NASA/CP-2009-215407

Proceedings of the First NASA Formal Methods Symposium

Edited by

Ewen Denney
Dimitra Giannakopoulou
Corina S. Păsăreanu

NASA Ames Research Center

April 2009

NASA/CP-2009-215407

Proceedings of the First NASA Formal Methods Symposium

Edited by

Ewen Denney
Dimitra Giannakopoulou
Corina S. Păsăreanu

NASA Ames Research Center

National Aeronautics and
Space Administration

Ames Research Center
Moffett Field, California

April 2009

Available from:

NASA Center for AeroSpace Information
7121 Standard Drive
Hanover, MD 21076-1320
(301) 621-0390

National Technical Information Service
5285 Port Royal Road
Springfield, VA 22161
(703) 487-4650

Preface

This NASA conference publication contains the proceedings of the First NASA Formal Methods Symposium (NFM 2009), held at the NASA Ames Research Center, in Moffett Field, CA, USA, on April 6 – 8, 2009.

NFM 2009 is a forum for theoreticians and practitioners from 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 is a new annual event intended to highlight the state of formal methods' art and practice. It follows the earlier Langley Formal Methods Workshop series and aims to foster collaboration between NASA researchers and engineers, as well as the wider aerospace, safety-critical and formal methods communities.

The specific topics covered by NFM 2009 included but were not limited to: formal verification, 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.

We considered two types of papers: regular papers describe fully developed work and complete results, and short papers describe interesting work in progress and/or preliminary results. Both categories must describe original work that has not been published elsewhere. We received 47 submissions (26 long papers and 19 short papers) out of which 22 were accepted (14 long, 8 short). All submissions went through a rigorous reviewing process, where each paper received a minimum of 3 reviews. The program selection was performed through an electronic Program Committee meeting.

In addition to the refereed papers, the symposium featured five invited talks given by Ed Clarke (CMU, Turing Award 2007) on *Model Checking – My 27-year Quest to Overcome the State Explosion Problem*, Bill Othon (NASA JSC) on *Applying Formal Methods to NASA Projects: Transition from Research to Practice*, Leslie Lamport (MSR) on *TLA+: Whence, Wherefore, and Whither*, Todd Farley (NASA Ames) on *Formal Methods Applications in Air Transportation*, and John O'Leary (Intel) on *TITLE*. The program also included a panel discussion on *Formal Methods meet NASA needs* that was chaired by Mats Heimdahl (U. Minnesota).

We would like to thank the program committee members and the external reviewers for their contribution in paper selection, and the NFM Organizing Committee for its support in setting up this event. Thanks also go to Allen Dutra, Domenico Bianculli, and Chris Fattarsi for their help with the NFM 2009 local organization and to Geoff Sutcliffe for help with the EasyChair style files. Finally, we thank SGT, in particular Larry Markosian, and CMU West, in particular Jim Morris, for their support.

The NFM 2009 website can be found at `http://ti.arc.nasa.gov/event/nfm09/`.

April 2009

Ewen Denney
Dimitra Giannakopoulou
Corina Păsăreanu

NFM 2009 Chairs

Conference Organization

General Chairs

Ewen Denney, SGT/NASA Ames
Dimitra Giannakopoulou, CMU/NASA Ames
Corina S. Păsăreanu, CMU/NASA Ames

Program Committee

Gilles Barthe, IMDEA, Madrid
Guillaume Brat, NASA Ames
Ricky Butler, NASA Langley
Charles Consel, INRIA, Bordeaux
Krzysztof Czarnecki, U. Waterloo
Luca de Alfaro, UC Santa Cruz
Ben Di Vito, NASA Langley
Matt Dwyer, U. Nebraska
Martin Feather, JPL
Klaus Havelund, JPL
Mats Heimdahl, U. Minnesota
Gerard Holzmann, JPL
John Kelly, NASA HQ
Mike Lowry, NASA Ames
John Matthews, Galois Inc.
César Muñoz, NASA Langley
John Penix, Google
Kristin Yvonne Rozier, NASA Ames
Wolfram Schulte, Microsoft Research
Koushik Sen, UC Berkeley
Natarajan Shankar, SRI
Doug Smith, Kestrel Institute
Mike Whalen, Rockwell Collins

Organizing Committee

Ewen Denney, NASA Ames
Ben Di Vito, NASA Langley
Dimitra Giannakopoulou, NASA Ames
Klaus Havelund, JPL
Gerard Holzmann, JPL
César Muñoz, NASA Langley
Corina S. Păsăreanu, NASA Ames
James Rash, NASA Goddard
Kristin Yvonne Rozier, NASA Ames

External Reviewers

Michal Antkiewicz
Thiago Bartolomei
Jacob Burnim
Zinovy Diskin
Alwyn Goodloe
Ethan Jackson
Nicholas Jalbert
Pallavi Joshi
Sudeep Juvekar
Herman Lee
Chang-Seo Park
Christos Stergiou

Table of Contents

Invited Talks

Model Checking: Techniques and Applications

Symbolic Execution and Testing

Design and Specification

Analysis: Evaluations and New Developments (Short Papers)

Deductive Verification

Modeling and Synthesis (Short Papers)

Model Checking – My 27-year Quest to Overcome the State Explosion Problem

Ed Clarke
Computer Science Department, Carnegie Mellon University
Pittsburgh, PA

Turing Award 2007

Abstract

Model Checking is an automatic verification technique for state-transition systems that are finite-state or that have finite-state abstractions. In the early 1980's in a series of joint papers with my graduate students E.A. Emerson and A.P. Sistla, we proposed that Model Checking could be used for verifying concurrent systems and gave algorithms for this purpose. At roughly the same time, Joseph Sifakis and his student J.P. Queille at the University of Grenoble independently developed a similar technique. Model Checking has been used successfully to reason about computer hardware and communication protocols and is beginning to be used for verifying computer software. Specifications are written in temporal logic, which is particularly valuable for expressing concurrency properties. An intelligent, exhaustive search is used to determine if the specification is true or not. If the specification is not true, the Model Checker will produce a counterexample execution trace that shows why the specification does not hold. This feature is extremely useful for finding obscure errors in complex systems. The main disadvantage of Model Checking is the state-explosion problem, which can occur if the system under verification has many processes or complex data structures. Although the state-explosion problem is inevitable in worst case, over the past 27 years considerable progress has been made on the problem for certain classes of state-transition systems that occur often in practice. In this talk, I will describe what Model Checking is, how it works, and the main techniques that have been developed for combating the state explosion problem.

E. Denney, D. Giannakopoulou, C.S. Păsăreanu (eds.); The First NASA Formal Methods Symposium, pp. 1-1

Applying Formal Methods to NASA Projects: Transition from Research to Practice

Bill Othon
NASA Johnson Space Center
Houston, TX

Abstract

NASA project managers attempt to manage risk by relying on mature, well-understood process and technology when designing spacecraft. In the case of crewed systems, the margin for error is even tighter and leads to risk aversion. But as we look to future missions to the Moon and Mars, the complexity of the systems will increase as the spacecraft and crew work together with less reliance on Earth-based support. NASA will be forced to look for new ways to do business. Formal methods technologies can help NASA develop complex but cost effective spacecraft in many domains, including requirements and design, software development and inspection, and verification and validation of vehicle subsystems. To realize these gains, the technologies must be matured and field-tested so that they are proven when needed. During this discussion, current activities used to evaluate FM technologies for Orion spacecraft design will be reviewed. Also, suggestions will be made to demonstrate value to current designers, and mature the technology for eventual use in safety-critical NASA missions.

E. Denney, D. Giannakopoulou, C.S. Păsăreanu (eds.); The First NASA Formal Methods Symposium, pp. 2-2

TLA+: Whence, Wherefore, and Whither

Leslie Lamport
Microsoft Research

Abstract

The evolution of my ideas on specification and verification, and how they led to the TLA+ specification language. What is good and bad about TLA+. A brief description of the next version of TLA+ and its tools.

E. Denney, D. Giannakopoulou, C.S. Păsăreanu (eds.); The First NASA Formal Methods Symposium, pp. 3-3

Formal Methods Applications in Air Transportation

Todd Farley
NASA Ames Research Center
Moffett Field, CA

Abstract

The U.S. air transportation system is the most productive in the world, moving far more people and goods than any other. It is also the safest system in the world, thanks in part to its venerable air traffic control system. But as demand for air travel continues to grow, the air traffic control system's aging infrastructure and labor-intensive procedures are impinging on its ability to keep pace with demand. And that impinges on the growth of our economy.

Air traffic control modernization has long held the promise of a more efficient air transportation system. Part of NASA's current mission is to develop advanced automation and operational concepts that will expand the capacity of our national airspace system while still maintaining its excellent record for safety. It is a challenging mission, as efforts to modernize have, for decades, been hamstrung by the inability to assure safety to the satisfaction of system operators, system regulators, and/or the traveling public.

In this talk, we'll provide a brief history of air traffic control, focusing on the tension between efficiency and safety assurance, and the promise of formal methods going forward.

E. Denney, D. Giannakopoulou, C.S. Păsăreanu (eds.); The First NASA Formal Methods Symposium, pp. 4-4

Theorem Proving in Intel Hardware Design

John O'Leary
Strategic CAD Labs, Intel Corporation

Abstract

For the past decade, a framework combining model checking (symbolic trajectory evaluation) and higher-order logic theorem proving has been in production use at Intel. Our tools and methodology have been used to formally verify execution cluster functionality (including floating-point operations) for a number of Intel products, including the Pentium® 4 and Core™i7 processors. Hardware verification in 2009 is much more challenging than it was in 1999 – today's CPU chip designs contain many processor cores and significant firmware content. This talk will attempt to distill the lessons learned over the past ten years, discuss how they apply to today's problems, outline some future directions.

E. Denney, D. Giannakopoulou, C.S. Păsăreanu (eds.); The First NASA Formal Methods Symposium, pp. 5-5

Building a Formal Model of a Human-interactive System: Insights into the Integration of Formal Methods and Human Factors Engineering

Matthew L. Bolton [*]
University of Virginia
Charlottesville, VA, United States of America
mlb4b@virginia.edu

Ellen J. Bass[†]
University of Virginia
Charlottesville, VA, United States of America
ejb4n@virginia.edu

Abstract

Both the human factors engineering (HFE) and formal methods communities are concerned with finding and eliminating problems with safety-critical systems. This work discusses a modeling effort that leveraged methods from both fields to use model checking with HFE practices to perform formal verification of a human-interactive system. Despite the use of a seemingly simple target system, a patient controlled analgesia pump, the initial model proved to be difficult for the model checker to verify in a reasonable amount of time. This resulted in a number of model revisions that affected the HFE architectural, representativeness, and understandability goals of the effort. If formal methods are to meet the needs of the HFE community, additional modeling tools and technological developments are necessary.

1 Introduction

Formal methods have proven themselves useful for finding problems in safety-critical systems that could lead to undesirable, dangerous, or disastrous system conditions. The traditional use of formal methods has been to find problems in a system's automation under different operating and/or environmental conditions. However, human operators control a number of safety critical systems and contribute to unforeseen problems. For example, human behavior has contributed to between 44,000 and 98,000 deaths nationwide every year in medical practice [12], 74% of all general aviation accidents [13], and a number of high profile disasters such as the Three Mile Island and Chernobyl accidents [14]. Human factors engineering (HFE) focuses on understanding human behavior and applying this knowledge in the design of systems that depend on human interaction: making systems easier to use while reducing errors and/or allowing recovery from them [17, 19].

Because both HFE and formal methods are concerned with the engineering of robust systems that will not fail under realistic operating conditions, researchers have leveraged the knowledge of both in order to evaluate safety-critical systems. Such synergy has been used for identifying the cognitive precondition for mode confusion and automation surprise [2, 5, 10, 15]; the automatic generation of user interface specifications, emergency procedures, and recovery sequences [6, 9]; and the identification of human behavior sequences (normative or erroneous) that contribute to system failures [3, 8].

When human factors engineers analyze human system interaction, they consider the goals, knowledge, and procedures of the human operator; the automated system and its human interface; and the operational environment. Cognitive work analysis is concerned with identifying constraints in the operational environments that shape the mission goals of the human operator [18]; cognitive task analysis is concerned with describing how human operators normatively and descriptively perform tasks when interacting with an automated system [11, 16]; and modeling frameworks such as [7] seek to find discrepancies between human mental models, human-device interfaces (HDIs), and device automation. In

E. Denney, D. Giannakopoulou, C.S. Păsăreanu (eds.); The First NASA Formal Methods Symposium, pp. 6-15

[*] Matthew L. Bolton is a PhD student in the University of Virginia's Department of Systems and Information Engineering

[†] Ellen J. Bass is an Associate Professor in the University of Virginia's Department of Systems and Information Engineering

this context, problems related to human-automation interaction may be influenced by the human operator's mission, the human operator's task behavior, the operational environment, the HDI, the device's automation, and their interrelationships.

In order to allow human factors engineers to exploit their existing modeling tools with the powerful verification capabilities of formal methods to identify potential problems that may be related to human-device interaction, we are developing a computational framework (Figure 1) for the formal modeling of human-interactive systems. This framework utilizes concurrent models of human operator task behavior, human mission directives, device automation, and the operational environment which are composed together to form a larger system model. Inter-model interaction is represented by variables shared between models. Environment variables communicate information about the state of the environment to the device automation, mission, and human task models. Mission variables communicate the mission goals to the human task model. Interface variables convey information about the state of the HDI (displayed information, the state of input widgets, etc.) to the human task model. The human task model indicates when and what actions a human operator would perform on the HDI. The HDI communicates its current state to the device automation via the interface variables. The HDI receives information about the state of the device automation model via the automation state variables.

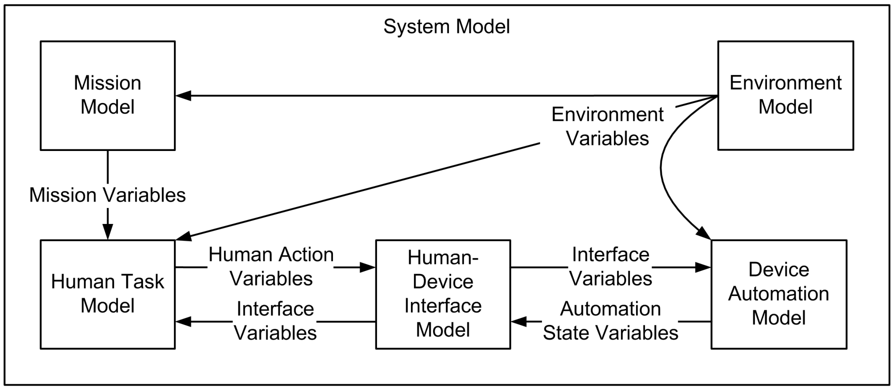


Figure 1: Framework for the formal modeling of a human-interactive system. Arrows between models represent variables that are shared between models. The direction of the arrow indicates whether the represented variables are treated as inputs or output. If the arrow is sourced from a model, the represented variables are outputs of that model. If the arrow terminates at a model, the represented variables are inputs to that model.

To be effective for the human factors community, the analysis framework must support models of the human task and the mission. Because an engineer may wish to rerun verifications using different missions, task models, human-device interfaces, environments, or automation behaviors, these components should remain decoupled (as is the case in Figure 1). Finally, the modeling technique must be capable of representing human-interactive systems with enough fidelity to allow the engineer to perform the desired verification, and do so in reasonable amount of time (this could mean several hours for a small project, or several days for a more complicated one).

In the first instantiation of this framework, we modeled a Baxter Ipump Pain Management System [1], a patient controlled analgesia (PCA) pump that administers pain medication in accordance with health care technician defined constraints. The pump is safety critical as a malfunctioning or miss-programmed pump could potentially overdose a patient. It has a simple HDI (see Figure 2) that can be used to specify a wide range of user mission options (different prescriptions that can be issued). The pump also has automation that checks user inputs and administers treatment.

The end goal of this modeling effort is to perform verifications related to the ability of documentation-derived human task behavior to successfully program the range of available prescriptions into the pump. The initial focus of this effort is to construct and debug the human-device interface and device automation models. An environmental model was not included because of the general stability of the environment in which an actual pump operates. Because this modeling phase was primarily concerned with ensuring that the human-device interface and device automation models were working as intended, the human task model was represented as an unconstrained operator - one that could issue any valid human action at any given time. Finally, the specification used in the verification process was only concerned with ensuring that the models were functioning properly.

Even though the target device being modeled was seemingly simple, the initial model was too difficult for the model checker to process quickly and too complex for it to verify. Thus in order to produce a model that was usable on a reasonable time scale, a number of model revisions ultimately impacted the goals of the framework. This paper discusses these compromises and uses them to draw conclusions about the feasibility of using formal methods to inform HFE.

2 Methods

2.1 The Target System

The Baxter Ipump is an automated machine that allows for the controlled delivery of sedative, analgesic, and anesthetic medication solutions [1]. Solution delivery via intravenous, subcutaneous, and epidural routes is supported. Medication solutions are typically stored in bags locked in a compartment on the back of the pump.

Pump behavior is dictated by internal automation, some of which is dependent on how the pump is programmed by a human operator. Pump programming is accomplished via its HDI (Figure 2) which contains a dynamic LCD display, a security key lock, and eight buttons. When programming the pump, the operator is able to specify all of the following: whether to use periodic or continuous doses of medications (i.e. the mode), whether to use prescription information previously programmed into the pump, the fluid volume contained in the medication bag, the units of measure used for dosage (ml, mg, or μg), whether or not to administer a bolus (an initial dose of medication), dosage amounts, dosage flow rates (for either basal or continuous rates as determined by the mode), the delay time between dosages, and one hour limits on the amount of delivered medication.

During programming, the security key is used to lock and unlock the compartment containing the medication solution. The unlocking and locking process is also used as a security measure to ensure that an authorized person is programming the pump. The start and stop buttons are used to start and stop the delivery of medication at specific times during programming. The on-off button is used to turn the device on and off. The LCD display can be used to choose from a variety of options that affect the way the pump operates. When the operator must choose between two or more options, the interface message indicates what is being chosen, and the initial or default option is displayed. Pressing the up button allows the programmer to scroll through the available options.

When a numerical value is required, the name of the required value is listed in the interface message, and the displayed value is presented with the cursor under one of the value's digits. The programmer can move the position of the cursor by pressing the left and right buttons. He or she can press the up button to scroll through the different digit values available at that cursor position. The clear button sets the displayed value to zero. The enter button is used to confirm values and treatment options.

Aside from the administration of treatment, the primary form of automation used by the pump is its dynamic checking and restriction of operator entered values. Thus, in addition to having hard limits on

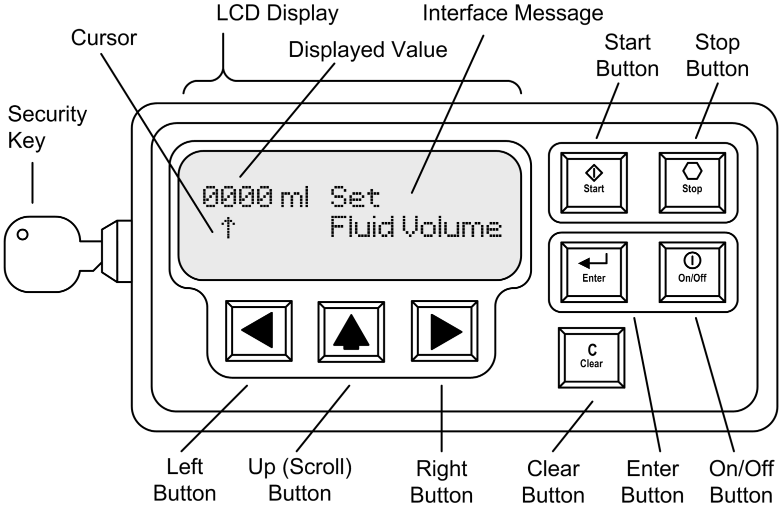


Figure 2: A simplified representation of the Baxter Ipump's human-device interface. Note that the actual pump contains additional controls and information conveyances.

the maximums and minimums a value can assume, the extrema can change dynamically in response to other user specified values.

2.2 Apparatus

All of the models were constructed using the Symbolic Analysis Laboratory (SAL) language [4]. This language was chosen because of its associated analysis and debugging tools, and because it allows for both the asynchronous and synchronous composition of different models (modules using SAL's internal semantics).

All verifications were done using SAL-SMC, the SAL symbolic model checker [1]. Verifications were conducted on a 3.0 gigahertz dual-core Intel Xeon processor with 16 gigabytes of RAM running Redhat Enterprise Linux 5.

2.3 Initial Model

The model that was initially produced was designed to conform to the architecture and design philosophy represented in Figure 2: the mission was represented as a set of viable prescriptions options; the mission, human operator, human-device interface, and device automation were modeled independently of each other; and the behavior of the automated system and HDI models was kept as representative as possible. To limit the scope of the human task model, an unconstrained human operator was constructed that was capable of issuing a valid human action to the human-device interface model at any given time. Further, because the PCA pump generally operates in a controlled environment, away from temperature and humidity conditions that might affect the performance of the pump's automation, no environmental model was included. Finally, because documentation related to the internal workings of the pump was limited, the system automation model was restricted to that associated with the pump programming procedure: behavior that could be gleamed from the operator's manual [1], correspondences with hospital staff, and direct interaction with the pump.

[1] Some model debugging was also conducted using SAL's bounded model checker.

2.4 Verification Specification

Because the model was limited to the prescription programming procedure and human behavior was unrestricted, full verification of the human operator's task behavior was not possible for this instantiation. However, verification could be conducted to ensure that prescriptions could be programmed into the pump. This was done using the SAL-SMC model checker and following specification written in linear temporal logic (state variables are presented in italics):

$$
\begin{array}{llll}
AG\neg(& InterfaceMessage & = & \text{TreatmentAdministering} \\
\wedge & PrescribedMode & = & EnteredMode \\
\wedge & PrescribedFluidVolume & = & EnteredFluidVolume \\
\wedge & PrescribedPCADose & = & EnteredPCADose \\
\wedge & PrescribedDelay & = & EnteredDelay \\
\wedge & PrescribedBasalRate & = & EnteredBasalRate \\
\wedge & PrescribedOneHourLimit & = & EnteredOneHourLimit \\
\wedge & PrescribedBolus & = & EnteredBolus \\
\wedge & PrescribedContinuousRate & = & EnteredContinuousRate \quad)
\end{array}
$$

Here, if the model is able to enter a state indicating that treatment is administering (*InterfaceMessage* = emphTreatmentAdministering) with the entered (or programmed) prescription values (variables with the ”Entered” prefix) matching the actual prescription values (variables with the ”Prescribed” prefix), a counter example is returned.

3 Model Revision

An attempt to verify the initial model resulted in two problems related to the feasibility and usefulness of the verification procedure. Firstly, the SAL-SMC procedure for translating the SAL code into a binary decision diagram (BDD) took excessively long (more than 24 hours), a time frame impractical for model debugging. Secondly, the verification process which followed the construction of the BDD, eventually ran out of memory, thus not returning a verification result. Thus, a number of revisions were required to make the model more tractable, some of which compromised the architectural, representation, and interpretability goals of the effort.

3.1 Model Coordination

Even before the initial verification, some additional model infrastructure was required in order to ensure that human operator actions were properly recognized by the HDI model. In an ideal modeling environment, human action behavior originating from the human operator model would be able to have both an asynchronous and synchronous relationship with the HDI model. Synchronous behavior would allow the HDI model to react to user actions in the same transition in which they were issued/performed by the human operator model. However, both the human operator and HDI models operate independently of each other, and may have state transitions that are dependent on internal or external conditions that are not directly related to the state of the other model. This suggests an asynchronous relationship. SAL only allows models to be composed with each other asynchronously or synchronously (but not both). Thus, it was necessary to adapt the models to support features associated with the unused composition.

Because of independent nature of the models in the framework, asynchronous composition was used to compose the human operator and HDI models. This necessitated some additional infrastructure to prevent the human operator model from issuing user inputs before the HDI model was ready to interpret them and to prevent the human operator model from terminating a given input before the interface could respond to it. This was accomplished through the addition of two Boolean variables: one indicating that input had been submitted (henceforth called Submitted) and a variable indicating the interface was ready to receive actions (henceforth called Ready). This coordination occurred as follows:

- If Ready is true, the human operator module sets one or more of the human action variables to a new input value and sets Submitted to true.
- If Ready and Submitted are true, the human-device interface module responds to the values of the human action variables and sets Ready to false.
- If Ready is not true and Submitted is true, the human operator module sets Submitted to false.
- If Ready and Submitted are both not true and the automated system is ready for additional human operator input, the human-system interface module sets Ready to true.

3.2 Representation of Numerical Values

In order to reduce the time needed to convert the SAL-code model to a BDD, a number of modifications were made to represent model constructs in a way that could be more readily processed by the model checker. As such, the modifications discussed here did not ultimately make the BDD representation of the model smaller, but merely expedite its construction.

3.2.1 Redundant Representation of Values

The dichotomy between the HDI and device automation models resulted in a situation where two different representations of user specified numerical values were convenient. Because the HDI required the human operator to enter values by scrolling through the available values for individual digits, it was conceptually simple to model these values as an array of integer digits in the HDI model. However, because the system automation was concerned with dynamically checking limits and using entered values to compute other values, a numerical representation of the actual value was more convenient for the automated system model.

Because this redundancy of information necessitated additional effort on the part of the BDD translator, it was remedied by eliminating the digit array representations. To accommodate this, a variety of functions needed to be created in order to allow actions from the human task model to dynamically change individual digits within a value.

3.2.2 Real Numbers and Integers

In the original model, all numerical values were represented as real values with restricted ranges. This was done because most user specified values were either integers or floating point numbers (precise to a single decimal point). Representing values this way proved especially challenging for the BDD translator. Thus, all values were modified so that they could be represented as restricted range integers. For integer variables representing floating point numbers, this meant that the model value was ten times the value it represented.

3.2.3 Variable Ranges

In the initial model, the upper bound on the range of all value based variables was set to the maximum amount that could be programmed into the pump: 99999[2]. However, in order to reduce the amount

[2] All lower bounds were set to 0.

of work required for the BDD conversion, the range of each numerically valued variable was given a specific upper bound that corresponded to the actual maximum value that a human operator could assign to it.

3.2.4 Model Reduction

In order to reduce the size of the model, a variety of elements were removed. In all cases these reductions were meant to reduce the number of state variables in the HDI or Device Automation models (slicing), or reduce the range of values a variable could assume (data abstraction). Unfortunately, each of these reductions also affected what user tasks could ultimately be modeled and thus verified in future work. All of the following reductions were undertaken:

- In the original model, the mission model could generate a prescription from the entire available range of valid prescriptions. The scope of this was significantly reduced so that only several prescription options were generated . While this significantly reduced the number of model states, it significantly reduced the number of prescriptions that could be used in verification procedures.

- In the original model, the HDI model would allow the operator to select what units to use when entering prescriptions (ml, mg, or μg). This was removed from the model, with the most general unit option (ml) becoming standard. This reduced the number of interface messages in the model, allowed for the removal of several variables (those related to the unit option selection, and solution concentration specification), and reduced the ranges required for several numerical values related to the prescription. This also eliminated the option of including unit selection task behavior in future work.

- In the original model, both the HDI and device automation models encompassed behavior related to the delivery of medication solution during the priming and bolus administration procedures. During priming, the HDI allows the operator to repeatedly instruct the pump to prime until all air has been pushed out of the connected tubing. During bolus administration, the HDI allows the operator to terminate bolus infusion by pressing the stop button twice. This functionality was removed from the models, thus eliminating interface message states and numerical values indicating how much fluid had been delivered in both procedures. This removed the option of incorporating task behavior related to pump priming and bolus administration.

- The original model mimicked security features found in the original device which required the operator to unlock and lock the device on startup and enter a security code. This functionality was removed from the model which reduced the number of interface messages in the model and removed the numerical variable associated with entering the security code (a 0 to 999 ranged variable). This eliminated the possibility of modeling human task behavior related to unlocking and locking the pump as well as entering the security code.

- In the original model, the interface message could automatically transition to being blank: mimicking the actual pump's ability to blank its screen after three seconds of operator inactivity. Because further operator inaction would result in the original device issuing a ”left in programming mode” alert, a blank interface message could automatically transition to an alert issuance. This functionality was removed from the model, eliminating several interface messages as well as variables that kept track of the previous interface message. Thus, the ability to model operator task response to screen blanking and alerts was also removed.

While these reductions resulted in a model that was much smaller and more manageable than the original, and one that could still be used concurrently with operator task behavior models for programming pump prescriptions, the ability to model some of task behaviors originally associated with the device had to be sacrificed.

3.2.5 Results

As a result of the revisions discussed in this section, the model was able to complete the entire verification procedure in approximately 5.9 hours, with the majority of that time (5.4 hours) being required to create the BDD representation. Because of the nature of the specification, a counterexample was produced which was used to verify that the model was behaving as intended[3].

4 Discussion

The modeling framework discussed in this paper had three primary goals:

1. Model constructs needed to be represented in a way that was intuitive to a human factors engineer who would be building and evaluating many of the models;

2. The sub-models would remain decoupled and modular (as seen in Figure 1) in order to allow for interchangeability of alternative sub-models; and

3. The models constructed around the framework needed to be capable of being verified in a reasonable amount of time.

While this effort was able to produce verifiable model of a human-interactive, safety-critical system, it did so at the cost of compromising all three of these goals. We discuss how each of these goals was violated and how related issues might be addressed.

4.1 Goals 1: Model Intuitiveness

Many of the model revisions were associated with representing model constructs in ways that were more readily interpretable by the model checker rather than the HFE modeler. These primarily took the form of converting floating point and digit array values as integers. There are two potential ways to address this issue. One solution would be to improve the model checkers themselves. Given that the modifications would not actually change the number of reachable states in the system, this would suggest that the model checker need only optimize the BDD conversion algorithms.

Alternatively, additional modeling tools could be used to help mitigate the situation. Such tools could allow human factors engineers to construct or import HDI prototypes, and translate them into model checker code. This would allow the unintuitive representations necessary for ensuring a model's efficient processing by the model checker to be removed from the modeler's view.

Tools could also be created to translate counterexample data into visualizations of HDI use cases. This would prevent the modeler from having to understand the code produced by the translation when examining counterexample data.

4.2 Goal 2: Decoupled Sub-models

Because the communication protocol used to coordinate human actions between the HDI and human task models assumes a particular relationship between variables shared by these models, they are more

[3] The completed model, SAL output, and counterexample can be found at `http://cog.sys.virginia.edu/formalmethods/`

tightly coupled than they would be otherwise. Unless SAL can be made to support both asynchronous and synchronous relationships between models more elegantly, there is little that can be done to eliminate the coordination infrastructure.

However, a solution may be found in an additional level of abstraction. Should a toolset exist for translating an HDI prototype into model checking code, then this translation could handle the construction of the coordination protocol, making this process effectively invisible to the modeler.

4.3 Goal 3: Model Verifiability

One of the reasons this particular modeling effort was so challenging was because the target system was dependent on a large number of user specified numerical values, all of which had very large acceptable ranges. This resulted in the scope of the model being reduced to the point where it could no longer be used for verifying all of the original human operator task behaviors.

Given the simplicity of the device that was modeled, it is clear that systems with a larger number of operator specified numerical values would be more challenging to verify, if not impossible. While the use of bounded model checkers may provide some verification capabilities for certain systems, there is little that can be done without advances in model checking technology and computation power.

However, it is possible that, had the target system been more concerned with procedural behaviors and less on the interrelationships between numerical values, the system model would have been much more tractable. Future work should identify the properties of human-interactive systems that lend themselves to be modeled and verified using the framework discussed here.

5 Conclusion

The work presented here has shown that it is possible to construct models of human-interactive systems for use in formal verification processes. However, this success was the result of a number of compromises that produced a model that was not as representative, understandable, or modular as desired. Thus, in order for formal methods to become more useful for the HFE community, the verification technology will need to be able to support a more diverse set of systems. Further, new modeling tools may be required to support representations that human factors engineers use. These advances coupled with efficient means of representing realistic human behavior in the models will ultimately allow formal methods to become a more useful tool for human factors engineers working with safety critical systems.

Acknowledgement

The project described was supported in part by Grant Number T15LM009462 from the National Library of Medicine and Research Grant Agreement UVA-03-01, sub-award 6073-VA from the National Institute of Aerospace (NIA). The content is solely the responsibility of the authors and does not necessarily represent the official views of the NIA, NASA, the National Library of Medicine, or the National Institutes of Health.

The authors would like to thank Radu I. Siminiceanu of the NIA and Ben Di Vito of NASA Langley for their help with the technical issues in this paper. They would like to thank Diane Haddon, John Knapp, Paul Merrel, Kathryn McGough, and Sherry Wood of the University of Virginia Health System for helping us understand the functionality of the Baxter Ipump and providing them with the necessary documentation, training materials, and device access.

References

[1] Baxter Heath Care Corporation. *Ipump Pain Management System Operator's Manual*, 1995.

[2] J. Crow, D. Javaux, and J. Rushby. Models and mechanized methods that integrate human factors into automation design. In *Proceedings of the International Conference on Human-Computer Interaction in Aeronautics*, 2000.

[3] P. Curzon, R. Ruksenas, and A. Blandford. An approach to formal verification of human-computer interaction. *Formal Aspects of Computing.*, 19(4):513–550, 2007.

[4] L. De Moura, S. Owre, and N. Shankar. The SAL language manual. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, 2003.

[5] A. Degani. *Modeling Human-machine Systems: On Modes, Error, and Patterns of Interaction*. PhD thesis, Georgia Institute of Technology, 1996.

[6] A. Degani, M. Heymann, and I. Barshi. A formal methodology, tools, and algorithm for the analysis, verification, and design of emergency procedures and recovery sequences. *NASA Internal White Paper*, 2005.

[7] A. Degani and A. Kirlik. Modes in human-automation interaction: initial observations about a modeling approach. In *IEEE International Conference on Systems, Man and Cybernetics, 1995. Intelligent Systems for the 21st Century.*, volume 4, 1995.

[8] R.E. Fields. *Analysis of Erroneous Actions in the Design of Critical Systems*. PhD thesis, University of York, 2001.

[9] M. Heymann and A. Degani. Formal analysis and automatic generation of user interfaces: Approach, methodology, and an algorithm. *Human Factors*, 49(2):311–330, 2007.

[10] D. Javaux and P.G. Polson. A method for predicting errors when interacting with finite state machines. *Reliability Engineering and System Safety*.

[11] B. Kirwan and L.K. Ainsworth. *A Guide to Task Analysis*. Taylor and Francis, 1992.

[12] L.T. Kohn, J. Corrigan, and M.S. Donaldson. *To Err is Human: Building a Safer Health System*. National Academy Press, 2000.

[13] N.C. Krey. 2007 Nall report: accident trends and factors for 2006. Technical report, 2007. `http://download.aopa.org/epilot/2007/07nall.pdf`.

[14] C. Perrow. *Normal Accidents*. Basic Books New York, 1984.

[15] J. Rushby. Using model checking to help discover mode confusions and other automation surprises. *Reliability Engineering and System Safety*, 75(2):167–177, 2002.

[16] J.M. Schraagen, S.F. Chipman, and V.L. Shalin. *Cognitive Task Analysis*. Lawrence Erlbaum Associates, 2000.

[17] N. Stanton. *Human Factors Methods: A Practical Guide for Engineering and Design*. Ashgate Publishing, Ltd., 2005.

[18] K.J. Vicente. *Cognitive Work Analysis: Toward Safe, Oroductive, and Healthy Computer-based Work*. Lawrence Erlbaum Assoc Inc, 1999.

[19] C.D. Wickens, J. Lee, Y.D. Liu, and S. Gordon-Becker. *Introduction to Human Factors Engineering*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 2003.

Model Checking for Autonomic Systems Specified with ASSL

Emil Vassev
Lero, University College
Dublin, Ireland
emil.vassev@lero.ie

Mike Hinchey
Lero, University of
Limerick, Ireland
mike.hinchey@lero.ie

Aaron Quigley
Lero, University College
Dublin, Ireland
aquigley@ucd.ie

Abstract

Autonomic computing augurs great promise for deep space exploration missions, bringing onboard intelligence and less reliance on control links. As part of our research on the ASSL (Autonomic System Specification Language) framework, we have successfully specified autonomic properties, verified their consistency, and generated prototype models for both the NASA ANTS (Autonomous Nano-Technology Swarm) concept mission and the NASA Voyager mission. The first release of ASSL provides built-in consistency checking and functional testing as the only means of software verification. We discuss our work on model checking autonomic systems specified with ASSL. In our approach, an ASSL specification is translated into a state-transition model, over which model checking is performed to verify whether the ASSL specification satisfies correctness properties. The latter are expressed as temporal logic formulae expressed over sets of ASSL constructs. We also discuss possible solutions to the state-explosion problem in terms of state graph abstraction and probability weights assigned to states. Moreover, we present an example case study involving checking liveness properties of autonomic systems with ASSL.

1 Introduction

As software increasingly exerts influence over our daily lives, the need for more reliable software systems has become critical. However, contemporary software systems are becoming extremely complex and so too is the task of ensuring their correctness. This is the reason why a great deal of research effort is devoted to developing software verification methods. A promising, and lately popular, technique for software verification is model checking [1], [2]. This approach advocates formal verification tools whereby software programs are automatically checked for specific flaws by considering correctness properties expressed in temporal logic. Numerous formal tools allowing verification by model-checking have been developed, such as Spin [3], Emc [4], Tav [5], Mec [6], XTL [7], etc. Despite best efforts and the fact that model checking has proved to be a revolutionary advance in addressing the correctness of critical systems, software assurance for large and highly-complex software is still a tedious task. The reason is that high complexity is a source of software failures, and standard model checking approaches do not scale to handle large systems very well due to the so-called state-explosion problem [1], [2].

NASA Engages in Autonomic Computing. Experience has shown that software errors can be very expensive and even catastrophic in complex systems such as spacecraft. An example is the malfunction in the control software of Ariane-5, which caused the rocket's crash 36 seconds after its launch [2].

There are a number of valid complexity-reduction approaches, including the use of formal methods and autonomic computing. The latter is an emerging field for the development of large-scale self-managing complex systems. The Autonomic Computing (AC) paradigm draws inspiration from the human (and mammalian) body's autonomic nervous system [8]. The idea is that software systems can manage themselves and deal with dynamic requirements, as well as unanticipated threats, automatically, just as the body does, by handling complexity through self-management based on high-level objectives.

NASA is currently approaching AC with great interest, recognizing in AC a bridge towards "the new age of space exploration" whereby spacecraft should be independent, autonomous, and "smart" [9]. NASA's New Millennium Project [9] addresses space-exploration missions where AC software controls spacecraft entirely. Both the Autonomous Nano-Technology Swarm (ANTS) concept mission [10] and

E. Denney, D. Giannakopoulou, C.S. Păsăreanu (eds.); The First NASA Formal Methods Symposium, pp. 16-25

the Deep Space One mission [9] involve the next generation of AC-based unmanned spacecraft systems. In such systems, autonomic computing software turns spacecraft into autonomic systems, i.e., capable of planning and executing many activities onboard to meet the requirements of changing objectives and harsh external conditions. NASA is engaging in AC research to find a solution to large-scale automation for deep space exploration. However, automation implies sophisticated software, which requires new development approaches and new verification techniques.

The ASSL Approach to Autonomic Computing. The Autonomic System Specification Language (ASSL) [11] is an initiative for self-management of complex systems where we approach the problem of formal specification, validation, and code generation of autonomic systems within a framework. Being dedicated to AC, ASSL helps AC researchers with problem formation, system design, system analysis and evaluation, and system implementation. The framework provides tools that allow ASSL specifications to be edited and validated. The current validation approach in ASSL is a form of consistency checking performed against a set of semantic definitions. The latter form a theory that aids in the construction of correct autonomic system (AS) specifications. From any valid specification, ASSL can generate an operational Java application skeleton. As a part of the framework validation and in the course of a new ongoing research project at Lero—the Irish Software Engineering Research Center, ASSL has been used to specify autonomic properties and generate prototype models of the NASA ANTS concept mission [12], [13], [14] and NASA's Voyager mission [15]. Both ASSL specifications and generated prototype models have been used to investigate hypotheses about the design and implementation of intelligent swarm-based systems and possible future Voyager-like missions incorporating the principles of AC.

Research Problem. Due to the synthesis approach of automatic code generation, ASSL guarantees consistency between a specification and the corresponding implementation. However, our experience with ASSL demonstrated that errors can be easily introduced while specifying large systems. Although the ASSL framework takes care of syntax and consistency errors, it cannot handle logical errors.

We are currently investigating a few possible approaches to ensure the correctness of the ASSL specifications and that of the generated autonomic systems:

- It is possible to improve the current ASSL consistency checker with assertion and debugging techniques. These should allow for a good deal of *static analysis* of an ASSL specification. Moreover, this approach should introduce flexibility to the ASSL consistency checker by providing various options that affect what static analysis the latter performs and what results are shown. Although currently under development, this approach will improve the verification process, but will not be sufficient to assert safety (e.g., freedom from deadlock) or liveness properties (cf. Section 2).
- ASSL generates operational Java code, which can be used to perform a sort of post-implementation model checking. For example, Java Pathfinder [16], developed at NASA Ames, can be used to verify the generated Java code. This approach is efficient when applied to smaller systems, but cannot be used to discover logical errors in the ASSL specifications because it is employed at a later stage of the software lifecycle.
- The most promising and most effective approach is model checking. In the course of this project, we consider two possible approaches:
  - Another paper presented at this event proposes mapping ASSL specifications to special Service Logic Graphs [17]. The latter support reverse model checking and games, and enable intuition of graphical models and expression of constraints in mu-calculus.
  - This paper presents a model-checking mechanism that takes as input an ASSL specification and produces as output a finite state-transition system such that a specific property in question is satisfied if and only if the original ASSL specification satisfies that property.

Benefits for Space Systems. An ASSL model-checking mechanism will be the next generation of the ASSL consistency checker based on automated reasoning. By allowing for automated system analysis and evaluation, this mechanism will complete the autonomic system development process with ASSL. The ability to verify the ASSL specifications for design flaws can lead to significant improvements in both specification models and generated autonomic systems. Subsequently, ASSL can be used to specify, validate and generate better prototype models for current and future space-exploration systems. These prototype models can be used for feature validation and simulated experimental results. The ability to simulate exploration missions with hypothesized possible autonomic features gives significant benefits.

2 Consistency Checking with ASSL

In general, ASSL considers ASs as being composed of autonomic elements (AEs) interacting over interaction protocols. To specify autonomic systems, ASSL exposes a multi-tier specification model that is designed to be scalable and to expose a judicious selection and configuration of infrastructure elements and mechanisms needed by an AS. By their virtue, the ASSL tiers are abstractions of different aspects of the AS under consideration, such as policies, communication interfaces, execution semantics, actions, etc. There are three major tiers (three major abstraction perspectives), each composed of sub-tiers [11]:

- AS tier — forms a general and global AS perspective, where we define the general system rules in terms of *service-level objectives (SLO)* and *self-management policies*, *architecture topology*, and *global actions*, *events*, and *metrics* applied in these rules.
- AS Interaction Protocol (ASIP) tier — forms a communication protocol perspective, where we define the means of communication between AEs. The ASIP tier is composed of *channels*, *communication functions*, and *messages*.
- AE tier — forms a unit-level perspective, where we define interacting sets of individual autonomic elements (AEs) with their own behaviour. This tier is composed of AE rules (*SLO* and *self-management policies*), an *AE interaction protocol (AEIP)*, *AE actions*, *AE events*, and *AE metrics*.

In general, we can group the ASSL tiers into groups of *declarative* (or imperative) and *operational* tiers. Whereas the former simply describe definitions in the AS under consideration, the latter not only describe definitions but also focus on the operational behavior of that AS. The ASSL framework evaluates an AS specification formally to construct a special *declarative specification tree* needed to perform both consistency checking and code generation.

Consistency checking (cf. Figure 1) is a framework mechanism for verifying specifications by performing exhaustive traversing of the *declarative specification tree*. In general, the framework performs two kinds of consistency-checking operations: *light* (checks for type consistency, ambiguous definitions, etc.) and *heavy* (checks whether the specification model conforms to special correctness properties). The

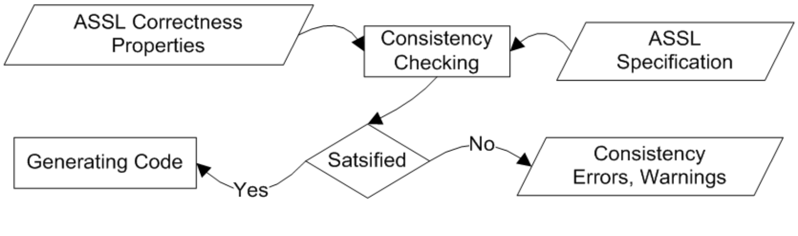


Figure 1: Consistency Checking with ASSL

correctness properties are ASSL semantic definitions [11] defined per tier. Although, they are expressed in First-Order Linear Temporal Logic (FOLTL) [1] [1], [2], currently ASSL does not incorporate a FOLTL engine, and thus, the consistency checking mechanism implements the correctness properties as Java statements. Here, the FOLTL operators $\forall$ (forall) and $\exists$ (exists) work over sets of ASSL tier instances. In addition, these operators are translated by taking their first argument as a logical atom that contains a single unbound tier variable. Ideally, this atom has a relatively small number of ground tier instances, so the combinatorial explosion generally produced by these statements is controlled.

It is important to mention that the consistency checking mechanism generates consistency errors or warnings. Warnings are specific situations, where the specification does not contradict the correctness properties, but rather introduces uncertainty as to how the code generator will handle it. Although considered efficient, the ASSL consistency checking mechanism has some major drawbacks.

- It does not consider the notion of time, and thus, temporal FOLTL operators such as $\Box$ (always), $\bigcirc$ (next), $\Diamond$ (eventually), $\mathscr{U}$ (until), $\mathscr{W}$ (waiting-for), etc., are omitted. Therefore, the ASSL consistency checking is not able to assert safety (e.g., freedom from deadlock) or liveness properties (e.g., a message sent is eventually received).

- The interpretation of the FOLTL formulas into Java statements is done in an analytical way and thus the introduction of errors is possible.

- There is no easy way to add new correctness properties to the consistency-checking mechanism.

3 Model Checking with ASSL

In general, model checking provides an automated method for verifying finite state systems by relying on efficient graph-search algorithms. The latter help to determine whether or not system behavior described with temporal correctness properties holds for the system's state graph.

In ASSL, the model-checking problem is: given autonomic system A and its ASSL specification a, determine in the system's state graph g whether or not the behavior of A, expressed with the correctness properties p, meets the specification a. Formally, this can be presented as a triple (a,p,g) where: a is the ASSL specification of the autonomic system A, p presents the correctness properties specified in FOLTL, and g is the state graph constructed from the ASSL specification in a labeled transition system (LTS) [2] format. The first step in the ASSL model-checking mechanism is to construct from an ASSL specification a the corresponding state graph g where the desired correctness properties p can be verified.

ASSL Tier States. The notion of state in ASSL is related to tiers. The ASSL operational semantics considers a state-transition model where the so-called *operational tiers* can be in different *tier states*, e.g., SLO can be evaluated as *satisfied* or *not satisfied* [11]. An ASSL autonomic system transits from one state to another when a particular tier evolves from a tier state to another tier state. Here, the LTS of an autonomic system specified with ASSL consists of high-level composite states composed of multilevel nested states; i.e., a tier state is determined by the states of the tiers (sub-tiers) nested in that tier.

In ASSL, special system operations cause ASSL tiers to evolve from one tier state to another tier state. Here, the tier state evolution caused by a system operation Op can be denoted as $\sigma \xrightarrow{Op(x_1,x_2,....,x_n)} \sigma'$ where the operation $Op(x_1,x_2,....,x_n)$ can be a parametric operation (e.g. *AS/AE action*, *ASIP/AEIP function*, or an abstract function denoting an operation performed by the framework) which takes n arguments. ASSL considers twenty system state-transition operations as shown in Figure 2.

[1] In general, FOLTL can be seen as a quantified version of linear temporal logic. FOLTL is obtained by taking propositional linear temporal logic and adding a first order language to it.

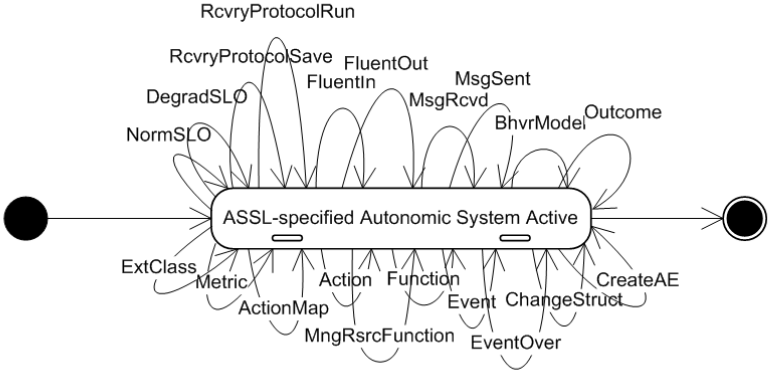


Figure 2: High-level LTS for ASs Specified with ASSL

Formally, an ASSL LTS can be presented as a tuple (S, Op, R, S_0, AP, L) [2] where: S is the set of all possible ASSL tier states; Op is the set of special ASSL state-transition operations; $R \subseteq S \times Op \times S$ are the possible transitions; $S_0 \subseteq S$ is a set of initial tier states; AP is a set of atomic propositions; $L : S \longrightarrow 2^{AP}$ is a labeling function relating a set $L(s) \in 2^{AP}$ of atomic propositions to any state s, i.e., a set of atomic propositions true in that state.

Constructing the ASSL LTS. The ASSL LTS is constructed by the ASSL framework by using the *declarative specification tree* (created by the framework when parsing an AS specification) and by applying the ASSL operational semantics [11]. The *declarative specification tree* contains the hierarchical tier structure of the actual specification. Thus, enriched with the possible tier states, it can be used to derive the composite multilevel structure of the specification's LTS by taking into consideration that all the tiers and sub-tiers run concurrently as state machines. The operational evaluation of the ASSL specification, can derive all the transition relations R and associate the tier states via the transition operations Op. Similar to the declarative specification tree, the generated ASSL LTS model is hierarchical, i.e., composed of multilevel composite tier states. Figure 3 depicts the transformation of the *declarative specification tree* into an ASSL LTS, where the latter is presented at the highest possible level of abstraction comprising a single composite state "AS Active" (cf. Figure 2).

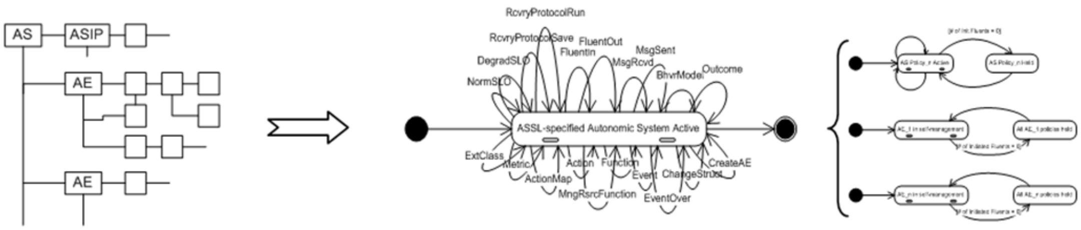


Figure 3: Transformation of the Declarative Specification Tree into an ASSL LTS

Considering implementation, the ASSL LTS can be implemented as a graph G, where the nodes will be tier states each encoded with transition relations $R \subseteq S \times Op \times S$.

ASSL Atomic Propositions and Labeling Function. In order, to make the ASSL LTS appropriate for model checking, we need to associate with each tier state a set of atomic propositions AP true in that state. In general, we should consider two types of AP — *generic* and *derivable*. Whereas the former should be generic (independent of the specification specifics) for particular tiers, the latter should be deduced by the operational evaluation of the ASSL specification. The ASSL *AP* should be about the ASSL tiers and should be expressed in propositional logic (to include them in the correctness properties expressed as FOLTL formulas). The full set of ASSL *AP* should be predefined and passed as input to a special labeling function $L(s)$ in the model checking mechanism, which labels (relates) each tier state s with appropriate *AP*. To perform this task, $L(s)$ should employ an algorithm that considers the ASSL operational semantics, the ASSL code generation rules [11], and concurrency among the ASSL tiers.

Therefore, the implementation graph of the ASSL LTS is composed of nodes, which can be presented formally as a tuple (s, R, AP_g, AP_d) where: s is the tier state; R is a set of transition relations connecting the state s to other states via system operations; AP_g is a set of generic atomic propositions related to s; AP_d is a set of deducted atomic propositions related to s.

Model Checking Algorithm. Given that Φ is a correctness property expressed in FOLTL (i.e., a temporal logic formula), determine whether the "AS Active" tier state (cf. Figure 2) satisfies Φ, which implies that all possible compositions of nested tier states satisfy Φ. Note that $s \vdash \Phi$ iff $L(s) \vdash \Phi$.

State Explosion Problem. A typical ASSL-specified system is composed of multiple AEs (in ANTS for example they could number over 1000) each composed of large number of concurrent processes. The latter are autonomic elements implementing *self-management policies*, which spread internally into special concurrent *fluents* [2] [11], *events*, *metrics*, etc. Thus, the biggest problem stemming from this large number of concurrent processes is the so-called state explosion problem [1], [2]. In general, the size of an ASSL state graph is at least exponential in the number of tiers running as concurrent processes, because the state space of the entire system is built as the Cartesian product of the local state of the concurrent processes (concurrently running tiers).

We are currently investigating two possible solutions to that problem — *abstraction* and *prioritized tiers*. The first solution is to use composite tier states to abstract their nested tier states. Thus, given original state graph G (derived from an ASSL specification) an abstraction is obtained by suppressing low-level tier states yielding a simpler and likely smaller state graph G^a. This reduces the total amount of states to be considered but is likely to introduce a sort of conservative view of the system [18] where the abstraction ensures only that correctness of G^a implies correctness of G.

The other possible solution, we have termed *prioritized tiers*, because the idea is to prioritize ASSL tiers by giving their tier states a special probability weight *pw*. Thus, the new formal presentation of the graph nodes will be (s, R, AP_g, AP_d, pw) where $0 \leq pw \leq 1$ is the probability weight assigned to each graph node. This can be used as a state-reduction factor to derive probability graphs G^{pw} with a specific level of probability weight, e.g., $pw > 0.3$. However, this approach is likely to introduce probability to the model-checking results, which correlates with the probability level of the graph G^{pw}.

Figure 4 depicts a global view of model checking in ASSL as it has been explained in this section. Note that in the case that a correctness property is not satisfied, the ASSL framework returns a *counterexample*. The latter is an execution path of the ASSL LTS for which the desired correctness property is not true. If model checking has been performed on the entire ASSL LTS, then the property does not hold for the original AS specification. Otherwise, in the case that a reduced ASSL LTS has been used (abstraction and/or prioritized tiers state-explosion techniques have been applied), the information provided by the counterexample is then used to refine the reduced model. For example, some composite states are presented with their low-level nested states, or a low-level probability weight is used in order to obtain a more accurate representation of the original model.

[2] An ASSL fluent is a special state with timed duration, e.g., a state like "performance is low". When the system gets into that specific condition, the fluent is considered to be initiated.

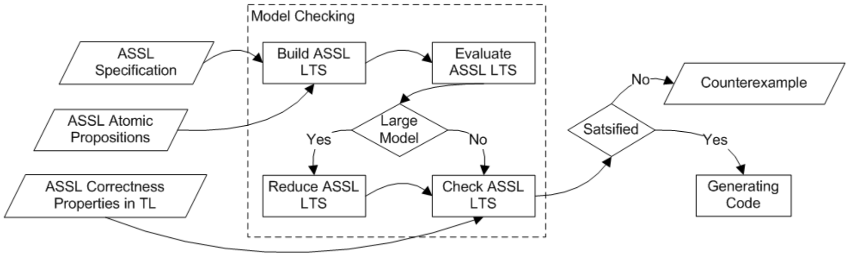
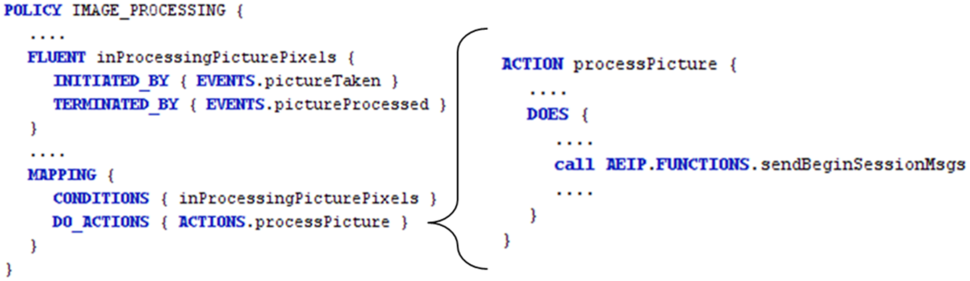


Figure 4: Model Checking in ASSL

4 Example: Checking Liveness Properties with ASSL

In this section, we demonstrate how the ASSL model checking mechanism can perform formal verification to check liveness properties of ASs specified and generated with ASSL. Our example is the ASSL specification model for the NASA Voyager Mission [15]. In this case study, we specified the Voyager II spacecraft and the antennas on Earth as AEs that follow their encoded autonomic behavior to process space pictures, and communicate those via predefined ASSL messages. Here we use a sample from this specification to demonstrate how a liveness property such as "*a picture taken by the Voyager spacecraft will eventually result in sending a message to antennas on Earth*" can be checked with the ASSL model-checking mechanism. Note that the ASSL specification model for the NASA Voyager Mission is relatively large (over 1000 lines of specification code). Thus, we do not present the entire specification but a specification sample. For more details on the specification, please refer to [15].

Figure 5 presents a partial ASSL specification of the `IMAGE_PROCESSING` self-management policy of the Voyager AE. Here the `pictureTaken` event will be prompted when a picture has been taken. This event initiates the `inProcessingPicturePixels` fluent. The same fluent is mapped to a `processPicture` action, which will be executed once the fluent gets initiated. As it is specified, the `processPicture` action prompts the execution of the `sendBeginSessionMsgs` communication function (cf. Figure 5), which puts a special message x on a special communication channel [15] (message x is sent over that channel). Note that the specification of both the `pictureTaken` event and the `sendBeginSessionMsgs` function is not presented here.



```
POLICY IMAGE_PROCESSING {
  ....
  FLUENT inProcessingPicturePixels {
    INITIATED_BY { EVENTS.pictureTaken }
    TERMINATED_BY { EVENTS.pictureProcessed }
  }
  ....
  MAPPING {
    CONDITIONS { inProcessingPicturePixels }
    DO_ACTIONS { ACTIONS.processPicture }
  }
}
```

```
ACTION processPicture {
  ....
  DOES {
    ....
    call AEIP.FUNCTIONS.sendBeginSessionMsgs
    ....
  }
}
```

Figure 5: ASSL IMAGE_PROCESSING policy at Voyager AE

As we have already mentioned in Section 3, the ASSL model checking mechanism should build the ASSL LTS from the ASSL specification. Here both the *declarative specification tree* and the ASSL operational semantics [11] will be used to derive tier states S and transition relations R, and to associate those tier states via the ASSL transition operations Op. Next the labeling function $L(s)$ (integrated in the model checking mechanism) will label each tier state s with appropriate atomic propositions AP.

Product Machine. Figure 6, presents a partial ASSL LTS of the sub-tiers of the Voyager AE. These sub-tiers are derived from the *declarative specification tree* constructed for the Voyager AE. Note that this LTS is a result of our analytical approach and for reasons of clarity it is simplified, i.e., not all the possible tier states are presented.

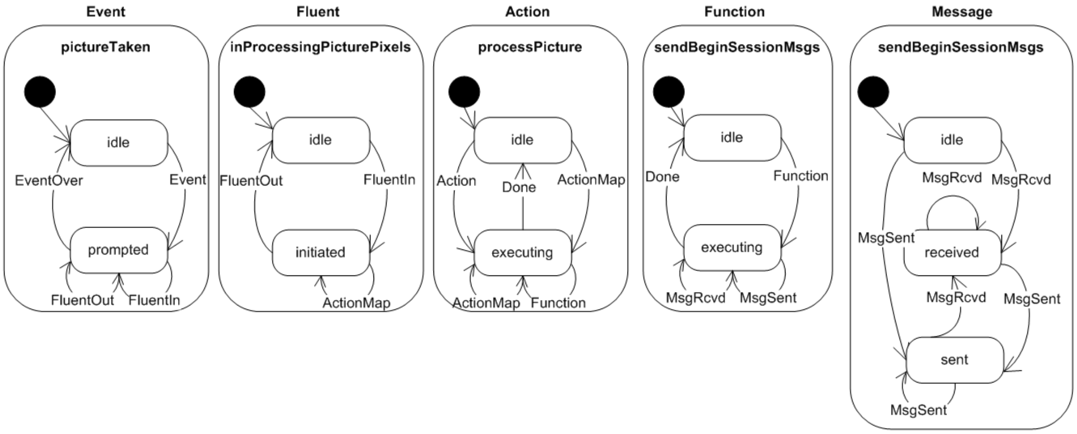


Figure 6: State machines of the Voyager AE sub-tiers

Here each sub-tier instance forms a distinct state machine (*basic* machine) within the AE state machine and the AE state machine is a *Cartesian product* of the state machines of its sub-tiers. It is important to mention that by taking the Cartesian product of a set of basic sub-tier machines, we form a *product machine* consisting of *product states*. The latter are tuples of concurrent basic sub-tier states. Moreover, in the AE product machine, the ASSL state-transition operations Op are considered *product transitions* that move from one product state to another. Note that the states in the state machine of the whole AS product machine can be obtained by the Cartesian product of all the AE product machines.

