{-1}$ is shown in *SCC_TC* in Figure 4. Then, function *TCtoSCC* extracts all states $\mathbf{i}$ such that $(\mathbf{i},\mathbf{i}) \in TC^{-1}$.

Unlike SCC enumeration algorithms like Xie-Beerel's or Lockstep, the $TC$-based approach does not necessarily suffer when the number of SCCs is large. Nevertheless, due to the complexity of building $TC^{-1}$, this approach is considered not feasible for complex systems. Thanks to the idea of saturation, our algorithm of computing $TC^{-1}$ completes on some large models, such as the dining philosopher problem with 1000 philosophers. For some models containing large numbers of SCCs, the $TC$-based approach shows its advantages. While the $TC$-based approach is not as robust as Lockstep, it can be used as the substitute for Lockstep when Lockstep fails to exhaustively enumerate all SCCs.

$TC^{-1}$ can also be employed to find recurrent states, i.e., terminal SCCs. As the other SCCs are not reachable from terminal SCCs, state $\mathbf{j}$ belongs to a terminal SCC iff $\forall \mathbf{i}, \mathbf{j} \rightarrow \mathbf{i} \Longrightarrow \mathbf{i} \rightarrow \mathbf{j}$. Given states $\mathbf{i},\mathbf{j}$, let $\mathbf{j} \mapsto \mathbf{i}$ denote that $\mathbf{j} \rightarrow \mathbf{i}$ *and* $\neg(\mathbf{i} \rightarrow \mathbf{j})$. We can encode this relation with a $2L$-level MDD, which can be obtained as $TC^{-1} \setminus TC$. The pseudocode of this algorithm is shown as *TSCC_TC* in Figure 5. The set of $\{(\mathbf{i},\mathbf{j}) | \mathbf{j} \mapsto \mathbf{i}\}$ is encoded with a $2L$-level MDD $L$. Then, the set of states $\{\mathbf{j} | \exists \mathbf{i}, \mathbf{j} \mapsto \mathbf{i}\}$, which do *not* belong to terminal SCCs, is computed by quantifying out the unprimed levels and can be stored in MDD *nontscc*. The remaining states in SCCs are recurrent states belonging to terminal SCCs.

To the best of our knowledge, this is the first symbolic algorithm for terminal SCC computation using the transitive closure. This algorithm is more expensive in both runtime and memory than SCC computation because of the computation of the $\mapsto$ relation. With the help of *TransClosureSat*, this algorithm works for most of the models we study. Moreover, for models with many terminal SCCs, this algorithm also shows its unique benefits.

## 5 Fairness

One application of the SCC computation is to decide language emptiness for an $\omega$-automaton. The language of an $\omega$-automaton is nonempty if there is a nontrivial *fair loop* satisfying a certain fair constraint. Thus, it is necessary to extend the SCC computation to finding fair loops. Büchi fairness (weak fairness) [5] is a widely used fair condition specified as a set of sets of states $\{\mathscr{F}_1, \ldots, \mathscr{F}_n\}$. A fair loop satisfies Büchi fairness iff, for each $i = \{1, \ldots, n\}$, some state in $\mathscr{F}_i$ is included in the loop.

Lockstep is able to handle the computation of fair loops as proposed in [2]. Here we present a $TC$-based approach. Assume $TC$ and $TC^{-1}$ have been built, let $\mathscr{S}_{weak}$ be the set of states $\mathbf{i}$ satisfying:

$$\bigcap_{m=1,\ldots,n} [\exists \mathbf{f}_m \in \mathscr{F}_m.(TC(\mathbf{f}_m,\mathbf{i}) \wedge TC^{-1}(\mathbf{f}_m,\mathbf{i}))]$$

According to the definition of weak fairness, it can be proved that $\mathscr{S}_{weak}$ contains all states in the fair loops. The pseudocode of computing $\mathscr{S}_{weak}$ is shown in Figure 6. $\mathscr{F}_i \times \mathscr{S}_{rch}$ returns a $2L$-level MDD encoding all pairs of states $(\mathbf{i},\mathbf{j})$ where $\mathbf{i} \in \mathscr{F}_i$ and $\mathbf{j} \in \mathscr{S}_{rch}$. The main complexity lies in computing $TC(\mathbf{i},\mathbf{j}) \wedge TC^{-1}(\mathbf{i},\mathbf{j})$, which is similar to computing the $\mapsto$ relation in the terminal SCC computation.

## 6 Experimental results

We implement the proposed approaches in SMART [4] and report experimental results obtained on an Intel Xeon 3.0Ghz workstation with 3GB RAM under SuSE Linux 9.1. All the models are described as the Petri nets expressed in the input language of SMART. These models include a closed queue networks

```
mdd SCC_TC(N^{-1})
 1 mdd TC^{-1} ← TransClosureSat(N^{-1});
 2 mdd SCC ← TCtoSCC(TC^{-1});
 3 return SCC;
```

```
mdd TransClosureSat(mdd n)
 1 if InCache_TransClosureSat(n,t) then return t;
 2 level k ← n.lvl; mdd t ← NewNode(k); mdd r ← N^{-1}_Unprimed(k)
 3 foreach i,j ∈ S_k s.t. n[i][j] ≠ 0 do  t[i][j] ← TransClosureSat(n[i][j]); endfor
 4 foreach i ∈ S_Unprimed(k) s.t. n[i] ≠ 0
 5   repeat                                                    • Build a local fixed point
 6     foreach j,j' ∈ S_Unprimed(k) s.t. n[i][j] ≠ 0 and r[j][j'] ≠ 0 do
 7       mdd u ← TCRelProdSat(t[i][j],r[j][j']);  t[i][j'] ← Or(t[i][j'],u);
 8     endfor
 9   until t does not change;
10 endfor
11 t ← UniqueTablePut(t);  CacheAdd_TransClosureSat(n,t);
12 return t;
```

```
mdd TCRelProdSat(mdd n, mdd r)
 1 if n = 1 and r = 1 then return 1;
 2 if InCache_TCRelProdSat(n,r,t) then return t;
 3 level k ← n.lvl;  mdd t ← 0;
 4 foreach i ∈ S_Unprimed(k) s.t. n[i] ≠ 0 do
 5   foreach j,j' ∈ S_Unprimed(k) s.t. n[i][j] ≠ 0 and r[j][j'] ≠ 0 do
 6     mdd u ← TCRelProdSat(n[i][j],r[j][j']);
 7     if u ≠ 0 then
 8       if t = 0 then t ← NewNode(k); endif
 9       t[i][j'] ← Or(t[i][j'],u);
10     endif
11   endfor
12 endfor
13 t ← TransClosureSat(UniqueTablePut(t));  CacheAdd_TCRelProdSat(n,r,t);
14 return t;
```

```
mdd TCtoSCC(mdd n)
 1 if n = 1 return 1;   if InCache_TCtoSCC(n,t) then return t;
 2 mdd t ← 0; level k ← n.lvl;
 3 foreach i ∈ S_Unprimed(k) s.t. n[i][i] ≠ 0 do
 4   if TCtoSCC(n[i][i]) ≠ 0 then
 5     if t = 0 then t ← NewNode(k); endif
 6     t[i] ← TCtoSCC(n[i][i]);
 7   endif
 8 endfor
 9 t ← UniqueTablePut(t);   CacheAdd_TCtoSCC(n,t);
10 return t;
```

Figure 4: Building the transitive closure using saturation.

(*cqn*) discussed in [15], two implementations of arbiters (*arbiter*1, *arbiter*2)[1], one which guarantees fairness and the other which does not, the N-queen problem (*queens*), the dining philosopher problem (*phil*) and the leader selection protocol (*leader*) [3]. The size for each model is parameterized with $N$.

```
mdd TSCC_TC(N^{-1})
 1 mdd TC^{-1} ← TransClosureSat(N^{-1});    mdd TC ← Inverse(TC^{-1});
 2 mdd SCC ← TCtoSCC(TC^{-1});
 3 mdd L ← TC^{-1} \ TC;
 4 mdd nontscc ← QuantifyUnprimed(L);
 5 mdd recurrent ← SCC \ nontscc;
 6 return recurrent;
```

Figure 5: Computing recurrent states using transitive closure.

```
mdd FairLoop_TC(S_rch, N^{-1}, {F_1, ..., F_n})
 1 mdd TC^{-1} ← TransClosureSat(N^{-1});    mdd TC ← Inverse(TC^{-1});
 2 mdd S_weak ← S_rch;
 3 foreach m ∈ {1, ..., n}
 4   mdd p ← QuantifyUnprimed(TC^{-1} ∧ TC ∧ (F_m × S_rch));
 5   S_weak ← S_weak ∩ p;
 6 endfor
 7 return S_weak;
```

Figure 6: Computing fair loops using transitive closure.

The number of SCCs (terminal SCCs) and states in SCCs (terminal SCCs) for each model obtained from (terminal) SCC enumeration is listed in column "SCC" ("TSCC") and column "States" respectively. The upper bounds for runtime and size of unique table (i.e., the storage for the MDD nodes) are set to 2 hours and 1GB respectively. The main metrics of our comparison are runtimes and peak memory consumption (for the unique table, storing the MDD nodes, plus the cache).

The top part of Table 1 compares three algorithms for SCC computation: the $TC$-based algorithm (column "TC") presented in Section 4, the improved Xie-Beerel algorithm (column "XBSat") presented in Section 3, and Lockstep (column "Previous algorithm") in Section 2.3. Coupled with saturation, the improved Xie-Beerel algorithm is better than Lockstep for most of the models in both runtime and memory. Compared with Lockstep, the $TC$-based algorithm is often more expensive. However, for two models, *queens* and $arbiter2$, the $TC$-based algorithm completes within the time limit while the other two algorithms fail. For $arbiter2$, our $TC$-based algorithm can explore over $10^{150}$ SCCs in a few seconds, while it is obviously not feasible for SCC enumeration algorithms to exhaustively enumerate all SCCs. To the best of our knowledge, this is the best result of SCC computation reported, stressing that the $TC$-based algorithm is not sensitive to the number of SCCs. With our new algorithm, the transitive closure can be built for some large systems, such as the dining philosopher problem with 1000 philosophers.

The bottom part of Table 1 compares the improved Xie-Beerel algorithm, *XBSaturation*, (column "XBSat") and algorithm *TSCC_TC_Sat* (column "TC"), presented in Section 3 and 4, respectively, for terminal SCC computation, with $XB_TSCC$ (column "Previous algorithm") in Section 2.3. The basic trends are similar to the results of SCC computations, *XBSaturation* works consistently better than the original method, while *TSCC_TC* is less efficient for most models. In the Xie-Beerel framework, it is faster to compute terminal SCCs than all SCCs because a larger set of states is pruned in each recursion. On the contrary, *TSCC_TC* is more expensive than *SCC_TC* due to the computation of the $\mapsto$ relation, which has large memory and runtime requirements. Nevertheless, for models with large numbers of terminal SCCs, such as *queens*, *TSCC_TC* shows its advantage over the Xie-Beerel algorithm.