Thus, by considering the sub-tier state machines we construct the Voyager AE product machine (cf. Figure 7). Note that this is again a simplified model where not all the possible product states are shown. Figure 7 presents the AE product states as large circles embedding the sub-tier states (depicted as smaller circles). Here we use the following aliases: **e** states for Event state machine; **f** states for Fluent state machine; **a** states for Action state machine; **y** states for communication function state machine; **x** states for Message state machine. Moreover, white circles present ”idle” state and gray circles present the corresponding ”active” state of the sub-tier state machine under consideration (such as *prompted* for events, *initiated* for fluents, etc.; cf. Figure 6).

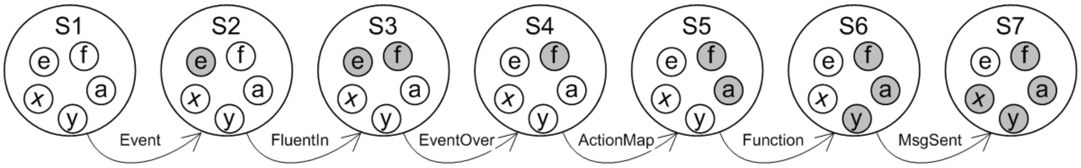


Figure 7: Partial Voyager AE product machine

Therefore, the formal presentation (S, Op, R, S_0, AP, L) (cf. Section 3) of the Voyager AE LTS is:

- $S = \{S1, S2, S3, S4, S5, S6, S7\}$
- $Op = \{\texttt{Event}, \texttt{FluentIn}, \texttt{EventOver}, \texttt{ActionMap}, \texttt{Function}, \texttt{MsgSent}\}$

- $R = \{(S1, S2, \texttt{Event}), (S2, S3, \texttt{FluentIn}), (S3, S4, \texttt{EventOver}), (S4, S5, \texttt{ActionMap}), (S5, S6, \texttt{Function}), (S6, S7, \texttt{MsgSent})\}$
- $S_0 = S1$ (initial state)
- *AP* ={event `pictureTaken` occurs, event `pictureTaken` terminates, action `processPicture` starts, fluent `inProcessingPicturePixels` initiates, function `sendBeginSessionMsgs` starts, sends message x}
- $L(S1)$ ={event `pictureTaken` occurs}; $L(S2)$ ={fluent `inProcessingPicturePixels` initiates}; $L(S3)$ ={event `pictureTaken` terminates}; $L(S4)$ ={action `processPicture` starts}; $L(S5)$ ={function `sendBeginSessionMsgs` starts}; $L(S6)$ ={sends message x}

Moreover, we consider the following *correctness properties* applicable to our case:

- *If an* event occurs *eventually a* fluent initiates.
- *If an* event occurs *next eventually it* terminates.
- *If a* fluent initiates *next* actions start.
- *If an* action starts *eventually a* function starts.
- *If a* function starts *eventually it* sends *a* message.

The ASSL model-checking mechanism shall use the correctness property formulae to check if these are held over product states considering the atomic propositions AP true for that state. Thus, the ASSL framework will be able to trace the *state path* shown in Figure 7 and to validate the liveness property stated above. Note that in this example, we intentionally presented a limited set of atomic propositions *AP* and correctness properties. The former are derivable, that is, deduced from the operational evaluation of the ASSL specification (cf. Section 3). Moreover, the Voyager AE product machine presents only product states relevant to our case study.

5 Conclusion and Future Work

We have presented our approach towards the development of a model checking mechanism in the ASSL framework. The principle technique is to transform an ASSL specification into a state graph enriched with atomic properties, which is used to verify whether special correctness properties are satisfied by that specification. Currently, ASSL provides a consistency checking mechanism to validate autonomic systems specified with ASSL against correctness properties. Although proven to be efficient with handling consistency errors, this mechanism cannot handle logical errors. Therefore, a model checking mechanism will complete the ASSL framework, allowing for automated system analysis and evaluation of ASSL specifications, and thus, it will help to validate *liveness* and *safety* properties of autonomic systems specified with ASSL.

Considering *liveness properties*, model checking will help ASSL to verify whether a self-management policy will fix a specific problem, a message sent by an AE will be eventually received by another AE, etc. In this paper, we have presented a case study example of checking with ASSL a special liveness property of the ASSL specification model for the NASA Voyager mission [15].

Considering *safety properties*, model checking will help ASSL to verify cases such as: an action will be performed when a policy is triggered; an action can be never started due to unreachable preconditions; an event will be prompted to stop a fluent (so the fluent is not endless); deadlocks among the AEs (waiting for messages from each other) and among the policies; contradictions among policies, SLO, and among policies and SLO.

Our plans for future work are mainly concerned with further development of the model checking mechanism for ASSL. Moreover, it is our intention to build an animation tool for ASSL, which will help to visualize counterexamples and trace erroneous execution paths.

It is our belief that a model checking mechanism for ASSL will enable broad-scale formal verification of autonomic systems. Therefore, it will make ASSL a better and more powerful framework for autonomic system specification, validation and code generation.

References

[1] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 2002.

[2] C. Baier, J.-P. Katoen. *Principles of Model Checking*. MIT Press, 2008.

[3] M. Ben-Ari. *Principles of the Spin Model Checker (Paperback)*. Springer, 2008.

[4] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[5] K. G. Larsen. Efficient Local Correctness Checking. *LNCS*, 663:30–43, 1992.

[6] A. Arnold, D. Begay, and P. Crubille. Construction and Analysis of Transition Systems with MEC. *Amast Series in Computing*, 3:192, 1994.

[7] R. Mateescu and H. Garavel. XTL: A Meta-Language and Tool for Temporal Logic Model-Checking. In *Proceedings of the International Workshop on Software Tools for Technology Transfer (STTT'98)*, 1998.

[8] P. Horn. Autonomic Computing: IBM's Perspective on the State of Information Technology. Technical report, IBM T. J. Watson Laboratory, October 2001.

[9] R. Murch. *Autonomic Computing: On Demand Series*. IBM Press, Prentice Hall, 2004.

[10] W. Truszkowski, M. Hinchey, J. Rash, and C. Rouff. NASA's Swarm Missions: The Challenge of Building Autonomous Software. *IT Professional*, 6(5):47–52, 2004.

[11] E. Vassev. *Towards a Framework for Specification and Code Generation of Autonomic Systems*. PhD thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, 2008.

[12] E. Vassev, M. Hinchey, and J. Paquet. A Self-Scheduling Model for NASA Swarm-Based Exploration Missions using ASSL. In *Proceedings of the Fifth IEEE International Workshop on Engineering of Autonomic and Autonomous Systems (EASe'08)*, pages 54–64. IEEE Computer Society, 2008.

[13] E. Vassev, M. Hinchey, and J. Paquet. Towards an ASSL Specification Model for NASA Swarm-Based Exploration Missions. In *Proceedings of the 23rd Annual ACM Symposium on Applied Computing (SAC 2008) - AC Track*, pages 1652–1657. ACM, 2008.

[14] E. Vassev and M. Hinchey. ASSL Specification and Code Generation of Self-Healing Behavior for NASA Swarm-Based Systems. In *Proceedings of the Sixth IEEE International Workshop on Engineering of Autonomic and Autonomous Systems (EASe'09)*. IEEE Computer Society, 2009.

[15] E. Vassev and M. Hinchey. ASSL specification model for the image-processing behavior in the nasa voyager mission. Technical Report Lero-2009-1, Lero—the Irish Software Engineering Research Center, 2009.

[16] K. Havelund and T. Pressburger. Model Checking JAVA programs using JAVA PathFinder. *STTT*, 2(4):366–381, 2000.

[17] M. Bakera, C. Wagner, T. Margaria, E. Vassev, M. Hinchey, and B. Steffen. Component-oriented behavior extraction for autonomic system design. In *The First NASA Formal Methods Symposium (NFM 2009)*. NASA, 2009.

[18] E. A. Emerson. The Beginning of Model Checking: A Personal Perspective. In *25 Years of Model Checking*, pages 27–45. Springer, 2008.

A Game-Theoretic Approach to Branching Time Abstract-Check-Refine Process

Yi Wang
The University of Tokyo
Meguro-ku, Tokyo, Japan
wangyi@graco.c.u-tokyo.ac.jp

Tetsuo Tamai
The University of Tokyo
Meguro-ku, Tokyo, Japan
tamai@graco.c.u-tokyo.ac.jp

Abstract

Since the complexity of software systems continues to grow, most engineers face two serious problems: the state space explosion problem and the problem of how to debug systems. In this paper, we propose a game-theoretic approach to full branching time model checking on three-valued semantics. The three-valued models and logics provide successful abstraction that overcomes the state space explosion problem. The game style model checking that generates counterexamples can guide refinement or identify validated formulas, which solves the system debugging problem. Furthermore, output of our game style method will give significant information to engineers in detecting where errors have occurred and what the causes of the errors are.

1 Introduction

Model checking is a major technique for verifying finite state systems [5]. The procedure normally uses an exhaustive search of the state space of the system to determine whether some specification is satisfied or not. Given sufficient resources, the procedure will always terminate with a yes/no answer. Since the size of a given system (concrete model) is usually very large, the state explosion problem may occur in model checking such a model. Abstraction as an indispensable means to reduce the state space makes model checking feasible [4, 5]. The traditional abstraction method uses two-valued semantics. It brings about two kinds of situations: under-approximation and over-approximation. In under-approximation, the abstract model exhibits behavior that exists in the concrete model but may miss some of its behavior. On the other hand, over-approximation may bring in additional behavior that does not exist in the concrete model. Unfortunately both of these two-valued abstractions have the unsoundness problem. That is for every existential property satisfied by an under-approximate model also holds in the concrete model but universal properties (for example safety) do not necessarily hold, and for every universal property satisfied by an over-approximate model also holds in the concrete model but existential properties (for example liveness) do not necessarily hold. Such unsoundness makes abstraction less useful and the state space explosion problem still remains. In the past ten years, three-valued models and logics have been studied. The main benefit of this approach is that both universal and existential properties are guaranteed to be sound. The three-valued semantics evaluates a formula to either *true, false* or an indefinite value as the third value. In Kleene's strongest regular three-valued propositional logic, the third value is understood as *unknown* that means it can take either *true* or *false*. The three-valued models, or modal transition systems, contain *may*-transitions which over-approximate transitions of the concrete model, and *must*-transitions which under-approximate transitions of the concrete model. To ensure logical consistency, truth of universal formulas is then examined over *may*-transitions, whereas truth of existential formulas is examined over *must*-transitions. We follow this approach.

Full branching time logic CTL* as an expressive fragment [6, 2, 5] of μ-calculus can describe properties in computation trees. There has been much work on model checking for sublogic of CTL* such as LTL and CTL [11, 3], but little on CTL*. This paper introduces ideas for full branching time temporal logic CTL*.

There are several approaches to studying computations: computational model approach, algebraic approach, logical approach and game-theoretic approach. In the field of model checking, one uses the

E. Denney, D. Giannakopoulou, C.S. Păsăreanu (eds.); The First NASA Formal Methods Symposium, pp. 26-35

logical approach to capture temporal ordering of events, the algebraic approach to model examined systems, and the computational model approach and the game-theoretic approach to do model checking. A typical model checking approach is the automata-theoretic approach. Kupferman, Vardi and Wolper have shown how to solve the model checking problem for branching time by using alternating automata [9]. In their approach the model checking problem is reduced to a non-emptiness checking problem of the alternating automaton composed as a product between a *Kripke structure* and an automaton expressing the interesting property. The game-theoretic approach to model checking can also be viewed as simulating alternating automata [10, 12]. Their winning conditions correspond to a special Rabin acceptance condition of the automata approach. In contrast to the automata-theoretic model checking approach it is not necessary to compose automata for the properties.

Related Work. Martin Lange and Colin Stirling proposed Model checking games for CTL* in [10]. They described a two-player CTL* *focus game* for *Kripke structure* on Boolean semantics. We propose a generalization of the two-player game for *Modal Transition System* on three-valued semantics. There were many papers of three-valued abstraction that proposed by Godefroid, Jagadeesan and Bruns [1, 7, 8]. They showed advantages of three-valued models. In this paper, we not only discuss advantages of three-valued models but also analyze the returning information for debugging, proving or refining such models. Sharon Shoham and Orna Grumberg proposed muti-valued games for μ-calculus [12]. We investigate games for CTL* which is the most expressive fragment of μ-calculus. To sum up, the contribution of this paper are: (1) new game-based approach to three-valued model checking, (2) new game-based algorithm showing winning strategy of each player for solving the game, (3) new analysis based on *focus game* for debugging, proving or refining abstract models.

2 Preliminaries

We introduce key notions behind the framework of abstraction and model checking. Let AP be a finite set of atomic propositions. We define that an atomic proposition p is in AP if and only if its negation $\neg p$ is in AP. In the rest of this paper we suppose that all models, both abstract and concrete, share the set AP.

The full branching time logic CTL* formulas are composed of propositions, negation, Boolean connectives, path quantifiers and temporal operators.

Definition 1 (Syntax of CTL*) *There are two types of formulas in CTL*: state formulas and path formulas. The syntax of state formulas is given by the following rules:*
If p is an atomic proposition, then p is a state formula.
If ψ_1 and ψ_2 are state formulas, then $\neg\psi_1$, $\psi_1 \vee \psi_2$ and $\psi_1 \wedge \psi_2$ are state formulas.
If ψ is a path formula, then $\mathrm{E}\psi$ and $\mathrm{A}\psi$ are state formulas.
If ψ is a state formula, then ψ is also a path formula.
If ψ_1, ψ_2 are path formulas then $\neg\psi_1, \psi_1 \vee \psi_2, \psi_1 \wedge \psi_2, \mathrm{X}\psi_1, \mathrm{F}\psi_1, \mathrm{G}\psi_1, \psi_1\mathrm{U}\psi_2$, and $\psi_1\mathrm{R}\psi_2$ are path formulas.

The F, G temporal operators can be replaced with U, R operators by rules: $\mathrm{F}\psi = \textbf{true}\ \mathrm{U}\psi$, $\mathrm{G}\psi = \textbf{false}\ \mathrm{R}\psi$. The set of subformulas $Sub(\varphi)$ for a given φ is defined in the usual way, except that
- $Sub(\varphi\mathrm{U}\psi) := \{\varphi\mathrm{U}\psi, X(\varphi\mathrm{U}\psi), \varphi \wedge X(\varphi\mathrm{U}\psi), \psi \vee (\varphi \wedge X(\varphi\mathrm{U}\psi))\} \cup Sub(\varphi) \cup Sub(\psi)$,
- $Sub(\varphi\mathrm{R}\psi) := \{\varphi\mathrm{R}\psi, X(\varphi\mathrm{R}\psi), \varphi \vee X(\varphi\mathrm{R}\psi), \psi \wedge (\varphi \vee X(\varphi\mathrm{R}\psi))\} \cup Sub(\varphi) \cup Sub(\psi)$.

We consider that every CTL* formula begins with a path quantifier "A" to ensure that it is a state formula. The following semantics of CTL* shows that this is not a restriction because of the equivalence $Q_1Q_2\varphi = Q_2\varphi$ for $Q_1, Q_2 \in \{\mathrm{A}, \mathrm{E}\}$.

Definition 2 (Semantics of CTL*) *Suppose that π^i denotes the suffix of π starting at s_i. Let ψ be CTL* formula and M be a model. If ψ is a state formula, the notation $M, s \models \psi$ means that ψ holds at state s*

in model M. Similarly, if ψ is a path formula, $M,\pi \models \psi$ means that ψ holds along path π in model M. The relation $\models$ is defined inductively as follows (assuming that ψ_1 and ψ_2 are state formulas and ψ'_1 and ψ'_2 are path formulas):

$M,s \models p \iff p \in L(s)$.
$M,s \models \psi_1 \vee \psi_2 \iff M,s \models \psi_1$ or $M,s \models \psi_2$.
$M,s \models \mathrm{E}\psi'_1 \iff$ there is a path π from s such that $M,\pi \models \psi'_1$.
$M,\pi \models \psi_1 \iff s$ is the first state of π and $M,s \models \psi_1$.
$M,\pi \models \psi'_1 \wedge \psi'_2 \iff M,\pi \models \psi'_1$ and $M,\pi \models \psi'_2$.
$M,\pi \models \mathrm{F}\psi'_1 \iff$ there exists a $k \geq 0$ such that $M,\pi^k \models \psi'_1$.
$M,\pi \models \psi'_1\mathrm{R}\psi'_2 \iff$ for all $j \geq 0$, if for every $i < j$ $M,\pi^i \not\models \psi'_1$ then $M,\pi^j \models \psi'_2$.

$M,s \models \neg\psi_1 \iff M,s \not\models \psi_1$.
$M,s \models \psi_1 \wedge \psi_2 \iff M,s \models \psi_1$ and $M,s \models \psi_2$.
$M,s \models \mathrm{A}\psi'_1 \iff$ for every path π starting from s, $M,\pi \models \psi'_1$.
$M,\pi \models \neg\psi'_1 \iff M,\pi \not\models \psi'_1$.
$M,\pi \models \psi'_1 \vee \psi'_2 \iff M,\pi \models \psi'_1$ or $M,\pi \models \psi'_2$.
$M,\pi \models \mathrm{X}\psi'_1 \iff M,\pi^1 \models \psi'_1$.
$M,\pi \models \mathrm{G}\psi'_1 \iff$ for all $i \geq 0$, $M,\pi^i \models \psi'_1$.
$M,\pi \models \psi'_1\mathrm{U}\psi'_2 \iff$ there exists a $k \geq 0$ such that $M,\pi^k \models \psi'_2$ and for all $0 \leq j < k$, $M,\pi^j \models \psi'_1$.

Consider that every concrete model is given as a *Kripke structure* (KS for short) over AP, denoted by M_c. A KS is four tuple $\langle S_c, S_c^0, R_c, L_c\rangle$ where S_c is a finite set of states. $S_c^0 \subseteq S_c$ is the set of initial states. $R_c \subseteq S_c \times S_c$ is a transition relation that must be total, that is, for every state $s_c \in S_c$ there is a state $s' \in S_c$ such that $s_c \rightarrow s'_c \in R_c$. $L_c : S_c \rightarrow 2^{AP}$ is a labeling function such that for every state s and every $p \in AP$, $p \in L_c(s)$ iff $\neg p \notin L_c(s)$. $[M_c \models \varphi] = \mathrm{tt}$ ($= \mathrm{ff}$) or $M_c \models \varphi$ ($M_c \not\models \varphi$) means that M_c satisfies (refutes) the CTL* formula φ.

An abstraction (S_a, γ) for S_c consists of a finite set of abstract states S_a and a total concretization function $\gamma : S_a \rightarrow 2^{S_c}$ that maps each abstract state to the (nonempty) set of concrete states it represents. The function $\alpha : 2^{S_c} \rightarrow S_a$ as the inverse of γ is said to be abstraction function. An abstract model M_a is said to be on two-valued semantics if it is a KS model. An abstract model is said to be on three-valued semantics if it is a *Modal Transition System* (MTS) model [1, 7, 8]. MTSs contain two types of transitions: *may*-transitions and *must*-transitions.

Definition 3 *A Modal Transition System M_a over AP is a tuple $\langle S_a, S_a^0, R_{may}, R_{must}, L_a\rangle$, where S_a is a nonempty finite set of states, $S_a^0 \subseteq S_a$ is the set of initial states, $R_{may} \subseteq S_a \times S_a$ and $R_{must} \subseteq S_a \times S_a$ are transition relations such that $R_{must} \subseteq R_{may}$. $L_a : S_a \rightarrow 2^{AP}$.*

R_{may} is the set of all possible transitions and R_{must} is the set of all inevitable transitions. Note that $R_{must} \subseteq R_{may}$, because all inevitable transitions are possible transitions. Consider a concrete KS M_c and an abstract MTS M_a of M_c. Let (S_a, γ) be the three-valued abstraction between M_c and M_a. Labelings in each state in S_a are constructed by the following rules:

$$\frac{p \in L_c(s) \wedge \neg p \in L_c(s')}{p \notin L_a(s_a)} \qquad \frac{p \in L_c(s) \wedge p \in L_c(s')}{p \in L_a(s_a)} \qquad \frac{\neg p \in L_c(s) \wedge \neg p \in L_c(s')}{\neg p \in L_a(s_a)}$$

where $s, s' \in \gamma(s_a)$. Note that it is possible that neither p nor $\neg p$ is in $L_a(s_a)$ though either p or $\neg p$ must be in $L_c(s_c)$. After state abstraction, transitions in M_a can be constructed by using the following rules:

$$\frac{\exists s_1 \in \gamma(s_a), \exists s_2 \in \gamma(s'_a) : s_1 \rightarrow s_2}{s_a \overset{may}{\rightarrow} s'_a} \qquad \frac{\forall s_1 \in \gamma(s_a), \exists s_2 \in \gamma(s'_a) : s_1 \rightarrow s_2}{s_a \overset{must}{\rightarrow} s'_a}$$

Other constructions of abstract models are based on *Galois connections*, which can be found in [7]. The three-valued semantics of CTL* over MTSs, denoted $[M \models^3 \varphi]$, preserving both satisfaction and refutation from the abstract model M_a to the concrete model M_c. However, a new truth value (the third value *unknown*), denoted by $\perp$, is introduced meaning that the truth value over the concrete model is not

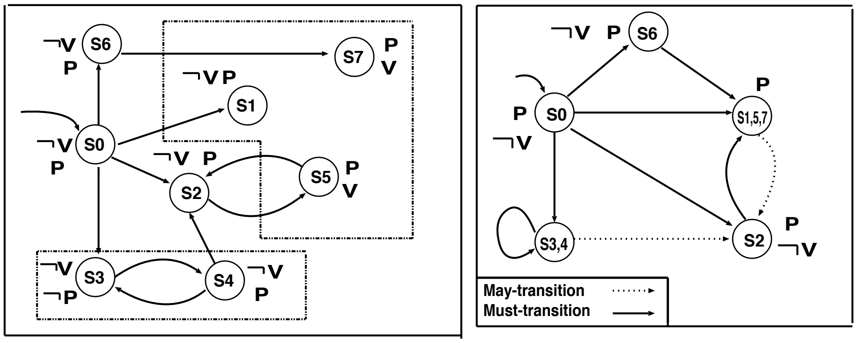


Figure 1: Example of Three-Valued Abstraction

known that can either be the truth value *true* or *false*.

Example 4 Let $AP = \{p, \neg p, v, \neg v\}$. Figure 1 shows a concrete KS model M_c (left) and its abstract MTS model M_a (right). Let (S_a, γ) be an (three-valued) abstraction, where $S_a = \{s_0, s_2, s_{3,4}, s_{1,5,7}, s_6\}$ and $\gamma(s_0) = \{s_0\}$; $\gamma(s_2) = \{s_2\}$; $\gamma(s_{3,4}) = \{s_3, s_4\}$; $\gamma(s_{1,5,7}) = \{s_1, s_5, s_7\}$; $\gamma(s_6) = \{s_6\}$.

3 Generalization of Focus Game

Model checking games for CTL* over two-valued models are two-player games, called *focus game*, which is proposed by Lange and Stirling [10]. There are two players in the *focus game*: the first player $\forall$ and the second player $\exists$. $\forall$'s task is to show that a formula is unsatisfied, while $\exists$'s task is to show the converse. In the two-player *focus game*, the set of *configurations* for a model (system) M and a formula φ, written in CTL*, is $conf(M, \varphi) = \{\forall, \exists, \perp\} \times S \times Sub(\varphi) \times 2^{Sub(\varphi)}$. A configuration is written $p, s, [\psi], \Phi$ where p is a player called the *path player*, $s \in S$, $\psi \in Sub(\varphi)$ and $\Phi \subseteq Sub(\varphi)$. Here ψ is said to be *in focus* and Φ are called side formulas. If p denotes a player then $\bar{p}$ denotes the other one in any round. A *play* between player p and $\bar{p}$ is a sequence of configurations. There are eighteen rules of the form

$$\frac{p, s, [\varphi], \Phi}{p', s, [\varphi'], \Phi'} p''$$

for transforming configurations, where $p, p', p'' \in \{\forall, \exists, \perp\}$ denote players. We follow these definitions and propose a three-player game for evaluating a CTL* formula φ on an abstract MTS model M_a with respect to three-valued semantics. We generalize the game by inserting a new player (the third player) $\perp$ and setting the game played in two rounds. Without loss of generality, we match $\forall$ *vs.* $\perp$ in the first round and $\exists$ *vs.* $\perp$ in the second round. We define that $\forall$ wins the game if $\forall$ wins the first round; $\exists$ wins the game if $\exists$ wins the second round; $\perp$ wins the game if $\perp$ wins both rounds. At each configuration the set of side formulas together with the formula in focus can be understood as a disjunction(resp. conjunction) of formulas in case the path player is $\forall$(resp. $\perp$) in the first round, $\perp$(resp. $\exists$) in the second round.

A play for model M with starting state s and a formula begins with the configuration $\forall, s, [\varphi]$ in the first round and with the configuration $\perp, s, [\varphi]$ in the second round. There are eighteen rules in the two-player focus game.

Path-chosen rules : Discarding rules :

$$(1)\frac{p, s, [A\varphi], \Phi}{\forall, s, [\varphi]} \qquad (2)\frac{p, s, [E\varphi], \Phi}{\exists, s, [\varphi]} \qquad (3)\frac{p, s, [\varphi], Q\psi, \Phi}{p, s, [\varphi], \Phi}\bar{p} \qquad (4)\frac{p, s, [\varphi], q, \Phi}{p, s, [\varphi], \Phi}\bar{p}$$

Boolean-connection rules :

$$(5)\frac{\forall, s, [\varphi_0 \wedge \varphi_1], \Phi}{\forall, s, [\varphi_i], \Phi}\forall \qquad (6)\frac{\forall, s, [\varphi_0 \vee \varphi_1], \Phi}{\forall, s, [\varphi_i], \varphi_{1-i}, \Phi}\exists \qquad (7)\frac{\exists, s, [\varphi_0 \vee \varphi_1], \Phi}{\exists, s, [\varphi_i], \Phi}\exists \qquad (8)\frac{\exists, s, [\varphi_0 \wedge \varphi_1], \Phi}{\exists, s, [\varphi_i], \varphi_{1-i}, \Phi}\forall$$

Unfolding rules :

$$(9)\frac{p, s, [\varphi \mathrm{U} \psi], \Phi}{p, s, [\psi \vee (\varphi \wedge \mathrm{X}(\varphi \mathrm{U} \psi))], \Phi} \qquad (10)\frac{p, s, [\varphi \mathrm{R} \psi], \Phi}{p, s, [\psi \wedge (\varphi \vee \mathrm{X}(\varphi \mathrm{R} \psi))], \Phi}$$

Progress rules :

$$(11)\frac{\forall, s, [\mathrm{X}\psi], \varphi_0 \wedge \varphi_1, \Phi}{\forall, s, [\mathrm{X}\psi], \varphi_i, \Phi}\forall (12)\frac{\forall, s, [\mathrm{X}\psi], \varphi_0 \vee \varphi_1, \Phi}{\forall, s, [\mathrm{X}\psi], \varphi_0, \varphi_1, \Phi} \qquad (13)\frac{\exists, s, [\mathrm{X}\psi], \varphi_0 \vee \varphi_1, \Phi}{\exists, s, [\mathrm{X}\psi], \varphi_i, \Phi}\exists (14)\frac{\exists, s, [\mathrm{X}\psi], \varphi_0 \wedge \varphi_1, \Phi}{\exists, s, [\mathrm{X}\psi], \varphi_0, \varphi_1, \Phi}$$

$$(15)\frac{p, s, [\mathrm{X}\chi], \varphi \mathrm{U} \psi, \Phi}{p, s, [\mathrm{X}\chi], \psi \vee (\varphi \wedge \mathrm{X}(\varphi \mathrm{U} \psi)), \Phi} \qquad (16)\frac{p, s, [\mathrm{X}\chi], \varphi \mathrm{R} \psi, \Phi}{p, s, [\mathrm{X}\chi], \psi \wedge (\varphi \vee \mathrm{X}(\varphi \mathrm{R} \psi)), \Phi}$$

To apply our three-player game, we restrict moves for different players. Since transitions may take place only in configurations with subformulas of the form $X\psi$, it is the only case where the rule (17) need to be applied to *may*-transitions and *must*-transitions.

$\forall$ and $\exists$ move on *must*-transitions: $\perp$ moves on *may* transitions:

$$(17a)\frac{p, s, [\mathrm{X}\varphi_0], \mathrm{X}\varphi_1, \cdots, \mathrm{X}\varphi_k}{p, t, [\varphi_0], \varphi_1, \cdots, \varphi_k} p \in \{\forall, \exists\}, s \overset{must}{\to} t \qquad (17b)\frac{\perp, s, [\mathrm{X}\varphi_0], \mathrm{X}\varphi_1, \cdots, \mathrm{X}\varphi_k}{\perp, t, [\varphi_0], \varphi_1, \cdots, \varphi_k} \perp, s \overset{may}{\to} t$$

The special rule (Change focus)

$$(18)\frac{p, s, [\varphi], \psi, \Phi}{p, s, [\psi], \varphi, \Phi}\bar{p}$$

A *move* in a play consists of two steps. First the path player and the focus determines which of the rules (1) – (17a) or (1) – (17b) apply, and hence which player makes the next choice. After that the path player's opponent has the chance to reset the focus by using rule (18). A play is finished after a full move if it has reached one of the following configurations (finish conditions).

1. $p, s, [q], \Phi$.
2. $C = \exists, s, [\varphi \mathrm{U} \psi], \Phi$ after the play already went through C and $\forall$ never applied (18) in between.
3. $C = \forall, s, [\varphi \mathrm{R} \psi], \Phi$ after the play already went through C and $\exists$ never applied (18) in between.
4. $p, s, [\varphi], \Phi$ for the second time possibly using rule (18) in between.
6. $\forall, s, [X\psi], \mathrm{X}\varphi_1, \cdots, \mathrm{X}\varphi_k$ and the rule (17a) can not be applied.
7. $\exists, s, [X\psi], \mathrm{X}\varphi_1, \cdots, \mathrm{X}\varphi_k$ and the rule (17a) can not be applied.

The winning criteria for three-player game are:

If the rule (17b) has been applied in a play and the play ends with one of the above finish conditions then $\perp$ wins, else

In the first round ($\forall$ **vs.** $\perp$)

1. When a play ends with the first finish condition, $\forall$ wins if $\neg q \in L(s)$, otherwise $\perp$ wins.
2. When a play ends with the second finish condition, $\forall$ wins.
3. When a play ends with the third finish condition, $\perp$ wins.
4. When a play ends with the fourth finish condition, the path player p wins if the second or the third finish condition does not apply.
5. Whenever a play ends with the fifth or the sixth finish condition, $\perp$ wins.

In the second round ($\exists$ **vs.** $\perp$)

1. When a play ends with the first finish condition, $\exists$ wins if $q \in L(s)$, otherwise $\perp$ wins.
2. When a play ends with the second finish condition, $\perp$ wins.
3. When a play ends with the third finish condition, $\exists$ wins.
4. When a play ends with the fourth finish condition, the path player p wins if the second or the third finish condition does not apply.
5. Whenever a play ends with the fifth or the sixth finish condition, $\perp$ wins.

The second round of the game is not always played. It is played if $\perp$ wins the first round, else the game is over and $\forall$ wins the game. Note that the game on three-valued semantics is an unfair game. Players $\forall$ and $\exists$ cannot move on all *may*-transitions whereas $\perp$ can move.

Let $\Gamma_{CTL^*}(M,s,\varphi)$ be a game over M for a CTL* formula φ. A game $\Gamma_{CTL^*}(M,s,\varphi)$ can be described as trees of all possible plays.

Definition 5 *A (game) tree is said to be a winning tree for player p, if p wins every branch (play) in it.*

We say that the player *p wins* or has a *winning strategy* for $\Gamma_{CTL^*}(M,s,\varphi)$ if p can force every play into a configuration that makes p win the play. A player p wins a round if p wins all possible plays in that round.

Example 6 Let $\varphi = AX(p) \vee EF(v)$ be a property that we want to check. Let M_a be the abstract model given in figure 1. We show that $\perp$ wins the game ($\perp$ wins both rounds) with the following winning trees.

The first round.

$\forall, s_0, [\varphi]$	
$\forall, s_0, [AX(p)], EF(v)$	$\forall, s_0, [EF(v)], AX(p)$
$\forall, s_0, [AX(p)]$	$\perp, s_0, [F(v)]$
$\forall, s_0, [X(p)]$	$\perp, s_0, [v \vee XF(v)]$
$\forall, s_{3,4}, [p]$	$\perp, s_0, [XF(v)]$
	$\perp, s_{1,5,7}, [F(v)]$
	$\perp, s_{1,5,7}, [v \vee XF(v)]$
	$\perp, s_{1,5,7}, [v]$

The second round.

$\perp, s_0, [\varphi]$	
$\perp, s_0, [AX(p)], EF(v)$	$\perp, s_0, [EF(v)], AX(p)$
$\perp, s_0, [EF(v)], AX(p)$	$\exists, s_0, [F(v)]$
$\cdots$	$\exists, s_0, [v \vee XF(v)]$
	$\exists, s_0, [XF(v)]$
	$\exists, s_{1,5,7}, [F(v)]$
	$\exists, s_{1,5,7}, [v \vee XF(v)]$
	$\exists, s_{1,5,7}, [v]$

4 Game-Based Algorithm

In the rest of this paper we assume that M_c, as a Kripke structure, represents a given concrete model and M_a, a Modal Transition System, denotes the abstract model of M_c. Let φ be a CTL* formula that represents the property we are interested in and $\Gamma_{CTL^*}(M_a,s,\varphi)$ be the three-player model checking game on the abstract MTS M_a.

We propose a game-based model checking algorithm, called **Mark Configuration**, for solving the game. **Mark Configuration** marks every configuration with one of symbols $\{\forall,\exists,\perp\}$ in each round. Let $C = p,s,[\varphi]$ be the starting configuration. **Mark Configuration** runs recursively and finally marks the starting configuration C with one of the three symbols.

Mark Configuration (the 1st round)

```
1.    BEGIN Mark($C$)
2.    INITIAL : history = $\emptyset$ ;
3.1   IF $\varphi = A\psi$ THEN Mark($\forall, s, [\psi]$)
3.2   ELSE IF $\varphi = E\psi$ THEN Mark($\perp, s, [\psi]$)
3.3   ELSE
3.4     SWITCH($\varphi$)
3.5       CASE 1 – CASE 6 ;
4.    END
```

Mark Configuration (the 2st round)

```
1.    BEGIN Mark($C$)
2.    INITIAL : history = $\emptyset$ ;
3.1   IF $\varphi = A\psi$ THEN Mark($\perp, s, [\psi]$)
3.2   ELSE IF $\varphi = E\psi$ THEN Mark($\exists, s, [\psi]$)
3.3   ELSE
3.4     SWITCH($\varphi$)
3.5       CASE 1 – CASE 6 ;
4.    END
```

From the syntax of CTL*, six types of formula φ can be considered when it is neither started by A nor E. That is $\varphi = q \mid \psi_1 \wedge \psi_2 \mid \psi_1 \vee \psi_2 \mid X\psi \mid \psi_1 U \psi_2 \mid \psi_1 R \psi$. Each of them corresponds to one case in this algorithm. When the focus formula φ is q, $\psi_1 \wedge \psi_2$ or $\psi_1 \vee \psi_2$, the configuration C's mark is decided by its children's marks. Let C_1', C_2' be configurations with subformula ψ_1, ψ_2, respectively. We represents case 1 – 3 as follows.

1. $C = p,s,[q]$ (where $p \in \{\forall,\exists,\bot\}$)
   If p wins in C then return p else return $\bar{p}$;

2. $C = p,s,[\psi_1 \wedge \psi_2]$ (where $p \in \{\forall,\exists,\bot\}$).
   if $\mathbf{Mark}(C_1') = \exists$ and $\mathbf{Mark}(C_2') = \exists$ then return $\exists$;
   if $\mathbf{Mark}(C_1') = \forall$ or $\mathbf{Mark}(C_2') = \forall$ then return $\forall$;
   if $\mathbf{Mark}(C_1') = \exists$ and $\mathbf{Mark}(C_2') = \bot$ or $\mathbf{Mark}(C_1') = \bot$ and $\mathbf{Mark}(C_2') = \exists$ then return $\bot$;

3. $C = p,s,[\psi_1 \vee \psi_2]$ (where $p \in \{\forall,\exists,\bot\}$).
   if $\mathbf{Mark}(C_1') = \forall$ and $\mathbf{Mark}(C_2') = \forall$ then return $\forall$;
   if $\mathbf{Mark}(C_1') = \exists$ or $\mathbf{Mark}(C_2') = \exists$ then return $\exists$;
   if $\mathbf{Mark}(C_1') = \forall$ and $\mathbf{Mark}(C_2') = \bot$ or $\mathbf{Mark}(C_1') = \bot$ and $\mathbf{Mark}(C_2') = \forall$ then return $\bot$;

The function *must-next* (*may-next*) is assumed to calculate all possible successors of a configuration by one move from rules 17a (17b). The configuration C's mark in case $\varphi = X\psi$ is determined not just by its children's marks but also by who the current player is and which the current round is. We distinguish different players in different rounds. The case 4 is as follows.

4a. $C = \forall,s,[X\psi]$ (1st round) or $\bot,s,[X\psi]$ (2nd round).
if there is a $C' \in$ *must-next*(C) and $\mathbf{Mark}(C') = \forall$ then return $\forall$; if for all $C' \in$ *must-next*(C): $\mathbf{Mark}(C') = \exists$ then return $\exists$; if there is $C' \in$ *may-next*(C): $\mathbf{Mark}(C') = \bot$ and for any other $C'' \in$ *may-next*(C): $\mathbf{Mark}(C'') = \exists$ or $\mathbf{Mark}(C'') = \bot$ then return $\bot$;

4b. $C = \bot,s,[X\psi]$ (1st round) or $\exists,s,[X\psi]$ (2nd round).
if for all $C' \in$ *may-next*(C) : $\mathbf{Mark}(C') = \forall$ then return $\forall$; if there is a $C' \in$ *must-next*(C) and $\mathbf{Mark}(C') = \exists$ then return $\exists$; if there is a $C' \in$ *may-next*(C) and $\mathbf{Mark}(C') = \bot$ and for any other $C'' \in$ *may-next*(C): $\mathbf{Mark}(C'') = \forall$ or $\mathbf{Mark}(C'') = \bot$ then return $\bot$;

To determine the configuration C's mark in case $\varphi = \psi_1 \mathrm{U} \psi_2$ and case $\varphi = \psi_1 \mathrm{R} \psi$, first we should look for who is the path player in C. Next we check whether C is the starting configuration of a loop. The variable *history* is used in recording checked configurations in loops on any path. The case 5, 6 are as follows.

5a. $C = p,s,[\psi_1 \mathrm{U} \psi_2]$.
(where $p = \forall$ in 1st round, $p = \bot$ in 2nd round)
if $C \in$ *history* then marks all $C' \in$ *history* with p;
history $:= \emptyset$; return p;
else *history* $:= \{C\} \cup$ *history*;
$\mathbf{Mark}(p,s,[\psi_2 \vee (\psi_1 \wedge X\psi_1 \mathrm{U} \psi_2)])$;

5b. $C = p,s,[\psi_1 \mathrm{U} \psi_2]$
(where $p = \bot$ in 1st round, $p = \exists$ in 2nd round)
if $C \in$ *history* then marks all $C' \in$ *history* with $\bar{p}$;
history $:= \emptyset$; return $\bar{p}$;
else *history* $:= \{C\} \cup$ *history*;
$\mathbf{Mark}(p,s,[\psi_2 \vee (\psi_1 \wedge X\psi_1 \mathrm{U} \psi_2)])$;

6a. $C = p,s,[\psi_1 \mathrm{R} \psi_2]$
(where $p = \forall$ in 1st round, $p = \bot$ in 2nd round)
if $C \in$ *history* then marks all $C' \in$ *history* with $\bar{p}$;
history $:= \emptyset$; return $\bar{p}$;
else *history* $:= \{C\} \cup$ *history*;
$\mathbf{Mark}(p,s,[\psi_2 \wedge (\psi_1 \vee X\psi_1 \mathrm{R} \psi_2)])$;

6b. $C = p,s,[\psi_1 \mathrm{R} \psi_2]$
(where $p = \bot$ in 1st round, $p = \exists$ in 2nd round)
if $C \in$ *history* then marks all $C' \in$ *history* with p;
history $:= \emptyset$; return p;
else *history* $:= \{C\} \cup$ *history*;
$\mathbf{Mark}(p,s,[\psi_2 \wedge (\psi_1 \vee X\psi_1 \mathrm{R} \psi_2)])$;

Proposition 7 (Terminating) *The algorithm* **Mark Configuration** *always terminates.*

Proof. The total number of all configuration can be calculated as $|S| \cdot 2^{|\varphi|}$. Every configuration is marked by **Mark Configuration** only once. □

Proposition 8 *Assume that* **Mark Configuration** *marks all configurations in the graph of* $\Gamma_{CTL^*}(M_a, s, \varphi)$. *The following two statements hold.*
1. Every configuration C is marked with one of $\forall$, $\exists$, $\bot$.
2. If a configuration C is marked with $p \in \{\forall, \exists, \bot\}$ *then p wins the (sub) game of* $\Gamma_{CTL^*}(M_a, s, \varphi)$ *that is started from C.*

Proof. The first statement follows from the fact that every case in **Mark Configuration** marks configurations with one symbol and each case corresponds to one syntax element of the focus formula in C. We show the second statement by the induction on structure of game tree that rooted by C. Without loss of generality, assume that $p \in \{\forall, \exists, \bot\}$ wins a game. When any winning tree of p consists of a single configuration, since the case 1 can be applied, C is marked with the symbol p. When any winning tree of p consists of an infinite sequence, in which the configuration C appears infinitely often, the focus formula of C must be either of the form $\psi_1 \mathrm{U} \psi_2$ or $\psi_1 \mathrm{R} \psi_2$. Suppose that the focus formula is U-formula and C is marked with $\forall$ or $\bot$. According to the case 5a or case 5b, $\forall$ or $\bot$ wins, since ψ_2 never holds. Suppose that the focus formula is R-formula and C is marked with $\exists$ or $\bot$. According to the case 6a or case 6b, $\exists$ or $\bot$ wins, since ψ_2 always holds. For any other structure of the game tree, there is at least a next configuration C' as a child of C that is marked with p. According to the remaining cases in **Mark Configuration**, C's mark is decided by the mark of C' in each corresponding case. By the induction hypothesis, C' is marked with p and it is deduced that p wins the game started by C. □

Proposition 9 *Consider a game has been marked by* **Mark Configuration**. *Let* C_s *be the starting configuration marked with* $\chi (\in \{\forall, \exists, \bot\})$. *For any configuration* C_i, *if* C_i *is marked with* χ *and may-next(*C_i*)* $\neq \emptyset$ *then there is at least one configuration* C_j *such that* $C_j \in$ *may-next(*C_i*) and* C_j *is marked with* χ.

Proof. This follows from the observation that the **Mark Configuration** recursively marks every configuration depending on the marks of its child vertices. The starting configuration is guaranteed to be marked eventually by the exhaustiveness of the search. □

Lemma 10 *The following statements hold.*
1. Every play terminates.
2. Every play has a uniquely determined winner.
3. item For every round of $\Gamma_{CTL^*}(M_a, s, \varphi)$ *one of the players has a winning strategy.*
4. One player wins the game iff the other players do not win.

Proof. Lange and Stirling [10] showed that these four statements hold in two-player games. In $\Gamma_{CTL^*}(M_a, s, \varphi)$, each round can be seen as a two-player game with more constraints. In particular, rules (17a) and (17b) do not introduce new moves to every player. Therefore, these four statements hold in each round, which derives this lemma. □

Theorem 11 (Soundness)
1. $\forall$ *wins* $\Gamma_{CTL^*}(M_a, s, \varphi) \;\Rightarrow\; M_c \not\models \varphi$.
2. $\exists$ *wins* $\Gamma_{CTL^*}(M_a, s, \varphi) \;\Rightarrow\; M_c \models \varphi$.
3. $\bot$ *wins* $\Gamma_{CTL^*}(M_a, s, \varphi) \;\Rightarrow\;$ *both* $M_c \models \varphi$ *and* $M_c \not\models \varphi$ *are possible.*

Proof. We now show the statement 1 (statement 2). Every play in $\forall$'s ($\exists$'s) winning tree terminates either in a loop or at a terminating configuration. If it terminates in a loop then there exists a configuration appearing twice and during the loop $\perp$ ($\exists$) cannot (can) win with the finish condition 2 or 4 (3 or 4). Otherwise it terminates at a configuration in which $\forall$ ($\exists$) wins with the finish condition 1. According to our constraint and winning criteria, the rule (17b) cannot be applied in whole $\forall$'s ($\exists$'s) winning tree. It implies that all plays in winning tree are based on *must*-transition in M_a. Therefore, $\forall$ ($\exists$) can also win on the model M_c. That is $M_c \not\models \varphi$ ($M_c \models \varphi$).
We show the statement 3. $\perp$ has winning trees for both rounds. There must be a play that either terminates at a terminating configuration, or uses the rule (17b) that is based on a transition in $R_{may} - R_{must}$ of M_a. Suppose it terminates at a configuration, denoted by $p, s_a, [q], \Phi$, with finish condition 1. According to abstraction, there are two states $s_c, s'_c \in \gamma(s_a)$ such that $q \in L_c(s_c)$ and $\neg q \in L_c(s'_c)$, or $\neg q \in L_c(s_c)$ and $q \in L_c(s'_c)$. Hence, both $\forall$ wins in M_c and $\exists$ wins in M_c are possible. Information is not sufficient to show $M_c \models \varphi$ or $M_c \not\models \varphi$. □

Lemma 12 *The following two statements hold.*
1. $M_c \models \varphi \;\Rightarrow\; [M_a \models^3 \varphi] = \text{tt}$ or $[M_a \models^3 \varphi] = \perp$.
2. $M_c \not\models \varphi \;\Rightarrow\; [M_a \models^3 \varphi] = \text{ff}$ or $[M_a \models^3 \varphi] = \perp$.

Proof. By the tree-valued abstraction, they are trivial. □

Lemma 13 *The following three statements hold.*
1. $[M_a \models^3 \varphi] = \text{tt} \;\Rightarrow\; \exists$ *wins* $\Gamma_{CTL^*}(M_a, s, \varphi)$.
2. $[M_a \models^3 \varphi] = \text{ff} \;\Rightarrow\; \forall$ *wins* $\Gamma_{CTL^*}(M_a, s, \varphi)$.
3. $[M_a \models^3 \varphi] = \perp \;\Rightarrow\; \perp$ *wins* $\Gamma_{CTL^*}(M_a, s, \varphi)$.

Proof. We show a winning strategy based on **Mark Configuration** for each player $p \in \{\forall, \exists, \perp\}$. Assume that **Mark Configuration** has been applied in the game $\Gamma_{CTL^*}(M_a, s, \varphi)$. Let $p \in \{\forall, \exists, \perp\}$ be a player and C be the current configuration. It does not matter whether p is the path player in C or not. Since $\perp$ can play $\exists$'s role in the first round and can play $\forall$'s role in the second round, we distinguish $\perp$ from other players.
$\forall$'s and $\exists$'s winning strategies: If there is a next configuration C' of C such that C' is marked with symbol p, then select C' by using one of rules 1 – 16, 17a or 18; Else do nothing.
$\perp$'s winning strategy: If there is a next configuration C' of C such that C' is not marked with the mark of $\perp$'s opponent, then select C' by using one of rules 1 – 16, 17b or 18; Else do nothing.
We use Proposition 8, 9 to prove that such strategy is a winning strategy for each corresponding player. Proposition 8 shows that a player p wins the game $\Gamma_{CTL^*}(M_a, s, \varphi)$ if the starting configuration is marked with p. Proposition 9 shows that for any configuration C marked with symbol p, there is a C' such that C' is a next configuration of C and it is also marked with p. By the definition of each player's task, we have $\exists$ wins if $[M_a \models^3 \varphi] = \text{tt}$, $\forall$ wins if $[M_a \models^3 \varphi] = \text{ff}$, $\perp$ wins if $[M_a \models^3 \varphi] = \perp$. □

Theorem 14 (Completeness)
- $M_c \models \varphi \;\Rightarrow\; \exists$ *wins* $\Gamma_{CTL^*}(M_a, s, \varphi)$ *or* $\perp$ *wins.*
- $M_c \not\models \varphi \;\Rightarrow\; \forall$ *wins* $\Gamma_{CTL^*}(M_a, s, \varphi)$ *or* $\perp$ *wins.*

Proof. It directly follows from the Lemma 12 and 13. □

Refinement issues. The main advantage of game-based model checking approach is availability of more precise debugging information on the examined system. Using games is not necessary to create an additional debugger, because the game-based approach annotates each state on the proofs/counterexamples or refinements with a sub-formula of the interesting temporal formula φ that is true/ false or unknown in that state. The annotating sub-formulas being true/false or unknown in the respective states, provide the reason for φ to be true/false or unknown. By analyzing such information we can figure out where errors have occurred and what the causes of the errors are.

Complexity issues. The best currently known complexity for CTL* model checking is in PSPACE time. So is our algorithm. Let all sub-games are started from formula $A\psi$ or $E\psi$. **Mark Configuration** can be applied in every sub-tree in each round. There are at most $|S| \cdot |\varphi|/2$ sub-games. **Mark Configuration** might have to be invoked $|S| \cdot |\varphi|/2$ times. After a sub-game, the space it needs can be released. Thus the algorithm for each round as a two-player game costs PSPACE time. The total complexity is the same class as the complexity of a round.

Conclusions. We presented two problems in the beginning of this paper: the state space explosion problem and the system debugging problem. To overcome both of these problems, we proposed an game-based approach by combining several powerful techniques: abstraction, refinement and three-valued logic. The abstraction on three-valued semantics was used to overcome the first problem. The analysis of the game-based model checking was used to solve the second problem. We also proposed a game-based algorithm for model checking, and proved its termination, soundness and completeness.

References

[1] Glenn Bruns and Patrice Godefroid. Model checking partial state spaces with 3-valued temporal logics. *In Proceedings of the 11th Conference on Computer Aided Verification Lecture Notes in Computer Science*, 1877, July 1999.

[2] Glenn Burns and Patrice Godefroid. Model checking with multi-valued logics. *In Proceedings of the 31st ICALP LNCS*, May 2004.

[3] Edmund M. Clarke and E. Allen Emerson. Synthesis of synchronization skeletons for branching time temporal logic. *Logic of Programs LNCS*, 131, 1981.

[4] Orna Grumberg Edmund M. Clarke and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16, 1994.

[5] Orna Grumberg Edmund M. Clarke, Jr. and Doron A. Peled. *Model Checking*. The MIT Press Cambridge, Massachusetts, London, England, 1999.

[6] Allen E. Emerson and Prasad A. Sistla. Deciding branching time logic. *In Proceedings of the 16th Annual ACM Symposium on Theory of Computing*, March 1984.

[7] Patrice Godefroid and Radaha Jagadeesan. Abstraction-based model checking using modal transition systems. *In Proceedings of CONCUR'(12th International Conference on Concurrency Theory),*, 2001.

[8] Patrice Godefroid and Radaha Jagadeesan. Automatic abstraction using generalized model checking. *In Proceedings of CAV'(14th Conference on Computer Aided Verification),*, July 2002.

[9] Orna Kupferman, Moshe Y. Vardi, and Pierre Wolper. An automata-theoretic approach to branching-time model checking. *Journal of the Association for Computing Machinery*, March 2000.

[10] Martin Lange and Colin Stirling. Model checking games for branching time logics. *Oxford University Press J.Logic Computat.*, 12, 2002.

[11] Amir Pnueli. The temporal logic of programs. *18th IEEE Symposium on the Foundations of Computer Science 46–57*, 1977.

[12] Sharon Shoham and Orna Grumberg. Muti-valued model checking games. *Automated Technology for Verification and Analysis*, 3707, 2005.

Software Model Checking without Source Code*

Sagar Chaki James Ivers
Software Engineering Institute, Carnegie Mellon University
Pittsburgh, PA
(chaki|jivers)@sei.cmu.edu

Abstract

We present a framework, called AIR, for verifying safety properties of assembly language programs via software model checking. AIR extends the applicability of predicate abstraction and counterexample guided abstraction refinement to the automated verification of low-level software. By working at the assembly level, AIR allows verification of programs for which source code is unavailable–such as legacy and COTS software–and programs that use features–such as pointers, structures, and object-orientation–that are problematic for source-level software verification tools. In addition, AIR makes no assumptions about the underlying compiler technology. We have implemented a prototype of AIR and present encouraging results on several non-trivial examples.

1 Introduction

Over the past decade, there has been considerable advancement in the theory and practice of automated formal software verification. One of the most promising paradigms to emerge in this area is software model checking (SMC) [3] – a combination of counterexample guided abstraction refinement [11] with predicate abstraction [18]. SMC verifies that a program P satisfies a specification ϕ iteratively, as follows:

1. **(Abstraction)** Construct a conservative model M from P via predicate abstraction. Go to Step 2.
2. **(Verification)** Model check $M \models \phi$. If this is the case, then terminate with result $P \models \phi$. Otherwise let CE be a counterexample to $M \models \phi$ returned by the model checker. Go to Step 3.
3. **(Validation)** Check whether CE corresponds to some concrete behavior of P. If this is the case, then we obtain a real counterexample and terminate with $P \not\models \phi$. Otherwise, CE is a spurious counterexample. Go to Step 4.
4. **(Refinement)** Construct a more precise model M' that does not admit CE as a behavior and repeat from Step 2 with $M = M'$.

Variations of the above process have been investigated by several research groups [3, 19, 9] with considerable success on *source code* derived from real-life examples. However, there has been considerably less work on applying SMC to verify *machine-level* programs. In this paper, we show that in spite of the absence of high-level information, such as variable names and branch conditions, the effectiveness of SMC extends to even low-level software. More specifically, we present a SMC-based procedure for verifying safety properties of PowerPC$^{\text{TM}}$ assembly programs. Our approach, which we call Assembly Iterative Refinement or AIR, consists of two broad phases:

1. **Decompilation:** In this stage, we translate the target assembly program A and safety property ϕ_A into an equivalent C program P and safety property ϕ_P. In essence, we treat each register as a global variable. Each procedure in A leads to a corresponding procedure in P, and each assembly instruction produces one or more C statements. We present further details of the decompilation process in Section 3. Note that C is particularly suited as the target language of decompilation because: (i) C supports bit-level operations that are critical for preserving the semantics of assembly instructions during decompilation; (ii) we are able to build on existing infrastructures for C verification to perform the next stage of AIR.

E. Denney, D. Giannakopoulou, C.S. Păsăreanu (eds.); The First NASA Formal Methods Symposium, pp. 36-45

*This research was conducted as part of the Predictable Assembly from Certifiable Components (PACC) initiative at the SEI. The SEI is a Federally Funded Research and Development Center sponsored by the US Dept. of Defense and operated by Carnegie Mellon University.

2. **Verification:** In this stage, we use SMC to check $P \models \phi_P$. Since decompilation is semantics-preserving, the result obtained in this stage for P also applies to the original assembly program A. Furthermore, any counterexample obtained with respect to P is transformed to a corresponding counterexample with respect to A. Further details about verification can be found in Section 4.

AIR extends the applicability of SMC to assembly program verification, and yields several tangible benefits. First, AIR does not require source code, and thus is applicable to software – such as legacy, proprietary, and commercial-off-the-shelf (COTS) software – for which source code is often not available. Second, unlike source code analysis, AIR analyzes exactly what is to be executed and makes no assumptions about any compiler technology being used. Thus, it eliminates the need to ensure compiler correctness, and reduces the size of the trusted computing base. This is especially desirable when analyzing safety-critical systems. Third, AIR is not tied to any specific high-level programming language, and consequently is more versatile than source-code verification. In particular, AIR is able to sidestep features, such as pointers, structures, and object-orientation, which are problematic for source-level analysis tools. Finally, since AIR decompilation is semantics preserving and targets the C language, it enables us to leverage existing and emerging C analysis tools for verification. Thus, even though we experiment with SMC-based tools, other C verifiers (e.g., CBMC [7] and F-Soft [21]) are also applicable.

We have implemented AIR and obtained encouraging results on several non-trivial benchmarks derived from Linux device drivers and an embedded OS. Further details can be found in Section 5. The rest of this paper is organized as follows. In Section 2, we survey related work. The decompilation and verification stages of AIR are described in Sections 3 and 4, respectively. Finally, we present experimental results in Section 5 and conclude in Section 6.

2 Related Work

Following SLAM [3], several other projects – e.g., MAGIC [9] and BLAST [19] – have investigated the use of the SMC paradigm for C source code verification. A number of projects – such as SPIN [20], Java PathFinder [32], BANDERA [17], BOGOR [16], Behave! [4], and ZING [1] – have also looked at software verification, but not necessarily via SMC. Instead, their focus has been on other languages, such as Java, and other program features, such as concurrency. Decompilers have been traditionally developed for binary understanding and reverse engineering, and not for verification *per se*. Nevertheless, the use of decompilation for verification has been suggested by Breuer and Bowen [6], and by Curzon [14] for verifying micro-code.

The verification of low-level software [12, 27] has also received a lot of attention. A number of approaches are based on either theorem proving, type checking, or static analysis. For example, Boyer and Yu have verified object code for the MC68020 processor using the Nqthm theorem prover [5]. Yu [33] has proposed the use of certified assembly programming and type preserving translations for ensuring the safety of low-level code. His techniques are powerful but require considerable manual intervention, e.g., via type annotations and the use of proof assistants. Yu and Shao [34] have also proposed a logic based type system for the static verification of concurrent assembly programs.

Reps et al. [30] have used static analysis algorithms to recover information about the contents of memory locations and how they are manipulated by executables. They have also created CodeSurfer/x86, a prototype tool for browsing, inspecting and analyzing x86 binaries. Our technique is based on model checking, is completely automated, and targets PowerPC$^{\text{TM}}$ assembly code. Balakrishnan et al. [2] have used model checking to analyze stripped device driver executables. Their approach is not based on decompilation to C, but on a tight combination of their own model checker and control-flow-graph-based internal representation for the target executables.

Another approach for ensuring the correctness of low-level programs is source code verification combined with compiler validation, i.e., proving that the compiler always produces a target code which correctly implements the source code. In practice, proving compiler correctness is extremely tedious. Furthermore, any change in the compiler necessitates its revalidation. Our technique is impervious to the underlying compiler technology. A complementary technique is translation validation [28] where instead of validating the compiler, each individual run of the compiler is followed by a validation phase which verifies that the target code produced on that run correctly implements the submitted source program. Both compiler validation and translation validation assume that the source code is available and has been independently verified. Our approach does not require such an assumption.

3 Decompilation

The first stage of AIR is the decompilation of the target assembly program A into an equivalent C program P. In this section we describe the decompilation procedure using a small assembly program as a running example. For ease of understanding, we start with the source code from which A was compiled. Fig. 1 shows a small C code fragment P_0 on the left and the result of compiling it with `gcc` on the right. P_0 is derived from MICRO-C, a lightweight operating system for real-time embedded applications. For brevity and simplicity we eliminated some irrelevant code from the actual MICRO-C sources. Also, the assembly A on the right of Fig. 1 does not contain some book-keeping information generated by `gcc`.

We use the 32-bit variant of the PowerPC^{TM} instruction set architecture (ISA) and assume little-endian mode. The PowerPC^{TM} architecture defines 32 32-bit general purpose registers (GPRs) – referred to in A as `%r0` through `%r31`. In addition, there are 32 64-bit floating point registers (FPRs) – referred to as `%f0` through `%f31` – and a few special registers (SPRs), e.g., condition register (`%cr`), link register (`%lr`), etc. An assembly program consists of a set of blocks, each beginning with a label. A label is either a procedure name (`OSMemNameSet`) or begins with a dot (`.L1 ... .L4`). The procedures `OS_ENTER_CRITICAL` and `OS_EXIT_CRITICAL` acquire and release a global lock for achieving mutual exclusion. We wish to verify whether our program satisfies the following property: **(Safety)** `OS_ENTER_CRITICAL` and `OS_EXIT_CRITICAL` are invoked alternately, beginning with a call to `OS_ENTER_CRITICAL`. Note that **Safety** is representative of a general class of safety specifications with respect to the acquisition and release of resources. Also, our example program does not satisfy **Safety**. Indeed, if the conditions of the first two `if` statements are both satisfied, then `OS_EXIT_CRITICAL` gets called twice in a row without any intervening call to `OS_ENTER_CRITICAL`. One possible fix for this problem is to add a `return` statement as indicated by the comment in the C code.

3.1 From assembly to C

The decompilation process converts the assembly program A to a C program, using the following general strategy:

- The 32 GPRs are declared as global `int` variables `r0` through `r31`. The 32 FPRs are declared as global `double` variables `f0` through `f31`. The SPRs are also declared as `int` variables `cr, lr` and so on. All integer data is assumed to be in signed (two's complement) 32-bit format and all double data is assumed to be in IEEE 64-bit double precision format.
- Each label corresponding to a procedure name yields a procedure declaration. Since an assembly program passes and returns all values via registers (i.e., global variables), our procedures are `void-void`, i.e., they have no parameters or return values. In our example, we obtain a single procedure declaration:

```
void OSMemNameSet(void)
```

```
struct os_mem {
  void *OSMemAddr ;
  void *OSMemFreeList ;
  unsigned int OSMemBlkSize ;
  unsigned int OSMemNBlks ;
  unsigned int OSMemNFree ;
  char OSMemName[32] ;
};

typedef struct os_mem OS_MEM;

void OSMemNameSet(OS_MEM *pmem,
        char *pname,unsigned char *err )
{
  unsigned char len;
  OS_ENTER_CRITICAL();
  if ((unsigned int )pmem == (unsigned int )((OS_MEM *)0)) {
    OS_EXIT_CRITICAL();
    (*err) = 116;
    //bug : there should most likely be a return here
    //return;
  }
  if ((unsigned int )pname == (unsigned int )((char *)0)) {
    OS_EXIT_CRITICAL();
    (*err) = 15;
    return;
  }
  if ((int )len > 31) {
    OS_EXIT_CRITICAL();
    (*err) = 119;
    return;
  }
  OS_EXIT_CRITICAL();
  (*err) = 0;
  return;
}
```

```
OSMemNameSet:
  stwu %r1,-48(%r1)
  mflr %r0
  stw %r31,44(%r1)
  stw %r0,52(%r1)
  mr %r31,%r1
  stw %r3,8(%r31)
  stw %r4,12(%r31)
  stw %r5,16(%r31)
  bl OS_ENTER_CRITICAL
  lwz %r0,8(%r31)
  cmpwi %cr7,%r0,0
  bne %cr7,.L2
  bl OS_EXIT_CRITICAL
  lwz %r9,16(%r31)
  li %r0,116
  stb %r0,0(%r9)
.L2:
  lwz %r0,12(%r31)
  cmpwi %cr7,%r0,0
  bne %cr7,.L3
  bl OS_EXIT_CRITICAL
  lwz %r9,16(%r31)
  li %r0,15
  stb %r0,0(%r9)
  b .L1
.L3:
  lbz %r0,20(%r31)
  rlwinm %r0,%r0,0,0xff
  cmplwi %cr7,%r0,31
  ble %cr7,.L4
  bl OS_EXIT_CRITICAL
  lwz %r9,16(%r31)
  li %r0,119
  stb %r0,0(%r9)
  b .L1
.L4:
  bl OS_EXIT_CRITICAL
  lwz %r9,16(%r31)
  li %r0,0
  stb %r0,0(%r9)
.L1:
  lwz %r11,0(%r1)
  lwz %r0,4(%r11)
  mtlr %r0
  lwz %r31,-4(%r11)
  mr %r1,%r11
  blr
```

Figure 1: A running example.

- Each label beginning with a dot results in a corresponding label in the C program. We strip off the initial dot to conform to valid ANSI-C syntax. Thus, the C program generated in our example contains four labels L1 through L4.
- Each assembly instruction gets translated to an equivalent sequence of C statements. In the rest of this section, we describe the translation process for the instructions that appear in our example. Note that the size of the resulting C program is linear in the size of the input assembly program.

3.2 Translating assembly instructions

PowerPC$^{\text{TM}}$ follows the Reduced Instruction Set Computer (RISC) or the load-store paradigm. Thus, there are no arithmetic, logical, or control-flow instructions that operate directly on data stored in memory. All operations are performed on GPRs, FPRs, and SPRs. In order to operate on memory data, the operands are loaded explicitly into registers, and the result is stored explicitly back to memory.

Assembly	**C statements**
Loads and stores	
`lwz %r0,8(%r31)`	`r0 = *((int*)(r31 + 8));`
`li %r0,116`	`r0 = 116;`
`lbz %r0,20(%r31)`	`r0 = (*((int*)(r31 + 20))) & (0xff);`
`stwu %r1,-48(%r1)`	`*((int*)(r1 - 48)) = r1; r1 = r1 - 48;`
`stw %r31,44(%r1)`	`*((int*)(r1 + 44)) = r31;`

`stb %r0,0(%r9)`	`*((int*)(r9 + 0)) =`
	`((*((int*)(r9 + 0))) & 0xffffff00) | (r0 & 0xff);`

Register operations	
`mr %r31,%r1`	`r31 = r1;`
`mflr %r0`	`r0 = lr;`
`mtlr %r0`	`lr = r0;`

`rlwinm %r0,%r0,0,255`	`r0 = (((r0 >> 32) & 0) | ((r0 << 0) & 0xffffffff))`
	`& (0xff);`

`cmpwi %cr7,%r0,0`	`cr = (r0 < 0) ? (cr | 0x8) : (cr & 0xfffffff7);`
	`cr = (r0 > 0) ? (cr | 0x4) : (cr & 0xfffffffb);`
	`cr = (r0 == 0) ? (cr | 0x2) : (cr & 0xfffffffd);`

`cmplwi %cr7,%r0,31`	`cr = (r0 >= 0) && (r0 < 31) ?`
	`(cr | 0x8) : (cr & 0xfffffff7);`
	`cr = (r0 < 0) || (r0 > 31) ?`
	`(cr | 0x4) : (cr & 0xfffffffb);`
	`cr = (r0 >= 0) && (r0 == 31) ?`
	`(cr | 0x2) : (cr & 0xfffffffd);`

Conditional and unconditional jumps	
`b .L1`	`goto L1;`
`ble %cr7,.L4`	`if(!(cr & 0x4)) goto L4;`
`bne %cr7,.L2`	`if(!(cr & 0x2)) goto L2;`
`bl OS_ENTER_CRITICAL`	`OS_ENTER_CRITICAL();`
`blr`	`return;`

Figure 2: Translation schema from assembly instructions to C statements.

Fig. 2 shows a table with assembly instructions on the left and the corresponding C statements on the right. Among the instructions in Fig. 2, the following have straightforward translations: `li = load immediate`, `mr = move register`, `mflr = move from link register`, `mtlr = move to link register`, `b = branch`, `bl = branch link`, and `blr = branch link return`. The translations for the other instructions can be understood on the basis of their semantics, as described below:

- `lwz = load word and zero`: loads a word from the memory location denoted by the second argument to the register denoted by the first argument.
- `lbz = load byte and zero`: loads a byte from the source memory location *S* denoted by the second argument to the target register *T* denoted by the first argument. The higher-order 24 bits of *T* are set to zero. Due to little-endianness, if we load an integer from *S*, our desired byte will be laid out at the lower order end. Now we have to zero out the higher-order 24 bits.
- `stwu = store word with update`: stores the word in source register *S* denoted by the first argument to the target memory location *T* denoted by the second argument, and then sets the value of *S* to *T*.
- `stw = store word`: stores the word in source register *S* denoted by the first argument to the target memory location *T* denoted by the second argument.
- `stb = store byte`: stores the lowest byte in source register *S* denoted by the first argument to the target memory location *T* denoted by the second argument. Again due to little-endianness, we load the current word stored at *T*, replace its lowest byte with the lowest byte of *S*, and store the new value back to *T*.

- `rlwinm = rotate left word immediate then AND with mask`: rotates left the contents of the source register *S* (denoted by the second argument) by the number of bits *B* (denoted by the third argument), then logically ANDs the rotated data with a 32-bit mask *BM* (denoted by the fourth argument), and stores the result in the target register *T* (denoted by the first argument).

To understand the comparison and jump instructions, we note that the condition register `cr` is logically partitioned into eight sub-registers `cr0 ... cr7`. The sub-registers are numbered from the higher order bits to the lower order bits of `cr` as shown by the following diagram.

cr0	cr1	cr2	cr3	cr4	cr5	cr6	cr7

Thus, `cr7` denotes the lowest four bits of `cr`. Further, suppose that the results of a comparison between X and Y are stored in a condition sub-register R. Then the bits of R must be interpreted as follows. The highest bit is 1 if and only if $X < Y$, the next bit is 1 if and only if $X > Y$, and the next bit is 1 if and only if $X = Y$. The lowest bit is reserved for overflows. We now present the translation scheme for the remaining instructions.

- `cmpwi = compare word immediate`: compares the contents of the register denoted by the second argument with the integer denoted by the third argument treating both values as signed integers, and stores the result in the condition sub-register denoted by the first argument.

- `cmplwi = compare logical word immediate`: compares the contents of the register denoted by the second argument with the integer denoted by the third argument treating both values as unsigned integers, and stores the result in the condition sub-register denoted by the first argument. Since all our C variables are signed, we guard the C statement to be executed on conditions that check for negative values in addition to the actual comparison being performed.

- `ble = branch less equal` : jumps to the label denoted by the second argument if the condition sub-register denoted by the first argument indicates a "less or equal" comparison result.

- `bne = branch not equal` : jumps to the label denoted by the second argument if the condition sub-register denoted by the first argument indicates a "not equal" comparison result.

This completes the description of the translation scheme for the instructions appearing in our example. In total, to perform our experiments, translations were written for eighty nine PowerPC assembly instructions. Complete details of the PowerPC$^{\text{TM}}$ ISA are available [29] online.

4 Verification

Once the target assembly program has been decompiled to a C program *P*, the second stage of AIR involves the verification of *P* via SMC. For our experiments, we used the COPPER [13] SMC tool for verification. For the purpose of AIR we only required the ability of COPPER to check for trace containment between sequential C programs and finite state machines. In addition, though our familiarity with COPPER lead to its use in our experiments, any other C verification tool based on the SMC paradigm, such as SLAM [3], MAGIC [9], or BLAST [19], is suitable for use in the verification stage of AIR.

Indeed, the main challenge involved in the use of SMC for AIR verification is tool-independent, and arises from the need for precise handling of bit-level semantics during SMC[1]. In all cases, the handling of bit-level semantics is delegated to the theorem prover used during predicate abstraction. Most often, the

[1] We note that the C bounded model checker CBMC [10] does obey precise bit-level semantics but does not use SMC.

theorem prover (usually Simplify [26] or Vampyre [31]) treats the C bit-wise operators as uninterpreted functions. For source code verification, this is not a major roadblock since many properties verified on source code do not rely on the precise interpretation of bitwise operators. However, in the case of AIR, precise interpretation of bitwise operations is crucial for verifying the C programs generated via decompilation. In our initial experiments, not a single non-trivial property could be verified by leaving the bitwise operators uninterpreted. We attempted several solutions to this problem, as discussed next.

Solution 1: Adding axioms. First, we added extra axioms about C bitwise operators to assist Simplify, the default theorem prover used by COPPER. Unfortunately, this solution is ad hoc, since we had no way of knowing if we had added enough axioms. Also, the performance of Simplify, in terms of both time and memory consumption, degraded dramatically with increasing numbers of axioms. Ultimately, we concluded that this approach would not scale to realistic programs.

Solution 2: Syntactic analysis. Next, we augmented COPPER with a set of syntactic bit-level analyses. Specifically, before invoking Simplify, COPPER performs some simplifications on the formula whose validity has to be checked. The transformations are targeted at specific patterns that arise in formulas due to the structure of assembly programs. For example, a common query to Simplify is the validity of `((E | 0x4) >> 2) & (0x1)`, where `E` is some C expression. Our technique is able to convert such formulas to `0x1`, whose validity can then be easily decided by Simplify. We call this solution *uninterpreted* since all bitwise operators are left completely uninterpreted by the theorem prover.

Solution 3: Using a bit-vector decision procedure. We also compared the above approach to the idea of replacing Simplify with the bit-vector decision procedure CPROVER [22] (we also experimented with CVC Lite [15] but found CPROVER to be faster). We tried two variations of this idea. In the *interpreted* variation, all formulas containing bitwise operators are solved using CPROVER. In the *semi-interpreted* variation, formulas containing bitwise operators are first solved using Simplify. CPROVER is used only if Simplify is unable to decide validity conclusively.

In our example, AIR is able to successfully report the bug in MICRO-C. When the bug is fixed, in accordance with the suggestion in the comment, AIR successfully verifies the safety property. Also our experiments indicate that the *uninterpreted* approach yields the best performance over a set of realistic benchmarks. We now present full details of the empirical validation of our technique.

5 Experimental Validation

We experimented with a set of benchmarks derived from MICRO-C and Linux device drivers. All our experiments were performed on a single core 2.4 GHz Pentium computer running RedHat 9. We imposed a time limit of one hour, and memory limit of one GB. We derived a set of eleven benchmarks – one from MICRO-C, and the rest from Linux 2.6.11.10 kernel drivers – by compiling C source code with `gcc-3.2`. For each example, we checked that a certain “lock” was being acquired and released properly. The nature of the lock varied with the example. For MICRO-C, the lock was an invocation of `OS_ENTER_CRITICAL`, while for the Linux drivers it was a call to `spin_lock`, `spin_lock_irq` or `spin_lock_irqsave`. The “unlock” was derived accordingly.

We initially observed that COPPER is easily able to verify the safety property for all our benchmarks because the locks and unlocks are paired up syntactically. In other words, an analysis of the control flow graph suffices and no further predicate abstraction is necessary. To make our benchmarks more interesting, we added data dependencies between the locks and unlocks. Essentially we guarded the

Name		Interpreted			Semi-Interpreted			Uninterpreted			BLAST			
	KLOC	T	M	I	T	M	I	T	M	I	CE	T	M	I
Micro-C	10	*	164	-	*	164	-	**305**	**70**	31	-	*	67	-
aha152x	33	2319	121	16	*	107	-	**198**	**74**	16	-	*	141	-
DAC960	45	*	193	-	*	142	-	**520**	**107**	16	×	61	128	6278
devices	17	1018	41	17	3523	45	18	**350**	**40**	19	×	202	60	96701
ide	26	510	61	15	557	61	15	**25**	**34**	15	-	*	105	-
ipr	35	*	285	-	*	288	-	**310**	**79**	19	×	30	100	1230
message	21	194	28	26	145	28	25	**29**	**27**	26	×	16	63	831
mxser	22	699	53	19	547	52	19	**129**	**46**	19	×	6	51	1302
synclink	34	123	33	15	106	33	15	**15**	**30**	15	×	40	89	4292
tg3	61	988	77	18	907	77	18	**168**	**70**	21	×	84	152	11496
tlan	31	312	63	7	242	63	7	**73**	**49**	7	×	25	88	2062

Figure 3: Results for non-buggy benchmarks. KLOC = 1000 lines of assembly; T = time in seconds; M = memory in MB; I = # of iterations; * means that the resource was exhausted; - means that no measurement was available; × means that the counterexample returned by BLAST is spurious. Best figures are highlighted.

Name		Interpreted			Semi-Interpreted			Uninterpreted			BLAST			
	KLOC	T	M	I	T	M	I	T	M	I	CE	T	M	I
Micro-C	10	*	164	-	*	164	-	*	70	-	-	*	67	-
aha152x	33	357	53	10	321	53	10	**84**	**49**	18	×	40	98	8681
DAC960	45	1208	138	13	1017	138	13	**388**	**107**	13	×	70	130	6272
devices	17	1260	*	14	*	46	-	*	43	-	×	281	59	96697
ide	26	*	75	-	*	67	-	*	39	-	-	*	105	-
ipr	35	2009	280	6	1949	280	6	**205**	**76**	6	×	37	99	1230
message	21	62	26	11	76	26	11	**6**	**24**	11	×	16	63	831
mxser	22	115	50	6	108	50	6	**76**	**45**	6	×	8	50	1302
synclink	34	120	35	10	212	36	16	**27**	**32**	16	×	34	89	4292
tg3	61	2115	77	17	849	77	11	**219**	**68**	12	×	88	152	11496
tlan	31	362	63	7	354	63	7	**98**	**49**	7	×	34	89	2062

Figure 4: Results for buggy benchmarks. KLOC = KLOC of assembly; T = time in seconds; M = memory in MB; I = # of iterations; * means that the resource was exhausted; - means that no measurement was available; × means that the counterexample returned by BLAST is spurious. Best figures are highlighted.

locks and unlocks with a non-deterministic value. Since the same value guards both lock and unlock the examples are still correct.

We experimented using the *interpreted*, *semi-interpreted* and *uninterpreted* approaches presented in Section 4. In the first two cases, the syntactic simplifications were also applied. As a control, we also used BLAST version 1.0. The results of our experiments with these benchmarks are summarized in Fig. 3. Next, for each benchmark, we created a buggy version by artificially inserting errors and repeated our experiments. The results for our experiments with the the buggy examples are summarized in Fig. 4.

We observe that the *uninterpreted* approach exhibits the best overall performance. The *interpreted* approach beats the *semi-interpreted* approach by successfully proving more examples. This indicates that almost all the formulas involving bitwise operators could not be proved by Simplify and hence had to be further delegated to CPROVER. This is also consistent with our initial failure with only Simplify (without the syntactic simplifications). BLAST returns counterexamples for both the correct and buggy examples. Upon closer inspection, all counterexamples returned by BLAST are found to be spurious. We note that this is essentially due to BLAST's dependency on Simplify.

6 Discussion and Conclusion

We have presented AIR, a framework for verifying safety properties of assembly language programs via SMC. We have proposed a number of approaches for more precise handling of bit-level semantics during SMC and empirically validated their relative effectiveness. Overall, our experiments indicate that AIR is effective on real-life benchmarks derived from an embedded OS and Linux device drivers.

It is worthwhile to consider a few issues concerning the AIR approach. Decompiling an assembly program, though much less difficult than verifying the resulting C program, requires careful attention to detail. Whether the target platform is big or little endian and whether a 0 or a 1 is shifted in on >> operations on signed integers are two such intricacies. Correctly modeling elements of the program's environment such as the contents of the PowerPC$^{\text{TM}}$ machine state register are more complicated. Other decisions must be guided by the capabilities of the model checker; for example, choosing whether to denote a comparison that treats two operands as unsigned quantities by using type casting and a simple comparison or by using more predicates and checking different conditions depending on the signs of the operands. Moreover, the correct handling of pointer aliasing during verification is crucial for maintaining the overall soundness of AIR. Finally, though AIR is applicable to any assembly program, it is not necessarily a good choice in many cases. The broad applicability of AIR comes at a cost in usability. Encoding properties in terms of elements of the assembly program may be more difficult then encoding the same property against, for example, a C program. Similarly, interpreting a counterexample expressed in terms of an assembly program is likely to be more difficult. However, these concerns are much less important when it is sufficient to know whether a property holds (or not), or when source is unavailable.

In summary, we believe that the AIR approach has important ramifications for the development of effective low-level software verification techniques. Specifically, AIR is applicable to verifying other low-level languages such as Java bytecode and MSIL. Programs in these languages generally contain additional information (such as variable names) that should further increase AIR's effectiveness. AIR is also adoptable for the purpose of using certifying model checking [23] for proof carrying code (PCC) [25]. Certifying model checking in combination with abstraction has been used [24, 8] to construct invariants and ranking functions for the purpose of certifying source code. By generating source code from binaries, AIR enables us to leverage the above technology for the PCC-style certification of binaries. Finally, there is a growing trend of implementing hardware functionality using software, such as microcode, in the domain of hardware-software co-design. We believe that AIR would also be applicable for the verification of such low-level programs.

References

[1] T. Andrews, S. Qadeer, S. Rajamani, J. Rehof, and Y. Xie. Zing: A model checker for concurrent software. In *Proceedings of CAV*, pages 484–487, 2004.

[2] G. Balakrishnan and T. W. Reps. "Analyzing Stripped Device-Driver Executables". In *Proceedings of TACAS*, pages 124–140, 2008.

[3] T. Ball and S. K. Rajamani. Automatically Validating Temporal Safety Properties of Interfaces. In *Proceedings of SPIN*, pages 103–122, 2001.

[4] BEHAVE! website. `http://research.microsoft.com/behave`.

[5] R. S. Boyer and Y. Yu. Automated proofs of object code for a widely used microprocessor. *Journal of the ACM (JACM)*, (1):166–192, 1996.

[6] P. T. Breuer and J. P. Bowen. Generating Decompilers. RUCS Technical Report RUCS/1998/TR/010/A, Department of Computing, The University of Reading, 1998.

[7] CBMC website. `http://www.cprover.org/cbmc`.

[8] S. Chaki. SAT-Based Software Certification. In *Proceedings of TACAS*, pages 151–166, 2006.

[9] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular Verification of Software Components in C. In *Proceedings of ICSE*, pages 385–395, 2003.

[10] E. Clarke, D. Kroening, and F. Lerda. A Tool for Checking ANSI-C Programs. In *Proceedings of TACAS*, pages 168–176, 2004.

[11] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM (JACM)*, (5):752–794, 2003.

[12] D. L. Clutterbuck and B. A. Carre. The verification of low-level code. *Software Engineering Journal (SEJ)*, (3):97–111, 1988.

[13] Copper website. `http://www.sei.cmu.edu/pacc/copper.html`.

[14] P. Curzon. *A Structured Approach to the Verification of Low Level Microcode*. PhD thesis, University of Cambridge, Computer Laboratory, 1991. Tech report no. 215.

[15] CVC Lite website. `http://verify.stanford.edu/CVCL`.

[16] M. B. Dwyer, J. Hatcliff, M. Hoosier, and Robby. Building Your Own Software Model Checker Using The Bogor Extensible Model Checking Framework. In *Proceedings of CAV*, pages 148–152, 2005.

[17] M. B. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C. S. Păsăreanu, H. Zheng, and W. Visser. Tool-supported program abstraction for finite-state verification. In *Proceedings of ICSE*, pages 177–187, 2001.

[18] S. Graf and H. Saïdi. Construction of Abstract State Graphs with PVS. In *Proceedings of CAV*, pages 72–83, 1997.

[19] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. In *Proceedings of POPL*, pages 58–70, 2002.

[20] G. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. 2003.

[21] F. Ivancic, Z. Yang, M. K. Ganai, A. Gupta, I. Shlyakhter, and P. Ashar. F-Soft: Software Verification Platform. In *Proceedings of CAV*, pages 301–306, 2005.

[22] D. Kroening. Application Specific Higher Order Logic Theorem Proving. In *Proceedings of the Verification Workshop (VERIFY'02)*, pages 5–15, 2002.

[23] K. S. Namjoshi. Certifying Model Checkers. In *Proceedings of CAV*, pages 2–13, 2001.

[24] K. S. Namjoshi. Lifting Temporal Proofs through Abstractions. In *Proceedings of VMCAI*, pages 174–188, 2003.

[25] G. C. Necula. Proof-Carrying Code. In *Proceedings of POPL*, pages 106–119, 1997.

[26] G. Nelson. *Techniques for Program Verification*. PhD thesis, Stanford University, 1980.

[27] I. O'Neill, D. Clutterbuck, P. Farrow, P. Summers, and W. Dolman. The formal verification of safety-critical assembly code. In *Proceedings of the International Federation of Automatic Control Safety of Computer Control Systems Conference (SAFECOMP '88)*, pages 115–120, 1988.

[28] A. Pnueli, M. Siegel, and E. Singerman. Translation Validation. In *Proceedings of TACAS*, pages 151–166, 1998.

[29] PowerPC$^{\text{TM}}$ ISA. `http://www.nersc.gov/vendor_docs/ibm/asm/mastertoc.htm`.

[30] T. W. Reps, G. Balakrishnan, J. Lim, and T. Teitelbaum. A Next-Generation Platform for Analyzing Executables. In *Proceedings of the third Asian Symposium on Programming Languages and Systems (APLAS '05)*, pages 212–229, 2005.

[31] Vampyre website. `http://www-cad.eecs.berkeley.edu/~rupak/Vampyre`.

[32] W. Visser, K. Havelund, G. P. Brat, and S. Park. Model Checking Programs. In *Proceedings of ASE*, pages 3–12, 2000.

[33] D. Yu. *Safety Verification of Low-Level Code*. PhD thesis, Graduate School. of. Yale University, 2004.

[34] D. Yu and Z. Shao. Verification of Safety Properties for Concurrent Assembly Code. In *Proceedings of the 2004 International Conference on Functional Programming (ICFP '04)*, pages 175–188, 2004.

Generalized Abstract Symbolic Summaries*

Suzette Person
University of Nebraska-Lincoln
Lincoln, NE
sperson@cse.unl.edu

Matthew B. Dwyer
University of Nebraska-Lincoln
Lincoln, NE
dwyer@cse.unl.edu

Abstract

Current techniques for validating and verifying program changes often consider the entire program, even for small changes, leading to enormous V&V costs over a program's lifetime. This is due, in large part, to the use of syntactic program differencing techniques which are necessarily imprecise. Building on recent advances in symbolic execution of heap manipulating programs, in this paper, we develop techniques for performing abstract semantic differencing of program behaviors that offer the potential for improved precision.

1 Introduction

Symbolic analysis techniques have been studied extensively since they allow a potentially infinite equivalence class of system behaviors to be reasoned about as a single unit. For example, symbolic simulation has been applied to reason about the equivalence between hardware designs. In this setting, researchers have identified the utility of *uninterpreted functions* as a means of coarsely abstracting common portions of hardware circuits in such a way that equivalence checking is preserved [2]. Uninterpreted functions achieve this by sacrificing the precise encoding of a function's meaning while preserving functional congruence, i.e., $x = y \implies f(x) = f(y)$.

Symbolic analysis for software has a long and rich history. Symbolic execution [9], which was developed in the early 1970s, records the program state as symbolic expressions over free variables that represent program input values, accumulates constraints on those input values that define the conditions under which a particular program path is executed, and produces expressions that characterize output values produced along the path. Symbolic execution has seen a recent resurgence of interest for test input generation [4, 7, 15]. While much of the existing work has focused on symbolic execution of scalar data, researchers have generalized symbolic execution to treat heap manipulating programs. In our work, we build on a technique called *lazy initialization* that materializes just the portion of heap that is referenced along the symbolically executed path [8].

Recent work has investigated the application of uninterpreted functions as an abstraction mechanism in the context of symbolic execution. Currie et al. [3] propose the use of uninterpreted functions to abstract common sequences of assembly instructions to make equivalence checking tractable. Siegel et al. [13, 14] manually map portions of sequential and parallel implementations onto uninterpreted functions to permit equivalence checking as a means of avoiding subtle errors in parallelization. In our recent work [11], we define a framework called *differential symbolic execution* (DSE) that compares the results of symbolic executions of two program versions to perform equivalence checking and *precise program differencing*. DSE uses uninterpreted functions to calculate *abstract summaries* of Java program blocks that have not changed between program versions. While some program changes are intended to retain program functionality, for example, performance optimizations, in many cases, program changes are meant to change the program's meaning, for example, when a bug is fixed. In such cases, one would like to be able to identify the equivalence class of program behaviors for which the programs differ. It is only these behaviors that must be subjected to rigorous testing, or in the case of critical systems, to

E. Denney, D. Giannakopoulou, C.S. Păsăreanu (eds.); The First NASA Formal Methods Symposium, pp. 46-55

*This material is based in part upon work supported by the National Aeronautics and Space Administration under grant number NNX08AV20A and by the National Science Foundation under awards CCF-0541263 and CNS-0720654. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NASA or NSF.

verification or manual inspection. Our goal in this paper is to extend the approach in [11] so that it is applicable to heap manipulating programs.

Generalizing abstract summaries to support dynamically allocated data used in modern programming languages, such as Java, presents numerous challenges. For hardware circuits [2] and assembly language programs [3] defining uninterpreted functions for portions of a hardware design or program is straightforward–the input lines or registers that are read define the function's domain and the output lines or registers written define the function's codomain. In more expressive programming languages, the *types* of values that are read or written and the *location* at which those values are referenced may be much more difficult to determine. While manual inspection and reasoning about the program can make such determinations [14], methods that are both broadly applicable and automated are needed.

Our work builds on generalized symbolic execution [8] which extends symbolic execution to heap manipulating programs. The key insight of their approach is that *on a given program path only the heap cells that are accessed can influence the computational effects*. This permits the analysis to create and initialize only the portion of the heap that is referenced along a path–this is called *lazy initialization*. When abstracting common program blocks, however, DSE intentionally *omits* precise tracking of the heap cells accessed within the block. In this paper, we integrate lazy initialization and abstract summaries to safely approximate the computational effects. The specific contributions of this paper involve (1) safely abstracting the computational-effects of common blocks using an adaptation of side-effects analysis on heap data; (2) instantiation of abstract summaries based on the state of the symbolic execution upon entry to each common block; (3) forcing the lazy *re-initialization* of reference fields that may have been written by the abstracted code block; and (4) adapting lazy initialization of heap fields to account for the situation where a common block may assign heap locations that were not present in the symbolic state on entry to the block.

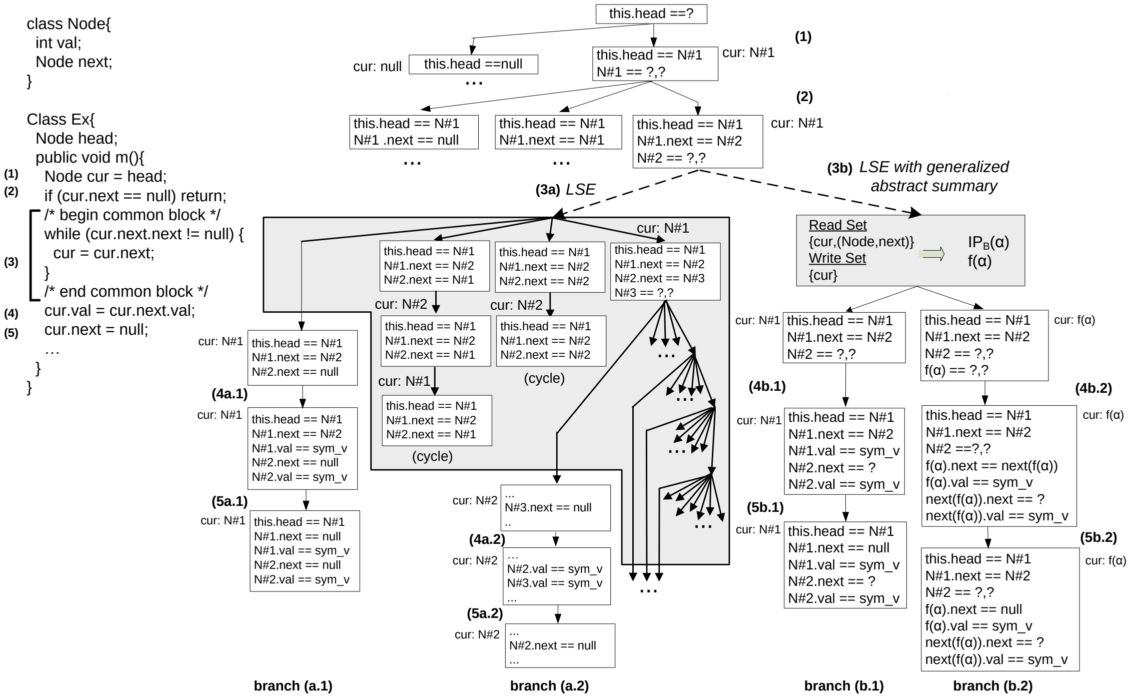
In the next section, we illustrate our approach to computing an abstract summary for a heap-manipulating program by way of example. Section 3 defines the symbolic execution and summarization concepts we build on. Section 4 explains how our technique differs from previous work and details the components of the approach that address the challenges presented by heap-manipulating programs. We conclude with a discussion of the implementation of these techniques in JavaPathfinder [12], our plans for evaluating the cost-effectiveness of the technique, and future work in Section 5.

2 Overview

Imagine having just completed V&V of a large complex system when a fault is detected or a performance optimization is identified. Unless great care is taken to isolate system components from one another, calculating the potential impact of a change (e.g., using data and control dependence analyses) is likely to identify that the majority of system components may behave differently as a result of the change. This, in turn, could lead to significant re-V&V activities even if the changes are actually behavior preserving or only affect a very narrow range of behaviors.

Differential symbolic execution addresses this problem by performing a detailed semantics-based analysis of the behaviors of two program versions. The results of the analysis are rendered as sets of logical formula each of which describes a distinct portion of the program's input space and the corresponding effects of program computation on all inputs in that space. DSE then systematically compares the logical formulae from one version with the formulae from the other. Equivalent pairs of formulae are eliminated since they indicate common behavior across program versions. The remaining formulae are then used to precisely characterize "when", i.e., for what portion of the program input space, and ”how", i.e., the computational effects, the program versions differ.

Our technique is an adaptation of lazy symbolic execution (LSE) [8], where LSE is performed on the



```
    class Node{
      int val;
      Node next;
    }

    Class Ex{
      Node head;
      public void m(){
(1)     Node cur = head;
(2)     if (cur.next == null) return;
        /* begin common block */
(3)     while (cur.next.next != null) {
          cur = cur.next;
        }
        /* end common block */
(4)     cur.val = cur.next.val;
(5)     cur.next = null;
        ...
      }
    }
```

Figure 1: Example with LSE and Abstract Summary-based Symbolic Execution (excerpts)

changed sections of code, and *abstract summaries* are used in place of performing detailed LSE on the common code blocks. Rather than describe the entire DSE method in detail, here we illustrate, by way of a small example, how abstract summaries in DSE can be used to exploit common code blocks, and compare the results of DSE with lazy symbolic execution of the entire method (on both the changed and common code blocks).

2.1 An Example

The left side of Figure 1 shows the source code for classes *Node* and *Ex*. We consider the method $Ex.m()$ in detail and assume that the `while` loop is shared with another version of the method; our technique considers this a *common block*.

The right side of Figure 1 depicts portions of a symbolic execution tree where a node encodes a symbolic representation of the program state and edges correspond to execution of a statement in $Ex.m()$. Figure 1 depicts two trees that share a common top portion. The lower-left side of the tree, beginning with branch (3a), illustrates the results of continuing LSE through the common code block. The lower-right side of the tree, beginning with branch (3b), illustrates how our approach abstracts the common code block and how the abstract summary is used during symbolic execution of the subsequent block of changed code. The shaded areas of the tree highlight the different treatment of the while loop under LSE and using our technique.

Lazy Symbolic Execution operates by maintaining an explicit heap that stores just the objects that have been referenced along a program execution path. Local variables and fields of scalar types are symbolically executed using traditional methods, but the treatment of heap data is closer to the concrete program semantics. LSE accounts for different program behaviors by considering multiple possible values when initializing reference fields at the point when they are first accessed; the term *lazy initialization* refers to the process of waiting to initialize a field until it is first referenced. LSE generates successor states in which the field has a `null` value, points to each type-compatible object that is currently in the heap, and points to a newly allocated object; we refer to this multiplicity of successor states as *branching* in the symbolic execution. LSE systematically explores a program component's behavior in this way until the method returns or some prescribed resource bound is reached.

In Figure 1, LSE of $Ex.m()$ begins with statements (1) and (2). At (1), the uninitialized field $this.head$ is read and assigned to the local variable *cur*. The field reference $this.head$ requires lazy initializion of the field $head$. In this case, because there are no allocated objects of type $Node$, LSE explores just two possibilities: either the field is `null`, or it is set to a new symbolic object of type $Node$. At statement (2), the *next* field of the object referenced by *cur* is read for the first time, so LSE creates a new $Node$ object on the symbolic heap (we detail only this path; other paths include assigning a value of `null`, or the previously created $Node$ object). On this path the new object is not equal to `null`, so symbolic execution continues into the abstracted code block at (3).

At the beginning of the common code block under LSE (3a), a four-way branch is created when *cur.next*$.next$ is read since the fields of *cur.next* have not yet been initialized and there are two existing objects of type $Node$. Execution of statements in the common block are depicted in the shaded area where the value of *cur.next*$.next$ is compared with `null` and *cur* is updated to *cur.next*. As shown in Figure 1, as LSE initializes new $Node$ objects in the $while$ loop, the tree will grow in depth (with new nodes for each loop iteration) and breadth (since iterations may initialize new objects on which LSE will branch). In practice, this expansion will be bounded by fixing the number of loop iterations considered or the maximum depth of the the tree in which case LSE under-approximates the method's behavior. On symbolic execution paths where the loop condition evaluates to false, LSE will continue by updating *cur.val* and *cur.next* at statements (4) and (5).

Abstract Summaries are incorporated into LSE when a common code block is reached as shown in branch (3b). At this point, LSE is suspended and a pre-computed summary of the effects of the block is used to update the symbolic state. Summaries are computed by approximating the set of memory locations that may be read and written by the code block. The shaded area on branch (3b) shows that *cur* may be written and read, and the field *next* of objects of type $Node$ may be read during execution of the block. The read and write sets and the current symbolic state are used to generate and instantiate two uninterpreted functions. $IP_B(\alpha)$ encodes the path condition through the block. $f(\alpha)$ represents the unknown value calculated in the block that is assigned to *cur*, where α parameterizes f with the values of the read set locations on entry to the block.

The symbolic state is updated by applying the functions to appropriate locations. Since *cur* may be written by the block, two successor states are created: one where *cur* is *not* updated (the leftmost branch), and one where *cur* is updated to its new, but unknown value, $f(\alpha)$ (the rightmost branch). Note that since $f(\alpha)$ is a reference value, the symbolic state is extended to hold a new object of type $Node$ whose fields have unknown values.

Considering just the rightmost branch, LSE resumes at (4b.2). Our approach treats object references computed within the block differently; for example, $f(\alpha)$ encodes an address and we use function composition to encode chains of dereferences i.e., $next(f(\alpha))$. At statement (4) *cur.next*$.val$ is read. Since *cur* is bound to a reference value from the abstracted block, whose address is $f(\alpha)$, the reference to its *next* field causes lazy initialization branching. The branch we illustrate creates a new $Node$ object, whose address is $next(f(\alpha))$, and sets the *cur* object's *next* field to that address (4b.2). Dereference of the non-reference field *val* of $next(f(\alpha))$ produces a fresh symbolic value, sym_v, which is then assigned to *cur*'s *val* field, i.e., $f(\alpha).val$. Finally, in statement (5) the field *cur.next* is accessed again, this time to set its value to `null` (5b.2).

2.2 Assessment

Using overapproximating *abstract symbolic summaries*, to represent common code blocks has advantages and disadvantages. There are program structures, such as loops and recursion, that are problematic for symbolic execution. LSE cannot process the common block in our example because there is no information about the length of the *Node* chain. Using abstract summaries avoids this problem, since the block is never executed in detail. Abstract summaries are a safe approximation because all of the behaviors present under LSE are also captured in the summary; branch (b.1) of our approach over-approximates branch (a.1) under LSE, and branch (b.2) of our approach accounts for the all of the remaining branches explored under LSE. Our approach may defer initialization of heap objects as shown on branches (b.1) and (b.2), however, it may also introduce infeasible paths into the analysis. While this is problematic for some applications of symbolic execution, such as test generation, for precise differencing our focus is on identifying only the behaviors on which two versions differ. For most program blocks, we can be assured of deterministic behavior, i.e., the block will perform the same computation when executed from the same state, thus even if abstract summaries introduce infeasible behaviors, many of those behaviors will be equivalent in both versions of the program.

3 Background on Symbolic Execution

Symbolic execution involves the non-standard interpretation of a program using *symbolic values* to represent input data values, and expressions over symbolic values to represent the values of program variables. The state of a symbolic execution is defined as a triple, (pp, pc, v). pp, the current *program point*, records the next statement to be executed. pc, the *path condition*, is the conjunction of branch conditions encoded as constraints along the current execution path. v, the *symbolic value store*, is a map from memory locations to symbolic expressions representing their current values; $v[j]$ is the symbolic value at location j. A store update $v\langle l, s\rangle$ defines a new store v' with the value s at location l.

Computation statements, $x = y$ op z, when executed symbolically in state (pp, pc, v) produce a new state $(pp+1, pc, v\langle x, op(v[y], v[z])\rangle)$ that reflects the update of the value store with a new symbolic value for x that applies op to its operand values. Branching statements, `if` x op y `goto` l, when executed symbolically in state (pp, pc, v) branch the symbolic execution to *two* new states $(l, pc \wedge v[x]$ op $v[y], v)$ and $(pp+1, pc \wedge \neg(v[x]\, op\, v[y]), v)$ corresponding to the “true” and “false” evaluation of the branch condition, respectively.

3.1 Generalized Symbolic Execution

In addition to the symbolic state stored for program variables, e.g., method locals, with generalized symbolic execution a portion of the symbolic state encodes a set of partially-defined heap objects. Lazy initialization is a technique for defining the values of only those object fields that have been referenced during symbolic execution along a path. It operates on statements with dereference expressions, e.g., $x = r.f$, and can update the state in five different ways as shown in Figure 2; where type() denotes the type of the location and Ref the set of reference types.

If the referenced field’s location is in the symbolic store, i.e., $r.f \in \mathrm{dom}(v)$, the successor state is given by (1), where the symbolic value store for x is updated with the value at location $r.f$. If the field’s location is not in the symbolic store and the field is a non-reference type i.e., primitive or scalar type, the successor state is given by (2), where *new* generates a new symbolic variable which is used to update $r.f$ in the symbolic value store ($v\langle r.f, \mathrm{new}\rangle$), and the value at $r.f$ is subsequently used to update the symbolic

$$
\begin{aligned}
&(pp+1, pc, v\langle x, v[r.f]\rangle) && \text{if } r.f \in \text{dom}(v) && (1)\\
&(pp+1, pc, v\langle r.f, \text{new}\rangle\langle x, v[r.f]\rangle) && \text{if } r.f \notin \text{dom}(v) \wedge \text{type}(f) \notin \text{Ref} && (2)\\
&(pp+1, pc, v\langle r.f, null\rangle\langle x, v[r.f]\rangle) && \text{if } r.f \notin \text{dom}(v) \wedge \text{type}(f) \in \text{Ref} && (3)\\
&(pp+1, pc, v\langle \text{new}, \bot\rangle\langle r.f, \text{new}\rangle\langle x, v[r.f]\rangle) && \text{if } r.f \notin \text{dom}(v) \wedge \text{type}(f) \in \text{Ref} && (4)\\
&(pp+1, pc, v\langle r.f, o\rangle\langle x, v[r.f]\rangle) && \text{if } r.f \notin \text{dom}(v) \wedge \text{type}(f) \in \text{Ref} \wedge && \\
& && o \in \text{dom}(v) \wedge \text{type}(v[o]) = \text{type}(f) && (5)
\end{aligned}
$$

Figure 2: Lazy initialization symbolic state updates for $x = r.f$ from (pp, pc, v)

value store for x ($\langle x, v[r.f]\rangle$). If the field is of reference type and it is not in the symbolic store, there are multiple possible successor states and the symbolic execution should explore each state. The field may have a `null` value yielding the successor state given by (3), or the field may store a reference to a new, uninitialized ($\bot$) object yielding the successor state given by (4), where the new object is added to the symbolic store, $r.f$ is updated to reference the new object, and x is updated with the value at $r.f$, and finally, the field may store a reference to a type-compatible object in the current state yielding a successor state given by (5) for each such object.

3.2 Symbolic Summaries

Recent work has proposed compositional symbolic execution to improve the scalability of inter-procedural symbolic execution [7, 1]. This approach works by symbolically executing a method and accumulating information for each path into a summary.

Definition 3.1 (Symbolic Summary [11]). *A symbolic summary, for a method m, is a set of pairs* $m_{sum} = \{(i_1, e_1), (i_2, e_2), \ldots, (i_k, e_k)\}$ *where* $\forall 1 \leq j \leq k \forall 1 \leq j' \leq k \wedge j \neq j' : i_j \wedge i_{j'}$ *is unsatisfiable.*

Each constraint i_j, called an input constraint, encodes a path condition. An effect constraint, $e_j = \bigwedge l \in Write(m, i_j) : l = v[l]$, encodes the symbolic state on exit from the method where the effect constraint is a conjunction of equality constraints for locations that are written by the method. $Write(m, \phi)$ denotes the locations written to by a method, m, when called on arguments that satisfy ϕ, and $Write(m) = Write(m, true)$.

Method call, $m(v)$, is executed symbolically in state (pp, pc, v) with summary m_{sum} as follows. For each pair $(i, e) \in m_{sum}$, let $pc' = pc \wedge i \wedge (\bigwedge j \in |v| : a[j] == f[j])$ which conjoins the current path condition with the input partition and with a set of equality constraints that equate symbolic variables for the formal parameters with the symbolic expressions for the actuals. If pc' is satisfiable, the execution continues with on a branch with successor state $(pp+1, pc', v\langle e\rangle)$.

4 Generalized Abstract Symbolic Summaries

In [11] we introduced the concept of an *abstract summary*, m_{abs}, to safely approximate common program behaviors without performing a precise symbolic execution. Our previous work supports symbolic execution of non-reference types and summarization of program methods. In this section, we detail abstract summaries for generalized symbolic execution of heap-manipulating programs and to summarize arbitrary program blocks. We begin by recalling the definition of abstract summary:

Definition 4.1 (Abstract Summary [11]). *An abstract summary for a method* $m(\vec{f})$ *with formal parameters* $\vec{f}$ *is a pair,* $m_{abs} = (i(\vec{f}), \bigwedge l \in Write(m) : l == e_l(\vec{f}))$ *where* $i : \vec{f} \to \{true, false\}$*, and* $e_l : \vec{f} \to$ type(l) *are uninterpreted functions defined over vectors of formal parameter values.*

An abstract summary is built up from a set of uninterpreted functions: i abstracts the path condition within the method and, for each written location, a function e_l which abstracts the value that the method may write to location l. Symbolically executing a call to method $m(\vec{a})$ in state (pp, pc, v) with m_{abs} involves instantiating the uninterpreted functions with the current symbolic state and effecting the updates to potentially written locations. The resulting state is $(pp+1, pc \wedge i(\vec{a}), v\langle \bigwedge l \in Write(m) : l = e_l(\vec{a})\rangle)$ where binding of formals to actuals is achieved by parameterizing the uninterpreted functions, and the state is updated to be consistent with the summary's effects.

We note that using a *may* analysis to calculate $Write(m)$ leaves open the possibility that an $l \in Write(m)$ may not actually be written by a call to m. Modeling all combinations of possible writes would result in $2^{|Write(m)|}$ possible successor states after instantiation of m_{abs}. Our approach mitigates this for non-reference types by using e_l which represents any possible value of the type of l.

There are three basic elements of this approach: (1) calculating the written (read) locations; (2) defining the set of uninterpreted functions; and (3) instantiating constraints involving those functions at the point of a method call during symbolic execution. Definition 4.1 captures how (1) and (2) are performed when summarizing methods that read and write non-reference types. As we explain in the remainder of this section, for heap manipulating programs element (1) must be generalized significantly, elements (2) and (3) must be performed together since the definition of uninterpreted functions is dependent on the symbolic state at the beginning of the summarized code block, and element (3) must be carried forward during symbolic execution to safely approximate lazy initialization of fields that may be written by an abstracted block.

4.1 Equality Preserving Heap Abstraction

Current approaches to symbolic execution of heap manipulating programs [4, 8, 12], use an explicit heap store where object references maintain their usual semantics. Our approach extends the explicit store to allow object references to be encoded as functions that encode dereference chains or access-paths through the heap; this is refered to as a storeless model of the heap [5].

When overapproximating abstract summaries are used, a symbolic execution may traverse a prefix of an access path, α, then enter an abstracted block where that access path is extended, with β, and then after exiting the block, further dereferencing along the access path may be performed, resulting in an overall access path of $\alpha\beta\gamma$. The explicit store used by LSE cannot faithfully represent an access path where the details of β are not known; it is insufficient to consider only the objects referenced along α since different objects may be referenced within the block.

We make the following observations: (a) in a given program state, an access path, e.g., α, evaluates to a single object, e.g., following α leads to o; and (b) an access path can be represented as the composition of field specific dereference functions applied to an initial object reference. For example, if $\beta = next.next$ where *next* is a field of class $Node$, then the function $Node_{next}(Node_{next}(o))$ would encode the object at the end of chain $\alpha\beta$.

When an abstracted block, b, executes in a given symbolic state, v, and traverses an access path to write to a field, f, of some object type, t, that object field can be uniquely defined by a function $t_f(o)$ where $o \in v$ is the root of the access path. This function stands for an unknown access path within the block. It is sufficient to have t_f be uninterpreted to judge equivalence between object references that are an extension of its access path, e.g., $t_f(o)\gamma$.

4.2 Computing Write (Read) Information

To produce a safely approximating abstract summary of a code block we over-estimate the set of written and read locations in the block. We use a combination of flow-insensitive field-sensitive type-based alias analyses [6] and side-effects analyses [10]. Every side-effects analysis defines a *naming scheme* for heap locations. Ours calculates a set of pairs, $(type, field)$, where a pair indicates a write (read) of $field$ of an object of $type$. Our analysis is initiated, on demand, to target just the common blocks that are encountered during symbolic execution.

The code block, b, is scanned to detect *local* reads and writes (e.g., in JVM bytecodes these include instructions such as `iload` and `istore`) and *heap* reads and writes (e.g., `getfield` and `putfield` bytecodes). The set of locations (e.g., local offsets) subject to local writes (reads) are recorded as $Write_l(b)$ ($Read_l(b)$). The set of $(type, field)$ pairs naming the heap locations that are written (read) in b are recorded as $Write_h(b)$ ($Read_h(b)$).

When a method invocation for $m()$ is encountered during the scan of b, the pre-computed $Write_h(m)$ and $Read_h(m)$ are unioned with the corresponding sets for the caller. If the pre-computed sets for $m()$ are not available, we scan $m()$'s body (and recursively any method it calls) to compute $Write_h(m)$ and $Read_h(m)$, which are cached, and unioned with the caller's sets.

4.3 Realizing Generalized Abstract Summaries

Realizing an abstract summary involves: (1) defining and instantiating the uninterpreted functions, (2) updating the current path condition and symbolic state, and (3) setting the program counter to point to the first instruction following the abstracted code block. When the symbolic program state on entry to a block includes heap locations, Definition 4.1 is inadequate because the abstracted block may read heap locations by dereferencing through parameters. All read locations must be included in the signature of the uninterpreted functions used in the abstract summary–leaving a location out risks judging two instances of a block as having equivalent effects when their behavior may be distinguished by the value at the missing location. This observation forces us to delay the generation of uninterpreted functions until the symbolic state on entry to the block has been computed.

We define sets of written and read locations within a symbolic state as follows:

Definition 4.2 (State-specific Locations)**.** *Given a symbolic state, v, and a code block, b, let*

$$\begin{aligned}
ReadIn(b,v) &= Read_l(b) \cup \{r.f \mid r.f \in \mathrm{dom}(v) \wedge (\mathrm{type}(r), f) \in Read_h(b)\}\\
WriteIn(b,v) &= \{l \mid l \in Write_l(b) \wedge \mathrm{type}(l) \notin \mathrm{Ref}\} \cup\\
&\quad\; \{r.f \mid r.f \in \mathrm{dom}(v) \wedge (\mathrm{type}(r), f) \in Write_h(b) \wedge \mathrm{type}(f) \notin \mathrm{Ref}\}\\
WriteIn_h(b,v) &= \{r.f \mid r.f \in \mathrm{dom}(v) \wedge (\mathrm{type}(r), f) \in Write_h(b) \wedge \mathrm{type}(f) \in \mathrm{Ref}\}\\
WriteIn_l(b,v) &= \{l \mid l \in Write_l(b) \wedge \mathrm{type}(l) \in \mathrm{Ref}\}
\end{aligned}$$

be the set of locations read in b, locations written in b of non-reference type, heap locations and locals written in b of reference type, respectively.

The need to make summaries state-specific forces us to extend the notion of an abstract summary from Definition 4.1. In particular, all definitions of uninterpreted functions and constraints on updated locations are deferred to the point where the abstract block is entered, and to subsequent uninitialized field references. Consequently, we modify the definition to simply capture the state-specific location sets that permit that deferred processing.

Definition 4.3 (Generalized Abstract Summary)**.** *An abstract summary for the execution of code block, b, in symbolic state v is the quadruple* $b_{gabs} = (ReadIn(b,v), WriteIn(b,v), WriteIn_h(b,v), WriteIn_l(b,v))$.

Let b be a code block with a single-entry, pp_{entry}, and single-exit, pp_{exit}, point. Instantiation of the generalized abstract summary, b_{gabs}, in state (pp_{entry}, pc, v) results in successor states of the form:

$$
\begin{array}{rl}
(pp+1, pc, v\langle r.f, e_{r.f}(top(w).2)\rangle\langle x, v[r.f]\rangle, w) & \text{if } r.f \notin \mathrm{dom}(v) \wedge \mathrm{type}(f) \notin \mathrm{Ref} \\
 & \wedge(\mathrm{type}(r), f) \in \bigcup w \qquad (6) \\
(pp+1, pc, v\langle e_{r.f}(top(w).2), \bot\rangle\langle r.f, e_{r.f}(top(w).2)\rangle\langle x, v[r.f]\rangle, w) & \text{if } r.f \notin \mathrm{dom}(v) \wedge \mathrm{type}(f) \in \mathrm{Ref} \\
 & \wedge(\mathrm{type}(r), f) \in \bigcup p \in w : p.1 (7)
\end{array}
$$

Figure 3: Lazy initialization updates for $x = r.f$ from (pp, pc, v, w)

$$
\begin{array}{l}
(pp_{exit}, pc \wedge i(ReadIn(b,v)), v\ \langle \bigwedge l \in WriteIn(b,v) : l = e_l(ReadIn(b,v))\rangle \\
\qquad\qquad\qquad\qquad \langle \bigwedge l \in L_{\mathrm{Ref}} : l = e_l(ReadIn(b,v))\rangle - WriteIn_h(b,v))
\end{array}
$$

where $L_{\mathrm{Ref}} \subseteq WriteIn_l(b,v)$, and i and e_l are uninterpreted functions with domains corresponding to the types of locations in $ReadIn(b,v)$ and with boolean and type(l) codomains, respectively. Writes within the block are treated in three different ways: (1) non-reference locations, $WriteIn(b,v)$, are bound to an uninterpreted function that represents the unknown value that may have been written to it by the block, (2) each successor state selects a subset of the locals of reference type and updates them with an uninterpreted function defining the reference, and (3) heap reference locations, $WriteIn_h(b,v)$, are eliminated from the symbolic state which makes them uninitialized so that LSE will re-initialize them as explained below. The second case is illustrated by the creation of branches (b.1) and (b.2) in Figure 1.

Initializing Abstracted Objects As LSE proceeds after an abstracted block it may create new locations. If such a location was written in b the summary instantiation shown above will not reflect that update, since that location did not exist in the state on entry to the block. Our solution is to carry $Write_h(b)$ and $ReadIn(b,v)$ forward during symbolic execution and use them to add two additional cases for lazy initialization.

Specifically, the symbolic state is extended to be a quadruple: (pp, pc, v, w). w is a stack of pairs $(Write_h(b), ReadIn(b,v))$ where b is a block in the current execution path and pairs are ordered from most to least recently executed on the path. When b_{gabs} is instantiated in state (pp_{entry}, pc, v, w) the resulting state is

$$
\begin{array}{l}
(pp_{exit}, pc \wedge i(ReadIn(b,v)),\ v\langle \bigwedge l \in WriteIn(b,v) : l = e_l(ReadIn(b,v))\rangle - WriteIn_h(b,v)), \\
\qquad\qquad\qquad\qquad (Write_h(b), ReadIn(b,v)) : w)
\end{array}
$$

where $x : y$ denotes that x is pushed on to stack y. Correspondingly, when symbolic execution backtracks across an abstracted block the fourth state component is popped.

We extend lazy initialization to account for uninitialized reference fields that may have been written by the abstracted code block with the adapted update rules in Figure 3: (6) is an adaptation of rule (2) from Figure 2 and (7) is an adaptation of rule (4). We denote the first and second components of pairs with .1 and .2, respectively. These add additional branching to the symbolic execution to set uninitialized reference fields that could have been written in an abstracted block to uninterpreted functions that encode the unknown object reference; rule (7) also creates new objects with an access-path address. Step (4b.2) in Figure 1 illustrates the application of rule (7) to create a new node with an access-path name whose fields are subsequently written at (4b.2) and (5b.2).

5 Conclusions and Future Work

It has long been understood that the cost of re-validating a system after a change is disproportionate to the size of the change. Lightweight techniques, such as *diff*, provide only the most superficial syntactic characterization of program differences. Semantics-based verification and certification techniques require much more precise methods that can accurately describe the portion of a system's input space for which it is known to retain its previous functionality. A logical characterization of that input sub-space can then be leveraged to prune the reasoning required by formal methods tools used to re-verify the changed system.

Symbolic execution coupled with abstract summaries on common program blocks are a powerful foundation for building these kind of differencing analyses. In this paper we have extended the summaries used by our DSE approach to accommodate more general definitions of common program blocks and to safely treat heap-manipulating blocks. We have implemented a prototype version of our technique for computing and realizing generalized abstract symbolic summaries in the Java PathFinder (JPF) symbolic execution framework [12]. At present, our prototype has been applied to small examples like the one in Figure 1. We are planning to conduct a more comprehensive evaluation of our approach and plan to present those results at the workshop. In the longer term, a focus of our future work is to couple DSE with testing, model checking, and deductive methods to realize *incremental* V&V techniques that can be applied cost-effectively across a system's lifetime.

References

[1] S. Anand, P. Godefroid, and N. Tillmann. Demand-driven compositional symbolic execution. In *Proceedings of TACAS*, 2008.

[2] R. E. Bryant, S. German, and M. N. Velev. Processor verification using efficient reductions of the logic of uninterpreted functions to propositional logic. *ACM Transactions Computational Logic*, 2(1):93–134, 2001.

[3] D. Currie, X. Feng, M. Fujita, A. J. Hu, M. Kwan, and S. Rajan. Embedded software verification using symbolic execution and uninterpreted functions. *Intl. Journal of Par. Progr.*, 34(1):61–91, 2006.

[4] X. Deng, J. Lee, and Robby. Bogor/kiasan: A k-bounded symbolic execution for checking strong heap properties of open systems. In *Proceedings of ASE*, pages 157–166, 2006.

[5] A. Deutsch. A storeless model of aliasing and its abstractions using finite representations using finite representations of right-regular equivalence relations. In *Proceedings of the ICCL*, pages 2–13, 1992.

[6] A. Diwan, K. S. McKinley, and J. E. B. Moss. Type-based alias analysis. In *Proceedings of PLDI*, pages 106–117, 1998.

[7] P. Godefroid. Compositional dynamic test generation. In *Proceedings of POPL*, pages 47–54, 2007.

[8] S. Khurshid, C. S. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proceedings of TACAS*, 2003.

[9] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[10] W. Landi, B. G. Ryder, and S. Zhang. Interprocedural modification side effect analysis with pointer aliasing. In *Proceedings of PLDI*, pages 56–67, 1993.

[11] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu. Differential symbolic execution. In *Proceedings of FSE*, pages 226–237, New York, NY, USA, 2008. ACM.

[12] C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape. Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In *Proceedings of the ISSTA*, pages 15–25, 2008.

[13] S. F. Siegel, A. Mironova, G. S. Avrunin, and L. A. Clarke. Using model checking with symbolic execution to verify parallel numerical programs. In *Proceedings of the ISSTA*, pages 157–168, 2006.

[14] S. F. Siegel and L. F. Rossi. Analyzing BlobFlow: A case study using model checking to verify parallel scientific software. In *Proceedings of European PVM/MPI Meeting*, volume 5205 of *LNCS*. Springer, 2008.

[15] W. Visser, C. S. Păsăreanu, and S. Khurshid. Test input generation with Java PathFinder. In *Proceedings of ISSTA*, pages 97–107, 2004.

A Comparative Study of Randomized Constraint Solvers for Random-Symbolic Testing

Mitsuo Takaki ‡, Diego Cavalcanti †, Rohit Gheyi †,
Juliano Iyoda ‡, Marcelo d'Amorim ‡, and Ricardo Prudêncio ‡

‡ Federal University of Pernambuco, Recife, PE, Brazil
† Federal University of Campina Grande, Campina Grande, PB, Brazil
{mt2,jmi,damorim,rbcp}@cin.ufpe.br
{diegot,rohit}@dsc.ufcg.edu.br

Abstract

The complexity of constraints is a major obstacle for constraint-based software verification. Automatic constraint solvers are fundamentally incomplete: input constraints often build on some undecidable theory or some theory the solver does not support. This paper proposes and evaluates several randomized solvers to address this issue. We compare the effectiveness of a symbolic solver (CVC3), a random solver, three hybrid solvers (i.e., mix of random and symbolic), and two heuristic search solvers. We evaluate the solvers on two benchmarks: one consisting of manually generated constraints and another generated with a concolic execution of 8 subjects. In addition to fully decidable constraints, the benchmarks include constraints with non-linear integer arithmetic, integer modulo and division, bitwise arithmetic, and floating-point arithmetic. As expected symbolic solving (in particular, CVC3) subsumes the other solvers for the concolic execution of subjects that only generate decidable constraints. For the remaining subjects the solvers are complementary.

1 Introduction

Software testing is important and expensive [8, 28, 35]. Several techniques have been proposed to reduce this cost. Automation of test data generation, in particular, can improve testing productivity. Random testing [13, 30] and symbolic testing [25] are two widely used techniques with this goal and with well-known limitations. On the one hand, random testing fails to explore a search space in a systematic manner: it can explore the same program path repeatedly and also fail to explore important paths (i.e., paths to which only a small portion of the space of input data can lead to an execution). On the other hand, pure symbolic testing is problematic for indexing arrays, dealing with native calls, detecting infinite loops and recursion, and, especially, dealing with undecidable constraints. Combined random-symbolic testing [22] has been recently proposed to circumvent these limitations. One important limitation it attempts to address is the *incapability of solving general constraints*. This is the focus of this paper. We study the impact of alternative randomization strategies for solving constraints. In this setting, random-symbolic testing reduces to random-symbolic constraint solving.