We conclude that saturation is effective in speeding up the SCC and terminal SCC computations within the framework of the Xie-Beerel algorithm. Also, our new saturation-based $TC$ computation can

| Model | | SCC/TSCC | States | TC | | XBSat | | Previous algorithm | |
|---|---|---|---|---|---|---|---|---|---|
| name | N | | | mem(MB) | time(sec) | mem(MB) | time(sec) | mem(MB) | time(sec) |
| **Results for the SCC computation** | | | | | | | | | |
| *cqn* | 10 | 11 | 2.09e+10 | 34.2 | 13.6 | 3.4 | < 0.1 | 4.0 | 3.9 |
| | 15 | 16 | 2.20e+15 | 64.4 | 73.8 | 5.0 | 0.2 | 89.1 | 44.5 |
| | 20 | 21 | 2.32e+20 | 72.7 | 687.8 | 25.8 | 0.5 | 118.7 | 275.0 |
| *phil* | 100 | 1 | 4.96e+62 | 5.0 | 0.5 | 3.2 | < 0.1 | 52.0 | 4.5 |
| | 500 | 1 | 3.03e+316 | 33.0 | 4.0 | 24.5 | 0.1 | – | to |
| | 1000 | 1 | 9.18e+626 | 40.5 | 7.8 | 29.1 | 0.3 | – | to |
| *queens* | 10 | 3.22e+4 | 3.23e+4 | 8.2 | 1.6 | 64.4 | 14.5 | 63.9 | 12.4 |
| | 11 | 1.53e+5 | 1.53e+5 | 45.8 | 9.0 | 94.2 | 108.6 | 96.3 | 93.6 |
| | 12 | 7.95e+5 | 7.95e+5 | 184.8 | 60.6 | 170.2 | 1220.4 | 281.9 | 1663.9 |
| | 13 | 4.37e+6 | 4.37e+6 | 916.5 | 840.6 | – | to | – | to |
| *leader* | 3 | 4 | 6.78e+2 | 6.0 | 1.4 | 20.8 | < 0.1 | 20.8 | < 0.1 |
| | 4 | 11 | 9.50e+3 | 70.3 | 73.1 | 25.4 | 1.1 | 23.8 | 0.3 |
| | 5 | 26 | 1.25e+5 | 116.6 | 3830.4 | 35.6 | 40.8 | 49.4 | 6.4 |
| | 6 | 57 | 1.54e+6 | – | to | 41.6 | 1494.9 | 417.2 | 387.9 |
| *arbiter*1 | 10 | 1 | 2.05e+4 | 24.1 | 1.2 | 21.4 | < 0.1 | 21.8 | 0.1 |
| | 15 | 1 | 9.83e+5 | 128.3 | 63.0 | 45.1 | < 0.1 | 62.1 | 6.8 |
| | 20 | 1 | 4.19e+7 | mo | – | 709.7 | < 0.1 | mo | – |
| *arbiter*2 | 10 | 1024 | 1.02e+4 | 20.3 | < 0.1 | 26.2 | 0.7 | 31.1 | 1.1 |
| | 15 | 32768 | 4.91e+5 | 20.4 | < 0.1 | 31.1 | 51.8 | 211.3 | 990.3 |
| | 20 | 1.05e+6 | 2.10e+7 | 20.4 | < 0.1 | 31.2 | 2393.3 | – | to |
| | 500 | **3.27e+150** | **1.64e+151** | **41.0** | **4.0** | – | to | – | to |
| **Results for the terminal SCC computation** | | | | | | | | | |
| *cqn* | 10 | 10 | 2.09e+10 | 37.9 | 15.5 | 21.4 | < 0.1 | 33.5 | 3.4 |
| | 15 | 15 | 2.18e+15 | 64.8 | 79.6 | 23.0 | 0.3 | 59.4 | 33.7 |
| | 20 | 20 | 2.31e+20 | 72.7 | 691.3 | 26.2 | 0.8 | 90.0 | 280.5 |
| *phil* | 100 | 2 | 2 | 26.5 | 0.5 | 20.9 | < 0.1 | 39.2 | 8.7 |
| | 500 | 2 | 2 | 34.3 | 4.1 | 23.2 | < 0.1 | – | to |
| | 1000 | 2 | 2 | 44.4 | 11.3 | 26.5 | 0.2 | – | to |
| *queens* | 10 | 1.28e+04 | 1.28e+4 | 36.2 | 3.0 | 46.7 | 2.8 | 62.3 | 35.1 |
| | 11 | 6.11e+04 | 6.11e+4 | 76.5 | 19.3 | 70.6 | 24.5 | 145.2 | 364.2 |
| | 12 | 3.14e+05 | 3.14e+5 | 244.1 | 205.4 | 98.8 | 179.4 | mo | – |
| | 13 | 1.72e+06 | 1.72e+6 | mo | – | 269.0 | 1940.81 | mo | – |
| *leader* | 3 | 3 | 3 | 26.6 | 1.5 | 20.7 | < 0.1 | 21.4 | 0.1 |
| | 4 | 4 | 4 | 70.6 | 75.1 | 24.4 | 0.9 | 38.0 | 4.5 |
| | 5 | 5 | 5 | 119.3 | 3845.3 | 30.6 | 26.9 | 41.1 | 87.6 |
| | 6 | 6 | 6 | – | to | 39.0 | 492.9 | 44.8 | 1341.5 |
| *arbiter*1 | 10 | 1 | 2.05e+4 | 24.1 | 1.2 | 20.4 | < 0.1 | 22.4 | 0.4 |
| | 15 | 1 | 9.83e+5 | 128.3 | 63.1 | 20.4 | < 0.1 | 65.3 | 23.3 |
| | 20 | 1 | 4.19e+7 | mo | – | 20.5 | < 0.1 | – | to |
| *arbiter*2 | 10 | 1 | 1 | 20.4 | < 0.1 | 20.9 | < 0.1 | 39.6 | 6.4 |
| | 15 | 1 | 1 | 20.5 | < 0.1 | 40.6 | 4.6 | – | to |
| | 20 | 1 | 1 | 20.5 | < 0.1 | 450.0 | 2897.8 | – | to |

Table 1: Results for SCC and terminal SCC computations.

tackle some complex models with up to $10^{150}$ states. Finally, for models with huge numbers of SCCs, the $TC$-based SCC computation has advantages over Lockstep, which detects SCCs one-by-one.

While our $TC$-based approach is not a replacement for Lockstep, we argue that it is an alternative worth further research. For a model with an unknown number of existing SCCs, employing both of these approaches at the same time could be ideal. Given current trends in multi-core processors, it is reasonable to run the two algorithms concurrently, possibly sharing some of the common data structures, such as the MDDs encoding the state space and next-state functions.

# 7 Conclusion

In this paper, we focus on improving two previous approaches to SCC computation, the Xie-Beerel algorithm and $TC$, using saturation. We first employ the saturation on the framework of the Xie-Beerel algorithm. In the context of the asynchronous models we study, the improved Xie-Beerel algorithm using saturation achieves a clear speedup. We also propose a new algorithm to compute $TC$ using saturation. The experimental results demonstrate that our $TC$-based algorithm is capable of handling models with up to $10^{150}$ of SCCs. As we argue, the $TC$-based approach is worth further research because of its advantages when used on models with large numbers of SCCs.

# References


[1] NuSMV: a new symbolic model checker. Available at http://nusmv.irst.itc.it/.

[2] Roderick Bloem, Harold N. Gabow, and Fabio Somenzi. An Algorithm for Strongly Connected Component Analysis in $n \log n$ Symbolic Steps. In *Formal Methods in Computer Aided Design*, pages 37–54. Springer-Verlag, 2000.

[3] Gianfranco Ciardo et al. SMART: Stochastic Model checking Analyzer for Reliability and Timing, User Manual. Available at http://www.cs.ucr.edu/∼ciardo/SMART/.

[4] Gianfranco Ciardo, Robert L. Jones, Andrew S. Miner, and Radu Siminiceanu. Logical and stochastic modeling with SMART. *Perf. Eval.*, 63:578–608, 2006.

[5] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999.

[6] Ramin Hojati, Hervé J. Touati, Robert P. Kurshan, and Robert K. Brayton. Efficient $\omega$-regular language containment. In *CAV*, pages 396–409, 1992.

[7] Timothy Kam, Tiziano Villa, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. Multi-valued decision diagrams: theory and applications. *Multiple-Valued Logic*, 4(1–2):9–62, 1998.

[8] Yonit Kesten, Amir Pnueli, and Li-on Raviv. Algorithmic verification of linear temporal logic specifications. In *ICALP '98: Proceedings of the 25th International Colloquium on Automata, Languages and Programming*, pages 1–16, London, UK, 1998. Springer-Verlag.

[9] Y. Matsunaga, P.C. McGeer, and R.K. Brayton. On computing the transitive closure of a state transition relation. In *Design Automation, 1993. 30th Conference on*, pages 260–265, June 1993.

[10] Tadao Murata. Petri nets: properties, analysis and applications. *Proc. of the IEEE*, 77(4):541–579, April 1989.

[11] Kavita Ravi, Roderick Bloem, and Fabio Somenzi. A comparative study of symbolic algorithms for the computation of fair cycles. In *FMCAD '00: Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design*, pages 143–160, London, UK, 2000. Springer-Verlag.

[12] Fabio Somenzi, Kavita Ravi, and Roderick Bloem. Analysis of symbolic SCC hull algorithms. In *FMCAD '02: Proceedings of the 4th International Conference on Formal Methods in Computer-Aided Design*, pages 88–105, London, UK, 2002. Springer-Verlag.

[13] Min Wan and Gianfranco Ciardo. Symbolic state-space generation of asynchronous systems using extensible decision diagrams. In M. Nielsen et al., editors, *Proc. 35th Int. Conf. Current Trends in Theory and Practice of Computer Science (SOFSEM)*, LNCS 5404, pages 582–594, Špindlerův Mlýn, Czech Republic, February 2009. Springer-Verlag.

[14] Aiguo Xie and Peter A. Beerel. Efficient state classification of finite-state Markov chains. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 17(12):1334–1339, 1998.

[15] Aiguo Xie and Peter A. Beerel. Implicit enumeration of strongly connected components and an application to formal verification. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 19(10):1225–1230, 2000.

# Towards the Formal Verification of a Distributed Real-Time Automotive System

Erik Endres* Saarland University Germany email@endrese.de
Christian Müller* Saarland University Germany cm@cs.uni-sb.de
Andrey Shadrin* Saarland University Germany shavez@cs.uni-sb.de
Sergey Tverdyshev*† SYSGO AG Germany stv@sysgo.com

**Abstract**

We present the status of a project which aims at building, formally and pervasively verifying a distributed automotive system. The target system is a gate-level model which consists of several interconnected electronic control units with independent clocks. This model is verified against the specification as seen by a system programmer. The automotive system is implemented on several FPGA boards. The pervasive verification is carried out using combination of interactive theorem proving (Isabelle/HOL) and model checking (LTL).

## 1 Introduction

There are many works on formal verification of hardware, software, and protocols. However, their interplay in a computer system is not to ignore because even if the hardware and software are correct there is no guarantee that this software is executed correctly on the given hardware. It becomes even more critical when considering distributed embedded systems due to the close interaction of software and hardware parts.

The goal of our project is to show that it is feasible to formally verify a *complex* distributed automotive system in a *pervasive* manner. Pervasive verification [10, 17] attempts to verify systems completely including the interaction of all components, thus minimizing the number of system assumptions. Our desired goal is a "single" top-level theorem which describes the correctness of the whole system.

The subproject Verisoft-Automotive aims at the verification of an automatic emergency call system, eCall [7]. The eCall system is based on a time triggered distributed real-time system which consists of distributed hardware and a distributed operating system.

We use Isabelle/HOL [12], an interactive theorem prover, as the design and verification environment. Our interactive proofs are supported by the model checking technique [21].

**Context and Related Work** Pervasive verification of a system over several layers of abstraction is introduced in the context of the CLI stack project [2]. However, the application of such verification techniques to an industrial scenario without strong restrictions (e.g. on the programming language) poses a grand challenge problem as by J. S. Moore [10]. Rushby [14] gives an overview of the formal verification of a Time-Triggered Architecture [15] and formally proves the correctness of some key algorithms. Automated correctness proofs for abstract versions of protocols for serial interfaces using k-induction are reported in [13]. There are also recent efforts on the fully automated verification of clock domain crossing issues [9]. It would be highly desirable to reuse results of this nature for a pervasive correctness proof of distributed automotive system. However, putting all these efforts together in a pervasive correctness proof arguing about several layers of abstraction has not been reported.

In the following section we present the automotive system and its components. In Section 3 we describe the hardware environment and system implementation. Section 4 exposes verification challenges. We conclude the paper by summary and future work.

*The authors were supported by the German Federal Ministry of Education and Research (BMBF) in the Verisoft project under grant 01 IS C38

†The reported work has been done while author was affiliated with Saarland University

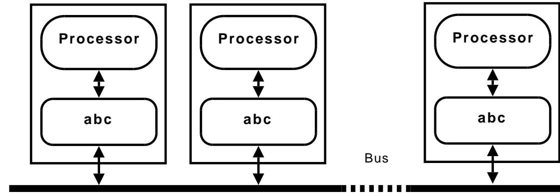



Figure 1: Several ECUs interconnected by a communication bus.

## 2 Automotive System

The automotive system [8] is inspired by the demands of the automotive industry and based on the FlexRay standard [4]. Our automotive system is a distributed asynchronous communication system represented by a set of electronic control units (ECUs) connected to a single bus. The overview of such a distributed system is illustrated in Figure 1. The ECU is built on the base of a formally verified generic gate-level platform [20]. This platform is a complex gate-level computer system, which consists of a pipelined processor with out-of-order execution and a number of memory mapped I/O devices.