One possible way to combine random and symbolic solvers is to first delegate to the random solver the parts of a constraint that build on theories a symbolic solver does not support. Then use the solution to simplify the original constraint. And finally combine the random solution with the one obtained from calling the symbolic solver on the simplified constraint. (For simplicity, we assume the constraint is satisfiable and that the random solver can find a solution.) Important to note is that, as for typical decision procedures in SMT solvers [20, 37], random and symbolic solvers are *not* independent in this combination; they *collaborate*. One practical consequence of this is that the more constraints the symbolic solver rejects the more complex random solving becomes, and conversely. Therefore, random solving is critical for the effectiveness of the combined solver.

We define **recall** as the fraction of constraints that a solver can find solutions out of the total number of *satisfiable constraints*. (We classify aproximately a constraint as satisfiable if at least one solver can find solution to it.) This metric quantifies completeness. Our goal is to increase recall. This paper makes the following contributions:

E. Denney, D. Giannakopoulou, C.S. Păsăreanu (eds.); The First NASA Formal Methods Symposium, pp. 56-65

- The proposal and implementation of several randomized constraint solvers. We implemented a plain random solver, three hybrid constraint solvers, and two search-based solvers. We use the random solver and the symbolic solver (in our case, CVC3 [1]) as baselines for comparison;
- Empirical evaluation of solvers with manually constructed constraints and constraints generated with a concolic execution of 8 subjects.

2 Technique: Randomized Solvers

This section presents randomized solvers with common input-output interface. **Input.** All solvers take as input (i) a constraint *pc* (in reference to a *path condition* from a symbolic execution), (ii) a random seed *s*, and (iii) a range of values $[lo, hi]$. An input constraint takes the form $\bigwedge b_i$, where b_i is a boolean expression constructed, in principle, with any logical system. For example, the expression $x > 0 \land x > y+1$ illustrates a valid input constraint. We often use the term **constraint** alone or **clause** in reference to a single boolean expression b_i and **constraint system** or **pc** in reference to the conjunction of all constraints. **Output.** A **solution** is a vector of variable assignments that satisfies one input constraint. For instance, $\langle x \mapsto 2, y \mapsto 0\rangle$ is a solution to the constraint $x > y+1$ (using integer variables). A solver returns a solution when it finds one or the flag *empty* otherwise.

Note on implementation. We wrote all solvers in the Java language, used the BCEL library [14] to instrument the bytecode of the experimental subject for concolic execution, and used part of the code from the JPF symbolic execution [5] for the integration with CVC3.

2.1 Baseline solvers

We use the solvers **ranSOL** and **symSOL** as representatives of plain random and symbolic solvers respectively. In our experiments we use these solvers as baselines for comparison. Figure 1 shows the pseudo-code for a random constraint solver ranSOL. The main loop generates random input vectors and selects those that satisfy *pc* (lines 1-6). The expression vars(*pc*) denotes the set of variables that occur in *pc*. Function *random* selects random integer values in the range $[lo, hi]$ and builds assignments to each variable in this set (line 2). (For simplification, we only show the case for integers.) The function $eval(pc, \overrightarrow{iv})$ checks whether the candidate solution $\overrightarrow{iv}$ models *pc*. This function evaluates the concrete boolean expression that *pc* encodes using the variable assignments in $\overrightarrow{iv}$. ranSOL returns $\overrightarrow{iv}$ at line 4 if it is a solution to *pc*, or returns *empty* on timeout. Symbolic constraint solvers are complete for a set of decidable theories. For example, CVC3 [1] supports rational and integer linear arithmetic (among others). However, these solvers are incomplete for solving constraints with non-linear arithmetic, integer division and modulo whose theories are undecidable. We use the label **symSOL** to refer to a symbolic solver. We used CVC3 in our experiments.

2.2 Heuristic search solvers

This section discusses two solvers based on well-known heuristic search techniques: genetic algorithms (GA) [23] and particle swarm optimization (PSO) [24]. Conceptually, these solvers attempt to optimize the random search that ranSOL drives. The basic task of these algorithms is to search a *space* of candidate solutions to identify the best ones in terms of a problem-specific *fitness function*. The search process usually starts with the selection of randomly-chosen individuals (i.e., candidate solutions to the search problem) in the search space. The search proceeds by making movements on each individual iteratively with *search operators* until the search meets some stop criteria (e.g., the result is good enough or the search time expired). The decision to move individuals in the search space depends on the evaluation of

```
Input: path condition $pc$
Input: random seed $s$, range $[lo,hi]$
while $\neg timeout$ do
    $\overrightarrow{iv} \Leftarrow$ random(vars($pc$), range)
    if eval($pc$, $\overrightarrow{iv}$) then
        return $\overrightarrow{iv}$
    end if
end while
return empty
```

Figure 1: Random (ranSOL)

```
Input: path condition $pc$
Input: random seed $s$, and range $[lo,hi]$
($pcgood$, $pcbad$) $\Leftarrow$ partition($pc$)
$\overrightarrow{iv_1} \Leftarrow$ symSOL.solve($pcgood$)
if ($\overrightarrow{iv_1}$ = $empty$) then
    return $empty$
end if
$newpc \Leftarrow pcbad \backslash \overrightarrow{iv_1}$
$\overrightarrow{iv_2} \Leftarrow$ ranSOL.solve($newpc$,$seed$,$range$)
return $\overrightarrow{iv_2} = empty\ ?\ empty : \overrightarrow{iv_1} + \overrightarrow{iv_2}$
```

Figure 2: Good constraints first (GCF)

```
Input: path condition $pc$
Input: random seed $s$, and range $[lo,hi]$
($pcgood$, $pcbad$) $\Leftarrow$ partition($pc$)
$sols \Leftarrow$ eRanSOL.solve($pcbad$,$seed$,$range$)
for all $\overrightarrow{iv_1}$ in sols do
    $newpc \Leftarrow pcgood \backslash \overrightarrow{iv_1}$
    $\overrightarrow{iv_2} \Leftarrow$ symSOL.solve($newpc$)
    if $\overrightarrow{iv_2} \neq empty$ then
        return $\overrightarrow{iv_1} + \overrightarrow{iv_2}$
    end if
end for
return empty
```

Figure 3: Bad constraints first (BCF)

```
Input: path condition $pc$
Input: random seed $s$, and range $[lo,hi]$
($goodvars$, $badvars$) $\Leftarrow$ partition($pc$)
while $\neg timeout$ do
    $\overrightarrow{iv_1} \Leftarrow$ random($badvars$)
    $newpc \Leftarrow pc \backslash \overrightarrow{iv_1}$
    $\overrightarrow{iv_2} \Leftarrow$ symSOL.solve($newpc$)
    if $\overrightarrow{iv_2} \neq empty$ then
        return $\overrightarrow{iv_1} + \overrightarrow{iv_2}$
    end if
end while
```

Figure 4: Bad variables first (BVF)

their current fitness values. The principle of these algorithms is that the movements across successive iterations will approximate the individuals to the solution space, i.e., each iteration potentially explores better regions in the search space. We discuss next two common aspects to GA and PSO central to our domain of application: (i) the representation of a solution (individual) and (ii) the fitness function. **Representation of a (candidate) solution.** One solution to a constraint solving problem is a mapping of variables in the constraint system to a concrete value from its domain. For instance, $\langle x \mapsto 2, y \mapsto 0\rangle$ is a solution to $x > y+1$ (using integer variables). **Fitness function.** The fitness function serves to evaluate the quality of **candidate** solutions. Two functions have been widely used for constraint solving problems: MaxSAT [17, 27, 33] and Stepwise Adaptation of Weights (SAW) [6, 16]. MaxSAT is a simple heuristic that counts the number of clauses that can be satisfied by a solution. Maximum fitness is obtained when the solution satisfies all clauses (boolean expressions) in a constraint system (conjunction of clauses). The main issue with MaxSAT is that the solver can sometimes favor solutions that satisfy several easy-to-solve constraints at the expense of solutions that satisfy only a few hard-to-solve. Bäck et al. proposed SAW [6] to reduce the impact of this issue. SAW associates a weight to each clause in a constraint. Each weight is updated with each iteration when it is not satisfied. The use of SAW helps to identify harder-to-solve clauses with the increase of iterations. The solver can use this information to favor individuals (i.e., to reduce movements on those individuals) that are more fit to satisfy harder to solve clauses. We used SAW to evaluate fitness in our GA and PSO implementations.

Summary of GA and PSO. A GA search starts with a population of individuals randomly selected from the search space. Each iteration produces a new population with special operators: a *crossover* combines two individuals to produce others and a *mutation* changes one individual. The individuals are probabilistically selected considering their fitness values. Similar to GA, PSO operates with an initial random population of candidate solutions called *particles*. The interactive collaboration of particles to compute a solution is the main difference between GA and PSO. Each particle has a *position* in the search space and a contribution factor to the population, typically called *velocity*, which PSO uses to update the next position of each particle. A typical PSO iteration updates the velocity of a particle according to global best and local best solutions. The next position of a particle depends on the old position and the new computed velocity. The mutually-recursive equations below govern the update of velocity and position across successive iterations t.

$$v_{t+1} = \omega * v_t + r_1 * c_1 * (best_{part} - x_t) + r_2 * c_2 * (best_{pop} - x_t)$$
$$x_{t+1} = x_t + v_{t+1}$$

Figure 5: Update of velocity and position in Particle Swarm Optimization (PSO).

The vectors v and x store respectively velocities and positions for each particle. We use the label t to refer to one iteration. This label is not the index of the vectors. The coefficient ω, typically called inertia, denotes the fraction of velocity in iteration (instant) t that the particle will inherit in iteration $t+1$. Coefficients r_1 and r_2 are numbers within the range [0,1] randomly generated according to some informed distribution. The vector $best_{part}$ stores the best solution each particle visited and c_1 indicates the confidence level to local solutions (i.e., to one individual particle). The term $best_{pop}$ indicates the best solution in the population and c_2 indicates the confidence level to global solutions. Note that the position of a particle at instant $t+1$ is computed by simply adding the contribution (velocity) v_{t+1}.

2.3 Hybrid solvers

This section describes solvers that conceptually combine ranSOL and symSOL. These hybrid solvers make different decisions in (i) what to randomize and in (ii) which order. **Note on terminology.** We use the term **eRanSOL** in reference to an extension of ranSOL that can return many solutions. We use the term $pc \backslash \overrightarrow{iv}$ to denote a substitution of variables in pc with their concrete values in $\overrightarrow{iv}$. For example, $(x > 0 \wedge x > y+1) \backslash \langle x \mapsto 2 \rangle$ is equivalent to $(2 > 0 \wedge 2 > y+1)$.

Good constraints first (GCF). Figure 2 shows the pseudo-code for the **GCF** solver. At line 1, the solver partitions the constraint *pc* in two: the first, named *pcgood*, contains decidable constraints. The second, *pcbad*, complements the first with undecidable constraints. Recall that *pc* consists of a conjunction of boolean expressions. The algorithm reduces to plain random solving if *pcgood* is empty and to plain symbolic solving if *pcbad* is empty. (We omit these checks for simplicity.) When both parts are non-empty, the combined solver uses the symbolic solver to first find a solution to *pcgood* (line 2). As *pcgood* only contains decidable constraints, an empty answer from symSOL indicates that *pcgood* is unsatisfiable (lines 3-5). Consequently, *pc* is also unsatisfiable since $\neg$*pcgood* implies $\neg$*pc* (from the partition function). In case symSOL finds a solution, the solver produces the constraint *newpc* with the substitution $pcbad \backslash \overrightarrow{iv_1}$. If the random solver can find one solution to *newpc* GCF returns $\overrightarrow{iv_1} + \overrightarrow{iv_2}$ as solution, i.e., variable assignments that the symbolic and random solvers produced, respectively. For illustration, GCF partitions the constraint $b \,\%\, a \neq 0 \wedge a > 0$ in two: $pcgood = a > 0$ and $pcbad = b \,\%\, a \neq 0$. (The modulo operator makes the constraint undecidable.) GCF passes *pcgood* to the symbolic solver, and uses the solution, say $\langle x \mapsto 2 \rangle$, to simplify *pcbad* and finally call the random solver on $b \,\%\, 2 \neq 0$.

Bad constraints first (BCF). Figure 3 shows the pseudo-code for the **BCF** solver. It differs from GCF in the order of randomization: it attempts to solve the undecidable parts first. BCF uses eRanSOL to find many solutions to *pcbad*. The main loop checks for each solution $\overrightarrow{iv_1}$ whether symSOL can find a solution to $pcgood \backslash \overrightarrow{iv_1}$ (lines 3-9) . Note that, differently from GCF, BCF calls symSOL once in each iteration. This algorithm corresponds to the one we discussed in Section 1.

Bad variables first (BVF). Figure 4 shows the pseudo-code for the **BVF** solver. It is similar to BCF in the order of calls to random and symbolic solvers. However, it partitions the problem differently. While the previous hybrid solvers partition the set of clauses from one input constraint, BVF *partitions the set of variables* that occur in that constraint. For example, BVF randomizes only the variable b to solve the constraint $a = b^2 + c$, while BCF and GCF randomizes all variables in this case as they appear in a clause involving non-linear arithmetic. BVF is similar to the one proposed in DART [22] as it randomizes a selection of variables for making the constraint decidable. DART, however, randomizes variables incrementally from left to right in the order they appear in the constraint. The constraint $a = b^2 \wedge \ldots \wedge b = a^2$ illustrates one diference between BVF and DART. BVF randomizes variables b and a while DART can avoid the randomization of a as its value depends only on b's value. We did not evaluate DART itself in this paper.

3 Evaluation

We evaluate the proposed solvers with two sets of experiments. The first compares the solvers we proposed and also the symbolic solver CVC3 [1] using a set of constraints written independently by the authors. The second compares the solvers using constraints generated from the concolic execution [36] of data-structures from a variety of sources.

3.1 Microbenchmark

The microbenchmark consists of 51 satisfiable constraints. We included 15 constraints with only linear integer arithmetic, 7 using the absolute value operator (not supported natively on CVC3), 5 using modulo and division (undecidable), 22 using non-linear integer arithmetic (undecidable), and 2 using floating-point arithmetic. Except for CVC3, we run each solver 10 times with different random seeds, using the range of values [-100,100], and a timeout of 1 second. We selected these input parameters arbitrarily. The experiments show that, except for the symbolic solver CVC3, the average recall of each solver was roughly the same: minimum average recall is 0.85 for BVF and maximum average recall is 0.92 for PSO. As expected CVC3 could not solve most of the constraints in this microbenchmark. It solved 21 out of the 52 constraints. But note that it could solve some special cases of undecidable constraints (only 15 decidable constraints in the microbenchmark). For each constraints except two (one involving non-linear integer arithmetic and the other floating-point) there was a solver that can solve it.

3.2 Concolic execution

Subjects and Setup. We used data-structure from a variety of sources. *bst* (P1) is an implementation of a binary search tree from Korat [11]. *filesystem* (P2) is a simplification of the Daisy file system [32]. *treemap* (P3) is a jdk1.4 implementation (`java.util.TreeMap`) of red-black trees. *switch* (P4) refers to one example program from the jCUTE distribution [36]. *ratpoly* (P5) is an implementation of rational polynomial operations from the Randoop distribution [30]. *rationalscalar* (P6) is another implementation of rational polynomials from the ojAlgo library [3]. *newton* (P7) is an implementation of the newton's method to iteratively compute the square root of a number [2]. *hashmap* (P8) is a jdk1.4 implementation (`java.util.HashMap`) of a map that uses hash values as keys. This experiment uses a concolic (concrete and symbolic) execution [36] to generate constraints for the subject programs described above. A concolic execution interprets the program simultaneously in a concrete and symbolic domain. On the one hand, the use of a concrete state enables a concolic execution to evaluate *deterministically* any program expression. This provides a means to handle infinite loops

```
Input: parameterized test $ptest$
Input: random seed $s$, range $[lo, hi]$
 1: $\overrightarrow{iv} \Leftarrow$ random(vars($ptest$), range)
 2: $result \Leftarrow \{\overrightarrow{iv}\}$
 3: $pcs \Leftarrow pcs$ + run($ptest$, $\overrightarrow{iv}$)
 4: while size($pcs$) $> 0$ do
 5:     $\overrightarrow{iv} \Leftarrow$ solve(pickOne($pcs$), $s$, $range$)
 6:     if $\overrightarrow{iv} \neq empty$ then
 7:         $result \Leftarrow result \cup \{\overrightarrow{iv}\}$
 8:         $pcs \Leftarrow pcs$ + run($ptest$, $\overrightarrow{iv}$)
 9:     end if
10: end while
11: return $result$
```

Figure 6: Concolic Execution Driver

	S1	S2	S3	**S4**	S5	S6	S7
P1	0.17	0.16	0.28	0.98	0.98	0.98	0.98
P2	0.2	0.03	0.21	1.00	1.00	1.00	1.00
P3	0.25	0.57	0.71	1.00	1.00	1.00	1.00
P4	0.48	0.25	0.48	1.00	0.04	0.25	0.15
avg.	0.28	0.25	0.42	**0.99**	0.71	0.77	0.74
P5	0.55	0.00	0.45	0.00	1.00	1.00	0.00
P6	0.92	0.02	0.98	0.00	0.00	0.92	0.18
P7	0.82	0.11	0.89	0.00	0.82	0.82	0.11
P8	0.36	0.64	0.86	0.00	0.36	0.36	0.00
avg.	0.73	0.08	**0.78**	0.00	0.50	**0.88**	0.09

Figure 7: Cell shows recall for each pair subject (row) and solver (column). S1 and S4 correspond to our baseline solvers.

and recursion, exploration of infeasible paths, and array indexing; which are typical limitations of a pure symbolic execution. On the other hand, the use of a symbolic state (which the concrete state is an instance of) enables a concolic execution to collect constraints that lead to non-visited paths along the execution of one concrete path. Figure 6 shows the pseudo-code of a test driver for a concolic execution. The driver takes as input (in addition to random seed and range of values) any procedure with parameters *ptest* and outputs inputs to *ptest* (that will lead execution to its different program paths). One iteration of the main loop explores one concrete path and produces several path constraints (corresponding to non-visited paths along that concrete path). A solution to a constraint, when found, will drive the next concolic execution of *ptest* (line 8). The operation solve at line 5 calls each solver with a 300 milliseconds timeout (based on average time from the microbenchmark). We set the overall timeout to 30 minutes. The concolic execution of the first four subjects only generates integer linear constraint, while the others construct non-linear constraints and unsupported constraints to CVC3.

Discussion. We use the following identifiers to label solvers: S1=ranSOL, S2=GA, S3=PSO, S4=CVC3, S5=GCF, S6=BCF and S7=BVF. Figure 7 shows a summary of the results. For the first 3 subjects the symbolic solver (S4) and consequently all hybrid solvers showed roughly the same average recall. Note that all constraints passed to the solver in this case are decidable. For switch (P4) which also builds decidable constraints, S4 timeouts often. For the last 4 subjects, S4 can rarely find a solution. In these cases, the search-based algorithms performed better on average. However, we observed that often one solver find solutions when the other misses. Figure 8 makes a pairwise comparison of the solvers. Line and row denote identifiers of solvers. A cell on line i and column j indicates that solver i solves a constraint that j misses. Note that, for the 4 experiments at the bottom and switch, the solvers vary significantly in the set of constraints they can solve. These results confirm our expectations that the solvers are complementary. It suggests that one may not be able to predict the heuristic that will fit best for a particular subject; it is preferable to run them all in parallel.

Impact of timeout in recall. Efficiency is important to enable symbolic testing: the number of queries submitted to the solver can be very high. One way to deal with this issue is to reduce the alloted time for constraint solving. However timeout reduction can reduce recall. To observe the impact of timeout in recall, we run each concolic execution experiment using timeouts from 100 to 500ms. We observed that

BST							
	S1	S2	S3	S4	S5	S6	S7
S1	-	5	2	0	0	0	0
S2	4	-	4	0	0	0	0
S3	13	16	-	2	2	2	2
S4	80	81	71	-	0	0	0
S5	80	81	71	0	-	0	0
S6	80	81	71	0	0	-	0
S7	80	81	71	0	0	0	-

Summary: 99 SAT, 354 UNK.
S1:17, S2:16, S3:28, S4:97, S5:97, S6:97,S7:97

FileSystem							
	S1	S2	S3	S4	S5	S6	S7
S1	-	12	0	0	0	0	0
S2	2	-	1	0	0	0	0
S3	1	12	-	0	0	0	0
S4	49	59	48	-	0	0	0
S5	49	59	48	0	-	0	0
S6	49	59	48	0	0	-	0
S7	49	59	48	0	0	0	-

Summary: 61 SAT, 475 UNK.
S1:12, S2:2, S3:13, S4:61, S5:61, S6:61,S7:61

TreeMap							
	S1	S2	S3	S4	S5	S6	S7
S1	-	1	1	0	0	0	0
S2	22	-	11	0	0	0	0
S3	31	20	-	0	0	0	0
S4	49	28	19	-	0	0	0
S5	49	28	19	0	-	0	0
S6	49	28	19	0	0	-	0
S7	49	28	19	0	0	0	-

Summary: 65 SAT, 470 UNK.
S1:16, S2:37, S3:46, S4:65, S5:65, S6:65,S7:65

Switch							
	S1	S2	S3	S4	S5	S6	S7
S1	-	30	0	0	43	32	38
S2	8	-	8	0	20	17	19
S3	0	30	-	0	43	32	38
S4	49	71	49	-	91	71	81
S5	1	0	1	0	-	3	3
S6	10	17	10	0	23	-	10
S7	6	9	6	0	13	0	-

Summary: 95 SAT, 235 UNK.
S1:46, S2:24, S3:46, S4:95, S5:4, S6:24,S7:14

RatPoly							
	S1	S2	S3	S4	S5	S6	S7
S1	-	27	5	27	0	0	27
S2	0	-	0	0	0	0	0
S3	0	22	-	22	0	0	22
S4	0	0	0	-	0	0	0
S5	22	49	27	49	-	0	49
S6	22	49	27	49	0	-	49
S7	0	0	0	0	0	0	-

Summary: 49 SAT, 295 UNK.
S1:27, S2:0, S3:22, S4:0, S5:49, S6:49,S7:0

RationalScalar							
	S1	S2	S3	S4	S5	S6	S7
S1	-	56	1	57	57	0	46
S2	0	-	0	1	1	0	0
S3	5	60	-	61	61	5	50
S4	0	0	0	-	0	0	0
S5	0	0	0	0	-	0	0
S6	0	56	1	57	57	-	46
S7	0	10	0	11	11	0	-

Summary: 62 SAT, 296 UNK.
S1:57, S2:1, S3:61, S4:0, S5:0, S6:57,S7:11

Newton							
	S1	S2	S3	S4	S5	S6	S7
S1	-	20	3	23	0	0	20
S2	0	-	0	3	0	0	0
S3	5	22	-	25	5	5	22
S4	0	0	0	-	0	0	0
S5	0	20	3	23	-	0	20
S6	0	20	3	23	0	-	20
S7	0	0	0	3	0	0	-

Summary: 28 SAT, 305 UNK.
S1:23, S2:3, S3:25, S4:0, S5:23, S6:23,S7:3

HashMap							
	S1	S2	S3	S4	S5	S6	S7
S1	-	0	0	5	0	0	5
S2	4	-	2	9	4	4	9
S3	7	5	-	12	7	7	12
S4	0	0	0	-	0	0	0
S5	0	0	0	5	-	0	5
S6	0	0	0	5	0	-	5
S7	0	0	0	0	0	0	-

Summary: 14 SAT, 379 UNK.
S1:5, S2:9, S3:12, S4:0, S5:5, S6:5,S7:0

Figure 8: Results of various solvers for constraints that concolic execution generates. Column and row show solver identifiers. A cell denotes the difference of constraints that a solver (from row) can solve and another (from column) cannot. The bottom line summarizes the results.

CVC3 typically finds a solution for decidable constraints in less than 100ms. However, for the Switch experiment the recall of CVC3 was 0.25 for 100ms, 0.75 for 200ms and 0.95 from 300 to 500ms. We could not determine the reason for this. In particular, we did not find a strong correlation between the size of the constraints or the number of variables in it and higher impact of time. We also observed a significant variation in recall on PSO for Newton and HashMap and on GA for TreeMap and HashMap. For these cases, we conjecture that the impact of time relates to the complexity of the search problem, i.e., the relative small size of the solution space compared to that of the search space.

4 Related Work

Random-symbolic testing has been widely investigated recently to automate test input generation [22, 26, 21]. It alternates concrete and symbolic execution to alleviate their main limitations. It is important to note that random-symbolic testing provides two orthogonal contributions: (i) constraint generation and (ii) constraint solving. Our goal is to improve constraint solving. In this context, DART [22] conceptually uses a random solver to simplify symbolic solving. We plan to evaluate the solvers we proposed with a DART solver as discussed in Section 2. Another approach to automate test input generation is random testing [10, 18, 31, 29]. The ranSOL solver differs from random testing in two important ways: (a) random testing generates inputs for program parameters; a classification of good input depends on the result of an actual execution, and (b) random testing typically generates test sequence and data simultaneously. We plan to combine random sequence generation together with random-symbolic input generation to automate testing.

We used the Satisfiability Modulo Theories (SMT) [37, 20, 12] solver CVC3 [1], which uses built-in theories for rationals and integer linear arithmetic (with some support to non-linear arithmetic). SAT solving research of undecidable theories has focused on the analysis of hybrid and control systems, as recently evidenced by the iSAT [19] and the ABSolver [7] systems. The first integrates the power of SMT solvers to solve boolean constraints with the capability of Interval Constraint Propagation (ICP) [9] to deal with non-linear constraint systems, while the second uses a DPLL-based [15] algorithm to perform the search and defers theory problems to subordinate solvers. As in hybrid and control systems, undecidable theories also arise in the domain of software systems. This paper shows simple algorithms that can be effective to solve both decidable and undecidable fragments of constraints that a concolic *program* execution generates. Another distinguishing feature of our solvers is that, in contrast to a DPLL(T) [20] solver, they are not dependent on a background theory T. One can use the solvers this paper describes in combination to any theory-specific solver to fully benefit from their complementary nature.

There are variations to the search-based solvers presented in Section 2 which we plan to investigate. Ru and Jianhua propose a hybrid technique which combines GA and PSO by creating individuals in a new generation by crossover and mutation operations [34]. Instinct-based PSO adds another criterion (the instinct) to influence a particle's behavior [4]. The instinct represents the intrinsic “goodness” of each variable of a particle's candidate solution. We also plan to analyze how test inputs generated from our solvers compare to those generated directly with a PSO algorithm whose fitness function is based on coverage [38].

5 Conclusions

This paper proposes and implements a plain random solver, three hybrid solvers combining random and symbolic solvers, and two heuristic search solvers. We use a random solver and a symbolic solver (CVC3) as baselines for comparison. We evaluate the solvers on two benchmarks. One with constraints the authors constructed and the other with constraints that a concolic execution generates on 8 subjects. For the concolic execution on subjects that generated only decidable constraints the the experiments reveal as expected that CVC3 is superior in all but 2 cases. CVC3 timed out in these cases. For solving undecidable constraints, no solver subsumes another. It suggests that one may not be able to predict the heuristic that will fit best for a particular subject; it is preferable to run them all in parallel.

Next we want to analyse several open source projects to quantify the number of constraints that would produce undecidable constraints. We believe this is a necessary step to provide evidence for the practical relevance of this research.

Acknowledgments. We are grateful to Augusto Sampaio, Leopoldo Teixeira and the anonymous reviewers for the comments on this work. This work was partially supported by the CNPQ grant 136172/2008.3 and FACEPE grants APQ-0093-1.03/07 and APQ-0074-1.03/07.

References

[1] CVC3 webpage. `http://www.cs.nyu.edu/acsys/cvc3/`.

[2] Newton's square root webpage. `https://trac.videolan.org/skin-designer/browser/trunk/src`.

[3] ojAlgo webpage. `http://ojalgo.org`.

[4] Ashraf Abdelbar and Suzan Abdelshahid. Instinct-based PSO with local search applied to satisfiability. In *IEEE International Joint Conference on Neural Networks*, pages 2291–2295, July 2004.

[5] Saswat Anand, Corina S. Pasareanu, and Willem Visser. JPF-SE: A symbolic execution extension to Java PathFinder. In *TACAS*, pages 134–138, 2007.

[6] Thomas Bäck, A. E. Eiben, and Marco E. Vink. A superior evolutionary algorithm for 3-sat. In *7th Conference on Evolutionary Programming VII*, pages 125–136, UK, 1998. Springer-Verlag.

[7] Andreas Bauer, Markus Pister, and Michael Tautschnig. Tool-support for the analysis of hybrid systems and models. In *Conference on Design, Automation and Test in Europe (DATE)*, pages 924–929, 2007.

[8] Boris Beizer. *Software Testing Techniques*. International Thomson Computer Press, 1990.

[9] F. Benhamou and L. Granvilliers. Continuous and interval constraints. In *Handbook of Constraint Programming*, Foundations of Artificial Intelligence, chapter 16. Elsevier, 2006.

[10] D. L. Bird and C. U. Munoz. Automatic generation of random self-checking test cases. *IBM Systems Journal*, 23(3):228–245, 1983.

[11] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: Automated testing based on Java predicates. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 123–133, 2002.

[12] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P. van Rossum, S. Schulz, and R. Sebastiani. An incremental and layered procedure for the satisf. of linear arith. logic. In *TACAS*, pages 317–333, 2005.

[13] C. Csallner and Y. Smaragdakis. JCrasher: An automatic robustness tester for Java. *Soft. Practice and Experience*, 34:1025–1050, 2004.

[14] Markus Dahm and Jason van Zyl. Byte Code Engineering Library, April 2003. `http://jakarta.apache.org/bcel/`.

[15] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of ACM*, 5(7):394–397, 1962.

[16] A.E. Eiben and J.K. van der Hauw. Solving 3-sat by gas adapting constraint weights. *Evolutionary Computation, 1997., IEEE International Conference on*, pages 81–86, Apr 1997.

[17] G. Folino, C. Pizzuti, and O. Spezzano. Combining cellular genetic algorithms and local search for solving satisfiability problems. In *IEEE Conference on Tools with Artificial Intelligence*, pages 192–198, 1998.

[18] J. Forrester and B. Miller. An empirical study of the robustness of windows NT applications using random testing. In *USENIX Windows Systems Symposium*, pages 59–68, 2000.

[19] M. Franzle, C. Herde, T. Teige, S. Ratschan, and T. Schubert. Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure. *Journal on Satisfiability, Boolean Modeling and Computation*, 1:209–236, 2007.

[20] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast decision procedures. In *Computer aided verification*, volume 3114 of *LNCS*, pages 175–188. Springer, 2004.

[21] Patrice Godefroid. Compositional dynamic test generation. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 47–54, New York, NY, USA, 2007. ACM.

[22] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed Automated Random Testing. In *Programming Language Design and Implementation*, volume 40, pages 213–223, 2005.

[23] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.

[24] J. Kennedy and R. Eberhart. Particle swarm optimization. In *IEEE Neural Networks*, pages 1942–1948, 1995.

[25] James C. King. Symbolic execution and program testing. *Communications of ACM*, 19(7):385–394, 1976.

[26] Rupak Majumdar and Koushik Sen. Hybrid concolic testing. In *ICSE*, pages 416–426, 2007.

[27] E. Marchiori and C. Rossi. A flipping genetic algorithm for hard 3-SAT problems. In *Genetic and Evolutionary Computation Conference*, volume 1, pages 393–400, Orlando, Florida, USA, 1999. Morgan Kaufmann.

[28] National Institute of Standards and Technology. The economic impacts of inadequate infrastructure for software testing. Planning Report 02-3, May 2002.

[29] C. Pacheco and M. Ernst. Randoop: feedback-directed random testing for Java. In *OOPSLA Companion*, pages 815–816, 2007.

[30] C. Pacheco, S. Lahiri, M. Ernst, and T. Ball. Feedback-directed random test generation. In *International Conference on Software Engineering*, pages 75–84, 2007.

[31] Carlos Pacheco and Michael D. Ernst. Eclat documents. Online manual, Oct. 2004. `http://people.csail.mit.edu/people/cpacheco/eclat/`.

[32] Shaz Qadeer. Daisy File System. Joint CAV/ISSTA Special Event on Specification, Verification, and Testing of Concurrent Software. 2004.

[33] Claudio Rossi, Elena Marchiori, and Joost N. Kok. An adaptive evolutionary algorithm for the satisfiability problem. In *SAC (1)*, pages 463–469, 2000.

[34] Nie Ru and Yue Jianhua. A GA and particle swarm optimization based hybrid algorithm. In *IEEE World Congress on Computational Intelligence*, Hong Kong, 1-6 June 2008.

[35] P. Santhanam and B. Hailpern. Software debugging, testing, and verification. *IBM Systems Journal*, 41:4–12, 2002.

[36] K. Sen and G. Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In *CAV*, pages 419–423, 2006.

[37] Cesare Tinelli. A DPLL-based calculus for ground satisfiability modulo theories. In *Proceedings of the European Conference on Logics in Artificial Intelligence*, pages 308–319, London, UK, 2002. Springer-Verlag.

[38] Andreas Windisch, Stefan Wappler, and Joachim Wegener. Applying particle swarm optimization to software testing. In Hod Lipson, editor, *Genetic and Evolutionary Computation Conference*, pages 1121–1128, 2007.

Component-Oriented Behavior Extraction for Autonomic System Design [*]

Marco Bakera, Christian Wagner, Tiziana Margaria
University of Potsdam, Germany
bakera,wagner,margaria@cs.uni-potsdam.de

Emil Vassev
Lero, University College Dublin, Ireland
emil.vassev@lero.ie

Mike Hinchey
Lero, University of Limerick, Ireland
mike.hinchey@lero.ie

Bernhard Steffen
TU Dortmund, Germany
steffen@cs.tu-dortmund.de

Abstract

Rich and multifaceted domain specific specification languages like the Autonomic System Specification Language (ASSL) help to design reliable systems with self-healing capabilities. The GEAR game-based Model Checker has been used successfully to investigate properties of the ESA ExoMars Rover in depth. We show here how to enable GEAR's game-based verification techniques for ASSL via systematic model extraction from a behavioral subset of the language, and illustrate it on a description of the Voyager II space mission.

1 Introduction

The SHADOWS project (**S**elf-**h**ealing **A**pproach to **D**esigning **C**o**m**plex Soft**w**are **S**ystems) [14, 15] aims at developing technologies that augment large software systems with a sort of immune response against various issues and contingencies that can occur at design-time or runtime. Focusing on functional healing at design time, we developed a number of enabling techniques for functional self-healing. In particular, we introduced game based model checking of behavioral models in the GEAR tool [1, 2] as a deep diagnosis tool for early realignment between behavioral models and requirements expressed as temporal properties that we applied to the analysis of the recovery behavior of the ESA ExoMars Rover.

In this paper we show 1) how we are able to link the behavioral modelling style of our techniques with ASSL [17], a rich domain-specific language for the specification of autonomous systems, equipped with a formal semantics [17], and 2) how we can easily and systematically translate (parts of) the specification of the Voyager's behavior into Service Logic Graphs (SLGs, introduced formally in Section 3.1), thus enabling the application of the SHADOWS technologies to the large class of autonomous systems describable in ASSL. The advantage of SLGs over other models is that they are closer to the field engineer's understanding, thus making advanced game-based diagnosis features accessible to non-experts in formal methods and models.

1.1 The NASA Voyager Mission Case Study

The NASA Voyager Mission started in 1977 and was designed for exploration of the outer planets of the Solar System. As the twin spacecraft Voyager I and Voyager II flew, they took pictures of planets and their satellites in 800x800 pixel resolution, then radiotransmitting them to Earth. Voyager II has two on-board television cameras - one for wide-angle images and one for narrow-angle images - that record images in black and white. Each camera is equipped with a set of colour filters, which help images to be reconstructed as fully-colored ones. Voyager II uses radar-like microwave frequencies to send the stream of pixels toward Earth. The signal suffers on this distance a 20 billion times attenuation [4]. In Vassev and Hinchey [18], the mission is specified as an autonomic system composed of the Voyager II spacecraft and four antennas on Earth, all specified as distinct autonomic elements. This paper is based on this specification and on results regarding the behavior of the system.

E. Denney, D. Giannakopoulou, C.S. Păsăreanu (eds.); The First NASA Formal Methods Symposium, pp. 66-75

[*] This work has been partially supported by the European Union Specific Targeted Research Project *SHADOWS* (IST-2006-35157), exploring a *S*elf-*H*ealing *A*pproach to *D*esigning c*O*mplex soft*W*are *S*ystems. The project's web page is at https://sysrun.haifa.ibm.com/shadows.

In the rest of this paper, we briefly sketch ASSL (Sect. 2) and then how to map the ASSL specification with our models (Sect. 3), and illustrate it on the model for the NASA Voyager mission. We then discuss verification issues (Sect. 4), related work (Sect. 5), and finally conclude (Sect. 6).

2 ASSL

The Autonomic System Specification Language (ASSL) is a framework that provides a multi-tier structure for specifying and validating autonomic systems and targets the generation of an operational prototyping model for any valid ASSL specification [17]. ASSL provides a multi-tier specification model that tackles autonomic systems (ASs) as composed of autonomic elements (AEs) interacting over interaction protocols (ASIP and AEIP). We concentrate here on the behavioral aspects of the AS and AE description, since they are the part of ASSL that finds direct counterpart in the GEAR behavioral models.

- The AS tier - provides a general and global AS perspective. It defines the general system rules in terms of *service-level objectives (SLO)* and *self-management policies*, *architecture topology*, and global *actions*, *events*, and *metrics* applied in these rules. It is similar to the mission and task level of the ExoMars description.
- the AE tier - provides a unit-level perspective, It defines interacting sets of individual autonomic elements (AEs) with their own behavior. This tier is composed of AE rules (*SLO* and *self-management policies*), an *AE interaction protocol (AEIP)*, *AE actions*, *AE events*, and *AE metrics*. It is similar to the Action level of the ExoMars description.

2.1 How the Voyager Takes Pictures

When a space picture must be taken and sent to Earth, the Voyager exhibits autonomous-specific behavior. The spacecraft must detect interesting objects on-the-fly and take their pictures. This reveals a sort of autonomic event-driven behavior that can be easily specified with ASSL at the three main tiers - AS (autonomic system) tier, ASIP (autonomic system specification protocol) tier, and AE (autonomic element) tier [17]. The Voyager II spacecraft and the antennas on Earth are specified at both AS and AE tiers as autonomic elements that follow their autonomic behavior encoded as a self-management policy called `IMAGE_PROCESSING`. ASSL specifies self-management policies with fluents[1], denoting specific system states, and mappings, that map fluents to ASSL actions, i.e. actions to be performed when the system gets into a fluent [17].

2.2 AS Tier Specification.

The `IMAGE_PROCESSING` self-management policy is specified at the AS tier to process images from four antennas on Earth located in Australia, Japan, California, and Spain. In fact, we consider this specification as forming the autonomic image-processing behavior of the Voyager Mission base on Earth.

As shown in Figure 1, the policy is specified with four policy *fluents*—one per antenna. Fluents denote specific system states. They are *initiated* by events prompted when an image has been received and *terminated* by events prompted when the received image has been processed. Further, all the four fluents are *mapped* to an ASSL *action*: that is to be performed when the system enters in one of the fluents. Figure 2 shows the specification of the events that initiate and terminate the fluent presented by Figure 1. Note that the first event is prompted to occur in the system when a special message has been received. In addition, a `processImage` action (see [18] for this action's specification) is specified to process images from all four antennas.

At the autonomic system interaction protocol (ASIP) tier, the image messages (one per antenna), a communication channel that is used to communicate these messages, and communication functions to send and receive these messages over that communication channel to the Earth are specified [18].

[1] ASSL adopts some AI-planning terminology: a fluent is comparable to a state variable in our transition system view.

```
FLUENT inProcessingImage_AntSpain {
  INITIATED_BY { EVENTS.imageAntSpainReceived }
  TERMINATED_BY { EVENTS.imageAntSpainProcessed }}

MAPPING {
  CONDITIONS { inProcessingImage_AntSpain}
  DO_ACTIONS { ACTIONS.processImage("Antenna_Spain") }}
```

Figure 1: An IMAGE_PROCESSING Fluent

```
EVENT imageAntSpainReceived {
  ACTIVATION { RECEIVED {
    ASIP.MESSAGES.msgImageAntSpain } }}

EVENT imageAntSpainProcessed { }
```

Figure 2: AS-tier Events

```
AESELF_MANAGEMENT {
  OTHER_POLICIES {
    POLICY IMAGE_PROCESSING {
      FLUENT inTakingPicture {
        INITIATED_BY { EVENTS.timeToTakePicture }
        TERMINATED_BY { EVENTS.pictureTaken }
      }
      FLUENT inProcessingPicturePixels {
        INITIATED_BY { EVENTS.pictureTaken }
        TERMINATED_BY { EVENTS.pictureProcessed }
      }
      MAPPING {
        CONDITIONS { inTakingPicture }
        DO_ACTIONS { ACTIONS.takePicture }
      }
      MAPPING {
        CONDITIONS { inProcessingPicturePixels }
        DO_ACTIONS { ACTIONS.processPicture }
      }
    }
  }
} // AESELF_MANAGEMENT
```

```
FLUENT inStartingGreenImageSession {
  INITIATED_BY { EVENTS.
          greenImageSessionIsAboutToStart }
  TERMINATED_BY { EVENTS.
          imageSessionStartedGreen }
}
FLUENT inCollectingImagePixelsBlue {
  INITIATED_BY { EVENTS.imageSessionStartedBlue }
  TERMINATED_BY { EVENTS.imageSessionEndedBlue }
}


EVENT greenImageSessionIsAboutToStart {
  ACTIVATION { SENT { AES.Voyager.
    AEIP.MESSAGES.msgGreenSessionBeginAus } }
}
EVENT imageSessionStartedBlue {
  ACTIVATION { RECEIVED { AES.Voyager.
    AEIP.MESSAGES.msgBlueSessionBeginAus } }
}
```

Figure 3: AE antenna self-management policies, fluents, events

2.3 AE Tier Specification.

At this tier, we have five autonomic elements: the Voyager II spacecraft and the four antennas on Earth. For each, an own part of the `IMAGE_PROCESSING` self-management policy is specified.

AE Voyager. The spacecraft's `IMAGE_PROCESSING` self-management policy (see Figure 3) uses two fluents. The `inTakingPicture` fluent is initiated by a `timeToTakePicture` event and terminated by a `pictureTaken` event. This event also initiates the `inProcessingPicturePixels` fluent, which is terminated by the `pictureProcessed` event. The fluents are mapped to the actions `takePicture` and `processPicture` respectively. Metrics are used, for example, to count all the detected interesting objects which the Voyager AE takes pictures of.

AE Antenna. The four antennas receiving signals from the Voyager II spacecraft are specified as autonomic elements. Their `IMAGE_PROCESSING` self-management policy uses pairs of fluents `inStartingImageSession` - `inCollectingImagePixels`, one for each color filter. These sets of fluents determine the states of the antenna AEs when an image-receiving session is starting and when an antenna AE is collecting the image pixels.

Since the `Voyager` AE processes the images by applying different filters and sends each filtered image separately, we have distinct fluents for each color and antenna. This allows an antenna AE to process a collection of multiple filtered images simultaneously.[2] It is the `Voyager` AE that notifies an

[2] Note that according to the ASSL formal semantics, a fluent cannot be re-initiated while it is initiated, thus preventing the same fluent from being initiated simultaneously twice or more times [17].

antenna that an image-sending session begins and ends. Figure 3 shows two of the `IMAGE_PROCESSING` fluents. They are further mapped to AE actions that collect the image pixels per filtered image (see [18]).

In Figure 3 we see how two of the events initiate the AE Antenna fluents. The `greenImageSessionIsAboutToStart` event is prompted (triggered) when the Voyager's `msgGreenSessionBeginSpn` message has been sent and the `imageSessionStartedBlue` event is prompted when the Voyager's `msgBlueSessionBeginSpn` message has been received by the antenna.

3 Mapping ASSL to GEAR Models

ASSL specifications describe all the different aspects of an autonomic system in one comprehensive document. This is practical, but by nature in realistic cases it becomes very complex, the complexity to a good extent due to the many cross references between the specification elements. A trace through the specified autonomic system may request jumping between different aspects (e.g. from messages → events → fluents → mappings → actions) and "pages". Another paper in this proceedings proposes mapping ASSL specifications to LTS, in order to verify LTL properties [19]. Here, we address a different mapping, that enables intuition of the graphical models, expression of constraints in any mu-calculus derivative, and a deep support to diagnosis by means of reverse model checking and games. Our models are Service Logic Graphs (SLG).

3.1 Behavioral models: Service Logic Graphs

To complement the original textual view, and in perspective to visualize and reify certain aspects of the SOS semantics of ASSL, we map selected behavioral elements of the specification to GEAR's behavioral models. These can be visualized as Service Logic Graphs (SLG) in the jABC framework [13, 16] (of which GEAR is the model checking plugin) and analyzed, guiding the user through the processes and workflows of the specified autonomic system. These models are directly amenable to model checking.

SLGs themselves are composed of reusable building blocks that are called *Service Independent Building Blocks* (SIBs) [9], and may represent both a single atomic service or a whole subgraph (i.e., another SLG). Thus SLGs can be hierarchical, which grants a high reusability not only of the building blocks, but also of the models themselves, within larger systems. SLGs formally stem from the concept of Kripke Transition Systems [11].

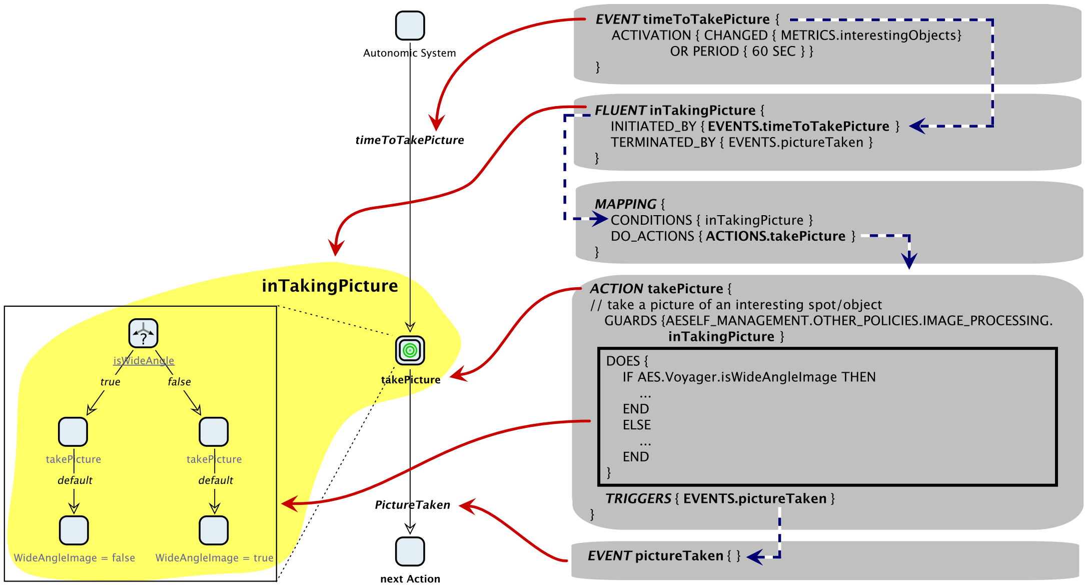
Kripke Transition System A Kripke Transition System K is defined as a tuple $(S, \mathsf{Act}, \rightarrow, I)$ over a set of atomic propositions AP, disjoint from Act, where

- S are the states of the model,
- Act is a set of actions,
- $\rightarrow \subseteq S \times \mathsf{Act} \times S$ are the possible transitions in the model, and
- a labeling interpretation function $I : S \rightarrow 2^{\mathsf{AP}}$ equips states with atomic propositions.

A KTS is best-suited for verification tasks that focus on transitions of the system as being the edges. On the contrary, one can think of an SLG as being the engineer's view of the system that focuses on the *actions* of the system as being the nodes.

Service Logic Graph A *Service Logic Graph (SLG)* is defined as a tuple $(S, \mathsf{Act}, \rightarrow, I)$ over a set of atomic propositions AP, disjoint from Act, where

- S represents the occurrences of the Service Independent building blocks (SIBs), which are the actions or functions in the graph
- *Act* is the set of possible branching conditions, to be determined upon execution of the preceding SIB,
- $Trans = (s, a, s)$ is a set of transitions where $s, s \in S$ and $a \in Act$, and
- a labeling interpretation function $I : S \rightarrow 2^{\mathsf{AP}}$ equips SIB occurrences with atomic propositions.



```
EVENT timeToTakePicture {
   ACTIVATION { CHANGED { METRICS.interestingObjects}
              OR PERIOD { 60 SEC } }
}

FLUENT inTakingPicture {
   INITIATED_BY { EVENTS.timeToTakePicture }
   TERMINATED_BY { EVENTS.pictureTaken }
}

MAPPING {
   CONDITIONS { inTakingPicture }
   DO_ACTIONS { ACTIONS.takePicture }
}

ACTION takePicture {
// take a picture of an interesting spot/object
   GUARDS {AESELF_MANAGEMENT.OTHER_POLICIES.IMAGE_PROCESSING.
            inTakingPicture }
   DOES {
      IF AES.Voyager.isWideAngleImage THEN
         ...
      END
      ELSE
         ...
      END
   }
   TRIGGERS { EVENTS.pictureTaken }
}

EVENT pictureTaken { }
```

Figure 4: Action, Event, Fluent, and *Mapping* in KTS behavioral model representation

The structural match, KTS and SLGs are both graph structures with labeled branches and nodes that are enriched with atomic propositional properties, suffices to adopt the established model checking technologies for the SLGs.[3]

In mapping the elements of the ASSL specification to a graphical representation in the behavioral model we focus on those constructs that describe behavioral and self-* aspects: these are the central elements which will be most frequently used to specify autonomic systems. We currently cover

- the AS tier: Service-Level Objectives, Self-Management Policies, Actions and Events.
- the AE tier: Service-Level Objectives, Self-Management Policies, Actions and Events, and additionally behavioral models and outcomes.

Architectural, communication, and quantitative aspects will be dealt in future work.

3.2 Mapping ASSL Elements

From the point of view of model generation, AS and AE specifications are structurally similar w.r.t. events, self management policies, and actions, but differ in the scoping—while the AS specification has a global scope, the AE specification is only valid for the local element. Due to the similarities, we focus on the description of the autonomic element (AE) tier. The AS tier is captured similarly, by means of hierarchy (where single nodes of the AS-level KTS are expandable to AE-level models).

We refer to Figure 4, showing a specification fragment for the Voyager (right) and the corresponding section of the behavioral model (left). In the textual specification (right), we have two events, one fluent with a mapping, and one action. Dashed arrows illustrate a trace of an event within the specs. Arrows indicate the correspondence between elements of the ASSL-specification and of the behavioral. The InTakingPicture cloud defines the current state of the system (an atomic proposition).

AE Event. Event is the central language element in ASSL. It specifies fluents, actions, and policies globally in the AS tier and locally in the AE tier. Events could be activated by messages, other events,

[3] In fact both system representation styles can easily be translated into each other by adequately mapping edges to nodes and vice versa.

actions or metrics. In our behavioral model, events are mapped to homonymous Branches. In Figure 4, the behavioral model starts with the event *timeToTakePicture*, activated for interesting objects or after a time period of 60 s. It initiates the self management policy (fluent) *inTakingPicture*.

AE Self Management Policy. It defines the behavior of the autonomic system by connecting specific system states with the intended (re)action. A policy consists of two elements:

- A *fluent*, similar to a state. It is initiated (ie., that state is reached) when the system satisfies specific *conditions*. It will be terminated (left) if specific events occur. Fluent activation and termination is driven by events.
- A *mapping* of certain conditions to *actions*. The conditions test fluents: in a certain state, certain actions (in the AS or AE tier) are performed. Actions activate specified actions.

They are central to the model extraction: the information contained in a self management policy is used and useful both for model construction and for verification.

Together, fluent and mappings define the control flow, i.e. create branches with the name of the initiating event. They define all possible incoming branches of an action. The specific condition that activates the fluent is stored in the *context* of the system's model. The context represents the current global state of the system, like a global Blackboard or shared memory-mechanism. For model checking purposes, the fluent is additionally associated as atomic proposition to the corresponding node(s) of the behavioral model. This enables global model checking. The fluent can be used as preconditions of actions. They hold on all states in the region between initiation and termination.

The fluent in our example is activated by the *timeToTakePicture*-event and the overall status of the autonomous system is changed to *intakingPicture*. This change activates an action: *takePicture* which is specified in the Mapping section of the self management object.

The self management policy which connects the event to actions is additionally used to annotate the nodes in the behavioral model with atomic propositions (AP). The name of the AP is equal to the name of the fluent. They can later be used for model checking.

AE Action. Actions are routines performed by AE or AS (global and local). In our behavioral model, they are the second essential element: the nodes of our behavioral model, named as the action. The different elements of an action are used to describe the nodes and for verification purposes. Action *parameters* become parameters of a node; the *does* part represents the body of a node. It can be a single action (then the node is an atomic node), but for complex *does* it is an entire behavioral model. We then model them as a SLG hierarchy, as in Figure 4: the node *takePicture* has a corresponding submodel, presented on the left. The *guards*, *returns* and *outcomes* are used for verification. We offer two possibilities for verification:

- The Localchecker uses the Guard to verify if an Action could be executed within the current system state (defined by the fluents and stored in a global context).
- We can use a model checker to verify relations of nodes and actions expressed as temporal logic constraints. Internally, GEAR uses the modal mu-calculus [10] enriched with forward and backward modalities, so it is best equipped, for example, to express dataflow properties, or other behavioral constraints such as CTL formulas.

The specified action in Figure 4 contains a guard which must conform to the AP annotated at the node.

How the outgoing branches of a node are defined depends on the information found in the action's specification. Actions can use events; triggers are communication functions to communicate with the autonomic system and its elements. We thus have several possibilities to detect outgoing branches.

- a trigger statement in the specification of an action will create an event which introduces the next fluent and/or action, and is comparable to an outgoing branch,

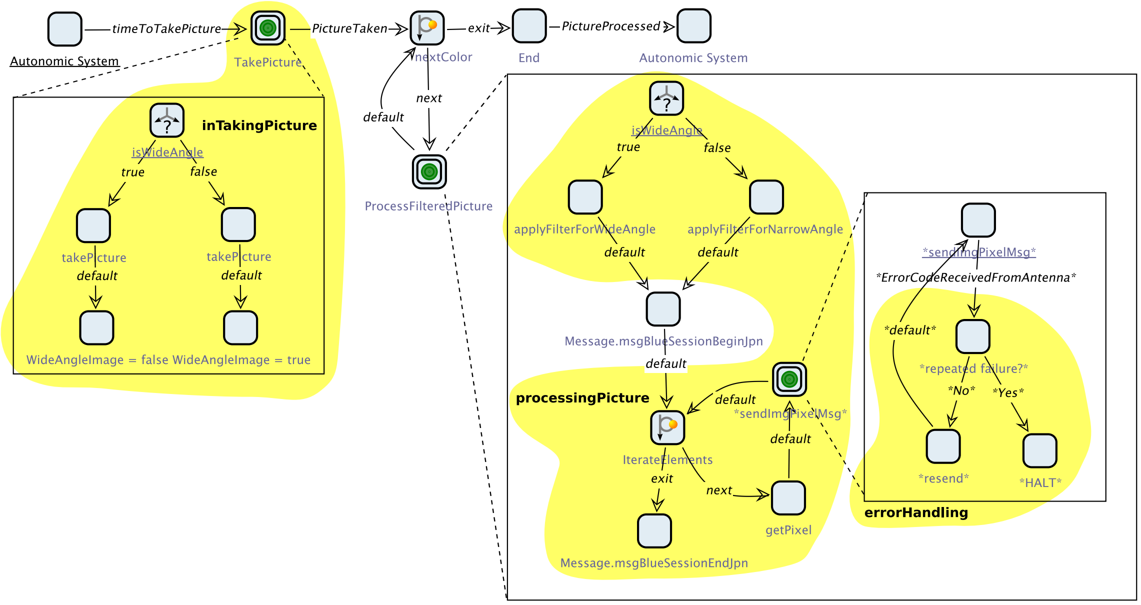


Figure 5: Behavioral model of the picture transmission process. Bottom right: a new error handling recovery mechanism.

- event statements in the Does part are added as possible outgoing branches,
- if communication functions are used, we follow the chain from the function to the communication channel to the events which will be activated by a specific message in the channel. It is not unusual that more than one event will be created from one message.

The *takePicture* action of Figure 4 is closed by a new event *pictureTaken*. This is specified in the triggers section and represents the outgoing branch of this node. The new event will again initiate a fluent, and it terminates the *inTakingPicture* fluent. Therefore, the next action has another AP.

AE Outcomes, AE Behavioral Models, and AE Recovery Protocol. These elements are not yet treated in depth. They will become relevant when applying the SHADOWS methodology. In short, **AE Outcomes** are post-conditions of actions or behavioral models — they are useful for verification purposes. An **AE Behavioral Model** is comparable, from the model generation point of view, to a further mapping in the self-management policy. It consists of conditions, a do element where an action is activated, and outcomes. We can model the behavioral model similarly to an action (atomic or hierarchical). Condition and outcomes become the pre- and post-condition and the action is the implemented behavior. An **AE Recovery Protocol** should guarantee fault-tolerant operation of the autonomous system (e.g. create snapshots, log messages, consistency checking). A recovery protocol specification is rather complex, and it is specified in a separate submodel.

4 Verifying the Voyager's Behavioral Model

Figure 5 contains the behavioral model of the Voyager II spacecraft. Note that the error handling graph at the right was not part of the original ASSL specification.

A simple verification issue that immediately emerges is whether the system takes care of an error-handling process whenever picture pixels are transmitted. This can be easily expressed in CTL [6] as

$$\mathsf{AG}(\mathsf{inProcessingPicturePixels} \Rightarrow \mathsf{EF}(\mathsf{errorHandling}))$$

This formula can be interpreted as follows:

> Wherever the system evolves to (the AG-part), whenever picture pixels are about to be processed (the atomic proposition inProcessingPicturePixels) it follows that the system has an option to evolve into an error-handling process (the EF(errorHandling)-part).

Since the original model of Figure 5 does not support any kind of self-healing capabilities, this property does not hold.

Therefore, in a first attempt to reconcile model and property, we added an error-handling routine directly in the model. We slightly changed the design manually, by refining the sendImgPixelMsg action, originally atomic, to an entire routine. Now, if problems during the transmission process occur, the system tries to resend those picture pixels that were not transmitted correctly. If the problem still exists afterwards, the system is halted and needs manual interaction from ground control.

A game-based approach as presented in Bakera et al. [1] would do much more than just allowing the identification of the missing recovery mechanism in the original specification. Enabling this investigation for self-healing and self-healing enactment is our aim. A domain-dependent guidance also enables to pinpoint that part of the model which is best-suited for integration of recovery mechanisms. Due to space limitation, we cannot discuss this process in detail in this contribution.

Enabling Model based Self-healing Within SHADOWS, we adopt a model-based approach, where models of desired software behavior direct the self-healing process. This allows for life cycle support of self-healing applicable to industrial systems. In particular, we show how to model the several abstraction levels of the system's behavior in a uniform and formal but intuitive way. This happens in term of processes in the jABC framework [16], a mature, model-driven, service-oriented process definition platform. Subsequently, we leverage the formality of these models to prove properties by model checking. In particular we exploit the interactive character of game-based model checking to show how to discover an error, then localize, diagnose, and correct it. Design-time healing technologies that naturally emerge when dealing with self-adaptive systems, as in the context of the SHADOWS project, demand for a deeper insight of design-time faults to effectively identify and overcome them.

The use of models rather than code is already a significant step towards the understandability of the actual behavior's descriptions to non programmers, like the engineers, in charge of designing a space module. This enables, for example, early discovery of misbehaviors, hazards, and ambiguities via design-time analysis. We strive to improve the diagnostic features making them as detailed as necessary yet as intuitive as possible. GEAR [1], our model checker capable for the full modal μ-calculus, has a rich user interface that allows pinpointing problems in system design. This is achieved by interactively exploring the problem space in a game-based way. The game-based nature of GEAR's verification algorithm supports the system designer at design-time to interactively explore the problem space upon property mismatches.

In the case of the Voyager mission case study, such properties can be used to check for complete picture transmission to the four antennas in case of transmission interrupts. Further, the verification process is able to assure the application of all four color filters before picture transmission. In addition it is essential for the picture transmission to send closing notification signals of transmission endings to the antennas. This as well can be assured by the aforementioned Model Checking techniques.

If problems occur in the verification task, one immediate result of the game-based algorithm of the Model Checker is an interactive counter-example. This counter-example both pinpoints the problem of the property mismatch and provides a strategy encoded into the counter-example to adapt and self-heal the system. GEAR [1] elaborates on the application of this technique on an ESA mission example.

5 Related Work

Nowadays, there is a growing consensus that model checking is most effective as an intelligent and early error-finding technique rather than a technique for guaranteeing correctness. This is partly due to the fact that specifications that can be checked automatically through model checking are necessarily partial in that they specify only certain aspects of the system behavior. Therefore, successful model checking runs, while reassuring, cannot guarantee full correctness. Rather, model checkers are increasingly conceived as elaborate debugging tools that complement traditional testing techniques.

Various model checkers are used to verify aerospace systems. Java Pathfinder [7], developed at NASA Ames, is a prominent example for verifying smaller systems. It assists developers at the Java code level, and therefore addresses a later phase than our approach. We aim at assertions on interactions between components or of the system as a whole, with a focus on demanding properties.