Each ECU has its own clock and contains a bus controller and a processor. The bus controller is attached to the processor via a device interface. Besides the control logic, each bus controller contains two buffers: a send and a receive buffer. We denote the controller "ABC" standing for automotive bus controller. Further we denote by $s_i^c$ the hardware state $s$ in cycle $c$ of the $i^{th}$ bus controller in our network. By $s_i^c.rb$ we denote the content of the receive buffer and by $s_i^c.sb$ the content of the send buffer. Moreover, since we argue about clock domain crossing, we model the translation of digital values to analogous and vice-versa. We use the function $hcy_i : \mathbb{R} \to \mathbb{N}$ to map real time to the corresponding hardware cycles on the ECU $i$. The function $asr_i(t) : \mathbb{R} \to \{0,1,\Omega\}$ provides the analogous value of the send register of ECU $i$ in cycle $hcy_i(t)$. Note, that the analogues value gets metastable ($\Omega$) for a short amount of time right after the send register is clocked. During a message transmission we write the content of the send buffer bitwise into this register. We write into it the idle value '1' otherwise. The real time clock period of the $i^{th}$ controller is denoted by $\tau_i$, i.e. one hardware cycle of the ECU $i$ lasts $\tau_i$ in the analogous world.

The ECUs communicate in a time-triggered static schedule. These time intervals are the so-called *communication rounds*. A communication round is a periodically recurring time unit which is divided into a fixed number of slots. In each slot, exactly one ECU is allowed to broadcast one message to the bus. Let $\alpha_i(r,s)$ be the point in real time when the slot $s$ of round $r$ is started on the ECU $i$, and $\omega_i(r,s)$ be the end time of this slot. The start of each round is signalized by one special ECU called *master*, all other ECUs are called *slaves*. As soon as a slave ECU receives this round start signal, it begins with execution of a fixed schedule. Each ECU counts the number of passed slots and, depending on its status in the given slot (sender or receiver), it either samples the bus value or it sends some data to the bus. When a slave ECU reaches the maximal slot number in one round, it goes to an idle state and waits for a new round start signal. The master ECU waits for some predefined amount of time when it is guaranteed that all slave ECUs are waiting for a new round. Only then the master broadcasts a start signal for the new round. The communication protocol as well as the clock synchronization mechanism are described in details in [3].

A system run scenario can be described as follows. Assume the ECU $m$ is acting as a sender in slot $s$ of a round $r$ and before slot $s$ ECU $m$ copied data $d$ from its memory to its send buffer, such that we have at the slot start $s_m^{hcy_m(\alpha_m(r,s))}.sb = d$. After the slot start the ECU $m$ waits *off* cycles before it starts the transmission. The number of cycles *off* has to be big enough s.t. the start time of the slot $s$ on all other ECUs is before $\alpha_m(r,s) + \tau_m \cdot off$. Then, $m$ broadcasts the content of $s_m.sb$ (data $d$) bitwise to the bus during the next $tc$ cycles (transmission length). At the end of the slot $s$ each receiver ECU contains $d$ in its receive buffer: $\forall i : s_i^{hcy_i(\omega_i(r,s))}.rb = d$.

Such a time-triggered communication requires that all ECUs have roughly the same notion of

time, such that each ECU is aware of the correct slot number and its role in this slot during each message transmission. This is one of the verification challenges (Section 4).

## 3 Implementation

The verified ECU design has been automatically translated to Verilog [18] directly from formal Isabelle/HOL hardware descriptions and has been synthesized with the Xilinx ISE software. The prototype implementation consists of several FPGA boards which are interconnected into a prototype of a distributed hardware network. Every ECU is running on either Xilinx Spartan-3 and Virtex-2 FPGA development boards [6]. Each board consists of a field programmable gate array (FPGA) and several devices (e.g. LEDs, switches) connected to the I/O-ports of the FPGA chip.Every board has its own clock source, thus, all ECUs are clocked independently. The boards are interconnected via Ethernet cable. The physical layer of the data transmission is tuned to the FlexRay standard and is provided by the low voltage differential signaling driver [11], which generates a differential signal of ± 350 mV. We successfully tested communication between FPGA boards with the help of the hardware logic analyzer Trektronix TLA5204 and the software Chipscope.

## 4 Verification Challenges

The correctness of the presented distributed system can be split into two parts: local correctness (single ECU) and distributed correctness (asynchronous communicating ECUs).

The local part focuses on the correctness of the processor, the ABC device and their communication, e.g. instructions are correctly executed, the device registers are written and read correctly. More details on the local correctness can be found in [19].

The distributed correctness states that during a run the exchange of data between ECUs is correct, e.g. the sent data of one ECU are the received data on another ECU. Obviously the distributed correctness requires the local one. Moreover, this exchange requires correct asynchronous communication via a FlexRay bus. The state of the bus is a conjunction of outputs of all send registers, i.e. it is an $\wedge$-bus:

$$bus(t) = \bigwedge_{\forall i} asr_i(t)$$

On the receiver side, the bus value $bus(t)$ will be clocked into the analogous receive register, digitalized, and clocked into the receive buffer afterwards. The correctness of the message exchange in the automotive system is based on two properties. First, we have to ensure that if a connection between a sender and receivers is established directly (i.e. we abstract bus by a link), then the low level bit transmission from the sender to all receivers is correct. One of the challenges here is to ensure that the value broadcast on the link is stable long enough so that it can be sampled correctly by the receiver. In our case, if the sender sends $n$ bits, the receiver will sample at least $n-i$ of these bits. The number $i$ is the number of lost bits due to the clock drift between different ECUs. This information loss happens only at the low-level bit transmission. At this level we transmit the message encoded according to the FlexRay standard (each bit is replicated eight times) which defines sufficient redundancy to guarantee the transmission of every single bit of the original information. The correctness of this low-level transmission mechanism cannot be carried out in a conventional, digital, synchronous model. It involves asynchronous and real-time-triggered register models taking into account setup- and hold-times as well as metastability. This part of the pervasive correctness has been formally verified and reported in [16].

The second part is the bus correctness where we have to prove that we can abstract the bus while a sender broadcasts data to the bus. Here, we show that the bus connection can be modeled as a direct link between sender and each receiver. The latter holds only if during each transmission only one sender (one ECU) is broadcasting and all receivers are listening and not sending something (i.e. they are not producing a bus contention). To avoid a bus contention each ECU has to be aware of the correct slot number, i.e. all ABCs have roughly the same notion of the slot start and end times

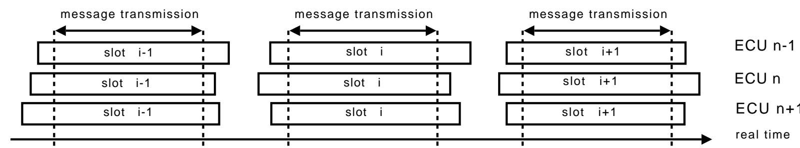



Figure 2: Time notion of an ECU.

(correctness of the scheduling unit). Therefore, due to drifting clocks a synchronization is necessary. We use a simple approach: in the beginning of each round all slave ECUs are waiting for the round start signal broadcast by the master ECU. After this signal all receivers are aware of the current slot number namely zero. All consecutive slots are started and ended locally on each ECU with respect to the maximal clock drift that can occur during a full round. Böhm has formally verified a theorem [3] that if a slave ECU $i$ is waiting and the master ECU $m$ starts a new round $r$, then the ECU $i$ is aware of each new slot $s$ of this round before the message transmission starts. He also proves that the transmission of each slot ends before the receiver ECU "thinks" that the slot is over. We have significantly extended these theorems and used them as an induction hypothesis to prove that on all ECUs **each** slot of **each** round starts before and ends after the message transmission:

$$\forall \text{ slot } s, \text{ round } r, \text{ ECU } m, \text{ ECU } i \;.\; m \text{ is sender in slot } s \rightarrow$$
$$\alpha_m(r,s) + \mathit{off}\cdot\tau_m > \alpha_i(r,s) \wedge \alpha_m(r,s) + (\mathit{off} + tc)\cdot\tau_m < \omega_i(r,s)$$

Thus, we have shown that each slot of an ECU overlaps with the same slot on all other ECUs during the message transmission as depicted in Figure 2. Since all receivers place '1' on the ∧-bus we show, that during any transmission, the bus contains value of the send register of the sender:

$$\forall \text{ ECU } m, \text{ slot s , round } r \;.\; bus(t) = asr_m(t) \text{ for } t \in [\alpha_m(r,s) + \mathit{off}\cdot\tau_m \;;\; \alpha_m(r,s) + (\mathit{off} + tc)\cdot\tau_m]$$

We prove all real-time properties and complex hardware theorems interactively in Isabelle/HOL. Some properties of hardware with "model-checkable state space" are expressed in LTL and proven automatically [21].

# 5 Summary

In this paper we presented the status of a pervasive verification of a distributed automotive system. The system is a distributed network of electronic control units interconnected by a single bus involving clock domain crossing. We have successfully built up a working gate-level prototype synthesized from our formal models. We have also partially verified the automotive system. This pervasive verification is very challenging because the system exists on three levels of abstraction: 1. a formal model of an asynchronous *real-time* triggered system on the bus side, 2. a formal gate-level design of *digital* hardware for local properties on the controller side, 3. a formal model as seen by an *assembler* programmer. Moreover, all our models are combined together and are formally specified in Isabelle/HOL theorem prover.

In our previous work we have formally verified a platform for electronic control unit [20], scheduler correctness [3], and low-level bit transmission [16]. As part of the current work we have *consolidated* previous results which was not an easy task due to the combination of results over several layers of abstractions. We are also finishing the verification of the asynchronous message transmission between several ABC devices. The latter includes the verification of the bus correctness (done) and a correct transmission of send and receive messages from the corresponding buffers to / from the bus (in progress).

For future work we see several interesting topics. First, finishing the current work. Then, we can extend the automotive system with fault tolerance, e.g. as sketched in [1]. Another work in progress at our chair is verification of a distributed operating system which runs on top of the presented

distributed system [5]. We also would like to put together formal proofs for the latter operating system and our distributed hardware.

## References


[1] Eyad Alkassar, Peter Boehm, and Steffen Knapp. Correctness of a fault-tolerant real-time scheduler algorithm and its h ardware implementation. In *MEMOCODE'2008*, pages 175–186. IEEE Computer Society Press, 2008.

[2] William R. Bevier, Warren A. Hunt, Strother Moore, and William D. Young. An approach to systems verification. *Journal of Automated Reasoning*, 5(4):411–428, 1989.

[3] Peter Böhm. Formal Verification of a Clock Synchronization Method in a Distributed Automotive System. Master's thesis, Saarland University, Saarbrücken, 2007.

[4] FlexRay Consortium. FlexRay – the communication system for advanced automotive control applications. `http://www.flexray.com/`, 2006.

[5] Matthias Daum, Norbert W. Schirmer, and Mareike Schmidt. Implementation correctness of a real-time operating system. In *(SEFM 2009), 23–27 November 2009, Hanoi, Vietnam*, pages 23–32. IEEE, 2009.

[6] Erik Endres. FlexRay ähnliche Kommunikation zwischen FPGA-Boards. Master's thesis, Wissenschaftliche Arbeit, Saarland University, Saarbrücken, 2009.

[7] European Commission (DG Enterprise andDGInformation Society). eSafety forum: Summary report 2003. Technical report, eSafety, March 2003.

[8] Steffen Knapp and Wolfgang Paul. Pervasive verification of distributed real-time systems. In M. Broy, J. Grünbauer, and T. Hoare, editors, *Software System Reliability and Security*, volume 9 of *IOS Press, NATO Security Through Science Series.*, 2007.

[9] Bing Li and Chris Ka-Kei Kwok. Automatic formal verification of clock domain crossing signals. In *ASP-DAC '09*, pages 654–659, Piscataway, NJ, USA, 2009. IEEE Press.

[10] J Strother Moore. A grand challenge proposal for formal methods: A verified stack. In Bernhard K. Aichernig and T. S. E. Maibaum, editors, *10th Anniversary Colloquium of UNU/IIST*, volume 2757 of *LNCS*, pages 161–172. Springer, 2002.

[11] National Semiconductor. *LVDS Owner's Manual*, 2008.

[12] Lawrence C. Paulson. *Isabelle: a generic theorem prover*, volume 828 of *LNCS*. Springer, New York, NY, USA, 1994.

[13] Lee Pike. Modeling time-triggered protocols and verifying their real-time schedules. In *FMCAD'07*, pages 231–238. IEEE, 2007.

[14] John M. Rushby. An overview of formal verification for the time-triggered architecture. In *FTRTFT '02*, pages 83–106, London, UK, 2002. Springer-Verlag.

[15] C. Scheidler, G. Heine, R. Sasse, E. Fuchs, H. Kopetz, and C. Temple. Time-triggered architecture (tta). *Advances in Information Technologies*, 1997.