For model checkers to be useful as debugging tools it is important that failing model checking attempts are accompanied by appropriate *error diagnosis* information that explains *why* the model check has failed. Model checkers may in fact also fail *spuriously*, i.e., although the property does not hold for the investigated abstraction it may still be valid for the real system. In order for model checking to be useful, it should therefore be easy for the user to rule out spurious failures and to locate the errors in the system based on the provided error diagnosis information. Therefore, it is important that error diagnosis information is easily accessible by the user.

Currently, ASSL provides a consistency checking mechanism to validate specifications of autonomic systems against correctness properties. Although proven to be efficient with handling consistency errors, this mechanism cannot handle logical errors. Another paper in this proceedings [19] proposes a different model checking approach for ASSL, based on Labelled Transition systems and LTL properties.

For linear-time logics (LTL), error diagnosis information is conceptually of a simple type: It is given by a (possibly cyclic) execution path of the system that violates the given property. Thus, in case model-checking fails, linear-time model checkers like SPIN [8] compute an output in the form of an *error trace* that represents a violating run, and is therefore valuable for the subsequent diagnosis and repair.

The produced error traces (also called counter-examples) can be used for simulation to reproduce the execution that leads to an error, e.g., a counter-example could be translated into a UML sequence diagram. However, SPIN supports multiple counter-examples; i.e., if a correctness property is not satisfied, there will be multiple error traces generated leading to the same error. Although, the shortest path can be determined manually, automatic analysis and generation of relevant executions requires a formal approach to the possible variations of a counter-example.

In general, the LTL model checking can be done though generalization to a broader class of linear formalism than LTL. This is possible through transformation into the so-called Büchi automaton and is often implemented on top of CTL model-checking algorithms by transforming the LTL model checking over one structure into checking fairness over another structure [5].

The situation is more complex for properties that embody recovery issues. These claim for more demanding properties expressible in branching-time logics like CTL or the modal μ-calculus. Such logics do not just specify properties of single program executions but properties of the entire execution tree, comprising the locale of decision points. Hence, meaningful error diagnosis information for branching-time logic model checking cannot be represented by linear executions in general. Here games help.

6 Summary and Conclusion

We have shown how to translate parts of an ASSL specification for autonomic systems into a behavioral model. This task implied mapping the ASSL specific *self-management policy, action*, and *event* parts that made up the system to corresponding counterparts in a behavioral system model that is based on a Service Logic Graph. We applied this translation step to the NASA Voyager II mission case study, opening up several options for verifying issues related to e.g. recovery issues. Having detected the absence of a recovery mechanism upon transmission error within the system specification, we may leverage GEAR to fix this problem. A game-based exploration of the problem space as already suggested a tool supported enhancement of the model-driven verification process [1] can help in identifying those parts of the model that need adaptation to overcome this specific problem. However, we did not elaborate on this exploration here since the translation of the specification is still incomplete.

We have previous experience of automatic generation of control flow graphs from a language's Structured Operational Semantics [12] (SOS). In [3] we showed how to do it for a process algebra, later extended for object oriented languages. Accordingly, we plan to examine the SOS for ASSL provided in [17] and possibly take it as a starting point for an SOS-driven generation of the SLGs. This way, the palette of model analyses developed in the jABC and the self-healing specific techniques developed in SHADOWS would become immediately applicable to all ASSL descriptions.

References

[1] M. Bakera, T. Margaria, C. Renner, and B. Steffen. Verification, diagnosis and adaptation: Tool supported enhancement of the model-driven verification process. *ISSE, Innovations in System and Software Engineering - a NASA Journal , Springer Verlag*, in print.

[2] M. Bakera, T. Margaria, C. Renner, and B. Steffen. Game-Based Model Checking for Reliable Autonomy in Space. *AIAA Journal of Aerospace Computing, Information and Communication(JACIC)*, to appear.

[3] V. Braun, J. Knoop, and D. Koschützki. *cool: A control-flow generator for system analysis*. Technical Report MIP-Bericht Nr. 9801, Faculty of Mathematics and Informatics, University of Passau, Germany, 1998.

[4] M. W. Browne. Technical magic converts a puny signal into pictures. *NY Times*, 1989.

[5] J. Burch, E. Clarke, K. Mcmillan, D. Dill, and L. Hwang. Symbolic model checking: 10 pow 20 states and beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 1–33. IEEE Computer Society Press, 1990.

[6] E. M. Clarke and E. A. Emerson. Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic. In *Logics of Programs — Proc. 1981 (LNCS Volume 131)*, pages 52–71. Springer-Verlag: Heidelberg, Germany, 1981.

[7] K. Havelund and T. Pressburger. Model Checking JAVA programs using JAVA PathFinder. *STTT*, 2(4):366–381, 2000.

[8] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, Boston, Massachusetts, USA, 2003.

[9] ITU. General recommendations on telephone switching and signaling - intelligent network: Introduction to intelligent network capability set 1, Recommendation Q.1211. Technical report, Standardization Sector of ITU, Geneva, March 1993.

[10] D. Kozen. Results on the propositional μ-calculus. In *ICALP*, volume 140 of *LNCS*, pages 348–359, Aarhus, Denmark, 12–16 July 1982. Springer-Verlag.

[11] M. Müller-Olm, D. A. Schmidt, and B. Steffen. Model-Checking: A Tutorial Introduction. In *SAS*, pages 330–354, 1999.

[12] G. D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[13] Ralf Nagel. jABC. `http://www.jabc.de`.

[14] SHADOWS. A self-healing approach to designing complex software systems. `https://sysrun.haifa.ibm.com/shadows/`.

[15] O. Shehory, S. Ur, and T. Margaria. Self-healing technologies in SHADOWS: Targeting performance, concurrency and functional aspects. In *10th (CONQUEST)*, 2007.

[16] B. Steffen, T. Margaria, R. Nagel, S. Jörges, and C. Kubczak. Model-Driven development with the jABC. In *Hardware and Software, Verification and Testing*, pages 92–108, 2007.

[17] E. Vassev. *Towards a Framework for Specification and Code Generation of Autonomic Systems*. PhD thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, 2008.

[18] E. Vassev and M. Hinchey. ASSL Specification Model for the Image-processing Behavior in the NASA Voyager Mission. Technical report.

[19] E. Vassev, M. Hinchey, and A. Quigley. Model checking for autonomic systems specified with ASSL. In *Proc. First NASA Formal Methods Symposium (NFM 2009)*. NASA, 2009.

Automated Verification of Design Patterns with LePUS3

Jonathan Nicholson* Epameinondas Gasparis* Amnon H. Eden* † Rick Kazman‡ §

http://essex.ac.uk/

Abstract

Specification and [visual] modelling languages are expected to combine strong abstraction mechanisms with rigour, scalability, and parsimony. LePUS3 is a visual, object-oriented design description language axiomatized in a decidable subset of the first-order predicate logic. We demonstrate how LePUS3 is used to formally specify a structural design pattern and prove ('verify') whether any Java$^{\mathrm{TM}}$ 1.4 program satisfies that specification. We also show how LePUS3 specifications (charts) are composed and how they are verified fully automatically in the Two-Tier Programming Toolkit.

Keywords: specification, automated verification, visual languages, design description languages, object-oriented design

Related Terms: first-order predicate logic, finite model theory, Java 1.4

1 Context

Software systems are the most complex artefacts ever produced by humans [1][2], and managing complexity is one of the central challenges of software engineering. According to the second Law of Software Evolution [3], complexity also arises because most programs are in a continuous state of flux. Maintaining consistency between the program and its design documentation is largely an unsolved problem. The result is most often a growing disassociation between the design and the implementation layers of representation [4]. Formal specification of software design and tools that support automated verification are therefore of paramount importance. Of particular demand are tools which, by a click of a button, can conclusively establish whether a given implementation is consistent with ('satisfies') the design. Attempts towards this end include Architecture Description Languages (ADLs) [5] and formal pattern specification languages [6]. Specifically, we are concerned with the following set of desired criteria:

- **Object-orientated:** suitable for modelling and specifying the building-blocks of the design of object-oriented programs and patterns
- **Visual:** specifications serve as visual presentations of complex (possibly hypothetical) systems
- **Parsimonious:** represent complex design statements parsimoniously, using a small vocabulary
- **Scalable:** abstraction mechanisms that scale well such that charts do not clutter as the size of programs increase
- **Rigorous:** mathematically sound and axiomatized such that all assumptions are explicit
- **Decidable:** fully-automated formal verification is possible at least in principle
- **Automatically verifiable:** accompanied by a specification and (fully automated) verification tool

LePUS3 (LanguagE for Patterns Uniform Specification, version 3) is an object-oriented Design Description Language [7] tailored to meet these criteria. It is particularly suited to representing design motifs such as structural and creational design patterns. LePUS3 'charts' are formal specifications, each of which stands for a set of recursive (fully Turing-decidable) sentences in the first-order predicate logic.

E. Denney, D. Giannakopoulou, C.S. Păsăreanu (eds.); The First NASA Formal Methods Symposium, pp. 76-85

*School of Computer Science and Electronic Engineering, University of Essex, UK

†Centre for Inquiry, Amherst, NY, USA

‡Software Engineering Institute, Carnegie-Mellon University, USA

§University of Hawaii, USA

LePUS3 logic is based on Core Specification Theory [8] which sets an axiomatized foundation in mathematical logic for many formal specification languages (including Z, B, and VDM). The axioms and semantics of LePUS3 are defined using finite model theory. The relation between LePUS3 specifications (*charts*) and programs is well-defined [9] (formulated using the notion of an *Abstract Semantics* function) and the problem of satisfaction is Turing-decidable. Furthermore, consistency between any LePUS3 chart and a standard Java 1.4 [10] program—henceforth, the problem of *verification*—can be established by a click of a button. This quality is demonstrated with the Two-Tier Programming Toolkit (discussed in §6), a tool which fully automates the verification of LePUS3 charts against Java 1.4 programs in reasonable time.

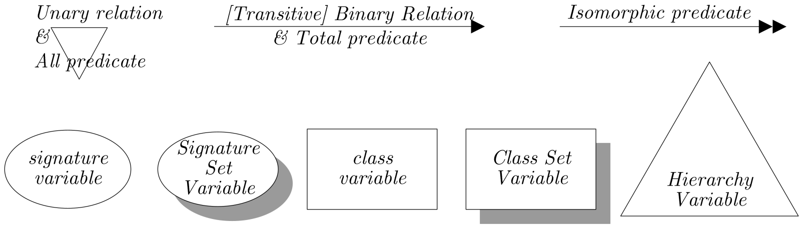


Figure 1: LePUS3 vocabulary, some visual tokens not used in this paper were omitted

The detailed syntax, axioms and truth conditions which constitute the logic of LePUS3 are laid out in [11]. Due to space limitations, our presentation focuses on a single example of the specification and verification of an informal hypothesis. LePUS3 also effectively specifies many other design patterns [12] and the design of object-oriented class libraries encoded in any class-based programming language (e.g. C++, Smalltalk). However, to maintain decidability the language is largely limited to the structural an creational aspects of such designs. In §2 we define informally our initial informal hypothesis, which we refine in §3 and §4 by specifying the design in LePUS3 and offer an abstract representation ('abstract semantics') of the implementation, respectively. In §5 we present a logic proposition that formalizes our original hypothesis and prove it. In §6 we present a tool which fully automates the verification process and discuss an experiment we are currently undertaking, which will test our predicted benefits in program comprehension, conformance and maintenance.

2 The problem

As a leading example we focus on a common claim (e.g. [13][14][15]) that the package `java.awt` in version 1.4 of the standard distribution ('Software Development Kit' [16]) of the Java programming language [10] 'implements' the Composite design pattern, quoted in Hypothesis 1.

Hypothesis 1. `java.awt` implements the Composite design pattern

In this section we examine the informal parts of Hypothesis 1: the Composite design pattern and package `java.awt`. The remainder of this paper is dedicated to formalizing and verifying this hypothesis.

2.1 The Composite Design Pattern

Design patterns have made a significant contribution to software design, each describing an abstract design motif—a recurring theme which in principle can be implemented by an unbounded number of programs in any class-based programming language:

> *A design pattern names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design . . . Each design pattern focuses on a particular object-oriented design problem or issue.*[17]

Table 1 quotes the solution advocated by the Composite design pattern. As is the custom of most pattern catalogues, it is described informally.

Intent: Compose objects into tree structures to represent part-whole hierarchies.
Participants:
– *Component:* Declares a basic interface, implementing default behaviour as appropriate.
– *Leaves:* Have no children, implements/extends superclass behaviour as appropriate.
– *Composite:* Has children, defines behaviour for components having children.
Collaborations: Interface of Component class is used to interact with objects in the structure. Leaves handle requests directly. Composite objects usually forward requests to each of its children, possibly performing additional operations before and/or after forwarding.

Table 1: The Composite design pattern [17] (abbreviated)

2.2 Package `java.awt`

Package `java.awt` ('Abstract Window Toolkit') is part of the standard distribution of Java Software Development Kit 1.4 [16] which provides user interface widgets (e.g. buttons, windows, etc.) and graphic operations thereon. Class `Component` represents a generic widget that is extended [in]directly by all non menu-related widgets (e.g. `Button`, `Canvas`). `Container` represents widgets which aggregate (hold an array of instances of) widgets. Excerpts of the package's source code that corroborate Hypothesis 1 are provided in Program 1. All references to `java.awt` shall henceforth refer exclusively to those aspects listed in Program 1.

```
public abstract class Component ... {
  public void addNotify() ...
  public void removeNotify() ...
  protected String paramString() ... }
public class Button extends Component ... {
  public void addNotify() ...
  protected String paramString() ... }
public class Canvas extends Component ... {
  public void addNotify() ...
  protected String paramString() ... }
public class Container extends Component {
  Component component[] = new Component[0];
  public Component[] getComponents() ...
  public Component getComponent(int) ...
  public void addNotify() { component[i].addNotify(); ... }
  public void removeNotify() { component[i].removeNotify(); ... }
  protected String paramString() { String str = super.paramString(); ... } ... }
```

Program 1: `java.awt` [16] (abbreviated)

3 Specification

Most contemporary modelling languages [18] and notations offer a means of representing implementation minutiae. Design patterns however describe design motifs: abstractions that are not tied in to specific programs. Therefore, the representation of design patterns requires accurately capturing generic abstractions involving collections of entities (e.g. ‘*composite*’, ‘*component*’) that are characterized not by a particular implementation but by their properties and relations (e.g., ‘*composite* is a class that has children of type *component*’). Our Design Description Language must therefore be useful in representing generically, among others, the category of entities and relations which constitute the building-blocks of design patterns, namely [sets of] classes, [sets of] methods, and their correlations.

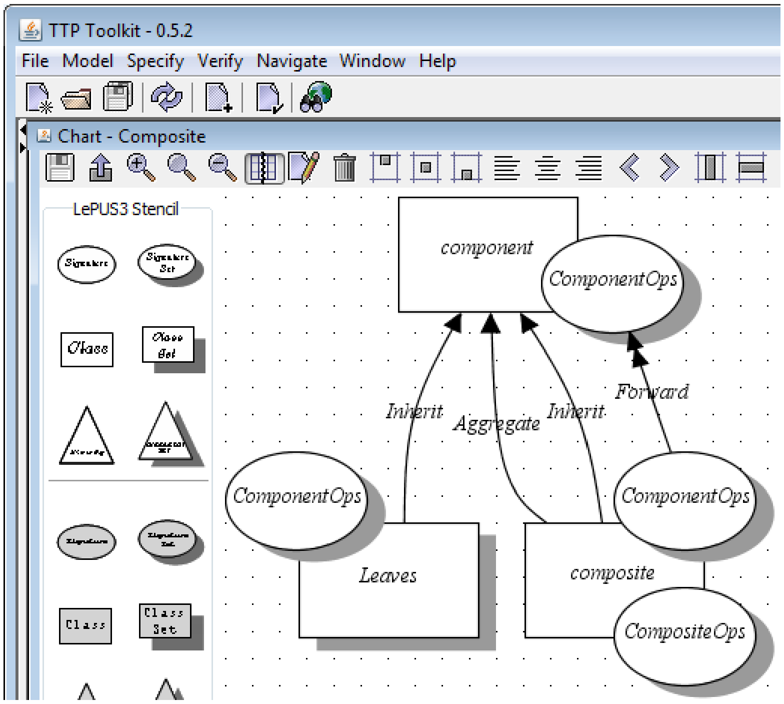


Chart 1: The Composite design pattern specified in LePUS3 (chart Composite) using the TTP Toolkit

LePUS3 was tailored specifically for this purpose, while keeping focus on automated verification. For example chart Composite (Chart 1) captures much of the informal description of the Composite design pattern (Table 1). A LePUS3 chart consists of a set of *well-formed formulas*, each of which is composed of a well-formed combination of visual tokens (Figure 1). Each formula consists of *terms*, which stand for [sets of] classes or [sets of] methods, a *relation* and possibly a *predicate symbol*, which represent correlations between the entities being modelled. The meaning of chart Composite is captured by the truth conditions spelled out in Table 2.

Given the formal specification of the pattern and the truth conditions of satisfying this specification, we may now rephrase our informal hypothesis in slightly more rigorous fashion as demonstrated in Hypothesis 2:

Hypothesis 2. `java.awt` ‘implements’ chart Composite

4 Abstract Semantics

When speaking of a ‘program’ or an ‘implementation’ we usually refer to the source code, normally represented by a complex collection of text files describing myriad implementation minutiae distributed

Terms
(a) *composite* and *component* are variables ranging over individual types (in Java: class, interface, or primitive type)
(b) *Leaves* is a variable that ranges over sets of types
(c) *CompositeOps* and *ComponentOps* are variables ranging over sets of method signatures
Formulas
(d) *composite* must have an 'aggregate' (an array or a Java collection) of instances of type *component* (or of subtypes thereof)
(e) *composite* must 'inherit' (in Java: `extend` or `implement`) (possibly indirectly) from class *component*
(f) Every class in *Leaves* must 'inherit' (possibly indirectly) from class *component*
(g) *composite* must define (or inherit) a method for each of the signatures in the set *CompositeOps*
(h) Every class in *Leaves* must define (or inherit) a method for each of the signatures in the set *ComponentOps*
(i) Each method defined in (or inherited by) *composite*, with a signature in *ComponentOps*, must at some point forward the method call (invocation) to that (unique) method with same signature that is a member of (or inherited by) *component*, and vice versa

Table 2: Truth conditions for chart Composite

across a directory structure. Source code is a difficult medium to reason about, not the least so because in practical circumstances it normally contains thousands (or millions) of lines of code. For example, the unabbreviated source code of only four classes from `java.awt` spans over ten thousand lines. Reasoning therefore requires a simplified picture of the program, hence the motivation for introducing the notion of *abstract semantics*.

In this context, an abstract semantics is a finite structure in model theory, which is simply a set of atomic entities and relations between them (implemented as a set of tables in a relational database). Specifically, a *finite structure* $\mathfrak{F}$ [11][19] is a pair $\mathfrak{F} = \langle \mathbb{U}, \mathbb{R} \rangle$ where $\mathbb{U}$ (the 'universe' of $\mathfrak{F}$) is the (finite) set of all (atomic) entities, and $\mathbb{R}$ is the (finite) set of relations between them, e.g.:

$$\mathbb{U} = \{\ \underline{\texttt{Container}}\ , \ldots,\ \underline{\texttt{addNotify()}}\ , \ldots\} \tag{1}$$

$$\mathbb{R} = \{\ \underline{\mathit{Class}}\ ,\ \underline{\mathit{Method}}\ ,\ \underline{\mathit{Signature}}\ ,\ \underline{\mathit{Inherit}}\ ,\ \underline{\mathit{Aggregate}}\ , \ldots\} \tag{2}$$

Note that to maintain their distinction, items in the model are underlined. Each atomic entity in the universe $\mathbb{U}$ is a class (an element of the unary relation $\underline{\mathit{Class}}$), a method, (an element of $\underline{\mathit{Method}}$) or a method signature (an element of $\underline{\mathit{Signature}}$, which identifies method name and argument types). In other words, $\mathbb{U}$ is the (disjoint) union of the unary relations $\underline{\mathit{Class}}$, $\underline{\mathit{Method}}$ and $\underline{\mathit{Signature}}$. Each unary (binary) relation in $\mathbb{R}$ is a finite set of 1-tuples (2-tuples) of atomic entities. For example, the unary relation $\underline{\mathit{Class}}$ is a set of 1-tuples, one for each of the four classes in `java.awt`. The binary relation $\underline{\mathit{Inherit}}$ is a set which contains all the pairs of types $\langle \mathit{cls}, \mathit{supercls} \rangle$ in `java.awt` such that *cls* extends/implements/is-subtype-of *supercls*. Likewise, the binary relation $\underline{\mathit{Aggregate}}$ contains pairs of classes $\langle \mathit{cls}, \mathit{elementType} \rangle$ such that *cls* contains a collection (or array) of instances of the class *elementType* (or subtypes thereof). The binary relation $\underline{\mathit{Forward}}$ represents a special kind of method call between two methods, $\langle \mathit{invoker}, \mathit{invoked} \rangle$, that share the same signature.

The precise relation between a program and its abstract semantics is formally captured using the abstract semantics function: a mapping from programs (expressions in the programming language) into finite structures. For example, $\mathcal{A}_{\mathit{Java}1.4}$ [9] is an abstract semantics function which represents the mapping from each Java 1.4 program to a finite structure.

$$\mathcal{A}_{\mathit{Java}1.4} : \mathbb{JAVA}1.4 \longmapsto \mathfrak{F}^* \tag{3}$$

where $\mathbb{JAVA}1.4$ stands for the set of all well-formed Java 1.4 programs and $\mathfrak{F}^*$ stands for the (enumerable) set of all possible finite structures.

Abstract semantics functions allow us to determine exactly how the source code of programs can be abstracted. For example, we may use $\mathcal{A}_{Java1.4}$ to define the finite structure for `java.awt` as follows:

$$\mathcal{A}_{Java1.4}(\texttt{java.awt}) \tag{4}$$

We require that abstract semantics functions are fully Turing-decidable such that they always terminate; in practical terms this means that $\mathcal{A}_{Java1.4}$ can, in principle, be implemented as a static analyzer. Such an analyzer is implemented in the Two-Tier Programming Toolkit (§6), a tool which parses any Java 1.4 program and generates a representation of a finite structure therefrom (a relational database) [9]. However, static analysis is not without its limitations, and as such we do not currently capture much of the behavioural aspects of programs, for example temporal information and program state.

Alternatively, other abstract semantics functions can be equally used to represent programs in any class-based object-oriented programming language (e.g. C++, C#, Smalltalk). For example, if an abstract semantics function is defined for the C++ programming language: $\mathcal{A}_{C++} : \mathbb{C}++ \longmapsto \mathfrak{F}^*$, the very same specification and verification mechanisms described in this paper can be applied to programs written in C++.

The notion of abstract semantics allows us to articulate informal claims concerning the relationship between a design pattern and a program precisely as a mathematical proposition. Specifically, we stipulate that a program p implements a design pattern if and only if the abstract semantics of p (a finite structure) *satisfies* that LePUS3 chart which specifies that pattern. Hypothesis 2 can thus be redefined in these terms as follows:

Hypothesis 3. $\mathcal{A}_{Java1.4}(\texttt{java.awt})$ *satisfies* Composite

In the following section we define this 'stisfies' relation, and recast Hypothesis 3 as a mathematical proposition.

5 Verification

By verification we refer to the rigorous, conclusive, and decidable process of establishing or refuting whether a particular program is consistent with a given LePUS3 specification (chart). An automated process of verification therefore consists of executing an algorithm which determines if a program p (modelled using the notion of abstract semantics) satisfies a LePUS3 chart Ψ.

The conditions for 'satisfying' Ψ are modelled after the standard Tarski's truth conditions for the classical logic, as demonstrated in Table 2. A satisfies proposition is represented using the standard semantic entailment symbol $\models$, allowing us to recast Hypothesis 3 as the following (decidable) proposition:

Hypothesis 4. $\mathcal{A}_{Java1.4}(\texttt{java.awt}) \models$ Composite

Charts modelling design motifs such as Composite contain variable terms. To show that such a chart is satisfied in the context of a specific program its variables must first be mapped to entities in the appropriate finite structure. Such a mapping is commonly referred to as an assignment. Formally the semantic entailment in Hypothesis 4 holds if and only if there exists an assignment that maps each variable in Composite to specific elements of a given program, in this case `java.awt`. Such an assignment is presented in Table 3, where the search for assignments is a matter of *pattern detection* and therefore beyond the scope of this paper.

Hypothesis 4 can now be recast as a proposition such that the abstract semantics of `java.awt` satisfy the Composite chart under assignment g, a claim represented in Hypothesis 5 using the standard notation.

Hypothesis 5. $\mathcal{A}_{Java1.4}(\texttt{java.awt}) \models_g$ Composite

$g(\mathit{composite})$	$=$	$\underline{\texttt{Container}}$
$g(\mathit{component})$	$=$	$\underline{\texttt{Component}}$
$g(\mathit{Leaves})$	$=$	$\{\,\underline{\texttt{Button}}\,,\underline{\texttt{Canvas}}\,\}$
$g(\mathit{ComponentOps})$	$=$	$\{\,\underline{\texttt{addNotify()}}\,,\underline{\texttt{removeNofity()}}\,,\underline{\texttt{paramString()}}\,\}$
$g(\mathit{CompositeOps})$	$=$	$\{\,\underline{\texttt{getComponents()}}\,,\underline{\texttt{getComponent(int)}}\,\}$

Table 3: Assignment g mapping variables in Composite to entities in `java.awt`

The proposition in Hypothesis 5 imposes specific conditions on the existence of specific entities and sets of entities in `java.awt` and on specific correlations amongst them. To prove it we refer back to Table 2, which constitutes two kinds of conditions:

1. **Truth conditions for terms.** For example, class *composite* is satisfied by virtue of assignment g, and the 1-tuple $\langle\,\underline{\texttt{Container}}\,\rangle$ in relation $\underline{\mathit{Class}}$.
2. **Truth conditions for formulas.** For example, the Inherit relation between class *composite* and class *component* is satisfied by virtue of g, and the pair $\langle\,\underline{\texttt{Container}}\,,\underline{\texttt{Component}}\,\rangle$ in the relation $\underline{\mathit{Inherit}}$.

Table 4 is demonstrates the proof for Hypothesis 5, the precise elements of $\mathcal{A}_{Java1.4}(\texttt{java.awt})$ which satisfy the truth conditions of Chart Composite (Table 2).

$\ldots,\langle\,\underline{\texttt{Container}}\,\rangle,\langle\,\underline{\texttt{Component}}\,\rangle \in \underline{\mathit{Class}}$	$\models$	(a)
$\ldots,\langle\,\underline{\texttt{Container}}\,,\underline{\texttt{Component}}\,\rangle \in \underline{\mathit{Aggregate}}$	$\models$	(d)
$\ldots,\langle\,\underline{\texttt{Container}}\,,\underline{\texttt{Component}}\,\rangle \in \underline{\mathit{Inherit}}$	$\models$	(e)
$\ldots,\langle\,\underline{\texttt{Container.getComponents()}}\,\rangle \in \underline{\mathit{Method}}$		
$\ldots,\langle\,\underline{\texttt{getComponents()}}\,,\underline{\texttt{Container.getComponents()}}\,\rangle \in \underline{\mathit{SignatureOf}}$		
$\ldots,\langle\,\underline{\texttt{Container}}\,,\underline{\texttt{Container.getComponents()}}\,\rangle \in \underline{\mathit{Member}}$	$\models$	(g)
$\ldots,\langle\,\underline{\texttt{Container.addNotify()}}\,,\underline{\texttt{Component.addNotify()}}\,\rangle \in \underline{\mathit{Forward}}$	$\models$	(i)
		$\ldots$

Table 4: Proof of Hypothesis 5 (abbreviated)

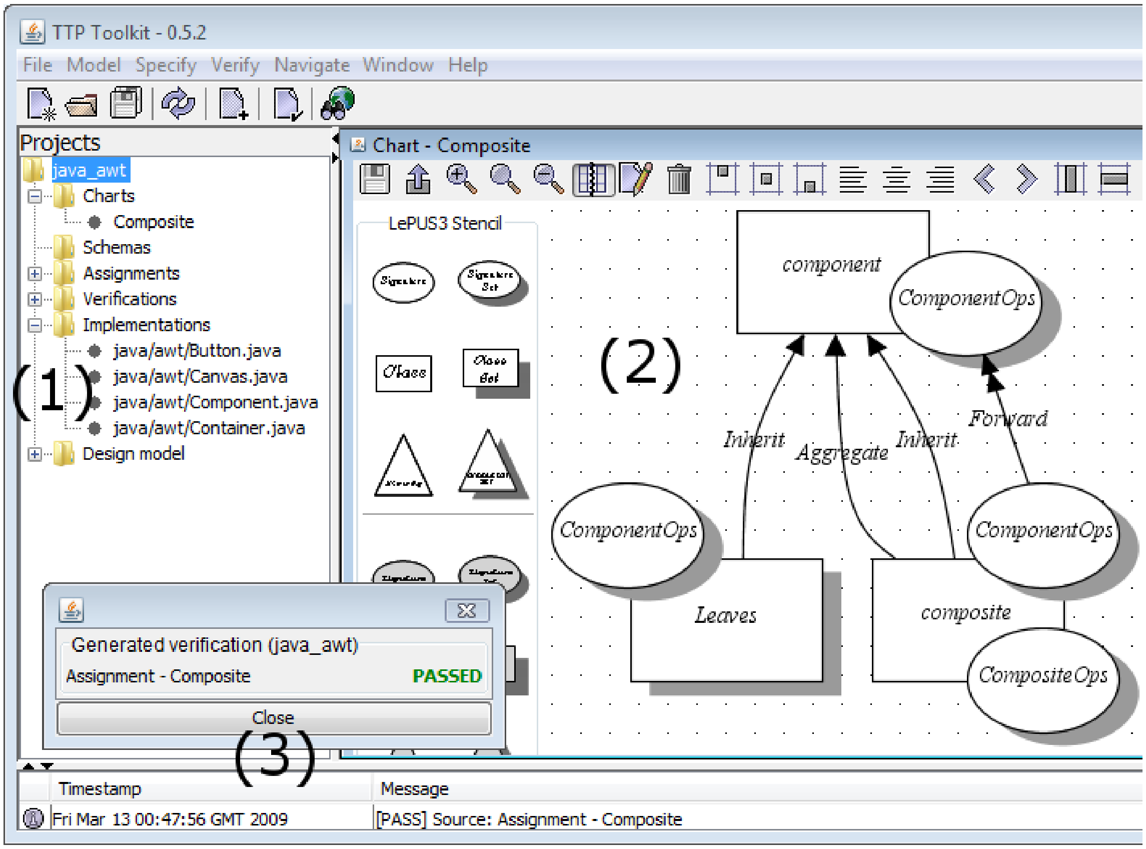
From this proof we conclude that `java.awt` indeed satisfies the Composite pattern. While the notion of verification as demonstrated is straightforward, manually producing the proof is a tedious, error-prone process. Such proofs require the software designer to check the validity of hundreds and thousands of clauses. It also requires intimate knowledge of both the abstract semantics of the implementation and of the specification language. Worse, the proof process would have to be repeated each time the implementation, or the design, change. However, verifying LePUS3 charts need not be a Herculean manual task as it can be fully automated, as described in the next section.

6 Tool Support

The Two-Tier Programming project [20] has recently completed implementing version 0.5.2 of the Two-Tier Programming (TTP) Toolkit. The TTP Toolkit is a prototype that integrates the representation of programs in two layers of abstraction: the design—a set of LePUS3 charts, and the implementation—a set of standard Java 1.4 source code files.

Our demonstration focuses on Figure 2, and begins with choosing the implementation (point 1); the TTP Toolkit supports the selection and static analysis (generating finite structures) of Java 1.4

source code, which in this case is the four .java files from `java.awt` (`Button.java`, `Canvas.java`, `Component.java` and `Container.java`). The TTP Toolkit also supports the composition and editing of specifications (LePUS3 charts), such as Chart Composite (point 2). Finally, the TTP Toolkit fully automates verification of programs against charts at the click of a button, such as the proof discussed in this paper. This is depicted by a window stating that verification was successful (point 3), as indicated by the text 'PASSED', as well as by a status message in the console.



Generated verification (java_awt)

Assignment - Composite PASSED

```
Fri Mar 13 00:47:56 GMT 2009   [PASS] Source: Assignment - Composite
```

Figure 2: Source files (1), chart Composite (2), and the verification result (3) in the TTP Toolkit

Additionally, it is extremely important that when verification fails it generates a simple explanation so that the inconsistency between specification and implementation can be rectified. For example, consider reversing the forwarding relation in chart Composite, a change which will fail verification against `java.awt`. The TTP Toolkit symbolically reports this failure to the user (Table 5), clearly indicating where the problem originates in chart Composite.

java.awt.Component.removeNotify() does not forward to any of the entities in: { java.awt.Container.removeNotify() , java.awt.Container.addNotify() }

Table 5: Summary of the explanation provided by the TTP Toolkit on verification failure

To summarize, the TTP Toolkit supports the following tasks:

- **Specifying.** The TTP Toolkit can be used to create, edit and view LePUS3 charts.
- **Analysing.** The TTP Toolkit statically analyses any (arbitrarily-large) well-formed Java 1.4 program and generates a relational database representing its abstract semantics, defined by $\mathcal{A}_{Java1.4}$.
- **Verifying.** The TTP Toolkit can conclusively and efficiently determine whether a given implementation satisfies a LePUS3 chart by a click of a button, and within reasonable time.

The TTP Toolkit has also been used to model, specify and verify other cases than that presented here[20]. For example it has been used to prove that `java.io` package is not consistent with the Decorator design pattern [12][17], while adding evidence to the claim that said package is consistent with a variation of the pattern [15].

Indeed, specification need not necessarily precede verification; verification being just one of many tools that aid in the development and understanding of programs. Specifically the TTP Toolkit also supports reverse engineering and program visualization. The Design Navigator [7][21] is a design recovery tool that allows a user to navigate through the design of Java 1.4 programs by reverse-engineering LePUS3 charts therefrom. To do so, the Design Navigator creates the abstract semantic representation of programs, uses the verification engine to detect correlations between [sets of] classes and methods, and represents them at the appropriate level of abstraction.

6.1 Empirical validation

We believe that TTP Toolkit can dramatically increase the productivity of software engineers. To test this claim we have designed and started conducting an experiment that compares the TTP Toolkit against a standard commercial integrated development environment. The experiment measures the performance of (mostly postgraduate) students in carrying out a variety of tasks under controlled conditions, designed to test the following specific claims:

- **Comprehension:** Effort required to understand the design and structure of arbitrarily-large programs, measured in time, is significantly reduced;
- **Conformance:** Overall dependability of programs, measured in terms of conformance of the implementation to the design specifications, can be significantly improved;
- **Evolution:** The cost of software maintenance and/or re-engineering, measured in terms of time, can be significantly reduced.

Preliminary results suggest that the TTP Toolkit radically decreases the length of time required to carry out software engineering tasks.

7 Summary

We presented LePUS3, an object-oriented Design Description Language, and demonstrated how LePUS3 can be used to specify (model) design patterns. We re-formulated an informal hypothesis, which claimed that the Composite design pattern is implemented by the `java.awt` package, as a mathematical proposition and sketched its proof. We also described the Two-Tier Programming Toolkit, a tool which can be used to compose object-oriented design specifications in LePUS3, statically analyse Java 1.4 programs, and verify them to establish whether they are consistent with the design specifications. Finally, we discussed an experiment designed to test our claims concerning the Two-Tier Programming Toolkit.

Acknowledgments. This work was partially funded by the UK's Engineering and Physical Research Council, and the University of Essex's Research Promotional Fund. The authors wish to thank Raymond Turner for his numerous contributions to this project.

References

[1] Frederick P. Brooks. No silver bullet: Essence and accidents of software engineering. *IEEE Computer magazine*, 20(4):10–19, April 1987.

[2] W. Wayt Gibbs. Software's chronic crisis. *Scientific American*, 271(3):72–81, 1994.

[3] M. M. Lehman. Laws of software evolution revisited. In *Proceedings of the 5th European Workshop on Software Process Technology*, pages 108–124, Nancy, France, October 1996. Springer-Verlag.

[4] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes*, 17(4):40–52, 1992.

[5] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Trans. Softw. Eng.*, 26(1):70–93, 2000.

[6] Toufik Taibi. *Design Patterns Formalization Techniques*. IGI Global, Hershey, USA, March 2007.

[7] Epameinondas Gasparis, Jonathan Nicholson, and Amnon H. Eden. LePUS3: an Object-Oriented design description language. In *5th Intl. Conf. on the Theory and Application of Diagrams*, Herrsching, Germany, September 2008.

[8] Raymond Turner. The foundations of specification. *J Logic Computation*, 15(5):623–662, October 2005.

[9] Jonathan Nicholson, Amnon H Eden, and Epameinondas Gasparis. Verification of LePUS3/Class-Z specifications: Sample models and abstract semantics for java 1.4. Tech. Rep. CSM-471, ISSN 1744-8050, School of Computer Science and Electronic Engineering, University of Essex, December 2007. `http://lepus.org.uk/ref/verif/verif.pdf`.

[10] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java Language Specification*. Addison-Wesley Professional, 3rd edition edition, 2005.

[11] Amnon H Eden, Epameinondas Gasparis, and Jonathan Nicholson. LePUS3 and Class-Z reference manual. Tech. Rep. CSM-474, ISSN 1744-8050, School of Computer Science and Electronic Engineering, University of Essex, December 2007. `http://lepus.org.uk/ref/refman/refman.pdf`.

[12] Amnon H Eden, Epameinondas Gasparis, and Jonathan Nicholson. The 'Gang of four' companion: Formal specification of design patterns in LePUS3 and Class-Z. Tech. Rep. CSM-472, ISSN 1744-8050, School of Computer Science and Electronic Engineering, University of Essex, December 2007. `http://lepus.org.uk/ref/companion/companion.pdf`.

[13] Jing Dong and Yajing Zhao. Experiments on design pattern discovery. In *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, page 12. IEEE Computer Society, 2007.

[14] Jochen Seemann and Jürgen Wolff von Gudenberg. Pattern-based design recovery of java software. In *Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 10–16, Lake Buena Vista, Florida, United States, 1998. ACM.

[15] Stephen A. Stelting and Olav Maassen. *Applied Java Patterns*. Prentice Hall, 2002.

[16] Sun Microsystems. *Java 2 SDK, Standard Edition Documentation, version 1.4.2*. Sun Microsystems, 2004.

[17] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, November 1994.

[18] Object Management Group. UML 2.0 superstructure specification. Technical report, August 2005. `http://www.omg.org/cgi-bin/apps/doc?formal/05-07-04.pdf`.

[19] Michael R. A. Huth and Mark D. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, Cambridge, England, 2000.

[20] Jonathan Nicholson, Epameinondas Gasparis, and Amnon H Eden. The Two-Tier programming project, 2008. `http://ttp.essex.ac.uk/`.

[21] Epameinondas Gasparis, Amnon H. Eden, Jonathan Nicholson, and Rick Kazman. The design navigator: Charting java programs. In *Companion of the 30th international conference on Software engineering*, pages 945–946, Leipzig, Germany, May 2008. ACM.

A Module Language for Typing by Contracts

Yann Glouche* Jean-Pierre Talpin Paul Le Guernic Thierry Gautier
INRIA, Unité de Recherche Rennes-Bretagne-Atlantique, Campus de Beaulieu, Rennes, France

Abstract

Assume-guarantee reasoning is a popular and expressive paradigm for modular and compositional specification of programs. It is becoming a fundamental concept in some computer-aided design tools for embedded system design. In this paper, we elaborate foundations for contract-based embedded system design by proposing a general-purpose module language based on a Boolean algebra allowing to define contracts. In this framework, contracts are used to negociate the correctness of assumptions made on the definition of a component at the point where it is used and provides guarantees to its environment. We illustrate this presentation with the specification of a simplified 4-stroke engine model.

1 Introduction

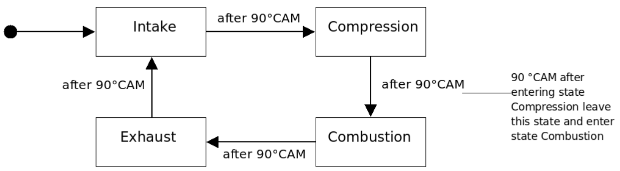
Methodological common sense for the design of large embedded architectures advises the validation of specifications as early as possible, and further advocates for an iterative validation of each refinement or modification made to any component of the initial specification, until the implementation of the system is finalized. As an example, in a cooperative industrial environment, designers often use and assemble components which have been developed by different suppliers. These components have to be provided with conditions of use and need to offer pre-validated guarantees on their function or service. The conditions of use and the service guarantee define a notion of *contract*. Design by contract, as advocated in [23], is now a common programming concept used in general-purpose languages such as C++ or Java. Assertion-based contracts express program invariants, pre and post conditions, as Boolean type expressions that have to be true for the contract being validated. We adopt the paradigm of *contract* to define a component-based validation process in the context of an embedded software modeling framework. It consists of an algebraic framework, based on two simple concepts, enabling logical reasoning on contracts [11]. First, the assumptions and guarantees of a component are defined as devices called filters: assumptions filter the behaviors a component accepts and guarantees filter the behaviors a component provides. Filters form a Boolean algebra and contracts define a Heyting algebra. This yields a rich framework to abstract, refine, combine and normalize contracts. In this paper, we put this algebra to work for the definition of a general purpose module system whose typing paradigm is based on the notion of contract. The type of a module is a contract holding assumptions made and guarantees offered by its behaviors. It allows to associate a module with an interface which can be used in varieties of scenarios such as checking the composability of modules or efficiently supporting modular compilation.

Plan We start with a highlight on some key features of our module system by considering the specification of a simplified engine controler, Section 2. This example is used through the article to illustrate our approach. We give a short outline of our contract algebra, Section 3, and demonstrate its capabilities for logical and compositional reasoning on the assumptions and guarantees of component-based embedded systems. Our main contribution, section 4, is to use this algebra as a foundation for the definition of a strongly-typed module system: contracts are used to type components with behavioral properties. Section 5 demonstrates the use of our module system by considering the introductory example and by illustrating the refinement mechanism of the language. We review related works, Section 6, before concluding, Section 7.

*Partially funded by the EADS Foundation.

2 An example

We illustrate our approach by considering a simplified automotive application presented in [4]. We define contracts to characterize properties of a four-stroke engine controller. In this specification, the cyclic behavior of the engine is rendered by four successive phases: *Intake*, *Combustion*, *Compression*, and *Exhaust*. These phases are driven by a camshaft whose angle is measured in degrees. This unit of measure defines a discrete and symbolic time sampling in the system. The angle of the camshaft marks an occurrence of a clock tick (the *cam*) at which sensors are sampled and a reaction is triggered. For example, at 90°, the intake valve is closed and a transition to compressin mode is triggered.


Intake
after 90°CAM
Compression
after 90°CAM
after 90°CAM
90 °CAM after entering state Compression leave this state and enter state Combustion
Exhaust
Combustion
after 90°CAM

In the module language, a specification is designated by the keyword **contract**. It defines a set of input and output variables subject to a contract. The interface defines the way the environment and the component interact through its variables. In addition, it embeds properties that are modeled by the composition of contracts. For instance, the specification of the intake mode of the engine controller could be defined by the assumption `0 <= (cam mod 360)< 90` and the guarantee `intake`. An implementation of the interface, designated by the keyword **process**, contains a compatible implementation of the contract assigning true to the signal `intake` when `0 <= (cam mod 360)< 90`.

```
module type IntakeType =              Module IntakeMode : IntakeType =
   contract                              process
     input integer cam;                    input integer cam;
     output event intake;                  output event intake;
     assume 0 <= (cam mod 360)< 90         intake := true when cam mod 360<90
     guarantee intake                    end;
   end;
```

The specification of the properties we consider for the engine consists of four contracts. Each contract specifies a phase of the engine. It is associated to a position of the camshaft and triggers an event. Instead of specifying four separate contracts, we define them as four instances of a generic contract. To this end, we define an encapsulation mechanism to generically represent a class of interfaces or implementations sharing a common pattern of behavior up to that of the parameter. In the example of the engine, for instance, we have used a functor to parameterize it with respect to its angle position.

```
module type phase =                     module type engine = contract
  functor (integer min, integer max)    input integer cam ; output event intake,
    contract                                    compression, combustion, exhaust;
      input integer cam ;               phase (0, 90) (cam, intake)
      output event trigger;             and phase (90, 180) (cam, compression)
      assume min <= (cam mod 360) < max and phase (180, 270) (cam, combustion)
      guarantee trigger                 and phase (270, 360) (cam, exhaust)
    end;                                end;
```

The generic contract, called `phase`, is parameterized with a trigger and with camshaft angle bounds. When the camshaft is within the specified angles, the engine controller activates the specified trigger. The contract of the engine is defined by the composition "**and**" of four applications of the generic phase contract. Each application defines a particular phase of the engine with its appropriate triggers and angles. The composition defines the greatest lower-bound of all four contracts. Each application of the phase functor produces a contract which is composed with the other ones in order to produce the expected contract. A component is hence viewed as a pair $M : I$ consisting of an implementation M that is typed by (or viewed as) an interface I of satisfiable contract. The semantics of the specification I, written ⟦I⟧, is a set of processes (in the sense of section 3) whose traces satisfy the contracts associated with I. The semantics ⟦M⟧ of the module M is the singleton containing the process ⟦IntakeMode⟧.

3 An algebra of contracts

A contract is viewed as a pair of logical devices that filter processes: the assumption **A** filters processes to select (accept or conversely reject) those of which that are to be asserted (accepted or conversely rejected) by the guarantee **G**. A process p fulfils (**A**,**G**) requirements (satisfies (**A**,**G**)) if either it is rejected by **A** (it is then out of the scope of (**A**,**G**) contract), or it is accepted by **A** and by **G**. We have chosen to represent a filtering device (such as **A** or **G**) by a *process-filter* which is the set of processes that it "accepts" (i.e., that it contains). We give more details of the algebra of contracts, including examples, additional properties and formal proofs in a technical report [11]. In the remainder, we only define the mathematical objects that are relevant to the presentation of our module system. They consist of two relations in the contract algebra, noted $\preccurlyeq$ for refinement and $\Downarrow$ for composition. In fact, our module system can be seen as a domain-specific language that syntactically reflects the structure of our contract algebra.

For $\mathbf{X}$ a nonempty finite set of variables, a behavior is a function $b : \mathbf{X} \rightarrow \mathscr{D}$ where $\mathscr{D}$ is a set of values (that may be traces), and a *process p* is a nonempty set of behaviors defined on $\mathbf{X}$. We denote by $\Omega = \{\emptyset\}$ the unique process defined on the empty set of variables: it has a single behavior that is the empty behavior. The set of processes is noted $\mathbb{P}$; it is extented to $\mathbb{P}^\star$ by adding the *empty "process"* noted $\mho = \emptyset$. A *process-filter* $\mathbf{R}$ is the set of processes that satisfy a given property on a set of variables $\mathbf{X}$; notice that if a process p is in $\mathbf{R}$ so are all processes defined on supersets of $\mathbf{X}$, whose sub-behaviors restricted to $\mathbf{X}$ are behaviors in p; moreover if a process p is in $\mathbf{R}$, so are all nonempty subsets of p. By convention, $\{\mho\}$ is a process-filter. We define an order relation $\sqsubseteq$ on the set of process-filters Φ extended by $\{\mho\}$ and establish that $(\Phi,\sqsubseteq)$ is a lattice. Also, $(\Phi,\sqsubseteq)$ is a Boolean algebra with $\mathbb{P}^\star$ as *1*, $\{\mho\}$ as *0*. The complementary of $\mathbf{R}$ is noted $\widetilde{\mathbf{R}}$. The conjunction $\mathbf{R} \sqcap \mathbf{S}$ of two process-filters $\mathbf{R}$ and $\mathbf{S}$ is the greatest process-filter $\mathbf{T} = \mathbf{R} \sqcap \mathbf{S}$ that accepts all processes whose behaviors are at once behaviors in some process accepted by $\mathbf{R}$ and behaviors in some process accepted by $\mathbf{S}$. The process-filter disjunction $\mathbf{R} \sqcup \mathbf{S}$ of two process-filters $\mathbf{R}$ and $\mathbf{S}$ is the smallest process-filter $\mathbf{T} = \mathbf{R} \sqcup \mathbf{S}$ that accepts all processes whose behaviors are at once behaviors in some process accepted by $\mathbf{R}$ or behaviors in some process accepted by $\mathbf{S}$. A *contract* $\mathbf{C} = (\mathbf{A},\mathbf{G})$ is a pair of process-filters and $\mathbb{C} = \Phi\times\Phi$ is the set of contracts. The refinement relation $(\preccurlyeq)$ is a partial order on contracts. $(\mathbb{C}, \preccurlyeq)$ is a distributive lattice of supremum $(\{\mho\},\mathbb{P}^\star)$ and infimum $(\mathbb{P}^\star,\{\mho\})$: it is a Heyting algebra. We define the nominal contract $p_{\succcurlyeq}$ of a process p as the contract that assumes any behavior and guarantees all the possible behaviors of p. Two contracts $\mathbf{C}_1$ and $\mathbf{C}_2$ have a greatest lower bound $\mathbf{C}_1 \Downarrow \mathbf{C}_2$ and a least upper bound $\mathbf{C}_1 \Uparrow \mathbf{C}_2$. The greatest lower bound $\mathbf{C} = (\mathbf{A},\mathbf{G})$ of two contracts $\mathbf{C}_1 = (\mathbf{A}_1,\mathbf{G}_1)$ and $\mathbf{C}_2 = (\mathbf{A}_2,\mathbf{G}_2)$ is defined by: $\mathbf{A} = \mathbf{A}_1 \sqcup \mathbf{A}_2$ and $\mathbf{G} = ((\mathbf{A}_1 \sqcap \widetilde{\mathbf{A}_2} \sqcap \mathbf{G}_1) \sqcup (\widetilde{\mathbf{A}_1} \sqcap \mathbf{A}_2 \sqcap \mathbf{G}_2) \sqcup (\mathbf{G}_1 \sqcap \mathbf{G}_2))$.

4 A module language for typing by contracts

In this section, we define a *module language* to implement our contract algebra and apply it to the validation of component-based systems. For the peculiarity of our applications, it is instantiated to the context of the synchronous language Signal, yet could equally be used in the context of related programming languages manipulating processes or agents. Its basic principle is to separate the interface, which declares properties of a program using contracts, and implementation, which defines an executable specification satisfying it.

4.1 Syntax

We define the formal syntax of our module language. Its grammar is parameterized by the syntax of programs, noted p or q, which belong to the target specification or programming language. Names are noted x or y. Types t are used to declare parameters and variables in the interface of contracts.

Assumptions and guarantees are described by expressions p and q of the target language. An expression exp manipulates contracts and modules to parameterize, apply, reference and compose them.

$$\begin{array}{lll@{\quad}ll}
x,y & & \text{name} & ag ::= [\textbf{assume}\ p]\ \textbf{guarantee}\ q; & \text{contract} \\
p,q & & \text{process} & \mid\ ag\ \textbf{and}\ ag \mid x(y^*) & \text{process} \\
b,c ::= & \textbf{event} \mid \textbf{boolean} \mid \textbf{short} \mid \textbf{integer} \mid \ldots & \text{datatype} & exp ::= \textbf{contract}\ dec;\ ag\ \textbf{end} & \text{contract} \\
t ::= & b \mid \textbf{input}\,b \mid \textbf{output}\ b \mid x \mid t \times t & \text{type} & \mid\ \textbf{process}\ dec;\ p\ \textbf{end} & \text{process} \\
dec ::= & t\ x\ [,dec] & \text{declaration} & \mid\ \textbf{functor}\ (dec)\ exp & \text{functor} \\
def ::= & \textbf{module}\,[\textbf{type}]\,x\ =\ exp & \text{definition} & \mid\ exp\ \textbf{and}\ exp & \text{composition} \\
\mid & \textbf{module}\,x\ [:t]\ =\ exp & & \mid\ x\ (exp^*) & \text{application} \\
\mid & def;def & & \mid\ \textbf{let}\ def\ \textbf{in}\,exp & \text{scoping}
\end{array}$$

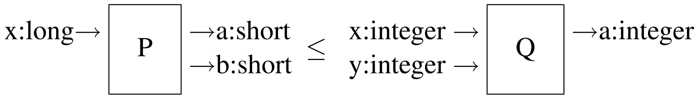
4.2 A type system for contracts and processes

We define a type system for contracts and processes in the module language. In the syntax of the module language, contracts and processes are associated with names x . These names can be used to type formal parameters in a functor and become type declarations. Hence, in the type system, type names that stand for a contract or a process are associated with a module type T. A base module type is a tagged pair $\tau(I,C)$. The tag τ is π for the type of a process and γ for the type of a contract. The set I consists of pairs $x : t$ that declare the types t for its input and output variables x. The contract C is a pair of predicates (p,q) that represent its assumptions p and guarantees q. The type of a functor $\Lambda(x : S).T$ consists of the name x and of the types S and T of its formal parameter and result. The role of the typing hypothesis Γ is to hold an association $x : T$ of names to types (we write $\Gamma(x)$ the type of name x in Γ). The role of the typing constraints Σ is to register inferred refinement relations of the form $C \preceq D$.

$$S,T ::= t \mid \tau(I,C) \mid S \times T \mid \Lambda(x:S).T\ \text{(type)} \quad \tau ::= \gamma \mid \pi\ \text{(kind)} \quad \Gamma ::= \emptyset \mid \Gamma \cup x:T \quad \Sigma ::= \emptyset \mid \Sigma \cup C \preceq D\ \text{(context)}$$

4.3 Contract refinement

We wish to define a subtyping relation on types t to extend the refinement relation of the contract algebra to the type algebra. In that aim, we wish to apply the subtyping principle $S \leq T$ to mean that the semantic objects denoted by S are contained in the semantic objects denoted by T (S refines T). Hence, a module of type T can safely be replaced or substituted by a module of type S. For example, consider a process P with one long input x and two short outputs a,b, and a process Q with two integer inputs x,y and one integer output, such that P refines Q.Then the type of a module M encapuslating P is a subtype of a module N encapsulating Q. M can replace N.



4.4 Subtyping as refinement

Accordingly, we say that s is a subtype of t under the hypothesis Σ, written $\Sigma \supset s \leq t$ iff Σ contains (or implies) the relation $s \leq t$. The inductive definition of the subtyping relation $\leq$ starts with the subtyping axioms for datatypes. Then, it is elaborated with algebraic rules, the rules for declarations and, finally, one rule for each kind of module type. In particular, a module type $S = \tau(I,C)$ is a subtype of $T = \tau(J,D)$, written $\Sigma \supset S \leq T$, iff the inputs in J subtype those in I, if the outputs in I subtype those in J, and if the contract C refines D. In the rule for functors, we write $V[y/x]$ for substituting the name x by y in V (we assume that y does not occur in V).

$$T \leq T \quad \textbf{event} \leq \textbf{boolean} \quad \textbf{short} \leq \textbf{integer} \leq \textbf{long} \qquad \frac{\Sigma \supset I \leq J \quad x \notin vars(I)}{\Sigma \supset I \leq (x : \textbf{input}\, b) \cup J} \quad \frac{\Sigma \supset I \leq J \quad x \notin vars(J)}{\Sigma \supset (x : \textbf{output}\, b) \cup I \leq J}$$

$$\frac{\Sigma \supset b \leq c}{\Sigma \supset \textbf{input}\, c \leq \textbf{input}\, b} \quad \frac{\Sigma \supset b \leq c}{\Sigma \supset \textbf{output}\, b \leq \textbf{output}\, c} \qquad \begin{array}{c}\tau \leq \gamma \\ \pi \leq \tau\end{array} \quad \frac{\Sigma \supset I \leq J \quad (C \preceq D) \in \Sigma \quad \tau \leq \tau'}{\Sigma \supset \tau(I,C) \leq \tau'(J,D)}$$

$$\frac{\Sigma \supset S \leq T \quad \Sigma \supset T \leq U}{\Sigma \supset S \leq U} \quad \frac{\Sigma \supset I \leq J \quad \Sigma \supset S \leq T}{\Sigma \supset (x : S) \cup I \leq (x : T) \cup J} \quad \frac{\Sigma \supset S \leq U \quad T \leq V}{\Sigma \supset S \times T \leq U \times V} \quad \frac{\Sigma \supset U \leq S \quad \Sigma \supset T \leq V[x/y]}{\Sigma \supset \Lambda(x : S).T \leq \Lambda(y : U).V}$$

Alternatively, we can interpret the relation $\Sigma \supset C \preceq D$ as a mean to register the refinement constraint between C and D in Σ. It corresponds to a proof obligation in the target language whose meaning is defined by the semantic relation $[\![C]\!] \preceq [\![D]\!]$ in the contract algebra, and whose validity may for instance be proved by model checking. The set Σ of refinement relations $C \preceq D$ is obtained by the structural decomposition of subtyping relations of the form $s \leq t$ into either elementary axioms, e.g. **event** $\leq$ **boolean**, or proof obligations $C \preceq D$.

4.5 Greatest-lower and lowest-upper bounds

Just as the subtyping relation, which implements and extends the refinement relation of the contract algebra in the typing algebra, the operations that define the greatest lower bound and least upper bound of two contracts are extended to module type by induction on the structure of types. The intersection and union operators are extended to combine the set of input and output declarations of a module. The side condition $^{(*)}$ is that $x \notin dom(J)$.

$$\begin{array}{rl}
\tau(I,C) \sqcap \tau(J,D) = \tau(I \sqcap J, C \Downarrow D) & (I \cup (x : S)) \sqcap (J \cup (x : T)) = (I \sqcap J) \cup (x : S \sqcap T) \\
S \times T \sqcap U \times V = (S \sqcap U) \times (T \sqcap V) & (I \cup (x : S)) \sqcup (J \cup (x : T)) = (I \sqcup J) \cup (x : S \sqcup T) \\
\Lambda(x : S).T \sqcap \Lambda(y : U).V = \Lambda(x : (S \sqcup U)).(T \sqcap V[y/x]) & (I \cup (x : \textbf{input}\, b)) \sqcap J = (I \sqcap J)^{(*)} \quad \emptyset \sqcap J = \emptyset \\
\tau(I,C) \sqcup \tau(J,D) = \tau(I \sqcup J, C \Uparrow D) & (I \cup (x : \textbf{output}\, b)) \sqcap J = (I \sqcap J) \cup (x : \textbf{output}\, b)^{(*)} \\
S \times T \sqcup U \times V = (S \sqcup U) \times (T \sqcup V) & (I \cup (x : \textbf{input}\, b)) \sqcup J = (I \sqcup J) \cup (x : \textbf{input}\, b)^{(*)} \\
\Lambda(x : S).T \sqcup \Lambda(y : U).V = \Lambda(x : (S \sqcap U)).(T \sqcup V[y/x]) & (I \cup (x : \textbf{output}\, b)) \sqcup J = (I \sqcup J)^{(*)} \quad \emptyset \sqcup J = J
\end{array}$$

4.6 Type inference in the module language

Our aim is to check the correctness of program construction. Hence, type inference shall produce a consistent type assignment to contract and process names in the module language and generate a proof obligation in the form of an observer function [12] (used for describing some properties) in the target programming language. The type inference system is defined by the sequent $\Gamma/\Sigma \vdash exp$ where Γ is the typing environment, Σ the typing constraints and exp is an expression of the module language. It embeds the type system of the target language: we assume that the relation $\Gamma \vdash p$ tells that p is well-typed in the target language under the hypothesis Γ. The sequent establishes a structural correspondence between expressions and types. It is defined by induction on the structure of expressions in a similar manner as that proposed for Standard ML in [17]. The operator $\tau \cdot T$ promotes the type T to the kind τ. It is defined by $\tau \cdot (\tau'(I,C)) = \tau \cdot (I,C)$ and by $\tau \cdot (\Lambda(x : S).T) = \Lambda(x : S).(\tau \cdot T)$. It is used to check that the definition of a module type is a contract and to promote the type of a module.

$$\frac{\Gamma/\Sigma \vdash s : S \quad \Gamma/\Sigma \vdash t : T}{\Gamma/\Sigma \vdash s \times t : S \times T} \quad \frac{\Gamma/\Sigma \vdash s : S \quad \Gamma/\Sigma \vdash t : T}{\Gamma/\Sigma \vdash \Lambda(x : s).t : \Lambda(x : S).T} \quad \frac{\Gamma/\Sigma \vdash t : T}{\Gamma/\Sigma \vdash t\, x : (x : T)} \quad \frac{\Gamma/\Sigma \vdash dec : I \quad \Gamma/\Sigma \vdash dec' : J}{\Gamma/\Sigma \vdash dec, dec' : I \cup J}$$

$$\frac{\Gamma/\Sigma \vdash p \quad \Gamma/\Sigma \vdash q}{\Gamma/\Sigma \vdash \textbf{assume}\, p\, \textbf{guarantee}\, q : (p,q)} \quad \frac{\Gamma/\Sigma \vdash ag : C \quad \Gamma/\Sigma \vdash ag' : D}{\Gamma/\Sigma \vdash ag\, \textbf{and}\, ag' : C \Downarrow D} \quad \frac{\Gamma/\Sigma \vdash x : \gamma(x_1 : T_1..x_n : T_n, C) \quad (\Gamma(y_i) \leq T_i)_{i=1}^n}{\Gamma/\Sigma \vdash x(y_{1..n}) : C[y_i/x_i]_{i=1}^n}$$

$$\frac{\Gamma/\Sigma \vdash dec : I \quad \Gamma \cup I \vdash ag : C}{\Gamma/\Sigma \vdash \textbf{contract}\, dec; ag\, \textbf{end} : \gamma(I,C)} \quad \frac{\Gamma/\Sigma \vdash dec : I \quad \Gamma \cup I \vdash p}{\Gamma/\Sigma \vdash \textbf{process}\, dec; p\, \textbf{end} : \pi(I, ((), p))} \quad \frac{\Gamma/\Sigma \vdash exp : S \quad \Gamma/\Sigma \vdash exp' : T}{\Gamma/\Sigma \vdash (exp\, \textbf{and}\, exp') : S \sqcap T}$$

$$\frac{\Gamma(x) = T}{\Gamma/\Sigma \vdash x : T} \quad \frac{\Gamma/\Sigma \vdash dec : I \quad \Gamma \cup I/\Sigma \vdash exp : T}{\Gamma/\Sigma \vdash \textbf{functor}\ (dec)\ exp\ \textbf{end} : \Lambda I.T} \quad \frac{\Gamma/\Sigma \vdash x : \Lambda(z : S).T \quad \Gamma/\Sigma \vdash y : U \quad \Sigma \subset U \leq S}{\Gamma/\Sigma \vdash x(y) : T[y/z]}$$

$$\frac{\Gamma/\Sigma \vdash exp : T \quad \gamma \cdot T \leq T}{\Gamma/\Sigma \vdash \textbf{module}\, \textbf{type}\, x = exp : (x : T)} \quad \frac{\Gamma/\Sigma \vdash exp : S \quad \Gamma/\Sigma \vdash t : T \quad \Sigma \subset S \leq T}{\Gamma/\Sigma \vdash \textbf{module}\, x : t = exp : (x : \pi \cdot T)} \quad \frac{\Gamma/\Sigma \vdash def : I \quad \Gamma \cup I/\Sigma \vdash exp : T}{\Gamma/\Sigma \vdash \textbf{let}\, def\, \textbf{in}\, exp : T}$$

4.7 Correctness

The correctness of our module system is stated by showing that a program is well-typed if the constraints implied by Γ/Σ are consistent. We say that the environment ρ is well-typed with Γ/Σ, written $\rho : \Gamma/\Sigma$, iff all definitions in ρ are well-typed with Γ under the constraints Σ. Notice that, in this implementation of the type system, the generated constraints Σ define the proof obligations that are needed for checking that the specification exp is well-typed. To establish this theorem we define the semantics $[\![exp]\!]_\rho$ of a term exp in the module system by induction on the structure of exp. It is a set of processes p of the model of computation that satisfy their specifications.