[16] J. Schmaltz. A formalization of clock domain crossing and semi-automatic verification of low level clock synchronization hardware. Technical report, Saarland University, 2006.

[17] The Verisoft Consortium. The Verisoft Project. `http://www.verisoft.de/`, 2003.

[18] S. Tverdyshev and A. Shadrin. Formal verification of gate-level computer systems (short paper). In Kristin Yvonne Rozier, editor, *LFM 2008*, NASA Scientific and Technical Information (STI), pages 56–58. NASA, 2008.

[19] Sergey Tverdyshev. *Formal Verification of Gate-Level Computer Systems*. PhD thesis, Saarland University, Saarbrücken, 2009.

[20] Sergey Tverdyshev. A verified platform for a gate-level electronic control unit. In *Formal Methods in Computer Aided Design, FMCAD'09*, IEEE, pages 164–171, 2009.

[21] Sergey Tverdyshev and Eyad Alkassar. Efficient bit-level model reductions for automated hardware verification. In *TIME 2008*, pages 164–172. IEEE, 2008.

# Slicing AADL Specifications for Model Checking*

Maximilian Odenbrett Viet Yen Nguyen Thomas Noll

Software Modeling and Verification Group
RWTH Aachen University, Germany

**Abstract**

To combat the state-space explosion problem in model checking larger systems, abstraction techniques can be employed. Here, methods that operate on the system specification before constructing its state space are preferable to those that try to minimize the resulting transition system as they generally reduce peak memory requirements.

We sketch a slicing algorithm for system specifications written in (a variant of) the Architecture Analysis and Design Language (AADL). Given a specification and a property to be verified, it automatically removes those parts of the specification that are irrelevant for model checking the property, thus reducing the size of the corresponding transition system. The applicability and effectiveness of our approach is demonstrated by analyzing the state-space reduction for an example, employing a translator from AADL to Promela, the input language of the SPIN model checker.

## 1 The Specification Language

The work that is described in this paper emanates from the European Space Agency COMPASS Project[1] (Correctness, Modeling, and Performance of Aerospace Systems). Within this project, a specification language entitled SLIM (System-Level Integrated Modeling Language) is developed which is inspired by AADL and thus follows the component-based paradigm. Each component is given by its type and its implementation. The *component type* describes the interface features: in and out data and event *ports* for exchanging data (instantaneous) and event messages (synchronously) with other components. The behavior is defined in the *component implementation* by *transitions* between *modes*, like in a finite automaton. Transitions can have an event port associated to them as a *trigger*. A transition with an out event port as trigger is enabled only if in at least one other component a transition with a corresponding in event port as trigger can synchronously be taken and thereby react to the trigger (and correspondingly the other way round). Furthermore, transitions can be taken only when their *guard* expression evaluates to true. Transitions can be equipped with *effects*, i.e., a list of assignments to data elements. Within a component implementation *data subcomponents*, comparable to local variables, can be declared. Together with in and out data ports we refer to them as *data elements*. All of them are typed and can have a *default value* which is used as long as it is not overwritten. The availability of each data subcomponent can be restricted with respect to modes of its supercomponent. In other than these modes the data subcomponent may not be used and on (re)activation it will be reset to its default value.

In addition to data subcomponents, components can be composed of other *non-data subcomponents*, possibly using multiple instances of the same component implementation. In the resulting nested component hierarchy, components can be connected to their direct subcomponents, to their neighbor components in the same supercomponent and to their direct supercomponent by *data flows* and *event port connections* between their ports. The connections can again be mode dependent. If a data flow becomes deactivated then its target port is reset to its default value. Fan-out is always possible whereas fan-in is not allowed for data flows and must be used carefully with event port connections. Cyclic dependencies are disallowed, too. A component can send an in event to or receive an out event from any of its subcomponents directly by using *subcomponent.eventport* as transition trigger.

*Partially funded by ESA/ESTEC under Contract 21171/07/NL/JD

[1] `http://compass.informatik.rwth-aachen.de/`

Listing 1 gives an example SLIM specification modeling an adder component provided with random input $x, y \in [0, 30]$. We refer to [2, 3] for a more detailed description of the language including a discussion of the similarities and extensions with respect to AADL. In particular, [2] presents a formal semantics for all language constructs, based on networks of event-data automata (NEDA).

```
system Main
end Main;

system implementation Main.Impl
  subcomponents
    random1: system Random.Impl
                    accesses aBus;
    random2: system Random.Impl
                    accesses aBus;
    adder: system IntAdder.Impl
                  accesses aBus;
    aBus: bus Bus.Impl;
  flows
    adder.x := random1.value;
    adder.y := random2.value;
  modes
    pick: initial mode;
  transitions
    pick -[random1.update]-> pick;
    pick -[random2.update]-> pick;
end Main.Impl;

bus Bus
end Bus;
bus implementation Bus.Impl
end Bus.Impl;

system IntAdder
  features
    x: in data port int;
    y: in data port int;
    sum: out data port int;
end IntAdder;

system implementation IntAdder.Impl
  flows
    sum := x + y;
end IntAdder.Impl;

system Random
  features
    value: out data port int default 2;
    update: in event port;
end RandomIntValue;

system implementation Random.Impl
  modes
    loop: initial mode;
  transitions
    loop -[update then value := 0]-> loop;
    ...
    loop -[update then value := 30]-> loop;
end RandomIntValue.Impl;
```

Listing 1: Integer Adder in SLIM

# 2 Slicing

The term "slicing" has been coined by Weiser [9], initially for sequential programs, and the approach was extended later on in several ways by many authors (cf. [8]). Directly related to our work is the extension of slicing to concurrent programs by Cheng [4] and, most important, the application of slicing to software model checking including formal notions of correctness by Hatcliff, Dwyer and Zheng [5].

The principal idea of slicing is to remove all parts of a program, typically variables and statements, that do not influence the behavior of interest, typically the values of some variables at some statements, described by a slicing criterion. To determine which parts are relevant, the transitive backward closure of the slicing criterion along different kinds of dependences, typically data and control dependences, is computed. However, finding a minimal sliced program is in general unsolvable since the halting problem can be reduced to it (cf. [9]).

## 2.1 Slicing of SLIM Specifications

Given a specification $S$ and a CTL$^*$ property $\varphi$ (without next operator), slicing should yield a smaller specification $S^{\varphi}_{sliced}$ that is equivalent to $S$ with respect to $\varphi$, i.e., $S \models \varphi$ iff $S^{\varphi}_{sliced} \models \varphi$ (cf. [5]). Consequently, comparable to a slicing criterion, the property defines the initially interesting parts that must not be sliced away: data elements and modes used in $\varphi$ (events are not allowed in our properties but could

be added). Subsequently, the closure of the set of interesting parts, i.e., all other aspects that have an (indirect) influence on them and thus on the property, is calculated in a fixpoint iteration. Obviously this iteration always terminates but in the worst case all parts of the specification become interesting.

In the following three paragraphs we describe in more detail the closure rules for adding data elements, events and modes to the set of interesting parts before the actual slicing algorithm is presented in pseudo-code.

**Identifying Interesting Data Elements** Like with data flow dependence for classic program slicing, all data elements used to calculate a new value for an interesting data element are interesting, too. Here, this affects the right hand sides of assignments to an interesting data element, either in transition effects or by data flows. Furthermore, comparable to control flow dependence, all data elements used in guards on interesting transitions (see below) must be kept in the sliced specification as the evaluation of the guard at runtime determines whether the transition can indeed be taken.

**Identifying Interesting Events** The main difference of SLIM specifications compared to sequential programs is that the components can synchronously communicate by sending and receiving events. Comparable to synchronization and communication dependences (cf. [4]), all events used as triggers on interesting transitions are important. As events can be forwarded by event port connections all events connected to an interesting event in any direction are interesting as well.

**Identifying Interesting Modes** Similarly to the program location in classical slicing, our algorithm does *not* treat the mode information as a data element which is either interesting or not but tries to eliminate uninteresting modes. The difficulty is that the questions whether a mode, a data element or an event is interesting are related to each other since all those elements can be combined in the transition relation: On the one hand, transitions are (partially) interesting when they change an interesting data element, have an interesting trigger or their source or target mode is interesting. On the other hand, triggers, guards, and source modes of those transitions are interesting. However, transitions themselves are not considered as elements of interest in the fixpoint iteration. Instead, modes are made interesting and with them implicitly all incoming and outgoing transitions. More concretely, besides the modes used in the property the following modes are interesting as well:

- Source modes of transitions changing an interesting data element. This obviously applies to transitions with assignments to interesting data elements in their effects but also to transitions reactivating an interesting data element, i.e., it is active in the target mode but not in the source mode, since it will be reset to its default value.
- All modes in which a data flow to an interesting data element or an event port connection to/from an interesting event is active. This guarantees that all transitions that deactivate a data flow to an interesting data element and thus reset it to its default value are included in the sliced specification.
- Source modes of transitions with interesting events as triggers because of their relevance for synchronous event communication.

Moreover, the reachability of interesting modes from the initial mode matters. Thus, every predecessor of an interesting mode, that is, the source modes of transitions to interesting target modes, is interesting as well.

Finally, for liveness properties it is additionally necessary to preserve the possibility of divergence since no fairness assumptions are made. For example, whether a component is guaranteed to reach a certain mode can depend on the fact whether another component can loop ad infinitum. To handle this, all modes on “syntactical cycles”, i.e., cycles of the transition relation without considering triggers and guards, are interesting as well. For safety properties this can be omitted.

## 2.2 The Slicing Algorithm

For the pseudo-code description given in Listing 2 we use the following notations: *Dat*, *Evt* and *Mod* are the sets of data elements, events and modes occurring in the specification, respectively. The relation *Trn* contains transitions of the form $m \xrightarrow{e,g,f} m'$ with source and target mode $m, m' \in Mod$, trigger $e \in Evt$, guard expression $g$ over data elements and $f$ a list of assignments. Data flows $d := a$ instantly assigning the value of an expression $a$ to a data element $d \in Dat$ are collected in the set *Flw*. Finally, *Con* contains connections $e \rightsquigarrow e'$ between event ports $e, e' \in Evt$. On this basis, the algorithm can compute the sets of interesting data elements ($D$), events ($E$) and modes ($M$).

Note that, in analogy to distinguishing calling environments of procedures, slicing is done for component instances and not for their implementation. This is more effective since different instances of the same implementation might be sliced differently. Therefore, we have to distinguish identical parts of different component instances. For example, the set *Dat* for the specification in Listing 1 does not simply contain the name `Random.value` but `random1.value` and `random2.value` to differentiate between the different instances of the `Random` component.

/* Initialization */
$D := \{d \in Dat \mid d \text{ occurs in } \varphi\}$;
$E := \emptyset$;
$M := \{m \in Mod \mid m \text{ occurs in } \varphi\}$;
/* Fixpoint Iteration */
**repeat**
  /* Transitions that update/reactivate interesting data elements or have interesting triggers */
  **for all** $m \xrightarrow{e,g,f} m' \in Trn$ with $\exists d \in D : f$ updates $d$
    or $\exists d \in D : d$ inactive in $m$ but active in $m'$
    or $e \in E$ **do**
    $M := M \cup \{m\}$;
  /* Transitions from/to interesting modes */
  **for all** $m \xrightarrow{e,g,f} m' \in Trn$ with $m \in M$ or $m' \in M$ **do**
    $D := D \cup \{d \in Dat \mid g \text{ reads } d\}$
      $\cup \{d \in Dat \mid f \text{ updates some } d' \in D \text{ reading } d\}$;
    $E := E \cup \{e\}$;
    $M := M \cup \{m\}$;
  /* Data flows to interesting data ports */
  **for all** $d := a \in Flw$ with $d \in D$ **do**
    $D := D \cup \{d' \in Dat \mid a \text{ reads } d'\}$;
    $M := M \cup \{m \in Mod \mid d := a \text{ active in } m\}$;
  /* Connections involving interesting event ports */
  **for all** $e \rightsquigarrow e' \in Con$ with $e \in E$ or $e' \in E$ **do**
    $E := E \cup \{e, e'\}$;
    $M := M \cup \{m \in Mod \mid e \rightsquigarrow e' \text{ active in } m\}$;
**until** nothing changes;

Listing 2: The Slicing Algorithm

## 2.3 The Sliced Specification

After calculating the fixpoint, the sliced specification $S^{\varphi}_{sliced}$ can be generated. Essentially, it contains only the interesting data elements ($D$), events ($E$) and modes ($M$) plus data flows and connections to them, i.e., $d := a \in Flw$ with $d \in D$ and $e \rightsquigarrow e' \in Con$ where $e \in E$, respectively. Their default values and the lists of modes in which they are active stay the same. The sliced transition relation contains all transitions $m \xrightarrow{e,g,f} m' \in Trn$ leaving an interesting mode $m \in M$ with slight modifications: If the target mode is not interesting ($m' \notin M$), it is replaced by a "sink mode" $m_o \notin Mod$ which is added to every component that had uninteresting modes. For each data subcomponent, this sink mode is also added to the list of modes in which the component is active. Furthermore, only those transition effects $d := a$ in $f$ are retained that assign to an interesting data element, i.e., $d \in D$. Finally, all "empty" components, i.e., those that neither have interesting data elements, interesting modes nor non-empty subcomponents, are completely removed in a bottom-up manner.