Theorem 1. *If $\rho : \Gamma/\Sigma$ for a satisfiable Σ and $\Gamma/\Sigma \vdash exp : T$ then $[\![exp]\!]_\rho \subseteq [\![T]\!]_\rho$.*

Proof sketch The proof of Theorem 1 is by induction on the structure of expressions exp (just as for a regular type system). For each syntactic category in the grammar of exp, it considers a satisfiable set of constraints Σ, a well-typed environment $\rho : \Gamma/\Sigma$ and assumes a proof tree for $\Gamma/\Sigma \vdash exp : T$. Induction hypotheses are made to assert that the sub-expressions $exp_{i\in 1..n}$ of exp satisfy the expected sub-goals $[\![exp_i]\!]_\rho \subseteq [\![T_i]\!]_\rho$ in order to deduce that $[\![exp]\!]_\rho \subseteq [\![T]\!]_\rho$ by definition of the denotational semantics.

5 Discussion

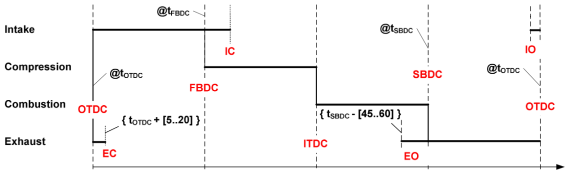
We illustrate the distinctive features of our contract algebra by reconsidering the specification of the four-stroke engine and its translation into observers in the target language of our choice: the multi-clocked synchronous (or polychronous) data-flow language Signal [7]. The separation of environmental assumptions and system guarantees is facilitated by the (unpaired) possibility to naturally express the complementary of a process-filter. Had we used automata to model $\mathbf{A}_{Intake}$, as in most of the related work, it would probably have been more difficult to model the engine not being in the intake mode: this would have required the definition of the complementary of an automaton and, most importantly, this could not have been done compositionaly. In our algebra, the complementary of the intake property is simply defined by $\widetilde{\mathbf{A}}_{Intake} = cam\ modulo\ 360^\circ \geq 90$. In general, the generic structure of observers specified in contracts will find a direct instance and compositional translation into the synchronous multi-clocked model of computation of Signal [16]. Indeed, a subtlety of the Signal language is that an observer not only talks about the value, true or false, of a signal, but also about its status, present or absent. Thanks to its encoding in three-valuated logic, an event (e.g. intake is present and true) can directly be associated with its complementary (e.g. intake is absent or false) without separating the status (the clock) and the value (the stream). Hence, the signal A_{intake} is present and true iff cam signal is present and its value is between 0 and 89 degrees.

$\mathbf{A}_{intake} = true\ when\ (0 \leq cam\ modulo\ 360 < 90)$ $\quad$ $\mathbf{G}_{intake} = (true\ when\ intake)\ default\ false$

The complementary of these assumptions is simply defined to be true iff the cam is absent or out of bounds: $\widetilde{\mathbf{A}}_{Intake} = (false\ when\ A_{intake})\ default\ true$. Notice that, for a trace of the assumptions $\mathbf{A}_{intake}$, the set of possible traces corresponding to $\widetilde{\mathbf{A}}_{Intake}$ is infinite (and dense) since it is not defined on the same clock as $\mathbf{A}_{intake}$.

$\mathbf{A}_{Intake} = 1\_0\_1\_0\_1\_0\_1\_0\_1\_$ and $\widetilde{\mathbf{A}}_{Intake} = 0\_\_\_0\_\_\_0\_\_\_0\_\_\_0\_$ or $0111011\_01\_10\_\_101\ldots$

It is also worth noticing that the clock of $\widetilde{\mathbf{A}}_{Intake}$ (its reference in time) need not be explicitly related to or ordered with $\mathbf{A}_{Intake}$ or $\mathbf{G}_{Intake}$: it implicitly and partially relates to the cam clock. Had we used a stricly synchronous model of computation, as in [18], it would have been necessary to know the clock of the system in order to define that of the complementary by a sample of it. Beside its Boolean structure, which allows for logical reasoning and normalization of contracts, our algebra supports the capability to compositionally refine contracts. For instance, consider a more precise model of the 4-stroke engine found in [4]. To additionally require the engine to reach the EC state (Exhaust closes) between 5 and 20 degrees, while in the intake mode, one will simply compose the initial contract with an additional constraint with : $A_{EC} = true\ when\ (5 <= cam\ modulo\ 360 < 21)$ and $G_{EC} = true\ when\ EC\ default\ false$. Similarly, event OTDC (Overlap Top Dead Center) occurs at the beginning of the cycle. The instant t_{OTDC} is a time observation of this event and the occurrence of the event EC is constrained by $\{t_{OTDC} + [5..20]\}$, hence $t_{OTDC} + 5 \leq t_{EC} \leq t_{OTDC} + 20$. We shall refine the specification of the engine to incorporate these additional constraints. This is done as follows:



```
module type better_engine = engine
  and contract
        input integer CAM ;
        output event EC, IC, EO, IO;
            phase(5,20)(CAM,EC)
        and phase(130,150)(CAM,IC)
        and phase(210,225)(CAM,EO)
        and phase(344,350)(CAM,IO)
end;
```

It is needless to say that a sophisticated solver, based for instance on Pressburger arithmetics, shall help us to verify the consistency of the above engine map. Nonetheless, an implementation of the above engine specification, for the purpose of simulation, can straightforwardly be derived. As a by-product, it defines an observer which may be used as a proof obligation against an effective implementation of the engine controller to verify that it implements the expected engine map. Alternatively, it may be used as a medium to synthesize a controller enforcing the satisfaction of the specified property on the implementation of the model. In Signal, process `phase` consists of one equation that is executed when its output `trigger` is needed (its clock is active). If the signal `cam` is present and within the specified bounds `min-max`, then `trigger` is present. Otherwise, `cam` is absent or simply out of bounds, the trigger is absent. The signals `cam` and `trigger` are respectively the input and output of the process whereas the names `min-max` are functor parameters of the process.

```
   process phase = {integer min, max;} (? integer cam; ! event trigger;)
                   (| trigger := when min<=(cam mod 360)<max |);
```

In the process `engine`, four instances of the `phase` equation are defined to specify the output signals that are relevant to a specific phase of the engine. Notice that these signals are not, *a priori*, synchronized one with the other: they are concurrent. This is done to favor a compositional specification of the system. Refinement precisely allows to iteratively build a sequentially executable specification by, e.g., synchronizing the signals `OTD`, `FBD`, `ITD` and `SBD`. This choice in favor of compositional modeling (polychrony) versus executability (synchrony) allows us to handle the additional specification of the *better engine* in a compositional manner, showing that the Signal language and our module system share the same concurrent/compositional design philosophy.

```
process engine =
 (? integer CAM; ! event OTD, FBD, ...
 (| OTD := phase {  0,  90} (CAM)
  | FBD := phase { 90, 180} (CAM)
  | ITD := phase {180, 270} (CAM)
  | SBD := phase {270, 360} (CAM)
  |);
```

```
process betterengine =
  (? boolean CAM ! event OTDC, FBDC, ITDC, ...
  (| (OTDC, FBDC, ITDC, SBDC) := engine (CAM)
   | EC := phase {  5,  20} (CAM)
   | IC := phase {130, 150} (CAM)
   | EO := phase {210, 225} (CAM)
   | IO := phase {344, 350} (CAM) |);
```

Contracts can be used to express exclusion properties. For instance, when the engine is in the intake mode, one should not start compression. We have $\mathbf{A}_{excl}=OTDC$ and $\mathbf{G}_{excl}=\neg FBDC$.

```
module type exclude = contract output event intake, compression;
                       assume intake guarantee (not compression default intake)
                      end;
```

In addition to the above safety properties, contracts can also be used to express liveness properties. For instance, consider the protocol for starting the engine. A battery is used to initiate its rotation. When the engine has successfully started, the battery can be turned off. We define a contract to guarantee that when the ignit button is pushed, the starter eventually stops.

```
module type StarterOff = contract input event ignit; output boolean starter;
                           assume ignit guarantee eventually not starter
                         end;
```

6 Implementation

The module system described in this paper, embedding data-flow equations defined in syntax, has been implemented in Java. It produces a proof tree that consists of 1/ an elaborated Signal program, that hierarchically renders the structure of the system described in the original module expressions, 2/ a static type assignment, that is sound and complete with respect to the module type inference system, 3/ a proof obligation consisting of refinement constraints, that are compiled as an observer or a temporal property in Signal.

The property is then tended to SIGNAL's model-checker, Sigali [20], which allows to prove or disprove that it is satisfied by the generated program. Satisfaction implies that the type assignment and produced SIGNAL program are correct with the initially intended specification. The generated property may however be used for other purposes. One is to use the controller synthesis services of Sigali [19] to automatically generate a SIGNAL program that enforces the property on the generated program. Another, in the case of infinite state system (e.g. on numbers) would be to generate defensive simulation code in order to produce a trace if the property is violated.

7 Related work

The use of contracts has been advocated for a long time in computer science [21, 13] and, more recently, has been successfully applied in object-oriented software engineering [22]. In object-oriented programming, the basic idea of design-by-contract is to consider the services provided by a class as a contract between the class and its caller. The contract is composed of two parts: requirements made by the class upon its caller and promises made by the class to its caller. The *assumption* specifies hypothesis which has to be satisfied by the component in order to provide the *guarantee*.

In the context of software engineering, the notion of assertion-based contract has been adapted for a wide variety of languages and formalisms but the central notion of time and/or trace needed for reactive system design is not always taken into account. For instance, extensions of OCL with linear or branching-time temporal logics have been proposed in [25, 10], focusing on the expressivity of the proposed constraint language (the way constraints may talk about the internals of classes and objects), and considering a fixed "sequence of states". This is a serious limitation for concurrent system design, as this sequence becomes an interleaving of that of individual objects.

In the theory of interface automata [1], the notion of interface offers benefits similar to our notion of contracts and for the purpose of checking interface compatibility between reactive modules. In that context, it is irrelevant to separate the assumptions from guarantees and only one contract needs to be and is associated with a module Separation and multiple views become of importance in a more general-purpose software engineering context. Separation allows more flexibility in finding (contra-variant) compatibility relations between components. Multiple views allow better isolation between modules and hence favor compositionality. In our contract algebra as in interface automata, a contract can be expressed with only one filter. To this end, the filtering equivalence relation (that defines the equivalence class of contracts that accept the same set of processes) may be used to express a contract with only one guarantee filter and with its hypothesis filter accepting all the processes (or, conversely, with only one hypothesis filter and a guarantee filter that accepts no process).

In the context of the EC project Speeds [6], a model of assume-guarantee contracts is proposed which extends the notion of interface automata with modalities and multiple views. This consists of labelling transitions that may be fired and other that must. By contrast to our domain-theoretical approach, the Speeds approach starts from an abstracted notion of modules whose only structure is a partial order of refinement. The proposed approach also leaves the role of variables in contracts unspecified, at the cost of some algebraic relations such as inclusion.

In [18], a notion of synchronous contracts is proposed for the programming language LUSTRE. In this approach, contracts are executable specifications (synchronous observers) timely paced by a clock (the clock of the system). This yields an approach which is satisfactory to verify safety properties of individual modules (which have a clock) but can hardly scale to the modeling of globally asynchronous architectures (which have multiple clocks).

In [8], a compositonal notion of refinement is proposed for a stream-processing data-flow language. The proposed type system allows reasoning on properties of programs abstracted by input-output types and causality graphs. In a similar way as Broy et al., we aim at using our module system to associate programs with compilation contracts, consisting of the necessary (and sufficient) synchronization and scheduling relations for modular code generation.

The system Jass [5] is somewhat closer to our motivations and solution. It proposes a notion of *trace*, and a language to talk about traces. However, it seems that it evolves mainly towards debugging and defensive code generation. For embedded systems, we prefer to use contracts for validating composition and hope to use formal tools once we have a dedicated language for contracts. Like in JML [15], the notion of agent with inputs/outputs does not exist in JASS, the language is based on class invariants, and pre/post-conditions associated with methods.

Another example is the language Synergy [9], that combines two paradigms: object-oriented modeling for robust and flexible design, and synchronous execution for precise modeling of reactive behavior. In [14], a method of encapsulation based on the object-oriented paradigm and behavioral inheritance for the description of synchronous models is proposed. [24] describes an ML-like model for decomposing complex synchronous structures in parameterizable modules.

Our main contribution is to define a type system starting from a domain theoretical algebra for assume-guarantee reasoning consisting of a Boolean algebra of process-filters and a Heyting algebra of contracts. This yields a rich structure which is

1/ generic, in the way it can be implemented or instantiated to specific models of computation;

2/ flexible, in the way it can help structuring and normalizing expressions;

3/ complete, in the sense that all combinations of propositions can be expressed *within* the model.

Finally, a temporal logic that is consistent with our model, such as for instance the ATL (Alternating-time Temporal Logic [3]) can directly be used to express assumptions about the context of a process and guarantees provided by that process.

8 Conclusion

Starting from an abstract characterization of behaviors as functions from variables to a domain of values (Booleans, integers, series, sets of tagged values, continuous functions), we introduced the notion of process-filters to formally characterize the logical device that filters behaviors from process much like the assumption and guarantee of a contract do. In our model, a process p fulfils its requirements (or satisfies) ($\mathbf{A}$,$\mathbf{G}$) if either it is rejected by $\mathbf{A}$ (i.e., if $\mathbf{A}$ represents assumptions on the environment, they are not satisfied for p) or it is accepted by $\mathbf{G}$. The structure of process-filters is a Boolean algebra and allows for reasoning on contracts with great flexibility to abstract, refine and combine them. In addition to that, and unlike the related work, the negation of a contract can formally be expressed from within the model. Moreover, contracts are not limited to expressing safety properties, as is the case in most related frameworks, but encompass the expression of liveness properties. This is all again due to the central notion of process-filter. We introduced a module system based on the paradigm of contract for a synchronous multi-clocked formalism, SIGNAL, and applied it to the specification of a component-based design process. The paradigm we are putting forward is to regard a contract as the behavioral type of a component and to use it for the elaboration of the functional architecture of a system together with a proof obligation that validates the correctness of assumptions and guarantees made while constructing that architecture.

References

[1] Alfaro L. and Henzinger T. A. Interface automata. In *ESEC / SIGSOFT FSE*, pp. 109–120, 2001.

[2] Alpern, B., Schneider, F. Proving boolean combinations of deterministic properties. In Proceedings of the Second Symposium on Logic in Computer Science. IEEE Press, 1987.

[3] Alur, R., Henzinger, T., and Kupferman, O. Alternating-time Temporal Logic. In *Journal of the ACM, v. 49*. ACM Press, 2002.

[4] André C., Mallet F., and Peraldi-Frati M-A. A multiform time approach to real-time system modeling; application to an automotive system. In *International Symposium on Industrial Embedded Systems*, pp. 234–241, 2007.

[5] Bartetzko D., Fischer C., Mueller M., and Wehrheim H. Jass - Java with assertions. *In Havelund, K., Rosu, G. eds : Runtime Verification*, Volume 55 of ENTCS(1):91–108, 2001.

[6] Benveniste A., Caillaud B., and Passerone R. A generic model of contracts for embedded systems. In *INRIA RR n. 6214*, 2007.

[7] Benveniste A., Le Guernic P., and Jacquemot C. Synchronous programming with events and relation: the SIGNAL language and its semantics. In *Science of Computer Programming*, volume v.16, 1991.

[8] Broy, M. Compositional refinement of interactive systems. Journal of the ACM, v. 44. ACM Press, 1997.

[9] Budde R., Poigné A., and Sylla K.-H. Synergy - an object-oriented synchronous language. *In Havelund, K., Rosu, G. eds: Runtime Verification*, Volume 153 of Electronic Notes in Theoretical Computer Science(1):99–115, 2006.

[10] Flake S. and Mueller W. An OCL extension for realtime constraints. In *Lecture Notes in Computer Science 2263*, pp. 150–171, 2001.

[11] Glouche Y., Le Guernic P., Talpin J.-P., and Gautier T. A boolean algebra of contracts for logical assume-guarantee reasoning. Research Report RR 6570, INRIA, 2008.

[12] Halbwachs N. and Raymond P. Validation of Synchronous Reactive Systems: From Formal Verification to Automatic Testing. In *Advances in Computing Science-Proceedings of the ASIAN'99 conference*, 1999.

[13] Hoare C.A.R. An axiomatic basis for computer programing. In *Communications of the ACM*, pp. 576–583, 1969.

[14] Kerbœuf M. and Talpin J.-P. Encapsulation and behavioural inheritance in a synchronous model of computation for embedded system services adaptation. In *Journal of languages, algebra and progr. Special issue on process algebra and syst. arch*. Elsevier, 2004.

[15] Leavens G. T., Baker A. L., and Ruby C. JML: A notation for detailed design. In Kilov H., Rumpe B., and Simmonds I., editors, *Behavioral Specifications of Businesses and Systems*, pp. 175–188. Kluwer Academic Publishers, 1999.

[16] Le Guernic P., Talpin J.-P., and Le Lann J.-C. Polychrony for system design. *Journal for Circuits, Systems and Computers*, Special Issue on Application Specific Hardware Design, 2003.

[17] Leroy X. A modular module system. Journal of Functional Programming, v. 10(3). Cambridge Academic Press, 2000.

[18] Maraninchi F. and Morel L. Logical-time contracts for reactive embedded components. In *In 30th EUROMICRO Conference on Component-Based Software Engineering Track, ECBSE'04, Rennes, France*, 2004.

[19] Marchand, H., Bournai, P., Le Borgne, M., Le Guernic, P. Synthesis of Discrete-Event Controllers based on the Signal Environment, Discrete Event Dynamic System: Theory and Applications, v. 10(4), 2000.

[20] Marchand, H., Rutten, E., Le Borgne, M., Samaan, M. Formal Verification of programs specified with Signal: Application to a Power Transformer Station Controller. Science of Computer Programming, v. 41(1). Elsevier, 2001.

[21] Martin A. and Lamport L. Composing specifications. In *ACM Trans. ProgramLang. Syst. 15*, pp. 73–132, 1993.

[22] Meyer B. *Object-Oriented Software Construction, Second Edition*. ISE Inc., Santa Barbara, 1997.

[23] Mitchell R. and McKim J. *Design by Contract, by Example*, Addison-Wesley, 2002.

[24] Nowak D., Talpin J.-P., Gautier T., and Le Guernic P. An ML-like module system for the synchronous language SIGNAL. In *European Conference on Parallel Processing (EUROPAR'97)*, August 1997.

[25] Ziemann P. and Gogolla M. An extension of OCL with temporal logic. In *Critical Systems Development with UML-Proceedings of the UML'02 workshop*, 2002.

From Goal-Oriented Requirements to Event-B Specifications

Benjamin Aziz, Alvaro E. Arenas, Juan Bicarregui
e-Science Centre, STFC Rutherford Appleton Laboratory
Oxford OX11 0QX, United Kingdom
{benjamin.aziz,alvaro.arenas,juan.bicarregui}@stfc.ac.uk
Christophe Ponsard, Philippe Massonet
Centre d'Excellence en Technologies de l'Information et de la Communication (CETIC)
B-6041 Charleroi, Belgium
{christophe.ponsard,philippe.massonet}@cetic.be

Abstract

In goal-oriented requirements engineering methodologies, goals are structured into refinement trees from high-level system-wide goals down to fine-grained requirements assigned to specific software/hardware/human agents that can realise them. Functional goals assigned to software agents need to be operationalised into specification of services that the agent should provide to realise those requirements. In this paper, we propose an approach for operationalising requirements into specifications expressed in the Event-B formalism. Our approach has the benefit of aiding software designers by bridging the gap between declarative requirements and operational system specifications in a rigorous manner, enabling powerful correctness proofs and allowing further refinements down to the implementation level. Our solution is based on verifying that a consistent Event-B machine exhibits properties corresponding to requirements.

1 Introduction

Goal-driven approaches focus on why systems are constructed, providing the motivation and rationale for justifying software requirements. Examples of goal-oriented requirements methodologies include KAOS [24] and *i**/Tropos [10], among others. In these methodologies, a goal is an objective, which the system under consideration and its environment must achieve. Hence, goals are *operationalised* into specifications of operations to achieve them. For instance, the KAOS language supports the specification of operations defined as constraints on state transitions. The language relies on a temporal state-based logic, where a global clock generates ticks regularly, creating new states and transitions. These constraints, described as pre/post/trigger conditions, restrict the possible values in pairs of successive states.

One of the main aspects of goal-oriented requirements engineering methodologies is that they are best at refining goals and capturing intentions and expectations about the system-to-be. However, they lack the ability to extend this refinement process to the design specification level. Instead, they are usually limited to the definition of high-level declarative specifications of the system design. One would like to continue the refinement process though these high-level specifications down to levels of detail that facilitate the implementation of such specifications.

In this paper, we define an approach for operationalising goal-oriented requirements into Event-B specifications [4], using model-checking techniques to demonstrate that an Event-B machine is a model of the system requirements expressed as linear temporal logic formulae. The approach is general, however, we propose a few operationalisation patterns that assist designers in the derivation of an Event-B machine from the system requirements. This can be considered a small step towards achieving an automatic construction of these machines.

Combining goal-orientation and Event-B has several benefits. First, they have a common scope in that they target the modelling of the system as a whole. Second, they are complementary in that goals help in identifying key properties and reasoning on them while Event-B helps in designing rigorously a more operational system by introducing more and more design details using the refinement approach. Finally, at tool level, the benefit is mutual: requirements level tools[22] help in ensuring consistent/complete requirements and guide the elaboration of the initial Event-B specification. Event-B industrial-level tools can then be used to perform more powerful verification, especially automated proofs [3].

The rest of the paper is structured as follows. Section 2 gives an overview of goal-oriented requirements engineering and introduces an example of a safety-critical system. Section 3 gives an overview of the Event-B language. Section 4 presents our approach for defining a correct operationalisation of requirements into Event-B machines. Section 5 compares our work with related literature and finally, Section 6 concludes our work and highlights future research directions.

2 Goal-Oriented Requirements Engineering

Requirements engineering involves eliciting, analysing and specifying the requirements of a system. The precise understanding of these requirements serves as a foundation to assess and manage the subsequent development phases. Goal-oriented requirements engineering methodologies, such as KAOS [24] and i^*/Tropos [10], focus on justifying why a system is needed through the specification of its high-level goals. These goals then drive the requirements elaboration process, which results in the definition of domain-specific requirements that can be *implemented* by the system components under development.

Goals may be organised in an AND-refinement hierarchy [24], where higher-level goals are in general strategic and coarse-grained whereas lower-level goals are technical and fine-grained. In such hierarchies, AND-links (represented here by circles) relate a goal to a set of sub-goals possibly conjoined with domain properties or environment assumptions; this means that satisfying all the subgoals in the refinement is a sufficient condition in the domain for satisfying the goal. Goal refinement ends when every sub-goal is realisable by some individual component assigned to it in the software-to-be.

Formally, goals at all levels can be represented in real-time linear temporal logic, which in addition to the usual first order logic operators ($\wedge \vee \neg \rightarrow \leftrightarrow$), it provides a number of temporal operators for the future: $\circ$ (next), $\Box$ (always), $\Diamond$ (eventually) and $\Diamond_{\leq d}$ (bounded eventually). We do not consider past LTL in the scope of this paper. Furthermore, we write $P \Rightarrow Q$ to mean $\Box(P \rightarrow Q)$.

2.1 Goals in Action: A Mine Sump

In order to demonstrate the goal-refinement methodology and introduce the requirements that will be operationalised, we consider the example of a mine sump [12]. In this system, water seeps into the sump from the mine and the level of water is kept within bounds by operating a pump. Additionally, a bell alarm must be immediately sounded if methane is detected in the sump and the pump must be shut down.

Figure 1 shows the goal model for such a mine sump system. The top goal of the system is to keep

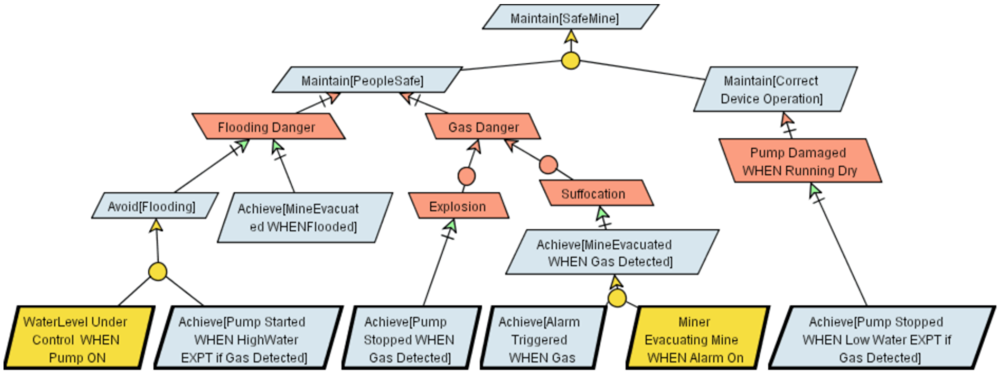

Maintain[SafeMine]
Maintain[PeopleSafe]
Maintain[Correct Device Operation]
Flooding Danger
Gas Danger
Pump Damaged WHEN Running Dry
Avoid[Flooding]
Achieve[MineEvacuat ed WHENFlooded]
Explosion
Suffocation
Achieve[MineEvacuated WHEN Gas Detected]
WaterLevel Under Control WHEN Pump ON
Achieve[Pump Started WHEN HighWater EXPT if Gas Detected]
Achieve[Pump Stopped WHEN Gas Detected]
Achieve[Alarm Triggered WHEN Gas
Miner Evacuating Mine WHEN Alarm On
Achieve[Pump Stopped WHEN Low Water EXPT if Gas Detected]

Figure 1: Goal/Obstacle Model of the Mine Sump System

the mine safe and the main refinement strategy is to avoid obstacles to safety. These obstacles (in red) are summarised by the flooding and methane dangers and the risk of damaging the pump when the sump runs dry. The methane danger is further refined to the dangers of explosion and suffocation. Mitigating those dangers yield four low level requirements under the responsibility of the system (in blue with thick outline at bottom line of the figure) with some necessary domain conditions and expectations (in yellow).

For sake of simplicity and because mapping of class-like model on Event-B has already been addressed [23], the domain is not structured in a complex object model but simply represented by five attributes of the *mine* system: *highWater*, *lowWater*, *pump*, *methane* and *bell*. Note that *highWater* and *lowWater* cannot hold at the same time. Based on this, our requirements can be formalised as follows.

Requirement Achieve[PumpStoppedWHENGasDetected]
Refines Avoid[Explosion]
FormalDef $(\forall m : Mine)\ m.methane = True \Rightarrow \ \circ\ m.pump = Off$

Requirement Achieve[AlarmTriggeredWHENGasDetected]
Refines Avoid[Suffocation]
FormalDef $(\forall m : Mine)\ m.methane = True \Rightarrow \ \circ\ m.bell = On$

Requirement Achieve[PumpStartedWHENHighWaterEXPTmethanePresent]
Refines Avoid[MineFlooded], Avoid[Explosion]
FormalDef $(\forall m : Mine)\ m.highWater = True \Rightarrow \ \Diamond_{\leq d_1}\ (m.methane \neq True \Rightarrow \ m.pump = On)$

Requirement Achieve[PumpStoppedWHENLowWaterEXPTmethanePresent]
Refines Avoid[MineFlooded]
FormalDef $(\forall m : Mine)\ m.lowWater = True \Rightarrow \ \Diamond_{\leq d_2}\ (m.methane \neq True \Rightarrow \ m.pump = Off)$

The dangers arising from the presence of methane have higher priority than flooding (people can still be evacuated when flooding occurs) and pump damage (not people). Therefore, the first two (gas related) requirements will have priority on the last two. This is why the pump is not started in the presence of gas (conflict resolution) and why *next* is used instead of bounded eventuality in the first two requirements.

The deadlines, d_i, represent the amount of time within which the predicates following $\Diamond_{\leq d_i}$ must become true. When d_i is equal to 0, then bounded eventually $\Diamond_{\leq d_i}$ becomes the next operator. Due to the above safety priority, both d_1 and d_2 must be more than 0.

It is worth noting that a number of verifications can already be addressed at this level: goal and obstacle refinements, conflicts resolution. Some tool support is available [22], mostly based on model-checking. The Event-B mapping will bridge the gap with the next development step but will also give access to a larger set of tools, including proof-based ones.

3 Event-B with Obligations

Event-B is a specification language for developing discrete systems [4]. Behavioural aspects of Event-B models are expressed by means of *machines*. A machine is defined in terms of a global *state* consisting of a set of *variables*, and some *events* that cause the state to change by updating the values of the variables as defined by the *generalised substitution* of the event. Events are guarded by a *condition*, which when satisfied implies that the event is permitted to execute by applying its generalised substitution in the current state of the machine. Event-B also incorporates a refinement methodology, which can be used by software architects to incrementally develop a model of a system starting from the initial most abstract specification and following gradually through layers of detail until the model is close to the implementation. *Invariants* denoting desirable behaviour can be specified at each layer of detail as well as across different layers.

In Event-B, an event is defined by the following syntax:

$$ev ::= \ \texttt{EVENT}\ e\ \texttt{WHEN}\ G\ \texttt{THEN}\ S\ \texttt{END}$$

where G is the guard, expressed as a first-order logical formula in the state variables, and S is the generalised substitution, defined by the following syntax:

$$\begin{array}{lll} S \ ::= & \texttt{SKIP} & \text{Do nothing} \\ & \mid\ x := \ E(v) & \text{Deterministic substitution} \\ & \mid\ \texttt{ANY}\ t\ \texttt{WHERE}\ P(t,v)\ \texttt{THEN}\ x := F(t,v)\ \texttt{END} & \text{Non-deterministic substitution} \\ & \mid\ S \parallel S' & \text{Parallel substitution} \end{array}$$

SKIP is a do-nothing substitution, which does not affect the machine's state. The deterministic substitution, $x := \ E(v)$, assigns to variable x the value of expression $E(v)$, defined over set of state variables v. In a non-deterministic substitution, $\texttt{ANY}\ t\ \texttt{WHERE}\ P(t,v)\ \texttt{THEN}\ x := F(t,v)\ \texttt{END}$, it is possible to choose non-deterministically local variables, t, that will render the logical guard $P(t,v)$ true. If this is the case, then the substitution, $x := \ F(t,v)$, can be applied, otherwise nothing happens. Finally, substitutions can be composed in parallel, $S \parallel S'$. For a comprehensive description of the Event-B language, we refer the reader to more detailed references such as [4, 18].

The operationalisation of requirements into Event-B machines requires extending the machinery of Event-B to incorporate the notion of obligations. The extra machinery has already been formally defined in [6]. Obligations are needed since the linear temporal logic-based specifications of requirements usually define not only a maximal set of permitted behaviours but also a minimal set of obliged ones. When the guard of an event is true, there is no obligation to perform the event and its execution may be delayed as a result of, for example, interleaving it with other permitted events. The choice of scheduling permitted events is made non-deterministically. In [6], we describe how obligations can be modelled in Event-B as events with *triggers*. The trigger of an event is a first-order logical formula in the state variables expressing an obligation on when the event must be executed.

We introduce here three types of triggered events:

- WITHIN events. These represent the general case of triggered events and are written as follows:

  $$\texttt{EVENT}\ e\ \texttt{WHEN}\ T\ \texttt{WITHIN}\ n\ \texttt{NEXT}\ S\ \texttt{END}$$

  where T is the trigger condition such that when T becomes true, the event must be executed within at most $n+1$ number of events, provided the trigger remains true. If the trigger changes to false within that number, the obligation to execute the event is canceled. This type of event represents a bounded version of the leads-to modality, represented by the obligation $(T \ \Rightarrow \ \ \Diamond_{\leq n}(T \rightarrow e))$.

- NEXT events. These events are a special case of the WITHIN events where $n = 0$. Their syntax is:

  $$ev ::= \ \texttt{EVENT}\ e\ \texttt{WHEN}\ T\ \texttt{NEXT}\ S\ \texttt{END}$$

  Whenever T becomes true, then the event will be the next one to be executed, which fulfills the obligation $(T \ \Rightarrow \ \ \circ e)$.

- EVENTUALLY events. These are also a special case of the WITHIN events, but where the value of n is unbounded and non-deterministically chosen. The syntax of the event is:

  $$ev ::= \ \texttt{EVENT}\ e\ \texttt{WHEN}\ T\ \texttt{EVENTUALLY}\ S\ \texttt{END}$$

  such that when T becomes true, then the event will eventually be executed. The choice of n (i.e. the deadline for executing the event) is made when the trigger becomes true and the value of n is known only internally. This event is modelling the obligation $(T \ \Rightarrow \ \ \Diamond(T \rightarrow e))$.

Here, our triggered events do not include a guard; we are interpreting the guard as being the same as the trigger, that is the event is triggered when it is permitted. All of the above triggered events are syntactic sugar and can be encoded and refined in the standard Event-B language, as described in [6]. In the rest of the paper, we adopt Event-B with obligations as our system specification language.

3.1 Machine Consistency

The semantics of Event-B machines is expressed via proof obligations, which must be proved in order for the machine to be well defined.

Definition 1 (Machine Consistency)**.** *Let M be a Event-B machine with, obligations, state variables v and invariant $I(v)$. Machine M is consistent if:*

1. *(Feasibility)*
   - *For each un-triggered event* `EVENT` e `WHEN` G `THEN` S `END` *in M the following property holds:* $I(v) \wedge G(v) \rightarrow \exists v' \cdot S(v,v')$.
   - *For each triggered event* `EVENT` e `WHEN` T `WITHIN` n `NEXT` S `END` *in M the following property holds:* $I(v) \wedge T(v) \rightarrow \exists v' \cdot S(v,v')$.
2. *(Invariant Preservation)*
   - *For each un-triggered event* `EVENT` e `WHEN` G `THEN` S `END` in *M the following property holds:* $I(v) \wedge G(v) \wedge S(v,v') \rightarrow I(v')$.
   - *For each triggered event* `EVENT` e `WHEN` T `WITHIN` n `NEXT` S `END` *in M the following property holds:* $I(v) \wedge T(v) \wedge S(v,v') \rightarrow I(v')$.
3. *(Deadlock Freeness for Triggered Events)*
   - *Every triggered event* `EVENT` e `WHEN` T `WITHIN` n `NEXT` S `END` *in M will be executed within at most $n+1$ events, provided trigger T remains true.*

Deadlock freeness is not easy to prove in general. In [6], we introduced strategies to prove such a property. For instance, if the machine consists only of `NEXT` triggered events, it will be deadlock-free if all triggers are mutually disjoint.

4 Operationalising Requirements into Event-B Specifications

The essence of goal refinement is to decompose a goal into sub-goals that can be implemented by the components of the software-to-be. Such sub-goals are called *requirements*. The process of assigning requirements (declarative property specifications) to their systems components (operational specifications) is called *operationalisation*. Our approach to operationalisation is to propose an Event-B-based specification, which represents a *consistent* machine, and then verify that the machine meets the requirements. In this sense, the machine is considered to be a *model* of the requirements.

We do not define a notion of correct operationalisation at the level of individual events, instead we develop a notion of correctness associated to an Event-B machine in relation to a set of requirements. This is formalised by the following.

Definition 2 (Correct Operationalisation)**.** *Given KAOS requirements $R_1, \cdots, R_k$, an Event-B machine M is a correct operationalisation of the requirements if the following conditions hold: 1) Machine M is consistent and 2) Machine M is a model for all requirements, i.e. $M \models R_i$ for $i = 1, \cdots, R_k$*

This definition provides a general solution to the problem of operationalisation, which could be achieved either through program verification or program construction methods. We use the notation $M \models P$ to indicate that the Event-B machine M is a model for the linear temporal logic property P according to the classical definition of a model checking problem [7].

Next, we demonstrate how patterns can assist the system designer in obtaining, in a constructive manner, a high-level system design in Event-B from system requirements expressed as real-time linear temporal logic formulae.

4.1 Operationalisation Patterns

In software engineering, patterns are defined as general reusable solutions to commonly recurring problems in software design. They define templates on how to solve a problem. Table 1 provides operationalisation patterns for three of the most frequently used goal patterns: *immediate generation* when using the next temporal operator, *eventually generation* when using the eventually temporal operator and *bounded generation* when using the eventually bounded operator. Here, we assume that C and S denote first-order logical formulae defined over the space of objects of the specification-to-be. In the Event-B machine, these objects are defined as variables and so $\overline{C}$ is the corresponding formula over Event-B state and $\overline{S}$ is the generalised substitution derived from predicate S, where S is seen as the post-condition of the substitution.

Requirement	**Formal Definition**	**Event-B Operationalisation**
Immediate Generation	$C \Rightarrow \circ S$	`EVENT` e `WHEN` $\overline{C}$ `NEXT` $\overline{S}$ `END`
Bounded Generation	$C \Rightarrow \Diamond_{\leq d} S$	`EVENT` e `WHEN` $\overline{C}$ `WITHIN` d `NEXT` $\overline{S}$ `END`
Eventually Generation	$C \Rightarrow \Diamond S$	`EVENT` e `WHEN` $\overline{C}$ `EVENTUALLY` $\overline{S}$ `END`

Table 1: Patterns for Operationalising Requirements into Event-B

In summary, our method to derive an Event-B machine from system requirements comprises the following steps:

1. Transform the object state (e.g. KAOS Object Model) into the Event-B state. UML-B defines it for Event-B and proposes a related tool [23].

2. Derive Event-B events from KAOS requirements following the patterns presented in Table 1.

3. Complete the Event-B machine with the initialisation part and other events.

4. Verify that the Event-B machine is a correct operationalisation of the KAOS requirements.

The resulting Event-B machine is an abstract specification of a system meeting the KAOS requirements, which can be refined down to the implementation following the Event-B refinement method [4].

One important issue to note here is the relationship between time in the definition of requirements and duration of events in Event-B specifications. In goal-oriented methodologies, real-time temporal logic is used to formally specify goals and requirements. The main advantage of this logic is that real time temporal properties can be expressed simply using temporal operators without the explicit use of time variables. A linear temporal structure is used and time is related to states using a history function. Given a current state and a time unit, the "next" temporal operator refers to the next state in the linear temporal structure. We have modelled the temporal structure in Event-B with obligations by defining each event as having a duration of one time unit and associating each triggered event with a counter that controls that the event must be executed within the associated time constraint [6]. An abstract scheduler enforces then the execution of an event when its associated counter is zero. For instance, we are interpreting the requirement of a "next" operation to indicate the next event in the machine operationalising the requirement, and this is enforced by making its counter zero when its trigger condition is true.

4.2 Operationalising the Mine Sump Requirements

In order to operationalise the requirements of the mine sump example introduced earlier, we need to represent the system's objects that are mentioned in those requirements as Event-B variables. This is done by mapping the objects to the variables as well as their corresponding value spaces using some bijection function. For simplicity, we assume this to be the identity function; each Event-B variable and its value space is named after its corresponding object and value space. After this step, it is possible to consider requirements as Event-B requirements that can be verified. We show in Table 2 the operationalisation of the requirements in the case of our mine sump example following the patterns suggested in Table 1.

Requirement	**Event-B Event**
Achieve[PumpStartedWHENHighWaterEXPTmethanePresent]	*high_water_detected*
Achieve[PumpStoppedWHENLowWaterEXPTmethanePresent]	*low_water_detected*
Achieve[PumpStoppedWHENmethaneDetected]	*methane_detected*
Achieve[AlarmTriggeredWHENmethaneDetected]	*methane_detected*

Table 2: Mapping the Mine Sump Requirements to Event-B Events

The derived definitions of these events are shown in the machine of Figure 2 modelled in Event-B with obligations [6]. The machine consists of events for detecting high and low water levels, detecting methane leak and changing the state of the high/normal/low water levels and the methane level. The machine is initialised in the normal state where the water level is between high and low levels, there is no

```
INVARIANTS
  lowwater, highwater:  Bool
  methane:  Bool
  pump, bell :  {ON, OFF}
  highwater ∧ lowwater = false

EVENTS
Initialisation
  BEGIN
    highwater := false ||
    lowwater := false ||
    methane := false ||
    pump := OFF ||
    bell := OFF
  END

high_water_detected
  WHEN highwater = true
  WITHIN d1 NEXT pump := ON
  END

low_water_detected
  WHEN lowwater = true
  WITHIN d2 NEXT pump := OFF
  END

methane_detected
  WHEN methane = true
  NEXT pump := OFF ||
       bell := ON
  END

methane_leak
  WHEN methane = false
  THEN methane := true
  END

high_to_normal
  WHEN highwater = true
       & lowwater = false
       & pump = ON
  THEN highwater := false
  END

normal_to_low
  WHEN highwater = false
       & lowwater = false
       & pump = ON
  THEN lowwater := true
  END

low_to_normal
  WHEN highwater = false
       & lowwater = true
       & pump = OFF
  THEN lowwater := false
  END

normal_to_high
  WHEN highwater = false
       & lowwater = false
       & pump = OFF
  THEN lowwater := true
  END
```

Figure 2: The Mine Sump Machine in Event-B with Obligations

methane leak and both the pump and the alarm bell are off. The machine then makes transitions across the high, normal and low water levels as long as no methane is leaked. Whenever the high (low) water level is sensed, the machine obliges the high (low) water detection event to fire. This obligation must be fulfilled within $d_1 + 1$ ($d_2 + 1$) number of events. If methane is leaked, the machine obliges the methane detection event to execute, which in turn shuts down the system by turning off the water sensors and the pump and sounds the alarm bell to evacuate the area. This event must be executed immediately in the next state following the methane leak since it is of a higher priority than any other event.

Finally, we establish the correctnes of our requirements operationalisation by the following theorem.

Theorem 1. *The Event-B machine presented in Figure 2 is a correct operationalisation of the requirements described in Section 2.1.*
Proof: The proof relies on showing that properties 1 and 2 of Definition 2 hold.

Property 1.1: the feasibility of the machine of Figure 2 can be proved trivially according to [18, Page 7, Figure 13], since all the events in the machine have deterministic substitutions.

Property 1.2: the invariant preservation property was expressed as five proof obligations for the (unsugared) machine when it was encoded in the Rodin tool (`http://www.event-b.org/platform.html`), all of which were discharged by the tool.

Property 1.3: to prove the deadlock freeness of the machine of Figure 2, we need to adopt a notion of schedulability as in [6] and then prove that the scheduler does not allow any one active counter (from the set $\{d1, d2\}$) to have the value of zero at any one computational slot. This implies that both counters must be always initialised with values above zero, i.e. bounded eventuality cannot be refined to next.

Property 2: for this property, we need to show that the machine of Figure 2 is a model of the requirements of Section 2.1. This was established by applying the ProB LTL model checker [15, 16], which verified that the machine is indeed a model of all the four requirements.

5 Related Work

There is a body of work on relating requirements and specifications. The standard operationalisation of KAOS requirements is presented in [14], where formal derivation rules map KAOS goal specifications, represented as real-time specifications, into specification of software operations, represented as sets of pre-, post- and trigger-conditions. This work has inspired us, but there are some differences. First, [14] requires a true-concurrency semantics for the operationalisation; by contrast, ours follows Event-B interleaving semantics. We consider the interleaving semantics to be more natural to developers. However it is more difficult for specifying timing constrains which required the Event-B extension presented previously [6]. Second, our definition does not require full equivalence between the specification and the Event-B machine. Third, our trigger condition is a state predicate while [14] is more general and allows for past formulaes. Finally, the use of well-established specification languages like Event-B allows the modeller to continue the development incrementally with a stepwise refinement method.

An early attempt to bridge requirements to specification in the context of the B method is presented in [21], which relates KAOS operations with B operations, and suggests to keep maintain properties as B invariants. The mapping is however totally informal and mainly focusing on deriving traceability links. In contrast our work is anchored in Event-B with a semantic link allowing both derivation and verification.

In [11], the authors propose a method for generating B operations from KAOS requirements that is driven by the aim of analysing desirable security properties in the system requirement and then preserving those properties when generating the system specification. However, their method is limited and lacks formality in that it assumes a syntactic relationship between KAOS and B operations. For example, the authors do not demonstrate how triggering conditions in KAOS requirements, which represent obliged behaviour, are transformed and preserved throughout the B specification and refinement process.

In [13], the authors propose an operationalisation of KAOS requirements into event-based transition systems specified in the language of Labeled Transition Systems (LTS) of [17]. However, their main aim is to be able to analyse, such as checking for consistency and implicit requirements, and animate the KAOS operation models rather than bridge the requirements to another operational model, such as Event-B, which has allowed us to continue the refinement process throughout the system specification.

Finally, Tropos and i* are alternative modeling framework supporting the goal-oriented paradigm [25, 9]. These frameworks are dedicated to the analysis of dependencies in socio-technical systems. i* is centered on the structural modeling and does not support formal layer. On the other hand, FormalTropos supports a formal language similar to the one of the KAOS method [9], supporting linear temporal logic for goals, first order logic for pre/post conditions on tasks (called creation/fulfillment conditions) and for invariants on various kinds of objects (such as tasks and resources). This framework also supports formal verification through model checking techniques [10]. However those checks are limited to global checks respectively addressing the detection contradiction, overspecification and underspecification. There is no direct operationalisation relation of goals on other model element and related model-checking capabilities.

6 Conclusion and Future Work

This paper presented a constructive verification-based approach to linking high-level system requirements, expressed as linear temporal logic formulae, to a system specification expressed as an Event-B machine extended with the notion of obligations. The source requirements are included as verification assertions that can be model-checked by tools like ProB, showing that the proposed specification indeed meets the system requirements. The significance of this work is that it integrates goal-driven requirements engineering methodologies with refinement-driven system specification and design methodologies. Such a technique helps in bridging the gap between requirements and formal specifications as well as providing the means for building system designs based on rigorous formal grounds.

The work presented here is part of an on-going effort to help in the industrial adoption of Event-B and of a more specific effort to model security requirements of large-scale distributed systems like Grids [19, 8], to derive rigorously security policies from requirements [20], and to exploit well-established techniques, such as refinement, in the development of these types of systems [5].

Future work will further elaborate the pattern library and explicit proofs of operationalisation correctness. It will consider larger problems, with more complex object and agent models. This would allow us to address the lack of structure in Event-B and to capture the agent model in the specification by tracing agent-goal responsibility relationships. At the tool level, industrial level tools already exist both for goal-oriented requirements engineering [1] and Event-B [2] however integration at the formal level still needs to be developed.

Acknowledgement

The authors would like to thank Renaud De Landtsheer for his helpful comments and review of the draft of this paper. This work is funded by the European Commission under the EU project GridTrust (project reference number 033817). Christophe Ponsard acknowledges financial support from the EU project DEPLOY (project reference number 214158).

References

[1] Objectiver. `http://www.objectiver.com`.

[2] Rodin. `http://sourceforge.net/projects/rodin-b-sharp/`.

[3] J-R. Abrial, M. J. Butler, S. Hallerstede, and L. Voisin. An Open Extensible Tool Environment for Event-B. In *Formal Methods and Software Engineering, ICFEM 2006*, volume 4260 of *Lecture Notes in Computer Science*, pages 588–605. Springer, 2006.

[4] J-R. Abrial and S. Hallerstede. Refinement, Decomposition, and Instantiation of Discrete Models: Application to Event-B. *Fundamenta Informaticae*, 77(1-2):1–28, 2007.

[5] A. E. Arenas, B. Aziz, J. C. Bicarregui, and B. Matthews. Managing Conflicts of Interest in Virtual Organisations. In *3rd International Workshop on Security and Trust Management, STM 2007*. Elsevier, 2007.

[6] J. Bicarregui, A. E. Arenas, B. Aziz, P. Massonet, and C. Ponsard. Toward Modelling Obligations in Event-B. In E. Borger, M. Butler, and J. P. Bowen, editors, *International Conference of ASM, B and Z Users*, volume 5238 of *Lecture Notes in Computer Science*, pages 181–194. Springer, 2008.

[7] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[8] G. Dallons, P. Massonet, J. F. Molderez, C. Ponsard, and A. E. Arenas. An Analysis of the Chinese Wall Pattern for Guaranteeing Confidentiality in Grid-Based Virtual Organisations. In *International Workshop on Security, Trust and Privacy in Grid Systems, Grid-STP 2007*. IEEE, 2007.

[9] A. Fuxman, R. Kazhamiakin, M. Pistore, and M. Roveri. Formal tropos: language and semantics, 2003.

[10] A. Fuxman, L. Liu, J. Mylopoulos, M. Pistore, M. Roveri, and P. Traverso. Specifying and Analyzing Early Requirements in Tropos. *Requirements Engineering*, 9(2):132–150, 2004.

[11] R. Hassan, S. Bohner, S. El-Kassas, and M. Eltoweissy. Goal-Oriented, B-Based Formal Derivation of Security Design Specifications from Security Requirements. In *ARES '08: Proc. of the 2008 Third Int. Conf. on Availability, Reliability and Security*, Washington, DC, USA, 2008. IEEE Computer Society.

[12] M. Joseph. *Real-Time Systems: Specification, Verification and Analysis*. Prentice Hall International, 1996.

[13] E. Letier, J. Kramer, J. Magee, and S. Uchitel. Deriving event-based transition systems from goal-oriented requirements models. *Automated Software Engg.*, 15(2):175–206, 2008.

[14] E. Letier and A. van Lamsweerde. Deriving Operational Software Specifications from System Goals. In *FSE'10: 10th ACM S1GSOFT Symp. on the Foundations of Software Engineering*, 2002.

[15] M. Leuschel and M. Butler. ProB: A Model Checker for B. pages 855–. Springer-Verlag, LNCS, 2003.

[16] M. Leuschel and D. Plagge. Seven at one stroke: LTL model checking for High-level Specifications in B, Z, CSP, and more. Technical report, 2007.

[17] J. Magee and J. Kramer. *Concurrency - State Models and Java Programs*. John Wiley and Sons, 1999.

[18] C. Métayer, J. R. Abrial, and L. Voisin. Event-B Language. Rodin Deliverable D3.2, 2005.

[19] S. Naqvi, P. Massonet, and A. E. Arenas. Security Requirements Model for Grid Data Management Systems. In *Critical Information Infrastructures Security*, volume 4347 of *LNCS*. Springer, 2007.

[20] S. Naqvi, C. Ponsard, P. Massonet, and A. E. Arenas. Security Requirements Elaborations for Grid Data Management Systems. *International Journal of System of Systems Engineering*, 2009. To appear.

[21] C. Ponsard and E. Dieul. From Requirements Models to Formal Specifications in B. In *Proc; of CAISE Workshops, Regulations Modelling and their Validation and Verification, Luxembourg*, June 2006.

[22] C. Ponsard, P. Massonet, J-F. Molderez, A. Rifaut, A. van Lamsweerde, and H. T. Van. Early verification and validation of mission critical systems. *Formal Methods in System Design*, 30(3):233–247, 2007.

[23] C. Snook and M. Butler. UML-B and Event-B: an integration of languages and tools. In *SE2008*, 2008.

[24] A. van Lamsweerde. *Requirements Engineering: From System Goals to UML Models to Software Specifications*. Wiley, 2009.

[25] E. Yu. *Modelling Strategic Relationships for Process Reengineering*. PhD thesis, Department of Computer Science, University of Toronto, 1995.

Introduction of Virtualization Technology to Multi-Process Model Checking

Watcharin Leungwattanakit[*]
University of Tokyo, Tokyo, Japan
watcharin@is.s.u-tokyo.ac.jp

Cyrille Artho
RCIS/AIST, Tokyo, Japan
c.artho@aist.go.jp

Masami Hagiya
University of Tokyo, Tokyo, Japan
hagiya@is.s.u-tokyo.ac.jp

Yoshinori Tanabe
University of Tokyo, Tokyo, Japan
y-tanabe@ci.i.u-tokyo.ac.jp

Mitsuharu Yamamoto
Chiba University, Chiba, Japan
mituharu@math.s.chiba-u.ac.jp

Abstract

Model checkers find failures in software by exploring every possible execution schedule. Java PathFinder (JPF), a Java model checker, has been extended recently to cover networked applications by caching data transferred in a communication channel. A target process is executed by JPF, whereas its peer process runs on a regular virtual machine outside. However, non-deterministic target programs may produce different output data in each schedule, causing the cache to restart the peer process to handle the different set of data. Virtualization tools could help us restore previous states of peers, eliminating peer restart. This paper proposes the application of virtualization technology to networked model checking, concentrating on JPF.

1 Introduction

Recently, software model checking [8] has become widespread. It is not only applied to application models but also actual implementations. However, many modern applications are very complex since they exchange data and interoperate with other entities via a network. Several techniques have been established to verify networked applications by a model checker. Centralization [4, 11] transforms every relevant process into a thread and verifies the application as a single process. However, this technique suffers from the state explosion problem, caused by the large number of interleavings among large numbers of processes and threads. NetStub [6] simplifies the complexity of the process by replacing peer processes with stub objects. The target process then talks with mock processes written only for verification. However, this approach still requires that a user write a stub for each peer process. Finally, Nakagawa et al. [10] proposed a technique that verifies every process in an application inside a multi-process model checker. This technique does not scale well because of the huge number of process interleavings, but the tool has shown that existing virtualization environments are feasible and efficient.

A single-process model checker like Java PathFinder (JPF) [9], without modification, cannot be applied to networked applications since some related processes are running outside the model checker and not backtracked together with the target process. For instance, two threads, T1 and T2, execute code shown in Figure 1. They write a character to and read a character from a peer process. The other side of the figure shows a subset of the state space generated by a model checker. Initially, the model checker executes the entire program under the first schedule (A). However, when it backtracks the program to state 1 and executes another branch, T2 sends a duplicate character to the peer process in the transition to state 4 (B). The duplicate characters may interfere with the correct functionality of the peer process, which is not backtracked. Furthermore, when T1 tries to read a character from the peer again in the transition 4–5 (C), the peer may reply nothing to the target process. This causes T1, and consequently the model checker, to wait forever for input without producing a verification result.

E. Denney, D. Giannakopoulou, C.S. Păsăreanu (eds.); The First NASA Formal Methods Symposium, pp. 106-110

[*]This work was supported by a *kakenhi* grant (2030006).

T1	T2

void run() {
W1: write("1");
R1: read();
}
	void run() {
	W2: write("2");
	R2: read();
	}

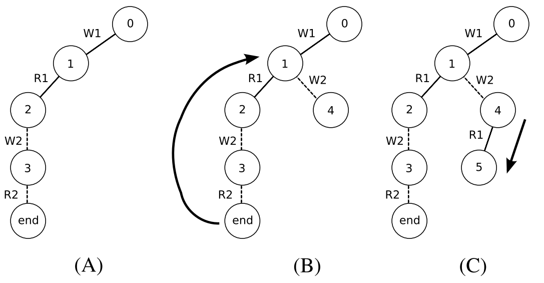


Figure 1: Left: Example networked program. Right: Development of the partial state space.

Our previous work [5] solves this problem by using a cache to capture the data transferred between a program under test and its peer process. Only one process is executed inside the model checker, whereas the other process runs on the host machine in native mode. The cache replays captured data appropriately so that the target program is given all necessary responses. Peers do not realize that they are connected to a process that runs in a model checking environment. The basic concept of the cache works well for simple request-response networked programs, but non-trivial programs need special care.

Some networked programs respond differently each time they receive a request. For example, dynamic web pages are reprocessed every time they are requested; the reproduced pages may differ from each other. Such non-determinism makes it impossible to replay cached data. Our current implementation solves this problem by having the cache create a new instance of the peer process to accept a new communication trace. Instead of restarting, it would be possible to run the peer process inside a virtual environment that the program state can be saved and restored. This method could reduce the time spent in starting a new process and control the number of connections generated by the cache. In this paper, we ignore non-determinism in peers for scalability. The cache approach compromises on soundness by verifying the target process with a subset of possible responses from peers, since we cannot determine the degree of non-determinism of a process running outside the model checker.

2 Concept

2.1 Virtualization

Virtualization is a technique that emulates several computing platforms on one physical machine. The virtualized systems use the computing resources of operating system (OS) on the host machine. There are several types of virtualization: *hardware emulation*, *full virtualization*, *paravirtualization* and *OS-level virtualization*. Hardware emulation creates virtual hardware devices required to run an unmodified guest OS. Full virtualization uses software called *hypervisor* as a mediator between guest systems and physical hardware. Guest systems may access physical hardware directly except for certain protected instructions, which are handled by the hypervisor. Paravirtualization offers better performance [7] by hosting modified guest systems, which are aware of the virtualization environment. OS-level virtualization requires a specially modified kernel to manage guest systems. Every guest is executed on the host OS, yielding nearly the same performance as in native mode. A general architecture of a virtualized system is shown in Figure 2. A *node*[1] represents a physical machine running both host and guest operating systems. The *hypervisor* is the layer of software that manages every guest system to run simultaneously. Each guest operating system is contained in a *domain*, a space that the guest system is allowed to access.

[1] These terms may be used differently in the documentation of some virtualization tools.

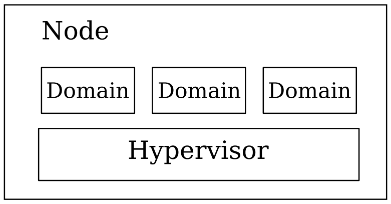


Figure 2: Architecture of a virtualized system.

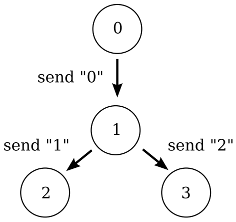


Figure 3: Part of the search space of the non-deterministic alphabet client.

2.2 Peer Processes in Virtualized Environment

Non-deterministic processes may send different data streams for each thread schedule. An example client/server system implemented as an alphabet client/server illustrates this. After connecting to the server, the client writes characters to its output data stream and reads characters from its input data stream. Read and write instructions are called separately by two threads, which are interleaved arbitrarily. The writer thread sends messages in set $N = \{``0", ``1", \ldots, ``9"\}$ to the server. The server side, when receiving a message $m \in N$, replies $f(m)$ to the client where function $f = \{(``0", ``a"), (``1", ``b"), \ldots, (``9", ``j")\}$. If the client sends "0" in the first step and randomly sends "1" or "2" in the next step, the search space that the model checker generates is as shown in Figure 3. After the model checker has finished the left branch, we may create a new connection to the server and re-send "0" followed by "2" to handle the other possibility. However, if the server is run inside a virtualization environment, its state could be stored and re-used later for backtracking.

The common operations of virtualization include `suspend`, `resume`, `kill`, `dump` and `undump`. Operation `suspend` temporarily pauses execution of a guest system, and `resume` starts execution again. `Kill` forcibly shuts down a guest system. `Dump` saves the current state of the running machine as a file, whereas `undump` recovers a program state from a file. Virtualization comes into play when the model checker encounters a decision in the search space. A model checker extension may call `suspend` and `dump` to keep peer's state before the decision. When backtracking, the extension should "kill" the current peer process and call `resume` and `undump` to restore the peer to the state before the decision.