The resulting specification $S^{\varphi}_{sliced}$ is again a valid SLIM specification. In particular, every object referenced in it is indeed declared as it was included in the fixpoint iteration, e.g., the data elements used in the guards of interesting transitions. Beyond that, sink modes indeed do not need outgoing transitions as it is impossible to change an interesting data element, to send or receive an interesting event or to reach an interesting mode as soon as an uninteresting mode has been entered by the original component.

## 3 Results and Conclusions

For model checking SLIM specifications we developed a translator [7] to Promela, the input language of SPIN [6]: Every component instance is transformed to a process whose program labels reflect the modes. Data elements are stored in global variables and communication is implemented using channels. Due to dependencies introduced by translation details, SPIN's slicing algorithm could not effectively reduce the resulting Promela code.

Comparing the model checking results of sliced and unsliced specifications served as a first sanity check for our algorithm while a general correctness proof based on the formal semantics of SLIM is to be developed. The idea is to show that the corresponding transition systems are related via a divergence-sensitive stuttering bisimulation, which is known to preserve the validity of CTL$^*$ properties without next [1, p. 560]. Furthermore, the differences in resource demands demonstrate the effectiveness of our approach, e.g., for the introductory example from Listing 1 as shown in the following table:

| Specification | Mem/State (bytes) | #States | Memory (MBs) | Time (seconds) |
|---|---|---|---|---|
| Unsliced (identical for $\varphi_1, \dots, \varphi_3$) | 136 | 1,676,026 | 272 | 6.0 - 7.3 |
| Sliced for $\varphi_1 \equiv \Box(0 \leq adder.sum \leq 60)$ | 116 | 1,437,691 | 211 | 5.4 |
| Sliced for $\varphi_2 \equiv \Box\left(\bigwedge_{\beta=1}^{2} 0 \leq random\beta.value \leq 30\right)$ | 84 | 553,553 | 84 | 1.4 |
| Sliced for $\varphi_3 \equiv \Box(0 \leq random1.value \leq 30)$ | 76 | 9,379 | 33 | 0.1 |

All three properties are valid invariants and thus require a full state space search. The difference is in the resulting set of interesting data elements: While for $\varphi_1$ every data port is needed, for $\varphi_2$ the whole `adder` component can be removed and for $\varphi_3$ only `random1.value` is interesting. The removal of the empty bus component accounts for the reduction from the unsliced specification to the one sliced for $\varphi_1$.

We conclude that our slicing algorithm can considerably reduce the state space, especially when whole components can be removed. We end with the remark that beyond the scope of this paper the algorithm has been extended to more involved language constructs (such as de- and reactivation of non-data subcomponents or hybridity) and that a refining distinction between weak and strong interesting data elements was made.

## References


[1] C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT Press, 2008.

[2] M. Bozzano, A. Cimatti, J.-P. Katoen, V. Y. Nguyen, T. Noll, and M. Roveri. Codesign of Dependable Systems: A Component-Based Modelling Language. In *7th Int. Conf. on Formal Methods and Models for Co-Design (MEMOCODE)*, pages 121–130. IEEE CS Press, 2009.

[3] M. Bozzano, A. Cimatti, J.-P. Katoen, V. Y. Nguyen, T. Noll, and M. Roveri. The COMPASS Approach: Correctness, Modelling and Performability of Aerospace Systems. In *28th Int. Conf. on Computer Safety, Reliability and Security (SAFECOMP 2009)*, volume 5775, pages 173–186. Springer, 2009.

[4] J. Cheng. Slicing concurrent programs - a graph-theoretical approach. In P. Fritzson, editor, *AADEBUG*, volume 749 of *Lecture Notes in Computer Science*, pages 223–240. Springer, 1993.

[5] J. Hatcliff, M. B. Dwyer, and H. Zheng. Slicing software for model construction. *Higher-Order and Symbolic Computation*, 13(4):315–353, 2000.

[6] G. J. Holzmann. *The SPIN Model Checker*. Addison-Wesley, 2003.

[7] M. Odenbrett. Explicit-State Model Checking of an Architectural Design Language using SPIN. Master's thesis, RWTH Aachen University, Germany, Mar. 2010. `http://www-i2.informatik.rwth-aachen.de/dl/noll/theses/odenbrett.pdf`.

[8] F. Tip. A survey of program slicing techniques. *J. Prog. Lang.*, 3(3), 1995.

[9] M. Weiser. Program slicing. In *ICSE*, pages 439–449, 1981.

# Model Checking with Edge Valued Decision Diagrams

Pierre Roux, École Normale Supérieure de Lyon, France. pierre.roux@ens-lyon.fr
Radu I. Siminiceanu, National Institute of Aerospace, Hampton, Virginia, USA. radu@nianet.org

**Abstract**

We describe an algebra of Edge-Valued Decision Diagrams (EVMDDs) to encode arithmetic functions and its implementation in a model checking library. We provide efficient algorithms for manipulating EVMDDs and review the theoretical time complexity of these algorithms for all basic arithmetic and relational operators. We also demonstrate that the time complexity of the generic recursive algorithm for applying a binary operator on EVMDDs is no worse than that of Multi-Terminal Decision Diagrams.

We have implemented a new symbolic model checker with the intention to represent in one formalism the best techniques available at the moment across a spectrum of existing tools. Compared to the CUDD package, our tool is several orders of magnitude faster.

## 1 Introduction

Binary decision diagrams (BDD) [3] have revolutionized the reachability analysis and model checking technology. Arithmetic decision diagrams [2], also called Multi-Terminal Binary Decision Diagrams (MTBDD) [8] are the natural extension of regular BDDs to arithmetic functions. They take advantage of the symbolic encoding scheme of BDDs, but functions with large co-domains do not usually have a very compact representation because there are less chances for suffixes to be shared.

Edge-valued decision diagrams have been previously introduced, but only scarcely used. An early version, the edge valued binary decision diagrams (EVBDD) [11], is particularly useful when representing both arithmetic and logic functions, which is the case for discrete state model checking. However, EVBDD have only been applied to rather obscure applications: computing the probability spectrum and the Reed-Muller spectrum of (pseudo)-Boolean functions.

Binary Moment Diagrams [4] were designed to overcome the limitations of BDDs/EVBDDs when encoding multiplier functions. However, their efficiency seems to be limited only to this particular type of functions. A new canonization rule for edge-valued decision diagrams enabling them to encode functions in $\mathbb{Z}\cup\{+\infty\}$ was introduced in [6] along with EVMDDs, an extension to multi-way diagrams (MDD) [9], but, again, this was applied to a very specific task, of finding minimum length counterexamples for safety properties. Later, EVMDDs have been also used for partial reachability analysis.

In this paper we first present a theoretical comparison between EVMDDs and MTMDDs for building the transition relation of discrete state systems before dealing with an implementation in a model checker along with state-of-the-art algorithms for state space construction.

## 2 Background

### 2.1 Discrete-state Systems

A discrete–state model is a triple $(S, S_0, T)$, where the discrete set $S$ is the *potential state space* of the model; the set $S_0 \subseteq S$ contains the *initial states*; and $T : S \to 2^S$ is the *transition function* specifying which states can be reached from a given state in one step, which we extend to sets: $T(X) = \bigcup_{i\in X} T(i)$.

We consider structured systems modeled as a collection of $K$ *submodels*. A (global) system state $i$ is then a $K$-tuple $(i_K,\dots,i_1)$, where $i_k$ is the *local state* for submodel $k$, for $K \geq k \geq 1$, and $S$ is given by $S_K \times \cdots \times S_1$, the cross–product of $K$ local state spaces $S_k$, which we identify with $\{0,\dots,n_k-1\}$ since

we assume that $S$ is finite. The *(reachable) state space* $R \subseteq S$ is the smallest set containing $S_0$ and closed with respect to $T$, i.e. $R = S_0 \cup T(S_0) \cup T(T(S_0) \cup \cdots = T^*(S_0)$. Thus, $R$ is the least fixpoint of function $X \mapsto S_0 \cup T(X)$.

### 2.2 Decision Diagrams

We discuss the extension of BDDs to integer variables, i.e., multi–valued decision diagrams (MDDs) [9]. We assume that the variables along any path from the root must follow the order $x_K, \ldots, x_1$. Ordered MDDs can be either *reduced* (no duplicate nodes and no node with all edges pointing to the same node, but edges possibly spanning multiple levels) or *quasi–reduced* (no duplicate nodes, and all edges spanning exactly one level), either form being *canonical*.

## 3 EVMDDs

**Definition 1.** *An* EVMDD *on a group* $(G,*)$, *is a pair* $A = \langle v,n \rangle$, *where* $v \in G$ *is the edge* value *also denoted as* $A$.val *and* $n$ *is a* node *also denoted* $A$.node*.*

*A* node $n$ *is either the unique terminal node* $\langle 0,e \rangle$ *where* $e$ *is the identity element of* $G$*, or a pair* $\langle k,p \rangle$ *where* $1 \leq k \leq K$ *and* $p$ *is an array of edges of size* $n_k$ *(cardinality of* $S_k$*). The first element of the pair will be denoted* $n$.level *and, when relevant, the i-th element in the array will be denoted by* $n[i]$.

**Definition 2.** *For a node* $n$ *with* $n$.level $= k$ *and* $(i_k,\ldots,i_1) \in S_k \times \cdots \times S_1$*, we define* $n(i_k,\ldots,i_1)$ *as* $n[i_k]$.val *if* $n[i_k]$.node.level $= 0$ *and* $n[i_k].\mathrm{val} * n[i_k].\mathrm{node}(i_{n[i_k].\mathrm{node.level}},\ldots,i_1)$ *otherwise.*

*The* function encoded *by an EVMDD A,* $f : S \to G, (i_K,\ldots,i_1) \mapsto A.\mathrm{val} * A.\mathrm{node}(i_{A.\mathrm{node.level}},\ldots,i_1)$ *is the repetitive application of law* $*$ *on the edge values along the path from the root to the terminal node, corresponding to arcs* $i_k$*, for* $K \geq k \geq 1$*:*

**Definition 3.** *A* canonical node *is either the terminal node or a node* $n$ *such that* $n[0]$.val $= e$.

*A* canonical *EVMDD contains only canonical nodes.*

It can be proved that any function $f$ has a unique canonical EVMDD representation [6].

Examples of graph representations of EVMDDs are given in Figure 1.

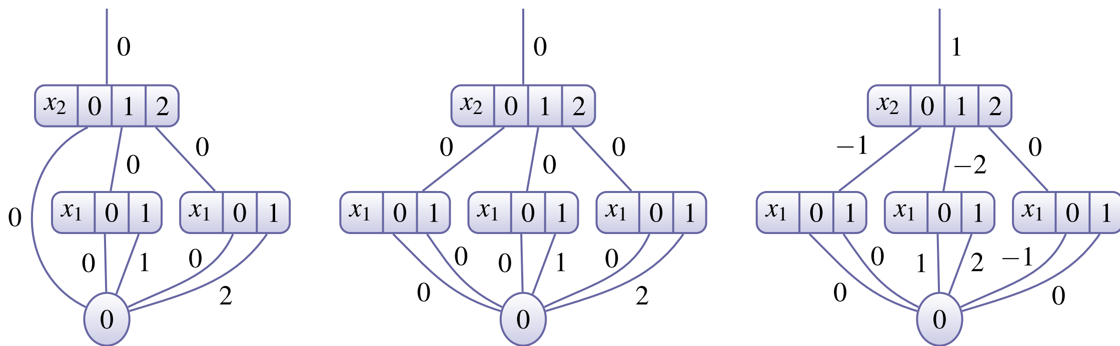


Figure 1: EVMDDs on $(\mathbb{Z},+)$ representing the same function $f : \{0,1,2\} \times \{0,1\} \to \mathbb{Z}, (x_2,x_1) \mapsto x_2 \cdot x_1$. The leftmost EVMDD is reduced while the others are quasi–reduced. The rightmost EVMDD is not canonical.

EVMDDs can be used when even the algebraic structure $G$ is not a group. For example, [6] offers a canonization rule for $\mathbb{N} \cup \{+\infty\}$. Also, $(\mathbb{Z},\times)$ that can be handled with the canonization rule "$\gcd\{n[i].\mathrm{val} \mid i \in S_{n.\mathrm{level}}\} = 1$ and $(n[0].\mathrm{val},\ldots,n[n_{n.\mathrm{level}}].\mathrm{val}) \geq_{\mathrm{lex}} 0$".

## 4 EVMDDs compared to MTMDDs

MTBDDs are commonly used in model checking to build the transition relation of discrete-state systems. In this section we show that EVMDDs are at least as suited to that purpose and oftentimes significantly better. In the following, we choose, without loss of generality, $(G,*) = (\mathbb{Z},+)$.

### 4.1 Space Complexity

**Theorem 1.** *For any function $f$, the number of nodes of the EVMDD representing $f$ is at most the number of nodes of the MTMDD representing the same function $f$.*[1]

### 4.2 Time complexity

Section 2 of [8] gives an algorithm to compute any binary operation on BDDs. The *apply* algorithm can be easily generalized to MDDs for any $n$-ary operator $\square_n$. It computes its result in time $\mathrm{O}\left(\prod_{i=1}^{n} |f_i|\right)$, where $|f_i|$ is the size (in nodes) of the MTMDD representing operand $i$.

Section 2.2 of [10] gives the equivalent *apply* algorithm for edge-valued decision diagrams.

**Theorem 2.** *The number of recursive calls of the generic* apply *algorithm for MTMDDs is equal to that for EVMDDs representing the same function [10].*

Hence, EVMDD computations are at least not worse than the MTMDD counterpart. However, particular operators $\square_n$ may enable much better algorithms on EVMDDs. Below is a synopsis of the basic algorithms to manipulate EVMDDs.

- Addition of constant $(f+c)$: $\mathrm{O}(1)$.
- Multiplication with scalar $(f \times c)$: $\mathrm{O}(|f|)$ [10].
- Addition $(f+g)$: $\mathrm{O}(|f|\,|g|)$ [10].
- Remainder and Euclidean Division: $\mathrm{O}(|f|c)$.
- Minimum and Maximum: $O(|f|)$.
- Relational Operator with constant $(f < c)$: not better in the worst case, but in practice the complexity can be improved, by using min and max.
- Relational Operators $(f < g)$: can be computed as $(f - g < 0)$.

### 4.3 Multiplication

As stated in [10], the result of a multiplication can have an EVMDD representation of exponential size in terms of the operands. For example, let $S$ be $\{0,1\}^K$, $f : (x_K,\ldots,x_1) \mapsto \sum_{k=2}^{K} x_k 2^{k-2}$ and $g : (x_K,\ldots,x_1) \mapsto x_1$, $f$ and $g$ both have an EVMDD representation with $K+1$ nodes whereas $f \cdot g$ has $2^K$ nodes. Therefore, we cannot expect to find an algorithm with better worst-case complexity. However, the following equation, coming from the decomposition of $\langle v,n\rangle$ in $v + \langle 0,n\rangle$ and $\langle v',n'\rangle$ in $v' + \langle 0,n'\rangle$

$$\langle v,n\rangle \times \langle v',n'\rangle = vv' + v\langle 0,n'\rangle + v'\langle 0,n\rangle + \langle 0,n\rangle \times \langle 0,n'\rangle$$

[1] All proofs and algorithms are given in a technical report [12], to appear.

suggests an alternative algorithm.

The first product is an integer multiplication done in constant time. The next two are multiplications by a constant done in $\mathrm{O}(|f|)$ and $\mathrm{O}(|g|)$, respectively. The last one is done through recursive calls. The first addition takes constant time, the second one takes $\mathrm{O}(|f|\,|g|)$ and produce a result of size at most $|f|\,|g|$, hence a cost of $\mathrm{O}(|f|\,|g|\,|fg|)$ for the last addition. The recursive function is called $\mathrm{O}(|f|\,|g|)$ times, hence a final complexity of $\mathrm{O}\left(|f|^2\,|g|^2\,|fg|\right)$.

Although we were unable to theoretically compare this algorithm to the generic *apply* algorithm, it seems to perform far better on practical cases.

# 5 Implementation

| Model size | Reachable states | CUDD (in s) | SMART (in s) | EVMDD (in s) |
|---|---|---|---|---|
| Dining philosophers | | | | |
| 100 | $4\times10^{62}$ | 11.42 | 1.49 | 0.03 |
| 200 | $2\times10^{125}$ | 3054.69 | 3.03 | 0.07 |
| 15000 | $2\times10^{9404}$ | — | — | 195.29 |
| Round robin mutual exclusion protocol | | | | |
| 40 | $9\times10^{13}$ | 4.44 | 0.44 | 0.08 |
| 100 | $2\times10^{32}$ | — | 2.84 | 1.17 |
| 200 | $7\times10^{62}$ | — | 20.02 | 9.14 |
| Slotted ring protocol | | | | |
| 10 | $8\times10^{9}$ | 1.16 | 0.19 | 0.01 |
| 20 | $2\times10^{20}$ | — | 0.71 | 0.04 |
| 200 | $8\times10^{211}$ | — | 412.27 | 25.97 |
| Kanban assembly line | | | | |
| 15 | $4\times10^{10}$ | 80.43 | 3.41 | 0.01 |
| 20 | $8\times10^{11}$ | 2071.58 | 8.23 | 0.02 |
| 400 | $6\times10^{25}$ | — | — | 74.89 |
| Knights problem | | | | |
| 5 | $6\times10^{7}$ | 1024.42 | 5.29 | 0.27 |
| 7 | $1\times10^{15}$ | — | 167.41 | 3.46 |
| 9 | $8\times10^{24}$ | — | — | 32.20 |
| Randomized leader election protocol | | | | |
| 6 | $2\times10^{6}$ | 4.22 | 8.42 | 0.86 |
| 9 | $5\times10^{9}$ | — | 954.81 | 18.89 |
| 11 | $9\times10^{11}$ | — | — | 109.25 |

Table 1: Execution times for building state space using our library or CUDD ("—" means "$>$ 1hour").

Symbolic model checkers, such as (Nu)SMV or SAL, are based on the library CUDD[1] which offers an efficient implementation of BDDs and MTBDDs. Our goal was to implement a new symbolic model checking library featuring EVMDDs for the transition relation construction and saturation[5] for state space generation. We also developed a basic model checking front-end to test the library and compare it to CUDD. Binaries and source code for both the EVMDD library and the model checker are available at `http://research.nianet.org/~radu/evmdd/`.

## 5.1 Encoding the Transition Relation

We represent the transition relation $T$ as a disjunction of events which is well suited for globally–asynchronous locally–synchronous systems, where each event encodes some local transition. To avoid the expensive coding of lot of identities, we use the *full-identity reduction* from [7].

## 5.2 State Space Construction

For state space construction, we use the Saturation algorithm [5] instead of the classical breadth first search exploration. This heuristic often gives spectacular improvements when building the state spaces of globally–asynchronous locally–synchronous systems. This is certainly the major source of improvement of our implementation over existing BDD libraries.

### 5.3 Experimental Results

Our new tool comprises 7K lines of ANSI-C code for the library and 4K lines for the simple model checker that provides a common interface to both our library and CUDD. Table 1 shows execution times for building the state space on a suite of classical models. Programs to generate all models can be found in the `examples` folder of our source code distribution.

We collected the results on a Linux machine with Intel Core 2 processor, 1.2GHz, 1.5GB of memory.

Note that using other existing tools, such as NuSMV or SAL on these models, we get execution times of the same order of magnitude as with the CUDD interface of our tool.

Compared to the first implementation of saturation algorithm [5] in the tool SMART, our new implementation is always several (up to a few dozens) times faster. This is due to both the encoding of the transition relation and our simple C implementation in comparison to the object-oriented C++ version.

## 6 Conclusions and Future Work

We have studied the advantages of the EVMDD data structure over the widely used MTBDDs for the construction of transition relations of finite state systems and implemented them in a library, along with state-of-the-art algorithms for state space generation. We obtained execution times several orders of magnitude faster than the CUDD library and classical algorithms, with a reduced memory usage enabling to handle extremely large systems. Future work should focus primarily on integrating our library into the SAL model checker.

Our results show that symbolic model checking remains an efficient technique for analyzing globally–asynchronous locally–synchronous systems and significant improvements are still possible.

## References

[1] Colorado University Decision Diagram library. http://vlsi.colorado.edu/~fabio/CUDD/.

[2] R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic decision diagrams and their applications. In *ICCAD*, pages 188–191, 1993.

[3] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35:677–691, 1986.

[4] R. E. Bryant and Y.-A. Chen. Verification of arithmetic functions with binary moment diagrams. Technical Report CS-94-160, CMU, 1994.

[5] G. Ciardo, G. Lüttgen, and R. Siminiceanu. Saturation: An efficient iteration strategy for symbolic state space generation. In *Proc. TACAS*, LNCS 2031, pages 328–342, Genova, Italy, Apr. 2001. Springer.

[6] G. Ciardo and R. Siminiceanu. Using edge-valued decision diagrams for symbolic generation of shortest paths. In *Proc. FMCAD*, LNCS 2517, pages 256–273, Portland, OR, USA, Nov. 2002. Springer.

[7] G. Ciardo and J. Yu. Saturation-based symbolic reachability analysis using conjunctive and disjunctive partitioning. In *Proc. CHARME*, October 2005.

[8] E. Clarke, M. Fujita, and X. Zhao. Application of multi-terminal binary decision diagrams. Technical report, IFIP WG 10.5 Workshop on Applications of the Reed-Muller Expansion in Circuit Design, 1995.

[9] T. Kam, T. Villa, R. Brayton, and A. Sangiovanni-Vincentelli. Multi-valued decision diagrams: theory and applications. *Multiple-Valued Logic*, 4(1–2):9–62, 1998.

[10] Y.-T. Lai, M. Pedram, and B. K. Vrudhula. Formal verification using edge-valued binary decision diagrams. *IEEE Trans. Comput.*, 45:247–255, 1996.

[11] Y.-T. Lai and S. Sastry. Edge-valued binary decision diagrams for multi-level hierarchical verification. In *DAC*, pages 608–613, 1992.

[12] P. Roux and R. Siminiceanu. Model checking with edge-valued decision diagrams. CR submitted, NASA, Langley Research Center, Hampton VA 23681-2199, USA, March 2010.

# Data-flow based Model Analysis

Christian Saad, Bernhard Bauer
Programming Distributed Systems Lab, University of Augsburg, Germany
{saad, bauer}@informatik.uni-augsburg.de

**Abstract**

The concept of (meta) modeling combines an intuitive way of formalizing the structure of an application domain with a high expressiveness that makes it suitable for a wide variety of use cases and has therefore become an integral part of many areas in computer science. While the definition of modeling languages through the use of meta models, e.g. in UML, is a well-understood process, their validation and the extraction of behavioral information is still a challenge.
In this paper we present a novel approach for dynamic model analysis along with several fields of application. Examining the propagation of information along the edges and nodes of the model graph allows to extend and simplify the definition of semantic constraints in comparison to the capabilities offered by e.g. the Object Constraint Language. Performing a flow-based analysis also enables the simulation of dynamic behavior, thus providing an "abstract interpretation"-like analysis method for the modeling domain.

## 1 Introduction and Motivation

Meta modeling is an easy and concise way to formalize the structure of an application domain. Today, it is widely spread in computer science, most notably in the field of software engineering where the internal design of software is often described using models. Approaches like the Model-driven Architecture (MDA) improve the development process e.g. through automated code generation. Arguably the most important industrial standard in this area is the Unified Modeling Language (UML) which in turn is based on the Meta-Object Facility (MOF) framework[1].

However, the abstract syntax as defined by the meta model is often not sufficient to guarantee well-formedness. Complex restrictions that cannot be expressed through the syntax are known as static semantics. To a certain degree, they can be implemented using the Object Constraint Language (OCL), thereby extending the expressiveness of the modeling language. Due to the static nature of OCL, it is not capable of validating dynamic properties that are highly dependent on the context in which the elements appear, e.g. the correct nesting of parallel paths in activities.