Care should be taken when saving a peer state so that the overhead from dumping the state does not dominate the time saved by preventing peer restarts. A hybrid solution combining caching and virtualization on the same system, may constitute the ideal trade-off. One could dump only some of the states, for example, the states after an I/O operation and/or an expensive operation. As long as the model checker does not backtrack to a state prior to the saved program state, these steps need not be repeated. The interaction with the peer will be only replayed from the last saved state, instead of the beginning, to the present state. Communication data needed for replaying execution can be stored in the cache.

2.3 System Architecture

The virtual machine hosting a domain needs to be configured beforehand. A virtual machine controller, which is a part of our extension, will handle this task. The peer including its configuration must be prepared to run inside a guest system. Users would write a shell script specifying how to start and configure a peer.

Peer processes may run either on a remote machine or the same machine as the target program. In normal testing, the target program connects directly to every peer process on the local host and the remote host. Each peer can access its *environment,* such as the file system and system variables of the host, that it runs on (see Figure 4, left). If the target program is verified by JPF, each peer will be run

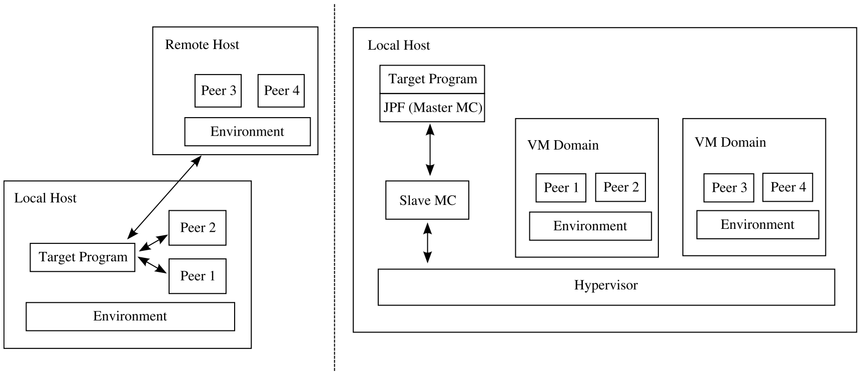


Figure 4: Left: Normal testing setup. Right: Model checking setup.

in the domain representing the host that it originally ran on (see Figure 4, right). Peers 1 and 2 are run in a domain representing the local host, whereas the peers on the remote host, peers 3 and 4, are run in another domain. Both domains, which contain the environment of the peers, are controlled by the same hypervisor. JPF, the *master model checker*, interacts with the *slave model checker*, which acts as an interface to peers. When a certain condition is satisfied, the slave model checker will save the state of every peer via the hypervisor. The slave model checker also restores previous states when JPF backtracks and replays I/O operations if necessary. This method differs from the multi-process model checker approach [10], in which the model checker checks every interleaving among all processes. In our proposed approach, JPF only explores thread interleavings in the target process, whereas the slave model checker synchronizes peer's states without investigating any interleavings.

2.4 Benefits of Virtualization

Creating extra connections would be a considerable overhead for programs with a high degree of nondeterminism. Virtualization could make this process simpler by executing each peer process inside a virtual environment, where it is possible to save and restore the state of the peer process. For example, in Figure 3, the cache saves the state of the server at node 1 before exploring the left branch. This saved state is restored when the model checker backtracks to node 1, where it is about to traverse the right branch. The cache does not need to create extra connections during search thanks to virtualization.

Performance in model checking would be improved when using a peer with high startup or processing time. Some networked programs take a considerable amount of time to load necessary modules and configuration files before being ready to handle requests. If such programs are used as a peer process without the help of a virtualization tool, the model checker will waste a considerable amount of time on restarting the peer.

Furthermore, the state of the peer becomes open to inspection when the peer runs inside a virtualized environment. In the current implementation, the cache waits a certain amount of time for a peer response after it has sent a message [5]. If no response has arrived, it assumes that the message sent so far is not a complete request that elicits a peer response. However, this assumption is not always true. Responses may come late due to slow request processing, or network congestion in case of the peer running remotely. It is difficult, if not impossible, to determine the status of a peer running in an environment that we do not control. Future work includes development of a method to determine whether a peer has finished producing output.

3 Conclusion and Ongoing Work

This paper shows the application of virtualization technology to multi-process model checking. The idea could be applied on any model checking system that executes actual processes during verification. Peer processes are executed under virtual environments rather than the host operating system. When a model checker backtracks the verified process, creation of extra connections is not necessary. Instead, the program state is recorded occasionally and is restored when backtracking. This idea will be implemented in a JPF extension called *net-iocache*.

This extension for JPF 4 has been developed to verify networked applications without virtualization, which is ongoing work [5]. We are now studying the strengths and weaknesses of each virtualization tool. Important factors like the speed of saving/restoring system states and size of a saved state will be tested and measured before selecting a tool. It is also preferable to store program states in memory rather than a disk to minimize overhead. Unfortunately, we have not found such a function in any tools that we are studying. We may consider adding this feature if necessary. Our development platform is Ubuntu Linux 8.04, for which the *Linux Kernel Virtual Machine* (KVM) [1] is one of the candidates of virtualization tools. It allows guest operating systems to run in the user-space of the host kernel. Although it requires a certain feature of hardware, KVM gives us a performance of guest programs close to running natively on the host system. Other candidates include Xen [3] and OpenVZ [2].

References

[1] KVM documentation. `http://kvm.qumranet.com/kvmwiki`.

[2] OpenVZ documentation. `http://wiki.openvz.org`.

[3] Xen web page. `http://www.xen.org`.

[4] C. Artho and P. Garoche. Accurate centralization for applying model checking on networked applications. In *Automated Software Engineering Conf.*, Tokyo, Japan, 2006.

[5] C. Artho, W. Leungwattanakit, M. Hagiya, and Y. Tanabe. Efficient model checking of networked applications. In *Proc. TOOLS EUROPE 2008*, volume 19 of *LNBIP*, pages 22–40, Zurich, Switzerland, 2008. Springer.

[6] Elliot D. Barlas and Tevfik Bultan. NetStub: A framework for verification of distributed Java applications. In *Automated Software Engineering Conf.*, Atlanta, Georgia, USA, 2007.

[7] V. Chaudhary, Minsuk Cha, J.P. Walters, S. Guercio, and S. Gallo. A comparison of virtualization technologies for HPC. *22nd International Conference on Advanced Information Networking and Applications*, pages 861–868, March 2008.

[8] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[9] Klaus Havelund and Thomas Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.

[10] Y. Nakagawa, R. Potter, M. Yamamoto, M. Hagiya, and K. Kato. Model checking of multi-process applications using SBUML and GDB. In *Workshop on Dependable Software: Tools and Methods*, pages 215–220, Yokohama, Japan, 2005.

[11] Scott D. Stoller and Yanhong A. Liu. Transformations for model checking distributed Java programs. In *SPIN '01: Proceedings of the 8th international SPIN workshop on Model checking of software*, pages 192–199, New York, NY, USA, 2001. Springer-Verlag New York, Inc.

Comparing Techniques for Certified Static Analysis*

David Cachera[†]
ENS Cachan, IRISA,
Rennes, France

David Pichardie
INRIA, Centre Rennes - Bretagne Atlantique,
Rennes, France

Abstract

A certified static analysis is an analysis whose semantic validity has been formally proved correct with a proof assistant. The recent increasing interest in using proof assistants for mechanizing programming language metatheory has given rise to several approaches for certification of static analysis. We propose a panorama of these techniques and compare their respective strengths and weaknesses.

1 Introduction

Nowadays safety critical systems are validated through long and costly test campaigns. Static analysis is a promising complementary technique that allows to automatically prove the absence of restricted classes of bugs. A significant example is the state-of-the-art ASTRÉE static analyzer for C [11] which has proven some critical safety properties for the primary flight control software of the Airbus A340 fly-by-wire system. Taking note of such a success, the next question is: should we completely remove the test campaign dedicated to the same class of bugs? If we trust the result of the analyzer, of course the answer is yes, but should we trust it? The analyzer itself can be certified by testing, but exhaustivity cannot be achieved. In this paper, we show how mechanized proofs can be used to certify static analyzers or their results.

Abstract interpretation [10] is a general theory that aims at designing provably correct static analyzers, but pencil-and-paper proofs hardly scale to real-size analyzers for real-size programming languages. Proof assistants allow to mechanically specify, program and prove correct static analyzers with respect to a formal model of the programming language semantics. If the feasibility of such a technique has been demonstrated for various kinds of analyses and programming languages [13, 2, 7, 18, 9, 5, 20], many approaches coexist and some of them differ in the kind of guarantee they give on the targeted static analysis. In this work, we make a comparison between different techniques, taking into account the proof effort, the obtained guarantee and maintenance problems. The paper is organized as follows: we first show how an analysis can be specified depending on the expected guarantees. We then address the question of computing a certified solution of the analysis. Finally we investigate the use of deductive verification to validate the invariant generated by an analysis.

For presentation purposes, all the analyzes we describe here will share a common semantical basis, which we give below, together with a summary of abstract interpretation principles. A program P is a graph (N,E) where $N \geq 1$ is the number of vertices (control points), and E a set of edges $(n,m) \in \{1..N\} \times \{1..N\}$. Each edge is labeled by an instruction $i_{n,m}$ from a set $\mathbb{I}$. Among nodes, we distinguish an entry point n_e such that there is no incoming edge in n_e. A *state* of a program P is composed of a control point n and an environment ρ: $\mathrm{State} = \{1..N\} \times \mathrm{Env}$. The *concrete semantics* of an instruction $i \in \mathbb{I}$ is given by a binary relation $\rightarrow_i$ over Env. The transfer function $F_i : \mathscr{P}(\mathrm{Env}) \rightarrow \mathscr{P}(\mathrm{Env})$ associated to instruction i is then defined by $F_i(S) = \{\rho' \mid \exists \rho \in S : \rho \rightarrow_i \rho'\}$. Given S_0 an initial set of environments, the collecting semantics $[\![P]\!]^c$ of P is the least solution $X \in \mathscr{P}(\mathrm{Env})^N$ of $S_{n_e} = S_0$ and $\forall m \in \{1..N\}$, $S_m = \bigcup_{(n,m)\in E} F_{i_{n,m}}(S_n)$.

The abstract interpretation formalism gives us a way to over-approximate the solution to these equations. An abstract semantics is expressed w.r.t. an abstract domain[1] $\mathrm{Env}^\sharp$ (usually a complete lattice

E. Denney, D. Giannakopoulou, C.S. Păsăreanu (eds.); The First NASA Formal Methods Symposium, pp. 111-115

*Work partially supported by EU project MOBIUS, ANR-U3CAT grant and FNRAE ASCERT grant.

[†]Currently delegated as a full time researcher at INRIA, Centre Rennes - Bretagne Atlantique.

[1] Here we focus on the specific case where the abstract states are mappings from $\{1..N\}$ to $\mathrm{Env}^\sharp$.

$(\mathrm{Env}^\sharp, \sqsubseteq_\sharp)$), and the relation between concrete and abstract semantics is given by a Galois connection $(\alpha, \mathscr{P}(\mathrm{Env}), \mathrm{Env}^\sharp, \gamma)$, *i.e.* a pair of monotone mappings such that $\forall S \subseteq \mathrm{Env} : S \subseteq \gamma(\alpha(S))$ and $\forall d \in \mathrm{Env}^\sharp : \alpha(\gamma(d)) \sqsubseteq_\sharp d$. The *abstraction function* α maps a set S of environments to an abstract environment, which can be seen as the least property satisfied by all elements of S. The *concretization function* γ maps an abstract environment to all the concrete environments satisfying the corresponding property. We are interested in computing an abstract semantics $[\![P]\!]^\sharp : \{1..N\} \to \mathrm{Env}^\sharp$ which is a certified *correct (over-)approximation* of the concrete collecting semantics, *i.e.* $\forall i \in \{1..N\} : [\![P]\!]^c(i) \subseteq \gamma([\![P]\!]^\sharp(i))$.

In addition to the systematic treatment of this safety issue, the theory provides an optimal specification of the abstract semantics: given a mapping $F : \mathscr{P}(\mathrm{Env}) \to \mathscr{P}(\mathrm{Env})$, a correct abstraction of F is a mapping $F^\sharp$ verifying $F \circ \gamma \subseteq \gamma \circ F^\sharp$, or equivalently $\alpha \circ F \circ \gamma \sqsubseteq_\sharp F^\sharp$. The case where $F^\sharp = \alpha \circ F \circ \gamma$ thus provides the best correct abstraction of F. The abstract semantics computed by the analyzer will then mimic the collecting semantics, in the sense that it also operates through abstract transfer functions $F_i^\sharp$. The result $[\![P]\!]^\sharp$ of the analysis is thus the least solution of

$$\alpha(S_0) \sqsubseteq_\sharp S^\sharp_{n_e} \quad \text{and} \quad \forall (n,m) \in E, F^\sharp_{i_{n,m}}(S^\sharp_n) \sqsubseteq_\sharp S^\sharp_m \tag{1}$$

and each $F_i^\sharp$ is proved an *optimal* correct abstraction of F_i. In order to save space, all properties like $\alpha(S_0) \sqsubseteq_\sharp S^\sharp_{n_e}$ dealing with the treatment of initial states will be discarded from the rest of this paper.

2 Analyzer Specification

Analysis soundness must be established with respect to the concrete semantics using a correctness relation that relates the concrete and the abstract domains. In this section we consider two formal frameworks that both enforce the following essential soundness property: any solution $S^\sharp$ of (1) is a correct approximation of $[\![P]\!]^c$. Various ways can be taken between these two opposite approaches, some of them have been investigated in [13, 7, 18, 9, 5].

2.1 Deep Analyzer Specification

This first approach (so far only experimented in [15]) exactly follows the Galois connection formalism by providing mechanized proofs of all the classical properties. We briefly recall the components of a static analyzer based on this formalism and make more precise the proof requirements.

component	mathematical structure	properties to prove
abstract domain	complete lattice $(\{1..N\} \to \mathrm{Env}^\sharp, \dot\sqsubseteq^\sharp, \dot\sqcup^\sharp, \dot\sqcap^\sharp)$	existence of a lub
correctness relation	Galois connection $(\alpha, \mathscr{P}(\mathrm{Env}), \mathrm{Env}^\sharp, \gamma)$	Galois connection definition
abstract semantics	abstract transfer functions	$F_i^\sharp = \alpha \circ F_i \circ \gamma$

Defining the abstract domain as a complete lattice constrains us to provide a proof of existence of a least upper bound for any subset of abstract elements (or a least fixpoint for any monotone function). Moreover, from a proof assistant point of view, the $\sqcup^\sharp$ operator is not constructive, which hampers its implementation. Taking a Galois connection as a correctness criterion also increases the number of proofs to be done, since this also provides an optimality property.

Let us take an example to illustrate this point. We consider a numerical abstraction based on the domain of intervals. The concrete domain is $(\mathscr{P}(\mathbb{Z}), \subseteq, \cup, \cap)$, and the abstract domain is the lattice of intervals $\{[a,b] \mid a \in \mathbb{Z} \cup \{-\infty\}, b \in \mathbb{Z} \cup \{+\infty\}, a \leq b\} \cup \bot_{\mathrm{Int}}$. The corresponding abstraction and concretization functions are defined by the following equations

$$\begin{array}{rcl} \gamma(\bot_{\mathrm{Int}}) & = & \emptyset \\ \gamma([a,b]) & = & \{x \in \mathbb{Z} \mid a \leq x \leq b\} \end{array} \qquad \alpha(P) = \left\{ \begin{array}{l} \bot_{\mathrm{Int}} \text{ if } P = \emptyset \\ [\inf(P), \sup(P)] \text{ otherwise} \end{array} \right.$$

If we want to compute $\alpha \circ F \circ \gamma$ for a given computable function F, the use of α might force us to provide a proof that a given subset of $\mathbb{Z}$ is not bounded to ensure that $\top = [-\infty, +\infty]$ is a correct result of the analysis. If no optimality property were involved, $\top$ could be taken as a correct result in any case.

2.2 Shallow Analyzer Specification

A second approach consists in focusing only on the soundness of the analysis, without considering the optimality issue. The choices made here are directly inspired from [12], where the authors describe the design of the ASTRÉE static analyzer.

component	**mathematical structure**	**properties to prove**
abstract domain	$(\mathrm{Env}^\sharp, \sqsubseteq_\sharp, \sqcup^\sharp, \sqcap^\sharp)$	no requirements on $\sqsubseteq_\sharp, \sqcup^\sharp$ or $\sqcap^\sharp$
correctness relation	$\gamma : \mathrm{Env}^\sharp \to \mathscr{P}(\mathrm{Env})$	$\forall c, d \in \mathrm{Env}^\sharp, c \sqsubseteq_\sharp d \Rightarrow \gamma(c) \subseteq \gamma(d)$
abstract semantics	abstract transfer functions	soundness: $F_i \circ \gamma \subseteq \gamma \circ F_i^\sharp$

If the shallow framework requires far less machine proofs than the deep framework, it ensures only a minimal amount of properties on the analysis which is doubtless sound but may still contain several precision bugs that are notoriously hard to debug.

3 Result Computation

The requirements made during the previous phase ensure that any solution of (1) is a correct approximation of $[\![P]\!]^c$, but do not specify *how* it is computed. One has to choose between various certification levels: from a complete proof of the whole analysis computation to a result-only certification.

3.1 Termination

If we aim at certifying the whole analyzer, we have to prove the termination of an algorithm computing a solution of (1). Termination proofs are known to be difficult and are seldom mechanized, or even precisely formalized. Even if we consider a complete lattice, the existence of a least fixpoint for monotone functions does not ensure the convergence of the computation in finite time. Except for the trivial case of finite height lattices, we have to exhibit specific properties of the domain, or to design operators that will ensure convergence.

A first approach consists in certifying that the lattice respects the ascending chain condition, *i.e.* that any increasing chain stabilizes in finite time. The main drawback of this approach is that it does not apply to popular abstract domains like intervals or polyhedra. A more common approach consists in designing widening and narrowing operators in order to accelerate the convergence [10]. In both approaches, a mechanized proof of termination has to cope with constructivity issues, and the criteria of ascending chain or termination of widening-based iteration have to be modified in consequence. In any case, a key issue for termination proofs is to provide modular lattice constructions, thus allowing for building a global proof out of basic blocks (usual numerical abstract domains for instance) [13, 17].

If one wants to avoid to perform tedious termination proofs, it is also possible to artificially bound the number of iterations in the abstract semantics computation [14]. In that case, one has to certify that any iteration yields a correct result, even if not the best one, or to check that the final result is indeed a correct approximation of the concrete semantics. This technique leads us to result certification.

3.2 Result-only Certification

In a safety-critical context, it is likely that the high confidence we are looking for is not for all results of the analyzer but for a few specific ones like those obtained for the next version of the flight-command program. Instead of globally certifying the analyzer itself, it might be interesting to provide a tool that checks the correctness of its result. This approach is directly related to the Proof Carrying Code technique [16, 1], where the code producer provides a formal proof that the code respects some safety requirements defined by the end-user. The user then verifies the proof with an automated and trusted checker. This use of a PCC technique for analysis certification has been proposed in [6]. The main requirement is the same as in the previous approaches: we still have to give a proof that $F_i \circ \gamma \subseteq \gamma \circ F_i^\sharp$. But now, instead of computing a solution of (1), we just have to check that a given certificate $S^\sharp$ is indeed a solution. Note that this proof can be automatically discharged by a computation.

4 Deductive Verification of Analysis Results

Since the main goal of static analysis is to generate invariants over program executions, it is natural to try to validate these invariants with deductive verification techniques that are traditionally applied for handwritten program invariants. In this section, we assume an axiomatic semantics given by a deductive judgment $\vdash \{\phi\}\ i\ \{\psi\}$ for each instruction $i \in \mathbb{I}$ such that, when property ϕ holds before executing i, ψ must hold after. Here, ϕ and ψ are formulas in a given logic language $\mathscr{L}$. We denote by $\pi : \{\phi\}i\{\psi\}$ a Hoare-proof derivation π that is a valid proof of $\vdash \{\phi\}\ i\ \{\psi\}$. We note $\rho \models \phi$ when an environment ρ satisfies property ϕ. To validate a set of invariants $\phi_1, \dots, \phi_N$ attached to each control point, it is sufficient to provide a set of Hoare proofs $\pi_{n,m} : \{\phi_n\}\ i_{n,m}\ \{\phi_m\}$ for all (n,m) in E.

An analysis result has to be first transformed into a set of assertions in the language $\mathscr{L}$. We thus assume that each abstract element $a^\sharp \in \mathrm{Env}^\sharp$ can be translated into a formula $\ulcorner a^\sharp \urcorner$ in $\mathscr{L}$. In this setting, an analysis result $S^\sharp$ is certified once the following statements have been formally machine checked.

Hoare logic soundness	$\vdash \{\phi\}\ i\ \{\psi\}$ implies $(\forall \rho, \rho', \rho \rightarrow_i \rho'$ and $\rho \models A$ implies $\rho' \models B)$
Provability of assertions	$\forall (n,m) \in E,\ \{\ulcorner S_n^\sharp \urcorner\}\ i_{n,m}\ \{\ulcorner S_m^\sharp \urcorner\}$is provable

The advantage of the approach is that the soundness of Hoare logic can be proved once and for all, and used for several static analyses. If we assume that the Hoare logic is not only sound but also complete, that transformation $\ulcorner \cdot \urcorner$ preserves satisfiability (*i.e.* for all $\rho \in \mathrm{Env}$ and $a^\sharp \in \mathrm{Env}^\sharp$, $\rho \in \gamma(a^\sharp)$ iff $\rho \models \ulcorner a^\sharp \urcorner$), and that each transfer function $F_i^\sharp$ is sound w.r.t. F_i, then the corresponding set of Hoare triples is provable for any solution $S^\sharp$ of the analysis. However, a proof of these triplets still has to be constructed, without entering a painful manual process for each of them. A first approach, proposed by Seo *et. al.* [19], instruments the analyzer to make it produce a proof derivation $\pi_{n,m}$ for all edges (n,m). This approach has also been followed by Beringer *et. al.* [3, 4] who translate type derivations into Hoare proofs. The approach proposed independently by Chaieb [8] relies on a weakest precondition computation. This time the analyzer is instrumented to produce proof terms for the verification conditions generated by a weakest precondition computation. If these approaches are elegant, they remain difficult to implement because generating proof terms requires more technical ability than transposing a pencil-and-paper proof into a proof assistant.

Ideally, the proof obligations (obtained for example with the weakest precondition calculus) should be automatically discharged by a trustworthy theorem prover, hence the uselessness of generating proofs. However, each analysis may require specific decision procedures. Instead of producing a specific proof for each verification condition, we believe it may be a better idea to strengthen the theorem prover with a decision procedure able to discharge exactly this kind of formula. In addition to validating the analysis result, it is likely to improve the prover itself. Since each analyzer addresses dedicated decision procedures in its transfer functions and partial order tests, it would be useful to share this capability with an automatic prover. The research line we advocate here is then to design abstract domains and decision procedures in parallel. To ensure the validity of the approach, the decision procedures must themselves be certified, *i.e.* must generate proof terms of validity.

5 Conclusion

We have considered several techniques for certifying the soundness of a static analyzer or of its result. Of course, there is no *silver-bullet* technique: if the deep approach is the most greedy in terms of proof effort, it is the only one that detects precision bugs. Revealing such bugs too late during a validation campaign may compromise the availability of the safety critical system that has to be validated in due time. Deductive verification appears as a promising technique but its apparent generality must be tempered: the underlying logic is not always expressive enough to translate the result of the analysis and automatically discharging verification conditions requires technical instrumentation of the analyzer. Building abstract domains and decision procedures in parallel seems an interesting research line to push further.

References

[1] E. Albert, G. Puebla, and M. Hermenegildo. Abstraction-carrying code. In *In Proc. of LPAR'04*, pages 380–397. Springer LNAI vol 3452, 2005.

[2] G. Barthe, G. Dufay, L. Jakubiec, and S. Melo de Sousa. A formal correspondence between offensive and defensive javacard virtual machines. In *VMCAI*, pages 32–45. Springer-Verlag LNCS vol. 2294, 2002.

[3] L. Beringer, M. Hofmann, A. Momigliano, and O. Shkaravska. Automatic certification of heap consumption. In *Proc. of LPAR 2004*, pages 347–362. Springer LNCS vol. 3452, 2004.

[4] L. Beringer, M. Hofmann, and M. Pavlova. Certification using the Mobius base logic. In *Proc. of FMCO'07*, pages 25–51. Springer LNCS vol. 5382, 2007.

[5] Y. Bertot, B. Grégoire, and X. Leroy. A structured approach to proving compiler optimizations based on dataflow analysis. In *Proc. of TYPES'04*, pages 66–81. Springer LNCS vol. 3839, 2006.

[6] F. Besson, T. Jensen, and D. Pichardie. Proof-Carrying Code from certified abstract interpretation to fixpoint compression. *Theoretical Computer Science*, 364(3):273–291, 2006.

[7] D. Cachera, T. Jensen, D. Pichardie, and V. Rusu. Extracting a data flow analyser in constructive logic. *Theoretical Computer Science*, 342(1):56–78, 2005.

[8] A. Chaieb. Proof-producing program analysis. In *Proc. of ICTAC 2006*. Springer LNCS vol. 4281, 2006.

[9] S. Coupet-Grimal and W. Delobel. A uniform and certified approach for two static analyses. In *Proc. of TYPES'04*, pages 115–137. Springer LNCS vol. 3839, 2006.

[10] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of POPL'77*, pages 238–252. ACM Press, 1977.

[11] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE analyser. In *Proc. of ESOP'05*, pages 21–30. Springer LNCS vol. 3444, 2005.

[12] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Combination of abstractions in the ASTRÉE static analyzer. In *Proc. of ASIAN'06*. Springer LNCS vol. 4435, 2007.

[13] G. Klein and T. Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. *ACM Transactions on Programming Languages and Systems*, 28(4):619–695, 2006.

[14] X. Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *Proc. of POPL'06*, pages 42–54. ACM Press, 2006.

[15] D. Monniaux. Réalisation mécanisée d'interpréteurs abstraits. Rapport de DEA, Université Paris VII, 1998.

[16] G. C. Necula. Proof-carrying code. In *Proc. of POPL'97*, pages 106–119. ACM Press, 1997.

[17] D. Pichardie. Building certified static analysers by modular construction of well-founded lattices. In *Proc. of FICS'08*, volume 212 of *Electronic Notes in Theoretical Computer Science*, pages 225–239, 2008.

[18] A. Salcianu and K. Arkoudas. Machine-Checkable Correctness Proofs for Intra-procedural Dataflow Analyses. In *Proc. of COCV'05*, pages 53–68. Elsevier ENTCS vol. 141:2, 2005.

[19] S. Seo, H. Yang, and K. Yi. Automatic construction of Hoare proofs from abstract interpretation results. In *Proc. of APLAS 2003*, pages 230–245. Springer, 2003.

[20] M. Wildmoser, A. Chaieb, and T. Nipkow. Bytecode analysis for proof carrying code. In *Proc. of Bytecode'05*, pages 19–34. Elsevier ENTCS vol. 141:1, 2005.

Towards a Framework for Generating Tests to Satisfy Complex Code Coverage in Java Pathfinder *

Matt Staats
Department of Computer Science and Engineering
University of Minnesota
staats@cs.umn.edu

Abstract

We present work on a prototype tool based on the JavaPathfinder (JPF) model checker for automatically generating tests satisfying the MC/DC code coverage criterion. Using the Eclipse IDE, developers and testers can quickly instrument Java source code with JPF annotations covering all MC/DC coverage obligations, and JPF can then be used to automatically generate tests that satisfy these obligations. The prototype extension to JPF enables various tasks useful in automatic test generation to be performed, such as test suite reduction and execution of generated tests.

1 Introduction

When using a rigorous coverage criterion to generate tests, significant resources are often required to create tests that satisfy the criterion. Approaches for reducing the cost of generating test cases are thus of great value. One such approach is to automate the test case generation process by using a model checker to generate the tests. While several variations of this approach exist [2, 5, 10], the approach generally operates in two steps. First, the required coverage obligations are expressed as either states or sequences of states of the system for which we wish to generate tests. Second, for each state or sequence of states corresponding to a coverage obligation, the model checker is used to determine the reachability of the state or the existence of the sequence of states. If the model checker determines that no states representing a particular coverage obligation are reachable (a plausible occurrence for a complex coverage criterion), the tester has conclusive proof that said obligation is not coverable; otherwise, the model checker emits a trace that can be used as a test case. This approach is largely indifferent to the coverage criterion used – essentially the same approach can be used to generate tests for any coverage criterion that can be expressed in terms of states. Developers can reuse the state space exploration functionality provided by a model checker to quickly create test generation capability for a variety of coverage criteria.

In this paper, we describe the development of a prototype tool built on the Java Pathfinder (JPF) model checker that allows developers to generate and execute tests for the MC/DC coverage criterion [3]. The tool has been developed as two components: an Eclipse plugin allowing developers and testers to instrument Java source files with MC/DC coverage obligations and an extension to JPF providing the functionality needed to generate test cases from said instrumentation.

While the prototype is useful in its current state, it was developed primarily to explore the use of JPF for automatic test generation for reactive systems common to the safety critical domain. Our interest in using JPF over other model checkers (as used in our previous work [5, 8, 10]) stems from its structure; unlike many model checkers, JPF has been developed to be understandable and extensible. For this reason, we believe that JPF may be an excellent model checker for rapidly developing and empirically testing coverage criteria, and we view this prototype as the first step towards an extensible framework for the rapid development of automatic test generation functionality in JPF. This framework would be of use not only for developers and testers who want to build automatic test generation functionality for new and currently unsupported coverage criteria, but also for software researchers exploring the effectiveness of various coverage criteria.

E. Denney, D. Giannakopoulou, C.S. Păsăreanu (eds.); The First NASA Formal Methods Symposium, pp. 116-120

*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.

2 Java Pathfinder

Java Pathfinder is a model checker for the Java language built on top of a custom Java Virtual Machine (JVM). Model checking is done via execution of Java bytecodes, an approach that allows different bytecode interpretations to be developed. This ability is used to support two major model checking modes in JPF: an explicit-state mode similar to SPIN [6], and a symbolic execution mode aptly named *Symbolic JPF* [1]. These modes differ primarily in how the state space is explored. During explicit-state execution, bytecode instructions are executed as in standard JVMs, such as the official Sun JVM. Nondeterminism can be either explicitly introduced by developers and testers as source code annotations in the program being analyzed or implicitly introduced when analyzing multi-threaded programs. JPF handles nondeterminism by branching the execution for each possible nondeterministic choice. Through this branching, the entire state space of the program being analyzed is explored. Multiple search algorithms are available and operate by influencing the order in which JPF explores branches (note that the default search is depth-first).

Symbolic JPF uses an extended interpretation of bytecodes to work with symbolic values, and contains functionality to create symbolic values in addition to concrete values. Operations over symbolic values are done algebraically in a straightforward manner. (The details of symbolic execution are omitted here; the interested reader is referred to [7] for specifics.) When Symbolic JPF encounters conditional branches incorporating symbolic values, it attempts to determine if the branch condition is satisfiable for both *true* and *false* possibilities using off-the-shelf decision procedures. The execution paths of any satisfiable possibilities are then explored, thus branching the search. Note that multi-threading, source code annotations (nondeterministic and otherwise) and operations over concrete values are handled as in explicit-state execution.

A substantial number of helper functions and classes have been developed for JPF. These allow developers and testers to annotate code (including the nondeterministic choice annotations mentioned previously), and allow extensions to modify and monitor the execution of JPF. One of the more powerful pieces of functionality is the ability to register Java listeners for various JPF events, such as execution of a bytecode instruction or the processing of a non-deterministic choice. This functionality allows extensions to access essentially the same information available internally to JPF. As we will see later, both the ability to annotate code and the ability to monitor JPF's execution greatly aides in the development of test generation functionality.

2.1 Existing Test Generation Functionality

JPF contains some preexisting functionality useful for generating tests. When generating tests exclusively using concrete execution mode, source code annotations are generally used. Nondeterministic choice annotations, such as *Verify.getBoolean()* and *Verify.getInt()*, provide a simple way for testers to simulate receiving inputs from the user and/or environment. Generating tests is accomplished through *storeTraceIf(condition, fileName)* statements. These statements, if executed when *condition* evaluates to *true*, output to *fileName* a list of values chosen for each nondeterministic choice. In principle, this list, or *trace*, can be examined to determine an execution path in which *condition* is true at the point *storeTraceIf* produced the trace. Once the execution path is identified, constructing a test should be relatively easy. In practice, however, these traces are only marginally useful, since (1) there exists no method of running or replaying the trace and (2) the trace is output in a format that is not readily understandable.

Using Symbolic JPF to generate tests can be done by assigning symbolic, rather than concrete, values to one or more variables. When Symbolic JPF is used to analyze the program, *path conditions* that algebraically represent the state of variables are produced. By solving these path conditions, concrete values satisfying the path conditions can be found. Thus, Symbolic JPF can be used to generate tests by finding a path of interest (e.g., a path leading to a coverage obligation) and solving the resulting path conditions. In fact, Symbolic JPF includes functionality for finding all paths within a method (up to some maximum length) and generating a test for each path, thus yielding path coverage over the method. However, while Symbolic JPF can be very effective for generating tests, no method of assigning symbolic values within a loop exists, thus precluding test generation for reactive systems (which are constructed as essentially large infinite loops).

3 Prototype MC/DC Test Generation Tool

We chose to develop the prototype to generate tests for the MC/DC coverage criterion because (1) it is a fairly complex coverage criterion and thus presents at least somewhat of a challenge to implement and (2) it is used in critical systems software such as avionics software. We do not describe the MC/DC coverage criterion in this paper; interested readers are directed to [3].

The prototype addresses some of the limitations present in JPF's existing test generation capability and introduces new functionality. The tool consists of two components: an extension to JPF that improves code annotations and adds support for a variety of common test generation tasks, and a plugin for the Eclipse IDE [4] to instrument Java source code with MC/DC test obligations.

3.1 JPF Extension

We developed the extension to JPF with two goals in mind. First, improve the usability and understandability of tests generated via code annotations. Second, provide functionality allowing tasks common in automatic testing such as test suite reduction and code coverage measurement to be performed directly in JPF.

Improving the code annotations was a fairly straightforward task. As we explained in Section 2, the *Verify* class can be used to generate traces via code annotations, but the traces generated are not practical for use as tests. We therefore re-implemented the code annotations to allow more useful traces to be generated. This resulted in a *Debug* class containing three key improvements to the code annotations.

First, we created choice annotations that produce symbolic values (i.e., we created the symbolic equivalents of *Verify.getInt()*) thus allowing developers and testers to explicitly assign symbolic values at any point in their source code. These annotations can be safely used within loops, thus bringing the benefits of symbolic execution to reactive systems.

Second, we extended choice annotations to allow them to be "tagged" with a string by developers and testers. Provided the developer or tester chooses tags which are meaningful (e.g., "*className.variableName*"), traces produced are far more understandable. Furthermore, these tags offer a simple way to associate information useful to JPF with a choice annotation.

Finally, we added the ability to execute traces generated by the *Debug* class. When starting JPF, developers or testers can specify a trace previously generated by *Debug*. During JPF's startup, the specified trace is parsed by *Debug*, which then creates a queue of values for each tag in the trace. After JPF begins running, when a choice annotation is executed, *Debug* checks the tag associated with the annotation and pops a value from the appropriate queue. If the appropriate queue is empty the standard choice annotation logic is used. Executing a trace thus results in JPF following the execution path implied by the trace until reaching the end of said path, after which JPF's usual operation resumes.

These code annotation improvements allow useful, executable tests to be generated using JPF. Given some method to instrument code for a coverage metric (which we provide as an Eclipse plugin, described next) they provide a viable method of generating tests for Java code. However, when using automated test generation, there are several tasks beyond test generation developers may wish to perform, including the ability to measure code coverage, reduce test suite size while maintaining code coverage, and execute entire test suites automatically (e.g., when performing regression testing). Rather than force developers and testers to develop ad-hoc methods of accomplishing these tasks, we provide functionality for these tasks in the form of JPF command line options.

We implemented much of this additional support by mimicking JPF's extensive use of Java listeners; specifically, we implemented a generic *DebugListener* class that can be registered with the *Debug* class to listen for certain test generation-related events during JPF's execution. Additionally, we created a test harness to execute tests in a test suite sequentially. By combining the test harness with listeners tracking *storeTraceIf* events, we perform test suite reduction and code coverage measurement for the MC/DC test coverage criterion. Note that while this functionality has been developed for generating and reducing test suites for the MC/DC coverage criterion, it could be easily adapted to other coverage criteria; most of the functionality exists in generic, extensible classes (in fact, it is only the tagging conventions used by the MC/DC Eclipse plugin that make this functionality coverage specific).

For complex coverage criteria such as MC/DC, even modestly sized systems may require thousand of coverage obligations to be generated, converted into the appropriate *Debug.storeTraceIf* statements, and inserted into the code at the appropriate points. Manually performing this is both tedious and error prone; thus while the functionality outlined above provides an approach for automatically generating tests, it is of limited use without a method of automatically instrumenting the source code to generate tests.

3.2 MC/DC Instrumentation Eclipse Plugin

The Eclipse plugin automates the tedious work of instrumenting a Java source file with the annotations needed to generate tests satisfying the MC/DC coverage criterion. Using the plugin is simple: when a user right clicks on a Java source code file in Eclipse, the options "Instrument to MC/DC Coverage" and "Remove MC/DC Instrumentation" are presented (in addition to the many other options Eclipse offers). When instrumenting, *Debug.storeTraceIf(obligation, fileName, tag)* statements are inserted into the source file, one for each MC/DC obligation. Each statement is inserted prior to the condition from which the *obligation* is generated from. Each *tag* contains a unique number for its corresponding obligation as well as the total number of obligations generated (used for calculating coverage). Given these obligations and the functionality described above, JPF can be used to (1) generate test cases meeting these obligations, (2) measure the MC/DC coverage achieved by an arbitrary set of test cases and (3) reduce a set of test cases while providing the same level of MC/DC coverage.

We have implemented this using Eclipse's Java abstract syntax tree. When instrumenting a Java source file, the file is parsed into an abstract syntax tree and each node in the tree is visited. When a condition node is encountered, the MC/DC obligations corresponding to the condition are generated using the method described in [10] and the appropriate *Debug.storeTraceIf(obligations, fileName, tag)* statements are inserted. Removal of the instrumentation is performed using a similar search.

4 Future Work and Conclusion

We view this prototype as the first step towards a framework for rapidly implementing automatic test generation capability for arbitrary coverage criteria in JPF. The key to this proposed framework is an approach (or combination of approaches) for quickly implementing automatic test generation for most coverage criteria using JPF. Though we have not conclusively identified a suitable approach, the development of our prototype has yielded several possibilities, including an Eclipse-based JPF code annotation library and a JPF extension for generating and monitoring code obligations during JPF's execution.

The development and use of a code annotation library seems well suited for structural coverage criteria. Many structural coverage criteria are designed to generate coverage obligations for specific constructs in the source code, and thus the approach used to annotate for one coverage criterion could be adopted for many other coverage criteria. For example, by modifying what obligations are generated for each condition, the MC/DC Eclipse plugin could be adapted to generate code annotations for branch or decision coverage.

The ability to generate and monitor code obligations during JPF's execution is appealing from an end user perspective, as it frees users from requiring anything other than JPF. However, note that while most structural coverage criteria are expressed in terms of source code constructs, JPF operates exclusively over bytecodes. This complicates attempts to monitor source code constructs, as a single construct (say, a branch) may correspond to multiple bytecodes. Determining if a branch evaluates to true or false thus requires understanding how the path taken through the bytecodes corresponds to the path taken through the source code.

As mentioned in the introduction, we are interested in using JPF to automatically generate tests for testing safety critical systems. In particular, we are interested in working with our collaborators at NASA Ames to explore potential coverage criteria for Java code generated from Simulink using translation tools developed at Vanderbilt University [9]. To perform this exploration, we plan to use our proposed framework to quickly develop the automatic test generation capability for potential coverage criteria. Using this automatic test generation capability, we plan to then empirically evaluate said criteria using realistic systems developed at NASA.

5 Acknowledgements

We would like to thank Corina Pasareanu of NASA Ames for her comments on this paper as well as her significant contributions towards the prototype. We would also like to thank NASA Ames, and Mike Lowrey in particular, for their support in developing the prototype as well as their suggestions for future work.

References

[1] S. Anand, C.S. Pasareanu, and W. Visser. JPF-SE: A Symbolic Execution Extension to Java PathFinder. *LECTURE NOTES IN COMPUTER SCIENCE*, 4424:134, 2007.

[2] C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on Java predicates. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 123–133. ACM New York, NY, USA, 2002.

[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] E. Foundation. Eclipse IDE, 2006.

[5] Mats P.E Heimdahl, S. Rayadurgam, Willem Visser, George Devaraj, and Jimin Gao. Auto-generating test sequences using model checkers: A case study. In *3rd International Worshop on Formal Approaches to Testing of Software (FATES 2003)*, 2003.

[6] Gerald J Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004.

[7] S. Khurshid, C.S. Pasareanu, and W. Visser. Generalized Symbolic Execution for Model Checking and Testing. *LECTURE NOTES IN COMPUTER SCIENCE*, pages 553–568, 2003.

[8] A. Rajan, M.W. Whalen, and M.P.E. Heimdahl. The Effect of Program and Model Structure on MC/DC Test Adequacy Coverage. In *Proceedings of the 30th International Conference on Software engineering*, pages 161–170. ACM New York, NY, USA, 2008.

[9] J. Sztipanovits and G. Karsai. Generative Programming for Embedded Systems. In *International Conference on Principles and Practice of Declarative Programming*, volume 6, pages 180–180. Springer, 2002.

[10] 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*, July 2006.

jFuzz: A Concolic Whitebox Fuzzer for Java

Karthick Jayaraman
Syracuse University
kjayaram@syr.edu

David Harvison, Vijay Ganesh, Adam Kieżun*
MIT
{dharv720, vganesh, akiezun}@mit.edu

Abstract

We present jFuzz, a automatic testing tool for Java programs. jFuzz is a concolic whitebox fuzzer, built on the NASA Java PathFinder, an explicit-state Java model checker, and a framework for developing reliability and analysis tools for Java. Starting from a seed input, jFuzz automatically and systematically generates inputs that exercise new program paths. jFuzz uses a combination of concrete and symbolic execution, and constraint solving.

Time spent on solving constraints can be significant. We implemented several well-known optimizations and name-independent caching, which aggressively normalizes the constraints to reduce the number of calls to the constraint solver. We present preliminary results due to the optimizations, and demonstrate the effectiveness of jFuzz in creating good test inputs. The source code of jFuzz is available as part of the NASA Java PathFinder.

jFuzz is intended to be a research testbed for investigating new testing and analysis techniques based on concrete and symbolic execution. The source code of jFuzz is available as part of the NASA Java PathFinder.

1 Introduction

We present jFuzz, a concolic whitebox fuzzer for Java built on top of the NASA Java PathFinder (JPF) [4]. jFuzz takes a Java program and a set of inputs for that program. For each input, jFuzz creates new inputs that are modified (or *fuzzed*) versions of the input and exercise new control paths in the program.

jFuzz (similarly to other concolic whitebox fuzz testing tools [7, 11]) executes the program both concretely and symbolically [7, 9, 14]. jFuzz converts the symbolic execution into a logical formula called a *path constraint*. jFuzz systematically negates every conditional along the execution path, conjoins the conditional with the corresponding path constraint, and queries a constraint solver. The solution, if one exists, is in terms of values for parts of the input. jFuzz uses the solution to fuzz (modify) these parts to obtain a new input. The appropriately fuzzed inputs can thus explore previously unexamined branches along the execution path. Thus, jFuzz can systematically explore every control-flow path.

The time spent in constraint solving can be significant [11], because (i) constraints may be hard to solve, or (ii) the solver may be repeatedly solving a large number of very similar problems. We implemented well-known optimizations such as constraint caching, constraint independence and subsumption [9, 11, 14] that seek to simplify the interaction of the testing tool with the solver. In addition, we also implemented name-independent caching, a new optimization that aggressively normalizes path constraints generated during concolic execution. This technique detects equivalence between two constraints modulo variable renaming. Thus, jFuzz caches solutions to already-solved constraints, and whenever jFuzz detects an equivalence between as yet unsolved constraint and an already-solved constraint, the cached solution is denormalized and reused, thus reducing the number of calls to the solver.

Contributions

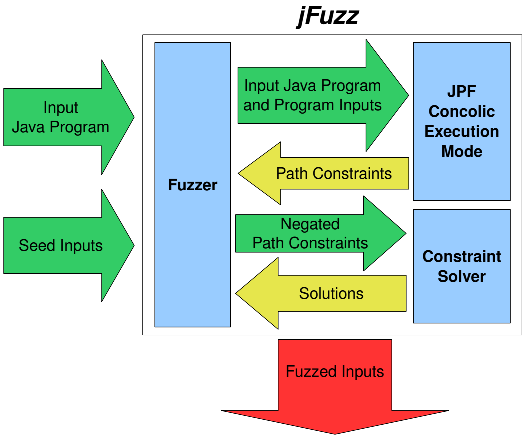
- **Concolic execution mode in JPF:** Concolic execution combines concrete and symbolic execution. Having this mode implemented in a reliable open-source framework such as JPF will facilitate further research in systematic software testing. The source code of the concolic execution mode is available as part of JPF.

E. Denney, D. Giannakopoulou, C.S. Păsăreanu (eds.); The First NASA Formal Methods Symposium, pp. 121-125

[*] To whom correspondence should be addressed.

```
public static int modtenadd(int x, Int y)        {

  int z;

  if(x >= 10 || y >=10)
  {
        throw new InvalidArgumentException();
  }

  z = x + y;

  if(z >= 10)
  {
        z = z - 10;
  }
return z;
}
```

(a) A Java function that performs modulo 10 addition



(b) jFuzz architecture. Given the program under test and seed inputs, jFuzz generates new inputs by modifying (fuzzing) the seed inputs so that each new input executes a unique control-flow path.

- **jFuzz:** jFuzz is a concolic whitebox fuzzer for Java, built on top of the JPF's concolic execution mode (which can be used independently of jFuzz). jFuzz is intended as a research vehicle for development of smart fuzzing techniques. The source code of the concolic execution mode is available as part of JPF.

- **Experimental Evaluation:** We present preliminary experimental results. We evaluated the efficiency of the concolic execution mode, effectiveness jFuzz's, and the performance of name-independent caching. In our experiments, the concolic mode added only 15% overhead above normal JPF execution. Tests created by jFuzz achieved slightly higher coverage than random fuzzing (18% vs. 15% line coverage). The constraint optimizations that we added reduced the time spent in constraint solving that ranged between 25%-30% to 1%.

2 Example

We illustrate whitebox fuzzing on an example (Figure 1(a)) Java function that performs modulo-10 addition on two integers. To start whitebox fuzzing, we provide the program and an initial concrete input $x = 3, y = 4$ to the fuzzer. Concolic execution produces a result of $z = 7$ and symbolic constraints $(x < 10) \bigwedge (y < 10) \bigwedge (x + y < 10)$.

Now, the whitebox fuzzer sequentially inverts each constraint and produces three sets of constraints.

1. $(x < 10) \bigwedge (y < 10) \bigwedge (x + y >= 10)$
2. $(x < 10) \bigwedge (y >= 10)$
3. $(x >= 10)$

The whitebox fuzzer solves these constraints using a constraint solver and obtains three new inputs $(6,6), (3,11), (11,3)$. Each of the three generated inputs exercises a distinct execution path in the program. The whitebox fuzzer repeats the process with the new inputs for either a pre-specified number of executions or time duration.

3 jFuzz Overview

jFuzz is built on top of the NASA Java PathFinder framework [4]. The Java PathFinder is an explicit state software model checker for Java bytecode, that also provides hooks for a variety of analysis techniques. Figure 1(b) illustrates the architecture of jFuzz. jFuzz works in three steps:

1. Concolic Execution: jFuzz executes the subject program in the concolic-execution mode on the seed input, and collects the path constraint. Each byte in the seed inputs is marked symbolic. The path constraint is a logical formula that describes the set of concrete inputs that would execute the same control-flow path as the seed input.

2. Constraint Solving: Once the concolic execution has completed, jFuzz systematically negates the conditionals encountered on the executed path. jFuzz conjoins the corresponding path constraint with the negated conditional, to obtain a new constraint query for the solver. The solution is in terms of input bytes, i.e., describes the values of the input bytes.

3. Fuzzing: For each solution, jFuzz changes the corresponding input bytes of the initial seed input to obtain a new fuzzed input for the program under test.

3.1 Concolic Execution Mode in Java PathFinder

One of the contributions of this paper is the concolic execution mode in JPF. This mode can be used independently of jFuzz, to construct new research tools that employ concolic techniques. The concolic mode is inspired by the symbolic execution mode already available in Java PathFinder [12]. JPF provides the facility to associate *attributes* with runtime values. Concolic (and symbolic) execution mode uses attributes to associate symbolic constraints with runtime values, and extends the Java bytecode instructions to update the symbolic constraints during concolic execution. The differences between concolic and symbolic mode are:

- Concolic mode preserves concrete values for runtime values, while symbolic mode loses them.
- Concolic mode does not fork and backtrack the execution. This improves performance because it does not require state matching. Debugging concolic execution is also much simpler.

3.2 Name-Independent Caching

A key issue with whitebox fuzzing is that the cumulative time spent in constraint solving can be a signification percentage of the time taken for producing new fuzzed inputs [11].

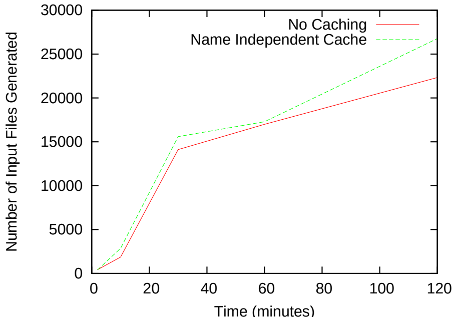
Path Condition 1:	Path Condition 2:
[1] a + b < 10	[1] x + y < 10
[2] b > 6	[2] y > 6
[3] a < 3	[3] x < 3
[4] a != 2	[4] x != 2

Figure 1: Two path conditions that are equivalent under name-independent caching.

In jFuzz, we implemented several well-known optimizations such as constraint caching, constraint independence and subsumption [9, 11, 14]. Additionally, we implemented name-independent caching, an optimization that normalizes path constraints generated during concolic execution. This technique detects equivalence between two constraints modulo variable renaming. Thus, solutions to already-solved constraints are cached, and whenever equivalences between as yet unsolved constraints and already-solved constraints are detected, the cached solutions are denormalized and reused, resulting in reduced number of calls to the solver. For example, consider the two path constraints in Figure 1. The name-independent cache detects that the two path constraints are structurally equivalence

		coverage	
mode	inputs	line	block
Seed inputs	15	13%	12%
Random fuzzing	300	14%	13%
jFuzz	264	18%	17%

(a) Coverage results for 1-hour test-runs.



(b) Number of input files generated with and without name-independent caching in a given amount of time.

Figure 2: Experimental Results

and passes only one to the constraint solver. Without this optimization, a fuzzing tool redundantly calls the constraint solver for both constraints.

4 Experimental Evaluation

We evaluated the efficiency of the concolic execution mode, jFuzz's effectiveness, and the performance of name-independent caching.

Experimental Setup. As a subject program, we used SAT4J [5], a Boolean SAT solver (19419 lines of Java code). We obtained 15 seed inputs from the SAT competition website [3] All experiments were performed on an Intel Centrino Duo 1.4 GHz processor with 2GB RAM running the GNU/Linux operating system (Ubuntu version 8.04). Each testing technique (or testing mode) was given 1 hour testing time. We measured line and block coverage using the Emma tool [2]. The time required by the random fuzzer and jFuzz to construct their test suites is included in the testing time. Consequently, both the random fuzzer and jFuzz had lower timeout per test.

Results. The concolic mode adds only modest overhead to JPF. We compared the execution times of 18 test programs when executed using the JPF concolic execution mode, Java PathFinder, and the Java virtual machine. Compared to the Java virtual machine, Java PathFinder (in simulation mode) has an average slow down of 12×, while the concolic execution mode has an average slow down of 14×.

jFuzz creates effective tests. Inputs generated by jFuzz achieve better coverage than randomly generated inputs and markedly increase coverage from seed inputs (Table 2(a)).

The optimizations are effective. The proportion of time spent in constraint solving ranged between 25% to 30% without optimizations, compared to 1% with the optimizations. Also, the caching increased the number of generated files, per unit of time, by 16% (Figure 2(b)). The optimizations reduced the number of redundant calls to the constraint solver and enables jFuzz to spend more time on generating new input files. The cache hit-rate, depending on the example, ranged between 30% and 50%.

5 Related Work

A number of testing tool are based on combined concrete and symbolic execution [1, 6, 7, 8, 9, 11, 13, 14]. However, as far as we know, only CREST [1] (a tool for testing C programs) is publicly available. Furthermore, most of these tools are *end-user* tools, and thus not necessarily extensible by other researchers.

jFuzz builds on top of extensible and mature technology, Java PathFinder explicit-state software model checker and dynamic-analysis framework. We believe that jFuzz will provide an extensible platform for researchers to try new concolic-based reliability techniques.

jFuzz's name-independent caching is a simple yet effective technique to reduce the cumulative time spent in constraint solver. Testing tools use constraint caching [7, 8, 10, 11], and other optimizations such as syntactic subsumption [11], and unrelated constraint elimination [9, 14]. However, as far as we know, the name-independent caching scheme that we implemented in jFuzz has not been used before in systematic testing.

jFuzz represents work in progress, and our results are preliminary. We plan to use jFuzz to test many larger Java programs. We believe that other researchers will find jFuzz useful as a testbed to try new concolic-based techniques.

Acknowledgments

We thank Corina Păsăreanu and Peter Mehlitz from the JPF team for their assistance with the development of the concolic mode and in getting jFuzz included in Java Path Finder.

References

[1] CREST: automatic test generation tool for c. http://code.google.com/p/crest.

[2] EMMA: A Java code coverage tool. http://emma.sourceforge.net/.

[3] The international SAT competitions web page. http://www.satcompetition.org/.

[4] NASA Java PathFinder. http://javapathfinder.sourceforge.net.

[5] SAT4J: A Java sat solver. http://www.sat4j.org/.

[6] Shay Artzi, Adam Kieżun, Julian Dolby, Frank Tip, Danny Dig, Amit Paradkar, and Michael Ernst. Finding bugs in dynamic Web applications. In *ISSTA*, 2008.

[7] Christian Cadar, Vijay Ganesh, Peter M. Pawlowski, David Dill, and Dawson R. Engler. EXE: automatically generating inputs of death. In *CCS*, 2006.

[8] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.

[9] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *PLDI*, 2005.

[10] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Active property checking. In *EMSOFT*, 2008.

[11] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Automated whitebox fuzz testing. In *NDSS*, 2008.

[12] Corina S. Păsăreanu, Peter C. Mehlitz, David H. Bushnell, Karen Gundy-Burlet, Michael Lowry, Suzette Person, and Mark Pape. Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In *ISSTA*, 2008.

[13] K. Sen and G. Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. *LECTURE NOTES IN COMPUTER SCIENCE*, 4144:419, 2006.

[14] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for C. In *FSE*, 2005.

Machine-Checkable Timed CSP

Thomas Göthel and Sabine Glesner
Department of Software Engineering and Theoretical Computer Science,
Berlin Institute of Technology, FR 5-6, Franklinstr. 28/29, 10587 Berlin, Germany
{tgoethel,glesner}@cs.tu-berlin.de

Abstract

The correctness of safety-critical embedded software is crucial, whereas non-functional properties like deadlock-freedom and real-time constraints are particularly important. The real-time calculus Timed CSP is capable of expressing such properties and can therefore be used to verify embedded software. In this paper, we present our formalization of Timed CSP in the Isabelle/HOL theorem prover, which we have formulated as an operational coalgebraic semantics together with bisimulation equivalences and coalgebraic invariants. Furthermore, we apply these techniques in an abstract specification with real-time constraints, which is the basis for current work in which we verify the components of a simple real-time operating system deployed on a satellite.

1 Introduction

The correctness of software in embedded safety-critical real-time systems is essential for their correct functioning. Failures in the software used in these areas may cause high material costs or even the loss of human lives. We address the problem of developing methods to increase confidence in these systems. We are looking for mathematical models needed to develop and reason about formal specifications of such systems, and at the same time we want to ensure that the reasoning process itself is correct.

The context of our work is the VATES[1] project [GHJ07], which is funded by the German Research Foundation (DFG). Our objective is the formal verification of embedded software systems, from an abstract specification to an intermediate representation (as used in compilers) and, finally, to machine code. The real-time operating system BOSS [MRH05], employed among others in a space satellite, is used as a case study to evaluate our techniques.

In our approach, we use the process calculus Timed CSP [Sch99] as the mathematical basis for formal proofs. Timed CSP is a formal modeling language that allows for the convenient specification and verification of reactive, concurrent real-time systems. We formalize our specifications and correctness proofs in a theorem prover, in which proofs are mechanized and (at least partly) automated. Unlike testing or simulating specifications, theorem proving allows us to gain absolute assurance of correct system behavior. In this paper, we propose a formalization of Timed CSP in the Isabelle/HOL theorem prover [NPW02] to combine both advantages, namely specifying real-time systems concisely and mechanizing correctness proofs for properties of their specifications.

The rest of this paper is organized as follows: In Section 2, we give some background information about Timed CSP, Bisimulations, Invariants and the Isabelle/HOL theorem prover. In Section 3, we present our formalization of the operational semantics of Timed CSP and the coalgebraic notions of bisimulations and invariants. In Section 4, we present an application of the formalization to verify a simple model of a satellite system. Related work is discussed in Section 5. Finally, in Section 6, we conclude and discuss ideas for future work.

2 Background

In this section, we give a brief overview of the background of our work. First, we summarize the essence of the real-time process calculus Timed CSP by introducing its operational semantics such that Timed CSP may be seen as a timed labeled transition system. Then, we proceed by introducing the well-known

E. Denney, D. Giannakopoulou, C.S. Păsăreanu (eds.); The First NASA Formal Methods Symposium, pp. 126-135

[1] VATES = Verification and Transformation of Embedded Systems

$$P := STOP \mid SKIP \mid a \to P \mid P;P \mid P \Box P \mid P \sqcap P \mid a : A \to P_a$$
$$\mid P \underset{A}{\parallel} P \mid P \setminus A \mid P \triangle P \mid P \overset{d}{\triangleright} P \mid P \triangle_d P \mid X$$

Figure 1: Syntax of Timed CSP

Let μ and t be variables with $\mu \in \Sigma \cup \{\tau, \surd\}$, $t \in \mathbb{R}^+ \setminus \{0\}$

$$\frac{}{STOP \overset{t}{\rightsquigarrow} STOP} \qquad \frac{}{(a \to P) \xrightarrow{a} P} \qquad \frac{}{(a \to P) \overset{t}{\rightsquigarrow} (a \to P)}$$

$$\frac{P \xrightarrow{\mu} P'}{P \overset{d}{\triangleright} Q \xrightarrow{\mu} P'}\ \mu \neq \tau \qquad \frac{P \xrightarrow{\tau} P'}{P \overset{d}{\triangleright} Q \xrightarrow{\tau} P' \overset{d}{\triangleright} Q} \qquad \frac{}{P \overset{0}{\triangleright} Q \xrightarrow{\tau} Q} \qquad \frac{P \overset{t}{\rightsquigarrow} P'}{P \overset{d}{\triangleright} Q \overset{t}{\rightsquigarrow} P' \overset{d-t}{\triangleright} Q}\ t \leq d$$

$$\frac{}{X \xrightarrow{\tau} asg(X)}\ X \in \mathcal{V} \qquad \text{with} \qquad asg : \mathcal{V} \Rightarrow TCSP$$

Figure 2: Part of the Operational Semantics of Timed CSP

notions of bisimulations [Mil89, JR97] and coalgebraic invariants [Jac97] for arbitrary labeled transition systems. We close this section with a short introduction to the Isabelle/HOL theorem prover which we have chosen for our formalization in Section 3.

2.1 Timed CSP

As the starting point in the development chain of the VATES project, we chose the real-time process calculus Timed CSP [Sch99], which extends CSP (Communicating Sequential Processes) [Hoa85] with timed process terms as well as timed semantics. Beside the specification and verification of reactive and concurrent systems, this also allows the verification of timelines. Due to not having enough space for presenting all details of (Timed) CSP, we refer to [Sch99] for a comprehensive introduction.

Let Σ be a (communication) alphabet and $\mathcal{V}$ a set of process variables. Furthermore, let a, d, A and X be variables with $a \in \Sigma$, $d \in \mathbb{R}^+$, $A \subseteq \Sigma$ and $X \in \mathcal{V}$. Then the Timed CSP processes are given by the grammar in Figure 1.

Timed CSP extends the CSP calculus with the timed primitives $P \overset{d}{\triangleright} Q$ (*Timeout*) and $P \triangle_d Q$ (*Timed Interrupt*). Intuitively, the meaning of Timeout is that the process P may be triggered by some event within d time units. When the time expires, the process Q of the Timeout construction handles this situation. The Timed Interrupt construction basically means the same, except that P may (successfully) terminate in d time units, otherwise Q is started. Owing to space limitations, we concentrate on the most interesting semantic rules, which are given in Figure 2. For a complete semantics, see [Sch99].