An advantage of using models as means of specifying systems is their versatility. In addition to code generation or model transformation, their use can also be leveraged by extracting implicitly contained information through an evaluation of model elements and their mutual relations. This may include metrics applicable for the modeling domain, e.g. Depth of Inheritance Tree (DIT) or Number of Children (NOC), or the compliance with a set of predefined modeling guidelines. Depending on the range of application, extracting knowledge about the dynamic properties of a model may even allow to identify inactive fragments in a way similar to dead code elimination in the translation of software programs. To statically deduce information about the dynamic behavior of a model, e.g. to calculate the valid execution paths of workflows and thereby approximating their runtime behavior, can be considered an abstract interpretation of dynamic semantics.

Current methods are usually not capable of performing a static analysis of dynamic aspects in order to express context-sensitive well-formedness rules or to simulate workflows. The approach discussed in this paper is designed to overcome these limitations by extending the static character of OCL constraints with a dynamic flow analysis, thus offering a powerful and generically applicable method for model validation and simulation. It is based on the data-flow analysis (DFA) technique, a well-understood formalism used in compiler construction for deriving optimizations from a program's control flow graph. The adaption of the DFA algorithm is simplified by the conceptual similarities between the areas of compiler construction and (meta) modeling: Both rely on the definition of domain specific languages (DSL) which operate on (at least) two layers of abstraction.

The concept of using DFA for model analysis has been introduced in [4]. In this paper we extend and update this description along with an evaluation of the advantages, possible shortcomings and proposed solutions based on the experience gained up to now. Additionally, we discuss several use cases which are in the focus of our research and present the current status of our implementation.

This paper is structured as follows: The basic concept of data-flow analysis for models along with a definition of how it can be specified and a corresponding evaluation algorithm is detailed in Section 2. Several use cases are presented in Section 3 before we give a summary of the concepts described in this paper and an outlook on future developments.

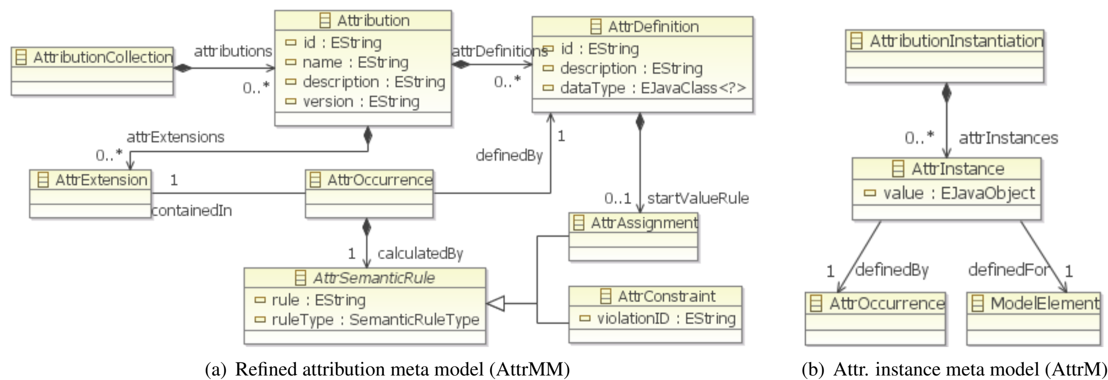
[1] Specifications are available at OMG's website: `http://www.omg.org/technology/documents/modeling_spec_catalog.htm`

# 2 Data-Flow Analysis on Models

As outlined in our previous literature, the conceptual similarities between context-free grammars, which form the basis for many algorithms in compiler construction, and MOF's multi-layered architecture of (meta) models allow to implement a DFA-like analysis method for the modeling domain. We aligned the four MOF levels (M3-M0) with the abstraction layers of attribute grammars[2] and devised a model-based attribution technique for assigning data-flow equations to meta model classes and evaluating them for arbitrary models.

It was proposed that the attributes containing the semantic rules (the equivalent to the data-flow equations) should be given a type definition consisting of a name, a data type and an initial value. Occurrences of these definitions can then be assigned to M2 classes along with a rule that will calculate the results for M1 objects depending on the values available at other attributes. If a model shall be analyzed according to a given attribution, these occurrences have to be instantiated at the according model objects. An evaluation algorithm is then responsible for executing the rules in a valid order ensuring that required input arguments are available at the time of execution. This process is repeated until all attribute values are stable (the fix-point). Several iterations of (re)evaluating the attribute instances may be necessary until the results have been propagated along all (cyclic) paths. To keep in line with the notion that everything is a model, a meta model describing the structure of the attribution was given.

## 2.1 Refined Definition




(a) Refined attribution meta model (AttrMM) (b) Attr. instance meta model (AttrM)

Figure 1: Meta models for data flow definitions

Figure 1(a) shows the refined attribution meta model *AttrMM*: *AttributionCollections* serve as containers for a set of *Attributions* which consist in turn of *AttributeDefinitions* and *AttributeExtensions*. *AttributeDefinitions* have to be assigned a *dataType*. Their initial value is returned by an *AttrSemanticRule* of the type *AttrAssignment*[3]. Occurrences (*AttrOccurrence*) can be attached to classes in the original meta model through *AttributeExtensions*. They possess an *AttrSemanticRule* which calculates their iteration value.

This design has the advantage of introducing dependencies only from the attribution to the meta model but not the other way round. This way, attributes can be defined and stored separate from the meta model, an important aspect when integrating with existing tools.

To ensure the compatibility with existing meta modeling frameworks, the resulting attribution instances were given their own meta model *AttrM* which can be seen in Figure 1(b). Each *AttributeInstance* has a *value* slot for storing intermediate and final evaluation results. Attribute instances representing the results are connected to their defining *AttrOccurrences* in the attribution (and thus to the attributed meta model class) as well as to the concrete model object to whose meta class the corresponding *AttributeExtension* was assigned.

An important aspect of modeling is the concept of generalization. Therefore, when creating attribute instances and attaching them to model elements, the generalization hierarchy of the meta model has to be considered, i.e. an attribute connected to class $A$ should implicitly be available at instances of subclasses of $A$. Also, in compliance with the MOF standard, support for the redefinition of attributes should be provided. This means if two *AttrOccurrences* $O_A$ and $O_B$ of the same *AttributeDefinition* $O$ were assigned to classes $A$ and $B$ and $B$ is a subclass of $A$, then $O_B$ overrides $O_A$ at all instances of $B$.

[2]It was shown that, while attribute grammars can be used to define data-flow equations, in their original form they are too restrictive and do not fit seamlessly into the modeling domain. Therefore, a model-based solution was chosen that drops most of these limitations at the cost of a slightly more complex - but also more versatile - evaluation algorithm.

[3]An assignment returns a value of the specified data type while a constraint just evaluates to "true" or "false", "false" representing a detected error in the model indicated to the user by the given *violationID*.

We assumed that the input dependencies, i.e. the occurrences whose instance values are required as arguments at other instances, will be explicitly specified by the user. However, this has proven to be impractical for two reasons: Aside from complicating the design of a DFA analysis, declaring relations on the meta layer has the drawback of introducing unnecessary instance dependencies if only a subset is actually needed. Instead, the evaluation algorithm is now responsible for requesting the calculation of required input for attribution rules during their evaluation.

### 2.2 Attribute Evaluation

DFA algorithms like the work-list approach assume that the output relationships are known beforehand in order to update the set of depending variables after each iteration. Since we are working with dynamic dependencies, another approach was chosen for evaluating an attribution for a given model: First, attributes must be instantiated in accordance to the given semantics and initialized with their start value. Then, the semantic rules have to be invoked in a valid order, backtracking their dependencies during execution, storing results and passing them as input arguments to the calling rules. This is repeated until the fix-point of the analysis has been reached.

The dependencies between attribute instances, stemming from their use as input arguments of semantic rules, impose the structure of a directed acyclic graph originating from a single root element on the execution order. This graph can be viewed as the result of a depth-first search along the input relations of the attributes, however in the case of cyclic dependencies the target $x$ of the responsible back edge $y \rightarrow x$ is now substituted by a newly created virtual node $x'$. The resulting tree-like representation (which may still contain forward and cross edges) is referred to as *dependency chain*. After each bottom-up evaluation run the virtual nodes are updated with the new value available at their reference node, i.e. the result at $x$ is transferred to $x'$. The evaluation is repeated until the value at each virtual node equals the result at its reference node which means that the fix-point has been reached.

As an example, consider that we need to determine the set of predecessors in a flow graph lying on a *direct path* from the start node while omitting optional routes. This can be achieved by adding the local node to an intersection of the same (recursively calculated) sets at preceding nodes. Nodes with no predecessors then correspond to leaves in the dependency chain while cyclic paths induce back edges and therefore the creation of virtual nodes.

A model may contain multiple dependency chains while at the same time a single chain may also split up into several chains if a root attribute instance was chosen that is not the absolute root, i.e. other attributes depend on it.

Because this algorithm executes the semantic rules according to their dependencies, the amount of redundant calculations is negligible. Nevertheless, there is still room for optimization, e.g. by calculating starting values only for leaves, through isolated recomputation of cycles or by parallelizing the evaluation. Also, a formal and practical evaluation of the algorithm's performance is necessary.

## 3 Use Cases

To demonstrate the applicability of the approach, we now give several use case examples which can be implemented using the model-based DFA technique: Section 3.1 demonstrates common compiler construction analysis while 3.2 and 3.3 deal with the domain of business processes (but are also applicable e.g. for UML activity diagrams).

Additional application fields which are currently under evaluation include the extraction of metrics in the area of model-driven testing and the definition of OCL constraints that avoid complex navigation statements and can be therefore more easily adjusted if the meta model changes.

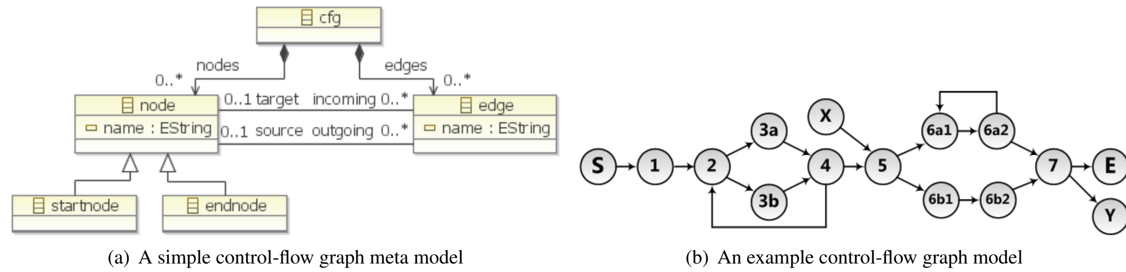
### 3.1 Analysis of Control-Flow Graphs

To simulate a traditional data-flow analysis we have implemented a meta model for control-flow graphs (CFG) (cf. Figure 2(a)) and instantiated the CFG model that can be seen in Figure 2(b) which will serve as an example.

An attribution model was created containing the following attributes for *node* along with their assignment rules based on an extended OCL syntax which allows to request attribute values by triggering the evaluator module:

```
is_reachable: self.incoming.source.is_reachable()→includes(true)
is_live: self.outgoing.target.is_live()→includes(true)
all_predecessors: self.incoming.source.name→ union(self.incoming.source.all_predecessors())→asSet()
scc_id: let self_pred : Set(String) = self.all_predecessors()→including(self.name) in
    if (self.incoming.source.all_predecessors()→asSet()=self_pred) then self_pred→hashCode() else 0 endif)
scc_nodes: if (not(self.scc_id() = 0)) then self.incoming.source→collect(predNode : node |
    if (predNode.scc_id()=self.scc_id())
    then predNode.scc_nodes() else Set{} endif) →flatten()→asSet()→including(self.name) else Set{} endif
```

While *is_reachable* determines if a node can be reached from the start node by checking if at least one predecessor is reachable (the start node has a distinct rule setting *is_reachable* to true), *is_live* checks if a path to the end node




(a) A simple control-flow graph meta model

(b) An example control-flow graph model

Figure 2: Control-flow graph example

exists. As expected, the evaluation returns that $X$ is not reachable and $Y$ is not live. Defining the semantic rules as constraints, corresponding exceptions are thrown indicating the type and the location of the error.

The list of transitive predecessor nodes is calculated through *all_predecessors* by creating the union of the *all_predecessors* attributes at the source nodes of incoming edges and adding to it the set of names of these nodes.

The attribute *scc_id* assigns a unique ID to each node that is part of a cyclic flow. This is accomplished by comparing the union of the *all_predecessors* attributes at preceding nodes to the local value of this attribute. If both sets contain the same elements, an ID is generated from the hash codes of the nodes that take part in the cycle.

Now, using the *scc_id*, the semantic rule for *scc_nodes* is able to determine which nodes take part in a cycle by building the set of predecessors with the same *scc_id*. Figure 3 shows the final values for this attribute which can again be used as input for the algorithms presented below.

Calculating *scc_nodes* requires 220-310 rule executions using an unoptimized evaluation algorithm. Once the OCL environment has been initialized, overall execution takes about 110ms on a standard desktop computer. Implementing the rules in Java leads to more verbose definitions but reduces the time required to about 50ms.

## 3.2 Business Process Decomposition

Decomposing a business process into a hierarchical layout of single-entry-single-exit (SESE) components is a common requirement, e.g. allowing to translate BPMN processes to BPEL or validate workflow graphs (cf. 3.3). In [3], the authors describe a decomposition algorithm based on token flow analysis that is able to handle cyclic

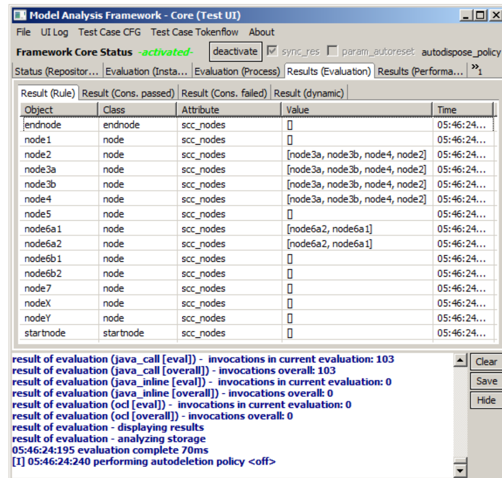
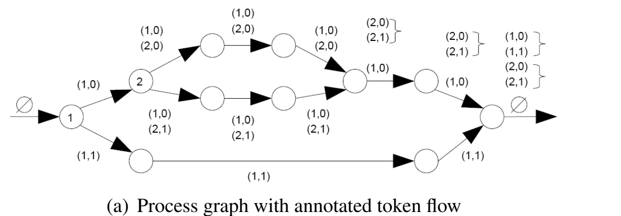
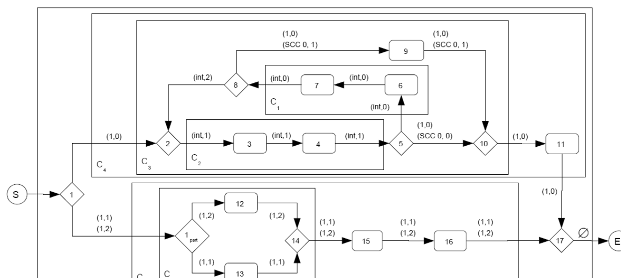



Figure 3: Evaluation result: Nodes that are part of cycles




(a) Process graph with annotated token flow




(b) A business process hierarchically divided into SESE components

Figure 4: Decomposition of (business) processes [3]

graphs and to classify the detected components. Tokens are created at branches and are propagated along the control-flow as can be seen in Figure 4(a). In a second step, the tokens originating from the same vertex converge and are removed (indicated by curly brackets). Similar token labelings identify the SESE components although unambiguous classification and cyclic paths require some additional handling. Figure 4(b) shows a decomposition.

Since the algorithm is based on the creation and propagation of information, it is an ideal use case for a generic DFA-based approach as opposed to a proprietary implementation. Several steps of this algorithm have been realized (including the cycle-detection presented above) with the goal of a comparison to the authors' implementation.

### 3.3 BP Validation and Transformation

Using the method presented in [5], the soundness of (a)cyclic business processes - i.e. the absence of local deadlocks and lack of synchronization - can be tested in linear time. This is accomplished by traversing the SESE hierarchy in bottom-up direction, applying heuristics to categorize each sub-graph according to the structure of its elements. This way, if an error is found, the SESE context in which it appears allows to track down its approximate location. Implemented using DFA definitions, this algorithm can be easily integrated into the decomposition process in order to perform the validation already during the component identification phase.

Making use of the SESE decomposition also enables to transform graph-oriented BPMN diagrams (Business Process Modeling Notation) to block-oriented BPEL code (Business Process Execution Language). The author of [1] describes data-flow equations that, if computed for the SESE fragments, yield information on how the respective component can be translated to BPEL code. Since this algorithm is already defined in the form of a data-flow analysis, the implementation using the model analysis approach is straightforward.

## 4 Conclusions and Future Investigations

In this paper we have described a refined version of our data-flow based approach to model analysis that introduces the notion of DFA to the modeling domain and is built upon widely-accepted standards and definitions.

To the best of our knowledge there exists no comparable methodology in the context of model analysis although different DFA techniques have been considered for use in the modeling domain: The authors of [2] discuss the advantages of control-flow information for the testing phase of the software development process. A concrete use case is presented in [6], using flow-analysis to derive definition/use relationships between actions in state machines.

The presented formal method completes common techniques like OCL with the possibility of describing (cyclic) information flows in the model graph based on local propagation and (re)calculation of attribute values. This allows a more robust definition of semantic constraints since complex navigation statements which are a common drawback of OCL can be avoided. Instead, the required information can be "transported" to where it is needed. Aside from validation scenarios, the ability to extract context-sensitive data enables to analyze dynamic aspects of models, e.g. valid execution paths in control-flows or the SESE components that make up a business process definition. This way, model-based DFA constitutes a generic and versatile "programming-language" for implementing a wide variety of algorithms that would otherwise each require a proprietary definition.

To verify the feasibility of this approach, the Model Analysis Framework (MAF) project was created to serve as basis for performance tests under realistic settings and allow to evaluate future extensions of the presented definitions and algorithms. It was designed to act as a parametrizable and modular research platform that for example allows to choose between different inheritance semantics and evaluation algorithms as well as at the same time being suited for productive use. MAF, which is based on the Eclipse Modeling Framework (EMF) and the Eclipse OCL interpreter, will soon be available at `http://code.google.com/p/model-analysis-framework/`.

Aside from formalizing and improving the evaluation algorithm to achieve a better theoretical and practical performance the main focus in the ongoing research is on the implementation and evaluation of further use cases.

## References


[1] Luciano García-Bañuelos. Pattern Identification and Classification in the Translation from BPMN to BPEL. *OTM 08: Proceedings of the OTM 2008 Confederated International Conferences, CoopIS, DOA, GADA, IS, and ODBASE 2008*, pages 436–444, 2008.

[2] Vahid Garousi, Lionel Bri, and Yvan Labiche. Control Flow Analysis of UML 2.0 Sequence Diagrams. 2005.

[3] Mathias Götz, Stephan Roser, Florian Lautenbacher, and Bernhard Bauer. Token Analysis of Graph-Oriented Process Models. *New Zealand Second International Workshop on Dynamic and Declarative Business Processes (DDBP), in conjunction with the 13th IEEE International EDOC Conference (EDOC 2009)*, September 2009.

[4] Christian Saad, Florian Lautenbacher, and Bernhard Bauer. An Attribute-based Approach to the Analysis of Model Characteristics. *Proceedings of the First International Workshop on Future Trends of Model-Driven Development in the context of ICEIS'09*, 2009.

[5] Jussi Vanhatalo, Hagen Völzer, and Frank Leymann. Faster and More Focused Control-Flow Analysis for Business Process Models Through SESE Decomposition. In *ICSOC '07: Proceedings of the 5th international conference on Service-Oriented Computing*, pages 43–55, Berlin, Heidelberg, 2007. Springer-Verlag.

[6] T. Waheed, M.Z.Z. Iqbal, and Z.I. Malik. Data Flow Analysis of UML Action Semantics for Executable Models. *Lecture Notes in Computer Science*, 5095:79–93, 2008.

# Author Index


Arcaini, Paolo .................... 4

Bauer, Bernhard ................ 227
Bjørner, Nikolaj ................. 1
Boldo, Sylvie ................... 14
Bollin, Andreas ................. 24
Brat, Guillaume ............ 2, 171
Butler, Michael ................ 182
Butler, Ricky ................... 34

Catano, Nestor .................. 47
Chaki, Sagar .................... 57
Ciardo, Gianfranco ............. 202
Cook, William R. ................ 97

Dowek, Gilles ................... 34

Endres, Erik ................... 212

Fu, Xiang ....................... 67

Gargantini, Angelo ............... 4
Gurfinkel, Arie ................. 57

Hagen, George ................... 34
Harrison, John ................... 3
Heimdahl, Mats ................. 161
Huang, Xiaowan .................. 77

Knight, John ................... 192

López, Eduardo Rafael ........... 87
Lemoine, Michel ................. 87
Lester, David .................. 139
Li, Chung-Chih .................. 67

Müller, Christian .............. 212
Maddalon, Jeffrey ............... 34
Moura, Arnaldo V. .............. 109
Muñoz, César .................... 34

Narkawicz, Anthony .............. 34
Nedunuri, Srinivas .............. 97
Nguyen, Thi Minh Tuyen .......... 14
Nguyen, Viet Yen ............... 217
Noll, Thomas ................... 217

Odenbrett, Max ................. 217

Pedrosa, Lehilton L. C. ........ 109
Pilotto, Concetta .............. 119
Price, Petra ................... 129

Rajan, Ajitha .................. 161
Rezazadeh, Abdolbaghi .......... 182
Riccobene, Elvinia ............... 4
Richards, Dominic .............. 139
Roux, Pierre ................... 222
Roy, Pritam .................... 149

Saad, Christian ................ 227
Shadrin, Andrey ................ 212
Shankar, Natarajan ............. 149
Siminiceanu, Radu .......... 47, 222
Singh, Anu ...................... 77
Smith, Douglas R. ............... 97
Smolka, Scott A. ................ 77
Staats, Matt ................... 161

Thompson, Sarah ................ 171
Turgeon, Greg .................. 129
Tverdyshev, Sergey ............. 212

Venet, Arnaud .................. 171

Whalen, Michael ................ 161
White, Jerome .................. 119

Yeganefard, Sanaz .............. 182
Yin, Xiang ..................... 192

Zhao, Yang ..................... 202

| REPORT DOCUMENTATION PAGE | *Form Approved* *OMB No. 0704-0188* |
|---|---|

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.
**PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.**

| **1. REPORT DATE** *(DD-MM-YYYY)* | **2. REPORT TYPE** | **3. DATES COVERED** *(From - To)* |
|---|---|---|
| 01- 04 - 2010 | Conference Publication | |

| **4. TITLE AND SUBTITLE** | **5a. CONTRACT NUMBER** |
|---|---|
| Proceedings of the Second NASA Formal Methods Symposium | |
| | **5b. GRANT NUMBER** |
| | |
| | **5c. PROGRAM ELEMENT NUMBER** |
| | |
| **6. AUTHOR(S)** | **5d. PROJECT NUMBER** |
| Muñoz, César | |
| | **5e. TASK NUMBER** |
| | |
| | **5f. WORK UNIT NUMBER** |
| | 411931.02.51.07.01 |

| **7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)** | **8. PERFORMING ORGANIZATION REPORT NUMBER** |
|---|---|
| NASA Langley Research Center<br>Hampton, VA 23681-2199 | L-19844 |

| **9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)** | **10. SPONSOR/MONITOR'S ACRONYM(S)** |
|---|---|
| National Aeronautics and Space Administration<br>Washington, DC 20546-0001 | NASA |
| | **11. SPONSOR/MONITOR'S REPORT NUMBER(S)** |
| | NASA/CP-2010-216215 |

**12. DISTRIBUTION/AVAILABILITY STATEMENT**

Unclassified - Unlimited
Subject Category 59
Availability: NASA CASI (443) 757-5802

**13. SUPPLEMENTARY NOTES**

An electronic version can be found at http://ntrs.nasa.gov

**14. ABSTRACT**

This publication contains the proceedings of the Second NASA Formal Methods Symposium sponsored by the National Aeronautics and Space Administration and held in Washington D.C. April 13-15, 2010.

**15. SUBJECT TERMS**

Formal methods; Formal model; Hardware assurance; Life critical; Model checking; Safety critical; Software assurance; Theorem proving; Verification

| **16. SECURITY CLASSIFICATION OF:** | | | **17. LIMITATION OF ABSTRACT** | **18. NUMBER OF PAGES** | **19a. NAME OF RESPONSIBLE PERSON** |
|---|---|---|---|---|---|
| **a. REPORT** | **b. ABSTRACT** | **c. THIS PAGE** | | | STI Help Desk (email: help@sti.nasa.gov) |
| | | | | | **19b. TELEPHONE NUMBER** ***(Include area code)*** |
| U | U | U | UU | 244 | (443) 757-5802 |

**Standard Form 298** (Rev. 8-98)
Prescribed by ANSI Std. Z39.18