The transitions of Timed CSP processes consist of instantaneous event transitions ($\xrightarrow{a}$) and timed transitions ($\overset{t}{\rightsquigarrow}$). The event transitions are best understood as communication with some environment. This means in particular that an event is only communicated if the environment requests it. This interpretation is enforced by the semantic rules of timed steps since there is no process construction forcing a visible event (all except τ) to happen; in other words, time can advance if and only if no internal transition can occur.

In contrast to [Sch99], we model recursion by using the process variables $X \in \mathcal{V}$. This means, for example, that the recursive process $P = a \to P$ is modeled by a process variable P_X, which is assigned to the (nonrecursive) process $a \to P_X$ with the assignment $asg : \mathcal{V} \Rightarrow TCSP$ ($TCSP$ denotes the set of all

Timed CSP processes). To define the semantics formally, it is therefore necessary to parameterize the rules of the operational semantics with such an assignment. The corresponding rule for process variables says that they are unfolded by an (invisible) internal event.

In this paper, the basic idea is to interpret Timed CSP as a timed labeled transtion system, in which the states are given by the Timed CSP processes and the transition relation is given by event and timed steps. This enables the application of bisimulations and invariants, explained in the following section.

2.2 Bisimulations and Invariants

A labeled transition system over an alphabet A is a tuple $LTS = (S,T)$, where S is a set of states and $T \subseteq (S \times A \times S)$ is the labeled transition relation. Instead of $(P,\alpha,P') \in T$, we also write $P \xrightarrow{\alpha} P'$. In addition, there is often a special event $\tau \in A$, which is interpreted as internal event.

A *timed* labeled transition system over an alphabet A is a triple $TLTS = (S,T,D)$, where S is a set of states, D is a time domain (which is closed under some addition) and $T \subseteq S \times (A \cup D) \times S$ is the timed transition relation. We assume A and D to be disjoint. Note that every timed labeled transition system can also be interpreted as a "simple" labeled transition system by joining the alphabet with the time domain.

Labeled transition systems are part of a more general theory, namely the theory of coalgebras [JR97]. Bisimulation turns out to be a strong and convenient proof principle for showing the equivalence of processes (states of the coalgebra). In addition, invariants [Jac97] allow the verification of liveness properties, as we show in Section 3.4. We introduce these notions for labeled transition systems in the following sections.

2.2.1 Strong Bisimulation

A relation $R \subseteq S \times S$ is called bisimulation on a labeled transition system $LTS = (S,T)$ over A if the following property holds: For all $(P,Q) \in R$ and $\alpha \in A$:

(i) If $P \xrightarrow{\alpha} P'$, then there is a Q' with $Q \xrightarrow{\alpha} Q'$ and $(P',Q') \in R$.

(ii) If $Q \xrightarrow{\alpha} Q'$, then there is a P' with $P \xrightarrow{\alpha} P'$ and $(P',Q') \in R$.

2.2.2 Weak Bisimulation

Let $LTS = (S,T)$ be a labeled transition system over A and $\tau \in A$ a special event, which is assumed to be internal or not visible to the environment. We define a relation $\rightarrow_{T^*} \subseteq S \times A \times S$ as follows[2]:

(i) If $P \xrightarrow{\tau}{}^{*} P'$, then $P \xrightarrow{\tau}_{T^*} P'$.

(ii) If $P \xrightarrow{\tau}{}^{*} P_1$, $P_1 \xrightarrow{\alpha} P_2$ and $P_2 \xrightarrow{\tau}{}^{*} P'$ $(\alpha \neq \tau)$, then $P \xrightarrow{\alpha}_{T^*} P'$.

A relation $R \subseteq S \times S$ is called weak bisimulation on a labeled transition system $LTS = (S,T)$ over A if the following property holds: For all $(P,Q) \in R$ and $\alpha \in A$

(i) If $P \xrightarrow{\alpha} P'$, then there is a Q' with $Q \xrightarrow{\alpha}_{T^*} Q'$ and $(P',Q') \in R$.

(ii) If $Q \xrightarrow{\alpha} Q'$, then there is a P' with $P \xrightarrow{\alpha}_{T^*} P'$ and $(P',Q') \in R$.

In contrast to strong bisimilations, every answering step may occur after and before arbitrarily many internal steps, which are assumed to be invisible to the environment.

2.2.3 Weak Timed Bisimulation

In the context of timed labeled transition systems $TLTS = (S,T,D)$ over A, we define weak timed bisimulations [dFELN99]. First, we define a set allowing us to compress timed steps $\rightarrow_{D^*} \subseteq S \times (A \cup D) \times S$

[2] $\xrightarrow{\tau}{}^{*}$ denotes the reflexive-transitive closure of the original transition system w.r.t. τ, i.e. $P \xrightarrow{\tau}{}^{*} P'$ means $P \xrightarrow{\tau} \bullet \xrightarrow{\tau} \ldots \xrightarrow{\tau} \bullet \xrightarrow{\tau} P'$.

with the help of $\rightarrow_{T^*}$ (defined over $A \cup D$):

(i) If $P \xrightarrow{\alpha}_{T^*} P'$ and $\alpha \in A$, then $P \xrightarrow{\alpha}_{D^*} P'$.

(ii) If for all $i \in \{0,\dots,n\}$ (for some arbitrary $n \in \mathbb{N}$): $P_i \xrightarrow{t_i}_{T^*} P_{i+1}$ with $t_i \in D$ and $\sum_{i=0}^{n} t_i = t$, then $P_0 \xrightarrow{t}_{D^*} P_{n+1}$.

A relation $R \subseteq S \times S$ is called weak timed bisimulation on a timed labeled transition system $TLTS = (S,T,D)$ over A if the following property holds:
For all $(P,Q) \in R$ and $\beta \in A \cup D$:

(i) If $P \xrightarrow{\beta} P'$, then there is a Q' with $Q \xrightarrow{\beta}_{D^*} Q'$ and $(P',Q') \in R$.

(ii) If $Q \xrightarrow{\beta} Q'$, then there is a P' with $P \xrightarrow{\beta}_{D^*} P'$ and $(P',Q') \in R$.

Here, a timed step may also be answered by many consecutive timed steps, where the summed duration is equal to the original time span. Furthermore, internal steps are allowed between these single timed steps.

All these kinds of bisimulation allow us to identify semantically equivalent processes with respect to the operational transition semantics. They all have exactly the same structure, i.e., for two equivalent processes P and Q, every simple step of process P must be answered by a "complex" step of Q and vice versa. The various complex steps are used for hiding details of single steps. In contrast to simulations, not too much information is lost because of the symmetry property of bisimulations.

2.2.4 Invariants

To reason about the states of a transition system, invariants can be used to identify invariant behavior. Let $LTS = (S,T)$ over A be a labeled transition system. A predicate $I \subseteq S$ is called an invariant if the following property holds: For all $P \in I$, $Q \in S$ and $\alpha \in A$:

If $P \in I$ and $P \xrightarrow{\alpha} Q$, then $Q \in I$

This means that invariants are closed under the transition steps of the transition system. Greatest invariants are of particular interest. Let $P \subseteq S$ be an arbitrary predicate on the state space of a labeled transition system. Then there exists a greatest invariant $\hat{P}$, $\hat{P} \subseteq P$. Intuitively, P is reduced until it fulfills the property of an invariant. In the extreme case, $\hat{P}$ is the empty set, which is also an invariant.

2.3 Isabelle

Isabelle is a generic interactive proof assistant. It enables the formalization of mathematical models and provides tools for proving theorems about them. A particular instantiation of Isabelle is Isabelle/HOL [NPW02], which is based on Higher Order Logic and comes with a very high expressiveness of specifications. Unlike model checking, proving theorems in a theorem prover like Isabelle/HOL is highly interactive. Specifications have to be designed carefully to be able to prove properties about them. Theorem provers require a high level of expertise but allow reasoning about models whose state space is too large to be automatically checked by, say, a model checker.

In our formalization, we benefit from the very well-developed formalizations of sets. More precisely, we make extensive use of inductively and coinductively defined sets, which come with induction and coinduction schemes, respectively.

3 Formalization of Timed CSP

In this section, we present our formalization of the operational semantics of Timed CSP. The syntax of Timed CSP is simply given by an inductive datatype $('v,'a)Process$ parameterized over an alphabet type $'a$ and a type representing the process variables $'v$. We omit its explicit definition here because it is a straightforward realization of the grammar in Figure 1. The operational semantics is given by two inductive sets defining the event and the timed steps. Furthermore, these sets are parameterized over a variable assignment $asg :: {}'v \Rightarrow ({}'v,{}'a)Process$. This is necessary to give (even mutually) recursive processes a semantics (see Section 2.1). As mentioned in Section 2.2, the operational semantics defines a timed labeled transition system. This means that the operational semantics is defined as an instance of the type $({}'s,{}'a)lts = ({}'s \times {}'a \times {}'s)set$. On this basis, we define and examine bisimulations and invariants for Timed CSP in Sections 3.2 and 3.4.

3.1 Operational Semantics

In Timed CSP, there are four different kinds of steps: event steps, non-visible steps, terminating steps and timed steps. We encode these by the datatype *'a eventplus*, where *'a* is assumed to be the process alphabet.

datatype *'a eventplus* = *ev 'a* | *tau* | *tick* | *time real*

We define one single type for all kinds of steps because we want to join event steps (*ev 'a, tau, tick*) and timed steps (*time real*) into one single transition relation. This is useful to be able to interpret Timed CSP as a timed labeled transition system (see 2.2).
Furthermore, we abbreviate process variable assignments by the type *procAsg*.

types $({}'v,{}'a)procAsg = {}'v \Rightarrow ({}'v,{}'a)Process$

Now we are able to define the operational semantics by defining the event steps and the timed steps (see Figure 3). The semantics of process terms depends on such a particular variable assignment. The inductively defined set of rules is a straightforward encoding of the rules in Figure 2.
The timed transitions are defined separately by a second inductively defined set. Note that the timed transitions depend on the formerly defined event transitions. Internal events are instantaneous in Timed CSP. This means that no time may advance when internal transitions are enabled. In the semantics of *Sequential Composition* and *Hiding*, it is thus necessary to allow time to advance only if internal transi-

```
inductive_set evstep:: $({}'n,{}'a)procAsg \Rightarrow (({}'n,{}'a)Process,{}'a\ eventplus)lts$
  for asg :: $({}'n,{}'a)procAsg$   where
    ...
    | Timeout_step1: $[\![(Q1,e,Q2) \in evstep\ asg; e = ev\ a \lor e = tick]\!]$
                     $\Longrightarrow ((Q1 \rhd^{d} P),e,Q2) \in evstep\ asg$
    | Timeout_step2: $[\![(Q1,tau,Q2) \in evstep\ asg]\!]$
                     $\Longrightarrow ((Q1 \rhd^{d} P),tau,(Q2 \rhd^{d} P)) \in evstep\ asg$
    | Timeout_step3: $((Q1 \rhd^{0} P),tau,P) \in evstep\ asg$
    ...

inductive_set tstep:: $({}'n,{}'a)procAsg \Rightarrow (({}'n,{}'a)Process,{}'a\ eventplus)lts$
  for asg :: $({}'n,{}'a)procAsg$   where
    ...
    | Timeout_step: $[\![(Q1,time\ t,Q2) \in tstep\ asg; t \leq d1]\!]$
                    $\Longrightarrow ((Q1 \rhd^{d1} P),time\ t,(Q2 \rhd^{d1-t} P)) \in tstep\ asg$
    ...
```

Figure 3: Event and Timed Steps

tions are not enabled (see [Sch99]). This is the case, for example, if the first process of the Sequential Composition can successfully terminate.

Since we wish to interpret Timed CSP as a labeled transition system, we join event and timed steps.

$$\begin{array}{l}\textbf{constdefs}\ \mathit{step} :: ('v,'a)\mathit{procAsg} \Rightarrow (('v,'a)\mathit{Process},'a\ \mathit{eventplus})\mathit{lts} \\ \quad \mathit{step\ asg} \equiv \mathit{tstep\ asg} \cup \mathit{evstep\ asg}\end{array}$$

This is done in the definition of *step*, which comprises the whole transition system of Timed CSP.

3.1.1 Properties of Steps

With our Isabelle/HOL formalization of the operational semantics of Timed CSP, we can prove some of the standard properties [Sch99]. We explain some examples which are useful in the rest of this paper.

Urgency of Internal Events: Although nothing can be guaranteed when an external event happens, it is important that internal steps have priority over timed steps. This property is captured by the lemma that internal steps and timed steps may not both be possible.

Constancy of Offers: Timed steps do not change the set of offered (visible) events of a process, i.e., only events (internal as well as external) may change the offers of a process.

Time Determinism: Timed steps do not introduce further nondeterminism. This means that every two timed steps of the same duration will reach the same target process.

Time Additivity: Two consecutive timed steps may be summarized into one timed step (of the summed duration). The other direction holds as well. Every timed step may be divided into two consecutive timed steps.

3.1.2 Definition of "complex" transitions

As explained in Section 2.2, the different kinds of bisimulation have the same structure in that some original transition must be answered adequately by a "complex" transition. It is straightforward to define the transition relations $\rightarrow_{T^*}$ and $\rightarrow_{D^*}$ from Section 2.2 on the timed labeled transition system of Timed CSP. In the context of our formalization, we call them *relWeak* and *relWeakt*, respectively. Their type is $('v,'a)\mathit{procAsg} \Rightarrow (('v,'a)\mathit{Process},'a\ \mathit{eventplus})\mathit{lts}$ in both cases.
We define $\mathit{time\_more} :: ('v,'a)\mathit{procAsg} \Rightarrow (('v,'a)\mathit{Process}\ ,\ \mathit{real} \times 'a\ \mathit{eventplus})\mathit{lts}$ as another transition relation which we use for the definition of liveness invariants in Section 3.4. Here, transitions are labeled by tuples indicating that some visible event may be communicated.

By $(P,(t,a),Q) \in \mathit{time\_more\ asg}$, we mean a sequence like:

$$P \xrightarrow{\tau^*} \bullet \overset{t1}{\rightsquigarrow} \bullet \xrightarrow{\tau^*} \bullet \quad \cdots \quad \xrightarrow{\tau^*} \bullet \overset{tn}{\rightsquigarrow} \bullet \xrightarrow{\tau^*} \bullet \xrightarrow{a} \bullet \xrightarrow{\tau^*} Q\ , \qquad \text{where} \quad \sum t_i = t.$$

Based on these definitions, we define bisimulations for Timed CSP processes in Isabelle/HOL, as explained in the following section.

3.2 Bisimulations

We define bisimulations abstractly for arbitrary labeled transition systems such that we can instantiate these definitions (and hence get the properties) for concrete bisimulations for Timed CSP. We define the greatest bisimulation *bisimilar T1 T2* using a coinductively defined set.

$$\begin{array}{l}\textbf{coinductive\_set}\ \mathit{bisimilar} ::\ ('s,'a)\mathit{lts} \Rightarrow ('s,'a)\mathit{lts} \Rightarrow ('s \times 's)\mathit{set} \\ \quad \textbf{for}\ \mathit{T1} :: ('s,'a)\mathit{lts}\ \textbf{and}\ \mathit{T2} :: ('s,'a)\mathit{lts}\quad \textbf{where} \\ \qquad [\![\ \forall\, t\ P2.\ (P1,t,P2) \in T1 \longrightarrow (\exists\, Q2.\ (Q1,t,Q2) \in T2 \wedge\ (P2,Q2) \in \mathit{bisimilar}\ T1\ T2)\ ;\end{array}$$

$$\begin{array}{l}\forall\, t\; Q2.\; (Q1,t,Q2) \in T1 \longrightarrow (\exists\, P2.\; (P1,t,P2) \in T2 \wedge\; (P2,Q2) \in \mathit{bisimilar}\; T1\; T2)\; \rrbracket \\ \Longrightarrow (P1,Q1) \in \mathit{bisimilar}\; T1\; T2\end{array}$$

The relations *T1* and *T2* are generalizations of the different kinds of complex transition relations mentioned in Section 2.2. The first transition relation (*T1*) is meant to be the original labeled transition system for which bisimulations are to be defined. The relation *T2* represents a complex transition relation that is used for answering steps in the original transition system. The benefit of defining the greatest bisimulation via a coinductive set is that the following coinduction proof scheme can easily be deduced.

$$\frac{\begin{array}{l}(P,Q) \in X \\ \forall t P2.(P1,t,P2) \in T1 \rightarrow (\exists Q2.(Q1,t,Q2) \in T2 \;\wedge\; (P2,Q2) \in X \cup \mathit{bisimilar}\; T1\; T2) \\ \forall t Q2.(Q1,t,Q2) \in T1 \rightarrow (\exists P2.(P1,t,P2) \in T2 \;\wedge\; (P2,Q2) \in X \cup \mathit{bisimilar}\; T1\; T2)\end{array}}{(P,Q) \in \mathit{bisimilar}\; T1\; T2}$$

Instantiating X with a concrete bisimulation (not necessarily the greatest), the scheme can be used to prove the bisimilarity of two processes. It has to be shown that they can communicate the same events (or let the same time span pass) arriving at processes that are again in X or are already bisimilar. Using this scheme, it can easily be shown that *bisimilar T1 T2* is indeed the greatest bisimulation (which is the union of all bisimulations). On this level of abstraction, we are able to prove that the greatest bisimulation is an equivalence relation. This is useful because these lemmas can be instantiated for the different kinds of bisimulations on Timed CSP processes.

The different kinds of bisimulations for Timed CSP can be defined by instantiating $T1$ with *step asg* and *T2* with *step asg*, *relWeak asg* or *relWeakt asg*. This leads to strong, weak and weak timed bisimulation, respectively. Thus, all these kinds are proved to be equivalence relations because this lemma holds for abstract bisimulations.

3.3 Properties of Bisimilarity

We have shown the important property that the three kinds of bisimilarity are congruence relations, i.e. from the bisimilarity of the parts of two processes it is possible to conclude the bisimilarity of the composed processes. Note that for these results the properties explained in Section 3.1.1 are of paramount importance because, for example, “internal_urgency” is needed to show that Sequential Composition and Hiding do not destroy the congruence property. Together with algebraic laws, it is possible to do bisimulation proofs almost without considering concrete bisimulation relations. Algebraic laws are that e.g. $P1 \overset{d1}{\rhd} (P2 \overset{d2}{\rhd} P3)$ is weak timed bisimilar to $(P1 \overset{d1}{\rhd} P2) \overset{d1+d2}{\rhd} P3$. Note that the algebraic laws based on weak timed bisimilarity are not complete w.r.t. the algebraic laws in, for example, the denotational Timed Failures semantics of Timed CSP [Sch99][3]. In Section 6 we discuss a solution for this problem.

3.4 Invariants

Invariants are able to express liveness conditions of processes. Their origin lies in the theory of coalgebras, where they are predicates on the state space of a coalgebra with certain properties. Invariants are closed under transitions, i.e. if an arbitrary state fulfills the predicate (the invariant), all successor states fulfill it as well. Here, we are interested in invariants on the state space of Timed CSP, i.e., the set of all Timed CSP processes. Of special importance are greatest invariants. As explained in Section 2.2.4, for

[3] In the denotational semantics, the Internal Choice construction is associative, whereas it is not valid when considering weak timed bisimilarity.

every predicate there exists a greatest invariant contained in the predicate. In Isabelle, we again take a coinductive set to define the greatest invariant w.r.t. to a predicate of interest.

coinductive_set $invariant{::}\ ('v,'a)procAsg \Rightarrow (('v,'a)Process \Rightarrow bool) \Rightarrow ('v,'a)Process\ set$
for $asg : ('v,'a)procAsg$ **and** $Pred : (('v,'a)Process \Rightarrow bool)$ **where**
$[\![Pred\ Q; \forall\ e\ Q1.\ (Q,e,Q1) \in step\ asg \longrightarrow Q1 \in invariant\ asg\ Pred]\!]$
$\Longrightarrow Q \in invariant\ asg\ Pred$

To define a particularly useful class of liveness conditions, we focus on predicates to be defined by *livePred*. A process fulfills such a predicate if there exists a step $(P,(t,a),P')$ in the sense of *time_more* (see 3.1) to another process and the inscription fulfills a certain property *IPred*. Remember that the transition relation *time_more* is defined as being labeled by tuples (t,a), where t represents the time and a is some event. The instantiation of the (greatest) invariant is given below.

constdefs $livePred{::}\ ('v,'a)procAsg \Rightarrow ((real \times 'a\ eventplus) \Rightarrow bool) \Rightarrow ('v,'a)Process \Rightarrow bool$
$livePred\ asg\ IPred\ P \equiv \exists\ e\ Q.\ (P,e,Q) \in time\_more\ asg \wedge IPred\ e$

constdefs $liveInvariant{::}\ ('v,'a)procAsg \Rightarrow ((real \times 'a\ eventplus) \Rightarrow bool \Rightarrow ('v,'a)Process\ set$
$liveInvariant\ asg\ IPred \equiv invariant\ asg\ (livePred\ asg\ IPred)$

The intent behind this special invariant is that some real-time process should always be able to react on an external event within a certain amount of time. This constraint can be embedded by defining the predicate *IPred* appropriately.

All invariants of the above form have a useful property: They are closed under weak timed bisimilarity. This is captured by the following theorem.

$$\begin{tikzcd}
Q \arrow[dd, "\cong"'] \arrow[dr, dashed, "\vdash"] & \\
& Inv \\
P \arrow[ur, "\vdash"'] &
\end{tikzcd}$$

lemma *liveInvariant_Bisim:* $[\![P \in liveInvariant\ asg\ IPred;$
$(P,Q) \in weak\_timed\_bisimilar\ asg]\!]$
$\Longrightarrow Q \in liveInvariant\ asg\ IPred$

To show a (liveness) property on a possibly complex process, we can show the property on a simpler bisimilar process instead. We will use this technique for our case study in the next section.

4 Application

To show the applicability of our formalization, we prove properties of a (rather simple) abstract specification of a satellite system. This satellite runs two main tasks. One is responsible for recognizing fire sources on Earth and sending messages to the terrestrial station. The other is responsible for rotating the solar panel in case of changes in the angle to the sun. This second task is the more important one as it ensures the satellite's survival by supplying it with energy. The behavior is realized by reacting on the *danger*, *rotation*, *fire* and *warning* event.

In our specification (Figure 4), the first task is realized by process *T1p* and the second by subprocesses *T2p* and *T3p*. The Interrupt construction (△) gives a higher priority to the second task. This means that whenever a *danger* event is received, the first task is aborted to adjust the position of the angle. We assume that it takes one time unit to send a message to earth, whereas the adjustment takes five time units. We wish to show that it is always possible to recognize a fire after at most five time units. This

$$T := (T1 = T1p \triangle (T2p \underset{\{rotation\}}{\|} T3p); \quad T1p := fire \rightarrow^{1} warning \rightarrow SKIP$$

$$T1) \setminus \{rotation, warning\} \quad T2p := danger \rightarrow rotation \rightarrow SKIP$$

$$T3p := rotation \rightarrow^{5} SKIP$$

$$Tn = (fire \rightarrow^{1} SKIP \triangle danger \rightarrow^{5} SKIP); Tn$$

Figure 4: Specification (T) and Abstract Specification (Tn) of a Simple Satellite

means that no deadlock may occur and that no situation may occur in which it is not possible to react on *fire* within five time units.

For this purpose, we first show that process T is weak timed bisimilar to the process *Tn* in Figure 4. Essentially, the equivalence proof of both processes can be performed by making use of the congruence property of weak timed bisimilarity. Together with simple algebraic laws, which can be proved abstractly, it is possible to prove them bisimilar without considering a concrete bisimulation relation. Since *Tn* has a simpler structure, the invariant is easier to prove than for the original process T.

Despite being a rather simple example, it does clearly demonstrate our proof principle: we prove a "complex" process to be correct by abstracting from its irrelevant internals and verifying on the abstract level. We show the correctness of the "complex" process T by instantiating our liveness invariant accordingly. Then we prove that process *Tn* fulfills this invariant such that we can use the lemma from the former section to deduce that T fulfills the invariant as well. We define the invariant as follows:

constdefs $fire\_pred\ (real \times event\ eventplus) \Rightarrow bool$
$fire\_pred\ e \equiv \exists t.\ e = (t, ev\ fire) \wedge t \leq 5$

Then our correctness criterion is expressed as $T \in liveinvariant\ pasg\ fire\_pred$. The proof for this is quite straightforward if we expand the state space of this process. For every reachable process, it must hold that it is possible to reach the initial process *Tn* again within five time units. The state space for this process is infinite because the Timeout construction is involved. Fortunately, it can be compressed by considering something similar to region graphs in the context of timed automata. This means for example, that the set of processes $\{(WAITd \triangle danger \rightarrow^{5} SKIP); Tn\ .\ \exists d.d \geq 0 \wedge d \leq 1\}$ can be considered as a whole because every process in this set can reach the process *Tn* in at most five time units.

5 Related Work

Related work on the formalization of CSP and Timed CSP in formal systems include the following: A denotational failures semantics of CSP formalized in Isabelle/HOL is presented in [TW97]. Recursive processes are represented by fixed points. Infinitely running processes are treated by considering arbitrarily long prefixes of their state transition sequences. The formalization is based on a shallow embedding. An extension of this work is described in [Int02]. In its current state, this work is incomplete. In addition, the chosen formalization decisions appear to have led to a rather clumsy specification that makes proofs unnecessarily complicated. A further formalization of CSP in Isabelle/HOL is described in [IR05]. This formalization is based on a denotational failures semantics and set up by a deep embedding. A different approach has been taken in [DHSZ06], where an operational semantics of Timed CSP has been set up in a Prolog-like style using a constraint-logic programming language. The transition rules of the operational semantics have been defined by rules and facts. Infinite traces are represented by allowing the unification process to run infinitely long, thus creating all traces. The time model only deals with discrete time.

None of this work formalizes Timed CSP fully, in particular with continuous time, as we have done in our work presented in this paper. We have based our formalization on an operational semantics, not on a denotational one, to fully exploit coalgebraic techniques and bisimulation in particular.

6 Conclusion

In this paper, we have presented a complete formalization of the operational semantics of the Timed CSP calculus. Our semantics models a transition system, which in turn allows us to employ coalgebraic proof techniques such as bisimulations and invariants. We have defined several forms of bisimulations (strong, weak and weak timed bisimulations) and have proved congruence properties about them. To simplify the use of bisimulations in formal proofs, we have shown that there exists an abstraction which is valid for all three kinds of bisimulation that can be used as a generalization in proofs. Moreover, we have shown that invariants can be used to verify liveness properties of processes. To simplify proofs involving invariants, we have shown that, under certain conditions, it suffices to verify invariants on simpler processes instead of on the original ones. As a sanity check of our formalization, we have formalized two simple tasks in a satellite system and have verified their liveness properties.

In future work, we aim to extend our formalization of the semantics such that weaker equivalences between processes can also be captured. We thus seek to adapt the ideas presented in [dFELN99], where the operationally defined original transition system is transformed appropriately to obtain a weaker notion of equivalence. Furthermore, we are currently extending our case study of the satellite operating system with the long-term goal of fully verifying this software system. We have developed a specification for a real-time scheduler and further components, for which we are currently conducting the correctness proofs in Isabelle.

References

[dFELN99] D. de Frutos-Escrig, N. López, and M. Núñez. Global Timed Bisimulation: An Introduction. In *FORTE XII / PSTV XIX '99*, pages 401–416. Kluwer, B.V., 1999.

[DHSZ06] J. S. Dong, P. Hao, J. Sun, and X. Zhang. A Reasoning Method for Timed CSP Based on Constraint Solving. In *ICFEM*, LNCS, pages 342–359. Springer, 2006.

[GHJ07] S. Glesner, S. Helke, and S. Jähnichen. VATES: Verifying the Core of a Flying Sensor. In *Proc. Conquest 2007*. dpunkt Verlag, 2007.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall International, 1985.

[Int02] M. Intemann. Semantische Codierung von Timed CSP in Isabelle/HOL. Master thesis, Uiversity of Bremen, 2002.

[IR05] Y. Isobe and M. Roggenbach. A Generic Theorem Prover of CSP Refinement. In *TACAS*, LNCS, pages 108–123. Springer, 2005.

[Jac97] B. Jacobs. Invariants, Bisimulations and the Correctness of Coalgebraic Refinements. In *AMAST*, pages 276–291, London, UK, 1997. Springer-Verlag.

[JR97] B. Jacobs and J. Rutten. A Tutorial on (Co)Algebras and (Co)Induction. *Bulletin of the European Association for Theoretical Computer Science*, 62:222–259, 1997.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice-Hall, Inc., 1989.

[MRH05] S. Montenegro, H.-P. Röser, and F. Huber. BOSS: Software and FPGA Middleware for the flying-laptop Microsatellite. In *DASIA*. Noordwijk: ESA Publications Division, 2005.

[NPW02] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[Sch99] S. Schneider. *Concurrent and Real Time Systems: The CSP Approach*. John Wiley & Sons, 1999.

[TW97] H. Tej and B. Wolff. A Corrected Failure Divergence Model for CSP in Isabelle/HOL. In *FME*, LNCS, pages 318–337. Springer, 1997.

Stochastic formal correctness of numerical algorithms

Marc Daumas[1], David Lester[2], Erik Martin-Dorel[1,3], and Annick Truffert[3]

[1]ELIAUS (EA 3679 UPVD) and [3]LAMPS (EA 4217 UPVD)
Perpignan, France 66860, {marc.daumas,erik.martin-dorel,truffert}@univ-perp.fr

[2]School of Computer Science, University of Manchester
Manchester, United Kingdom M13 9PL, david.r.lester@manchester.ac.uk

Abstract

We provide a framework to bound the probability that accumulated errors were never above a given threshold on numerical algorithms. Such algorithms are used for example in aircraft and nuclear power plants. This report contains simple formulas based on Lévy's and Markov's inequalities and it presents a formal theory of random variables with a special focus on producing concrete results. We selected four very common applications that fit in our framework and cover the common practices of systems that evolve for a long time. We compute the number of bits that remain continuously significant in the first two applications with a probability of failure around one out of a billion, where worst case analysis considers that no significant bit remains. We are using PVS as such formal tools force explicit statement of all hypotheses and prevent incorrect uses of theorems.

1 Introduction

Formal proof assistants are used in areas where errors can cause loss of life or significant financial damage, as well as in areas where common misunderstandings can falsify key assumptions. For this reason, formal proof assistants have been much used for floating point arithmetic [1, 2, 3, 4, 5] and probabilistic or randomized algorithms [6, 7]. Previous references link to a few projects using proof assistants such as ACL2 [8], HOL [9], Coq [10] and PVS [11].

All the above projects that deal with floating point arithmetic aim at containing worst case behavior. Recent work has shown that worst case analysis may be meaningless for systems that evolve for a long time, as encountered in industry. A good example is a process that adds numbers in ± 2 with a measure error of $\pm 2^{-24}$. If this process adds 2^{25} items, then the accumulated error is ± 2, and note that 10 hours of flight time at operating frequency of 1 kHz is approximately 2^{25} operations. Yet we easily agree that provided the individual errors are not correlated, the actual accumulated errors will continuously be much smaller than ± 2.

We present in Section 2 a few examples where this work can be applied. We focus on applications for *n counting in billions* and a probability of failure of about *one out of a billion*. Should one of these constraints be removed or lessened, the problems become much simpler. The main contributions of this work are the selection of a few theorems *amenable to formal methods* in a reasonable time, their application to *software and systems reliability*, and our work with PVS. Section 3 presents the formal background on probability with Markov's and Lévy's inequalities and how to use this theory to assert software and system reliability.

Doob-Kolmogorov's inequality was used in previous work [12]. It is an application of Doob's inequality, but it can be proved with elementary manipulations for second order moments. It is better than Lévy's inequality in the sense that it can applied to any sum of independent and centered variables. Yet it is limited by the fact that it bounds only second order moments.

2 Applications

Lévy's inequality works with independent symmetric random variables, as we safely assume in Sections 2.1 and 2.2. The applications presented in Section 2.3 cannot be treated by Lévy's inequality. Yet

E. Denney, D. Giannakopoulou, C.S. Păsăreanu (eds.); The First NASA Formal Methods Symposium, pp. 136-145

Listing 1: Accumulation of n values, which can also be viewed as a dot product with $d_i = b_i \times c_i$

```
$a_0 = 0$;
for ($i = 1$; $i <= n$; $i = i + 1$)
  $a_i = a_{i-1} \oplus d_i$; // is replaced by $a_i = a_{i-1} + d_i + X_i$
    // to accommodate round-off errors and existing errors on $d_i$
```

we may obtain similar results with Doob's inequality which is not restricted to symmetric variables. Alas, we foresee that the time needed to develop Doob's inequality in any of the formal tools available today is at least a couple of years. Automatic treatment of all the following applications may use interval arithmetic that has been presented in previous publications and is now available in formal tools [4, 5].

2.1 Long accumulations and dot products

A floating point number represents $v = m \times 2^e$ where e is the exponent, an integer, and m is the mantissa [13]. IEEE 754 standard [14] on floating point arithmetic uses sign-magnitude notation for the mantissa and the first bit b_0 of the mantissa is implicit in most cases ($b_0 = 1$), leading to the first definition in equation (1). Some circuits such as the TMS320 [15] use two's complement notation for m, leading to the second definition in equation (1). The sign s and all the b_i are either 0 or 1.

$$v = (-1)^s \times b_0.b_1 \cdots b_{p-1} \times 2^e \qquad \text{or} \qquad v = (b_0.b_1 \cdots b_{p-1} - 2 \times s) \times 2^e \tag{1}$$

In fixed point notation e is a constant provided by the data type and b_0 cannot be forced to 1. We define for any representable number v, the *unit in the last place* (ulp) function below, with the notations of equation (1).

$$\mathrm{ulp}(v) = 2^{e-p+1}$$

The example given in Listing 1 sums n values, $(a_1, \ldots, a_n)$. When the accumulation is performed with floating point arithmetic, each iteration introduces a new round-off error X_i. One might assume that X_i follows a continuous or discrete uniform distribution on the range $\pm u$ with $u = \mathrm{ulp}(a_i)/2$, as trailing digits of numbers randomly chosen from a logarithmic distribution [16, pp. 254–264] are approximately uniformly distributed [17]. A significantly different distribution may mean that the round-off error contains more than trailing digits.

Errors created by operators are discrete and they are not necessarily distributed uniformly [18]. As they are symmetric, we only have to bound the moments involved in our main result, as in equation (2).

Mean	Variance	4th order moment	6th order moment	8th order moment
$\mathbb{E}(X_i) = 0,$	$\mathbb{E}\left(X_i^2\right) \leq \frac{u^2}{3},$	$\mathbb{E}\left(X_i^4\right) \leq \frac{u^4}{5},$	$\mathbb{E}\left(X_i^6\right) \leq \frac{u^6}{7},$	$\mathbb{E}\left(X_i^8\right) \leq \frac{u^8}{9}.$

(2)

If a_i uses a directed rounding mode, we introduce $X_i' = X_i - \mathbb{E}(X_i)$ and we use equation (2) again. We may also assume that d_i carries some earlier round-off errors. In this case, X_i is a linear combination of ℓ round-off errors satisfying equation (2) for a given u.

If d_i is a data obtained by an accurate sensor, we may assume that the difference between d_i and the actual value $\overline{d_i}$ follows a normal distribution very close to a uniform distribution on the range $\pm u$, with some new value of u. In this case, we model the error $d_i - \overline{d_i}$ by a symmetric random variable and X_i is the sum of two random variables ($\ell = 2$) satisfying equation (2) for a given u.

Table 1: Number of significant bits with a probability of failure $\mathbb{P}\left(\max_{1\leq i\leq n}(|S_i|)\geq\varepsilon\right)$ bounded by P

u	n	ℓ	P	$2k$	$\varepsilon \approx$	Number of significant bits
2^{-24}	10^9	2	10^{-9}	2	68.825	−6.10
				4	0.42832	1.22
				6	0.085786	3.54
				8	0.040042	4.64
				44	0.010153	6.62
2^{-24}	10^9	10	10^{-10}	2	486.66	−8.92
				4	1.7031	−0.77
				6	0.28156	1.82
				8	0.11939	3.06
				48	0.023873	5.38
u	$u^{-3/2}$	1	$u^{3/2}$	2	$\sqrt[4]{4u^{-2}/9}$	$(\log_2 u - 1 + \log_2 3)/2$
				4	$\sqrt[8]{4u^{-1}/9}$	$(\log_2 u - 2 + 2\log_2 3)/8$
				6	$\sqrt[12]{100/81}$	$(2\log_2 3 - \log_2 10)/6$
				8	$\sqrt[16]{4900u/729}$	$(-\log_2 u - 2\log_2 70 + 6\log_2 3)/16$

After n iterations and assuming that all the errors introduced are independent, we want the probability that the accumulated errors have exceeded some user specified bound ε:

$$\mathbb{P}\left(\max_{1\leq i\leq n}(|S_i|)\geq\varepsilon\right)\leq P \quad\text{with}\quad S_n=\sum_{i=1}^{n}X_i \quad\text{and}\quad \begin{array}{l}X_i \text{ is the sum of } \ell \text{ symmetric random}\\ \text{variables satisfying equation (2) for a}\\ \text{given } u.\end{array} \tag{3}$$

Previous work used Doob-Kolmogorov's inequality. We will see in Section 3 that we can exhibit tighter bounds using Lévy's inequality followed by Markov's one. Table 1 presents the number of significant bits of the results (i.e. $-\log_2\varepsilon$) for some values of u, n, ℓ, and P in equation (3). These values are obtained by using one single value of u, as large as needed. Tighter results can be obtained by using a specific value of u for each random variable and each iteration.

2.2 Recursive filters operating for a long time

Recursive filters are commonly used in digital signal processing and appear, for example, in the programs executed by Flight Control Primary Computers (FCPC) of aircraft. Finite impulse response (FIR) filters usually involve a few operations and can be treated by worst case error analysis. However, infinite impulse response (IIR) filters may slowly drift.

The theory of signal processing provides that it is sufficient to study second order IIR with coefficients b_1 and b_2 such that the polynomial $X^2 - b_1X - b_2$ has no zero in $\mathbb{R}$. Listing 2 presents the pseudo-code of one such filter. A real implementation would involve temporary registers.

When implemented with fixed or floating point operations, each iteration introduces a compound error X_i that is a linear combination of ℓ individual errors satisfying equation (2) for a given u. As these filters are linear, we study the response to $d_0 = 1$ and $d_i = 0$ otherwise, to deduce the accumulated effect

Listing 2: Infinite impulse response (IIR) filter operated on n iterations

```
$y_{-1} = 0$; $y_0 = d_0$;
for ($i = 1$; $i <= n$; $i = i + 1$)
  $y_i = d_i \ominus b_1 y_{i-1} \ominus b_2 y_{i-2}$; // is replaced by $y_i = d_i + X_i - b_1 y_{i-1} - b_2 y_{i-2}$
     // to accommodate round-off errors and existing errors on $d_i$
```

of all the errors on the output of the filter. This response is defined as the sequence of real numbers such that

$$y_{-1} = 0, \qquad y_0 = 1, \quad \text{and} \quad y_n = -b_1 y_{n-1} - b_2 y_{n-2} \quad \text{for all} \quad n \geq 1.$$

This sequence can also be defined by the expression

$$y_n = b_2^{n/2} \sqrt{b_2 + 2b_1^2} \cos(\omega_0 + n\omega) \quad \text{with constants } \omega_0 \text{ and } \omega.$$

If the filter is bounded-input bounded-output (BIBO) stable, $0 < b2 < 1$ and the accumulated effect of the round-off errors is easily bounded by $\sqrt{b_2 + 2b_1^2} \ / \ (1 - \sqrt{b_2})$. Worst case error analysis is not possible on BIBO unstable systems. Our work and the example of Table 1 can be applied to such systems.

2.3 Long sums of squares and Taylor series expansion of programs

The previous programs introduce only first order effect of the round-off errors. We present here systems that involve higher order errors such as sum of square in Listing 3 that introduces D_i^2 and power series of all the random variables as in equation 4.

Assuming that d_i carries an error X_i in Listing 3, its contribution to the sum of square cannot be assumed to be symmetric. Lévy's inequality cannot be applied, but Doob's inequality provides a similar result, though a large number of foundational results are still lacking in existing formal libraries.

Listing 3: Sum of n squares

```
$a_0 = 0$;
for ($i = 1$; $i <= n$; $i = i + 1$)
  $a_i = a_{i-1} \oplus d_i \otimes d_i$; // is replaced by $a_i = a_{i-1} + X_i + (d_i + D_i)^2 = a_{i-1} + X_i + 2d_i D_i + d_i^2 + D_i^2$
     // to accommodate round-off errors and existing errors on $d_i$
```

The output of a system can always be seen as a function F of its input and its state $(d_1, \ldots, d_q)$. This point of view can be extended by considering that the output of the system is also a function of the various round-off errors $(X_1, \ldots, X_n)$ introduced at run-time. Provided this function can be differentiated sufficiently, Taylor series expansion at rank r provides that

$$\begin{aligned} F(d_1, \ldots, d_q, X_1, \ldots, X_n) &= F(d_1, \ldots, d_q, 0, \ldots, 0) \\ &+ \sum_{m=1}^{r} \frac{1}{m!} \left(\sum_{i=1}^{n} X_i \frac{\partial F}{\partial X_i} \right)^{[m]} (d_1, \ldots, d_q, 0, \ldots, 0) \\ &+ \left(\sum_{i=1}^{n} X_i \frac{\partial F}{\partial X_i} \right)^{[r+1]} (d_1, \ldots, d_q, \theta_1, \ldots, \theta_n), \end{aligned} \tag{4}$$

where θ_i is between 0 and X_i, and $(\cdot)^{[m]}$ is the symbolic power defined as

$$\left(\sum_{i=1}^{n} X_i \frac{\partial F}{\partial X_i}\right)^{[m]} = \sum_{1\leq i_1,\ldots,i_m\leq n} X_{i_1}\cdots X_{i_m} \frac{\partial^m F}{\partial X_{i_1}\cdots\partial X_{i_m}}.$$

When the Taylor series is stopped after $m = 2$, we can use Doob's inequality to provide results similar to the ones presented in Table 1, provided the X_i are symmetric and independent. Higher order Taylor series don't necessarily create sub-martingales but weaker results can be obtained by combining inequalities on sub-martingales.

3 Formal background on probability

3.1 A generic and formal theory of probability

We rebuilt the previously published theory of probability spaces [12] as a theory of Lebesgue's integration recently became fully available. The new PVS development in Figure 1, still takes three parameters: $\mathtt{T}$, the sample space, $\mathscr{S}$, a σ-algebra of permitted events, and $\mathbb{P}$, a probability measure, which assigns to each permitted event in $\mathscr{S}$, a probability between 0 and 1. Properties of probability that are independent of the particular details of $\mathtt{T}$, $\mathscr{S}$, and $\mathbb{P}$ are then provided in this file.

A random variable X is a measurable application from $(\mathtt{T},\mathscr{S})$ to any other measurable space $(\mathtt{T}',\mathscr{S}')$. In most theoretical developments of probability, $\mathtt{T}$, $\mathscr{S}$, and $\mathbb{P}$ remain generic, as computations are carried on $\mathtt{T}'$. Results on real random variables use $\mathtt{T}' = \mathbb{R}$ whereas results on random vectors use $\mathtt{T}' = \mathbb{R}^n$. Yet both theories refer explicitly to the Borel sets of $\mathtt{T}'$, the elements of the smallest σ-algebra containing the open sets of $\mathtt{T}'$.

As the Borel sets of $\mathbb{R}$ and $\mathbb{R}^n$ are difficult to grasp, most authors consider finite $\mathtt{T}$ and $\mathscr{S} = \mathscr{P}(\mathtt{T})$ for discrete random variables in introductory classes. This simpler analysis is meant only for educational purposes. Handling discrete and continuous random variables through different $\mathtt{T}$ and $\mathscr{S}$ parameters is not necessary and it is contrary to most uses of probability spaces in mathematics. Both discrete and continuous variables can be described on the same generic $\mathtt{T}$, $\mathscr{S}$, and $\mathbb{P}$ parameters in spite of their differences. Similarly, many authors work on sections $\{X \leq x\}$ rather than using *the inverse images of Borel sets* of $\mathtt{T}'$ because the latter are difficult to visualize. Such a simplification is valid thanks to Dynkin's systems. But using abstract Borel sets rather than sections in formal methods often leads to easier proofs.

3.2 A concrete theory of expectation

The previous theory of random variables [12] made it possible to define them and to use and derive their properties. Very few results were enabling users to actually compute concrete results on random variables. Most of such results lie on a solid theory of the expected value. As most theorems in the latter theory are corollaries of a good theory of Lebesgue's integration, we have developed a formal measure theory based on Lebesgue's integration and we develop formal theorems on expected values, as needed in our applications.

The expected value is the (unique) linear and monotonous operator $\mathbb{E}$ on the set of $\mathbb{P}$-integrable random variables that satisfies Beppo-Lévy's property and such that $\mathbb{E}(\chi_A) = \mathbb{P}(A)$ for all $A \in \mathscr{S}$. We can also use the following definition when Lebesgue's integral exists:

$$\mathbb{E}(X) = \int_{\mathtt{T}} X \,\mathrm{d}\mathbb{P}.$$

Markov's inequality below is heavily used to obtain concrete properties on random variables.

```
probability_space[T:TYPE+,           (IMPORTING topology@subset_algebra_def[T])
                  S:sigma_algebra,   (IMPORTING probability_measure[T,S])
                  P:probability_measure]: THEORY
BEGIN
   IMPORTING topology@sigma_algebra[T,S],
             probability_measure[T,S],
             continuous_functions_aux[real],
             measure_theory@measure_space[T,S],
             measure_theory@measure_props[T,S,to_measure(P)]

   limit: MACRO [(convergence_sequences.convergent)->real]
                                                  = convergence_sequences.limit
   h  : VAR borel_function
   A,B: VAR (S)
   x,y: VAR real
   n0z: VAR nzreal
   t:   VAR T
   n:   VAR nat
   X,Y: VAR random_variable
   XS:  VAR [nat->random_variable]

   null?(A)         :bool = P(A) = 0
   non_null?(A)     :bool = NOT null?(A)
   independent?(A,B):bool = P(intersection(A,B)) = P(A) * P(B)

   zero: random_variable = (LAMBDA t: 0)
   one:  random_variable = (LAMBDA t: 1)

   ; % Needed for syntax purposes!
   <=(X,x): (S) = {t | X(t) <= x}; % < = >= > /= omitted

   complement_eq: LEMMA complement(X = x) = (X /= x) % More omitted

   ; % Needed for syntax purposes!
   +(X,x): random_variable = (LAMBDA t: X(t) + x); % More omitted

   borel_comp_rv_is_rv: JUDGEMENT o(h,X) HAS_TYPE random_variable
   partial_sum_is_random_variable:
     LEMMA random_variable?(LAMBDA t: sigma(0,n,LAMBDA n: XS(n)(t)))

   distribution_function?(F:[real->probability]):bool
                                   = EXISTS X: FORALL x: F(x) = P(X <= x)
   distribution_function: TYPE+ = (distribution_function?) CONTAINING
                                  (LAMBDA x: IF x < 0 THEN 0 ELSE 1 ENDIF)
   distribution_function(X)(x):probability = P(X <= x)

   convergence_in_distribution?(XS,X):bool
     = FORALL x: continuous(distribution_function(X),x) IMPLIES
                 convergence((LAMBDA n: distribution_function(XS(n))(x)),
                                        distribution_function(X)(x))

   invert_distribution:   LEMMA LET F = distribution_function(X) IN
                                P(x < X) = 1 - F(x)
   interval_distribution: LEMMA LET F = distribution_function(X) IN
                                x <= y IMPLIES
                                P(intersection(x < X, X <= y)) = F(y) - F(x)
   limit_distribution:    LEMMA LET F = distribution_function(X) IN
                                P(X = x) = F(x) - limit(LAMBDA n: F(x-1/(n+1)))

   F: VAR distribution_function
   distribution_0:                LEMMA convergence(F o (lambda (n:nat): -n),0)
   distribution_1:                LEMMA convergence(F,1)
   distribution_increasing:       LEMMA increasing?[real](F)
   distribution_right_continuous: LEMMA right_continuous(F)
END probability_space
```

Figure 1: Abbreviated probability space file in PVS

Theorem 1 (Markov's inequality)**.** *For any random variable X and any constant $\varepsilon > 0$,*

$$\mathbb{P}(|X| \geq \varepsilon) \leq \frac{\mathbb{E}(|X|)}{\varepsilon}.$$

Many theorems relate to independent random variables and their proofs are much easier once independence is well defined. The family $(X_1, \ldots, X_n)$ is independent if and only if, for any family of Borel sets $(B_1, \ldots, B_n)$,

$$\mathbb{P}\left(\bigcap_{i=1}^{n} (X_i \in B_i)\right) = \prod_{i=1}^{n} \mathbb{P}(X_i \in B_i).$$

The following characteristic property is used a lot on families of independent variables:
For any family of Borelean functions $(h_1, \ldots, h_n)$ such that the $h_i(X_i)$ are $\mathbb{P}$-integrable,

$$\mathbb{E}\left(\prod_{i=1}^{n} h_i(X_i)\right) = \prod_{i=1}^{n} \mathbb{E}(h_i(X_i)).$$

It is worth noting that the fact that n random variables are independent is not equivalent to the fact that any pair of variables is independent, and cannot be built recursively from $n-1$ independent random variables.

Future work may lead us to implement a theory of the law $\mathbb{P}_X$ associated to each random vector $X : \mathtt{T} \longrightarrow \mathbb{R}^n$, with a "transfer" theorem for any Borelean function $h : \mathbb{R}^n \longrightarrow \mathbb{R}$ below, and most properties of Lebesgue's integral including Fubini's theorem.

$$\mathbb{E}(h(X)) = \int_{\mathtt{T}} h(X)\, \mathrm{d}\mathbb{P} = \int_{\mathbb{R}^n} h\, \mathrm{d}\mathbb{P}_X$$

3.3 Almost certain a priori error bound

What we are actually interested in is whether a series of calculations might accumulate a sufficiently large error to become meaningless. In the language we have developed, we are computing the probability that a sequence of n calculations has failed because it has exceeded the ε error-bound somewhere.

Theorem 2 (Corollary of Lévy's inequality)**.** *Provided the (X_n) are independent and symmetric the following property holds for any constant ε.*

$$\mathbb{P}\left(\max_{1 \leq i \leq n}(|S_i|) \geq \varepsilon\right) \leq 2\,\mathbb{P}(|S_n| \geq \varepsilon)$$

Proof. We use a proof path similar to the one published in [19]. We define $S_n^{(j)}$ below with Dirichlet's operator δ_P, which is equal to 1 if the predicate holds and 0 otherwise. As the X_n are symmetric, the random variables S_n and $S_n^{(j)}$ share the same probability density function.

$$S_n^{(j)} = \sum_{i=1}^{n} (-1)^{\delta_{i>j}} X_i$$

We now define $N = \inf\{k \text{ such that } |S_k| \geq \varepsilon\}$ with the addition that $\inf \varnothing = +\infty$ and similarly $N^{(j)} = \inf\{k \text{ such that } |S_k^{(j)}| \geq \varepsilon\}$. Events $\max_{1 \leq i \leq n}(|S_i|) \geq \varepsilon$ and $N \leq n$ are identical. Furthermore,

$$\mathbb{P}(|S_n| \geq \varepsilon) = \sum_{j=1}^{n} \mathbb{P}(|S_n| \geq \varepsilon \cap N = j) = \sum_{j=1}^{n} \mathbb{P}\left(|S_n^{(j)}| \geq \varepsilon \cap N = j\right).$$

As soon as $j \leq n$, $2S_j = S_n + S_n^{(j)}$ and $2|S_j| = |S_n| + |S_n^{(j)}|$. Therefore, the event $\{|S_j| \geq \varepsilon\}$ is included in $\{|S_n| \geq \varepsilon\} \cup \{|S_n^{(j)}| \geq \varepsilon\}$ and

$$\mathbb{P}(N \leq n) = \sum_{j=1}^{n} \mathbb{P}(|S_j| \geq \varepsilon \,\cap\, N = j) \leq \sum_{j=1}^{n} \mathbb{P}(|S_n| \geq \varepsilon \,\cap\, N = j) + \sum_{j=1}^{n} \mathbb{P}\left(|S_n^{(j)}| \geq \varepsilon \,\cap\, N = j\right).$$

This ends the proof of Lévy's inequality. □

Should we need to provide some formula beyond the hypotheses of Lévy's inequality, we may have to prove Doob's original inequality for martingales and sub-martingales [20] in PVS. It follows a proof path very different from Doob-Kolmogorov's inequality but it is not limited to second order moment and it can be applied to any sub-martingale S_i^{2k} with $k \geq 1$ to lead to

$$\mathbb{P}\left(\max_{1\leq i\leq n}\left(|S_i|\right) \geq \varepsilon\right) \leq \frac{\mathbb{E}\left(S_n^{2k}\right)}{\varepsilon^{2k}}.$$

Here, we use Markov's inequality applied to S_n^{2k} in order to obtain the results of Table 1:

$$\mathbb{P}(|S_n| \geq \varepsilon) = \mathbb{P}\left(S_n^{2k} \geq \varepsilon^{2k}\right) \leq \mathbb{E}\left(S_n^{2k}\right)/\varepsilon^{2k}.$$

Formulas

$$\begin{aligned}
\mathbb{E}(S_n^2) &= u^2\left(\tfrac{1}{3}n\right) \\
\mathbb{E}(S_n^4) &= u^4\left(\tfrac{1}{5}n + \tfrac{1}{3}n(n-1)\right) \\
\mathbb{E}(S_n^6) &= u^6\left(\tfrac{1}{7}n + n(n-1) + \tfrac{5}{9}n(n-1)(n-2)\right) \\
\mathbb{E}(S_n^8) &= u^8\left(\tfrac{1}{9}n + \tfrac{41}{15}n(n-1) + \tfrac{14}{3}n(n-1)(n-2) + \tfrac{35}{27}n(n-1)(n-2)(n-3)\right)
\end{aligned}$$

are based on the binomial formula for independent symmetric random variables

$$\mathbb{E}\left(S_n^{2k}\right) = \sum_{k_1+k_2+\cdots+k_n=k} (2k)!\frac{\mathbb{E}\left(X_1^{2k_1}\right)}{(2k_1)!}\frac{\mathbb{E}\left(X_2^{2k_2}\right)}{(2k_2)!}\cdots\frac{\mathbb{E}\left(X_n^{2k_n}\right)}{(2k_n)!}.$$

Proof. We first prove the formula below by induction on n for any exponent m.

$$\mathbb{E}\left(S_n^m\right) = \sum_{m_1+m_2+\cdots+m_n=m} m!\frac{\mathbb{E}\left(X_1^{m_1}\right)}{m_1!}\frac{\mathbb{E}\left(X_2^{m_2}\right)}{m_2!}\cdots\frac{\mathbb{E}\left(X_n^{m_n}\right)}{m_n!}$$

It holds for $n = 1$. We now write the following identity based on the facts that X_n are independent and symmetric. $\mathbb{E}\left(S_{n+1}^m\right) = \mathbb{E}\left((S_n + X_{n+1})^m\right)$ is also equal to

$$\mathbb{E}\left(\sum_{m_{n+1}=0}^{p} \frac{m!}{(m-m_{n+1})!m_{n+1}!} X_{n+1}^{m_{n+1}} S_n^{m-m_{n+1}}\right) = \sum_{m_{n+1}=0}^{p} \frac{m!}{(m-m_{n+1})!m_{n+1}!} \mathbb{E}\left(X_{n+1}^{m_{n+1}}\right)\mathbb{E}\left(S_n^{m-m_{n+1}}\right)$$

We use the induction hypothesis on $\mathbb{E}\left(S_n^{m-m_{n+1}}\right)$ and we collapse the summations. We end the proof for the even values of m after noticing that $\mathbb{E}\left(X_i^{2k+1}\right) = 0$ for any i and any k, since the X_n are symmetric. □

4 Perspectives and concluding remarks

To the best of our knowledge, this paper presents the first application of Lévy's inequality to software and system reliability of very long processes with an extremely low rate of failure. Our results allow any one to develop safe upper limits on the number of operations that a piece of numeric software should be permitted to undertake. In addition, we are finishing certification of our results with PVS. The major restriction lies in the fact that the slow process of proof checking has forced us to insist that individual errors are symmetric.

At the time we are submitting this work, the bottleneck is the full certification of more results using PVS proof assistant. Yet this step is compulsory to provide full certification to future industrial uses. We anticipate no problem as these results are gathered in textbooks in computer science and mathematics. This library and future work will be included into NASA Langley PVS library[1] as soon as it becomes stable.

The main contribution of this work is that we selected theorems that produce significant results for extremely low probabilities of failure of systems that run for a long time, and that are amenable to formal methods. During our work, we discarded many mathematical methods that would need too many operations or that would be too technical to be implemented with existing formal tools.

Notice that this work can be applied to any sequence of independent and symmetric random variables that satisfy equation (2). It is worth pointing out one more time that violating our assumption (independence of errors) would lead to worse results, so one should treat the limit we have deduced with caution, should this assumption not be met.

Acknowledgment

This work has been partially funded by CNRS PICS 2533 and by the EVA-Flo project of the ANR. It was initiated while one of the authors was an invited professor at the University of Perpignan Via Domitia.

References

[1] D. M. Russinoff, "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*, vol. 1, pp. 148–200, 1998. [Online]. Available: http://www.onr.com/user/russ/david/k7-div-sqrt.ps

[2] J. Harrison, "Formal verification of floating point trigonometric functions," in *Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design*, W. A. Hunt and S. D. Johnson, Eds., Austin, Texas, 2000, pp. 217–233. [Online]. Available: http://www.springerlink.com/link.asp?id=wxvaqu9wjrgc8l99

[3] S. Boldo and M. Daumas, "Representable correcting terms for possibly underflowing floating point operations," in *Proceedings of the 16th Symposium on Computer Arithmetic*, J.-C. Bajard and M. Schulte, Eds., Santiago de Compostela, Spain, 2003, pp. 79–86. [Online]. Available: http://perso.ens-lyon.fr/marc.daumas/SoftArith/BolDau03.pdf

[4] M. Daumas and G. Melquiond, "Certification of bounds on expressions involving rounded operators," *ACM Transactions on Mathematical Software*, vol. 37, no. 1, 2010, to appear. [Online]. Available: http://hal.archives-ouvertes.fr/hal-00127769

[5] M. Daumas, D. Lester, and C. Muñoz, "Verified real number calculations: A library for interval arithmetic," *IEEE Transactions on Computers*, vol. 58, no. 2, pp. 226–237, 2009. [Online]. Available: http://dx.doi.org/10.1109/TC.2008.213

[1] `http://shemesh.larc.nasa.gov/fm/ftp/larc/PVS-library/pvslib.html`.

[6] J. Hurd, “Formal verification of probabilistic algorithms,” Ph.D. dissertation, University of Cambridge, 2002. [Online]. Available: http://www.cl.cam.ac.uk/~jeh1004/research/papers/thesis.pdf

[7] P. Audebaud and C. Paulin-Mohring, “Proofs of randomized algorithms in Coq,” in *Proceedings of the 8th International Conference on Mathematics of Program Construction*, T. Uustalu, Ed., Kuressaare, Estonia, 2006, pp. 49–68. [Online]. Available: http://dx.doi.org/10.1007/11783596_6

[8] M. Kaufmann, P. Manolios, and J. S. Moore, *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.

[9] M. J. C. Gordon and T. F. Melham, Eds., *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[10] G. Huet, G. Kahn, and C. Paulin-Mohring, *The Coq proof assistant: a tutorial: version 8.0*, 2004. [Online]. Available: ftp://ftp.inria.fr/INRIA/coq/current/doc/Tutorial.pdf.gz

[11] S. Owre, J. M. Rushby, and N. Shankar, “PVS: a prototype verification system,” in *11th International Conference on Automated Deduction*, D. Kapur, Ed. Saratoga, New-York: Springer-Verlag, 1992, pp. 748–752. [Online]. Available: http://pvs.csl.sri.com/papers/cade92-pvs/cade92-pvs.ps

[12] M. Daumas and D. Lester, “Stochastic formal methods: an application to accuracy of numeric software,” in *Proceedings of the 40th IEEE Annual Hawaii International Conference on System Sciences*, Waikoloa, Hawaii, 2007, p. 7 p. [Online]. Available: http://hal.ccsd.cnrs.fr/ccsd-00081413

[13] D. Goldberg, “What every computer scientist should know about floating point arithmetic,” *ACM Computing Surveys*, vol. 23, no. 1, pp. 5–47, 1991. [Online]. Available: http://doi.acm.org/10.1145/103162.103163

[14] D. Stevenson *et al.*, “An American national standard: IEEE standard for binary floating point arithmetic,” *ACM SIGPLAN Notices*, vol. 22, no. 2, pp. 9–25, 1987.

[15] *TMS320C3x — User’s guide*, Texas Instruments, 1997. [Online]. Available: http://www-s.ti.com/sc/psheets/spru031e/spru031e.pdf

[16] D. E. Knuth, *The Art of Computer Programming: Seminumerical Algorithms*. Addison-Wesley, 1997, third edition.

[17] A. Feldstein and R. Goodman, “Convergence estimates for the distribution of trailing digits,” *Journal of the ACM*, vol. 23, no. 2, pp. 287–297, 1976. [Online]. Available: http://doi.acm.org/10.1145/321941.321948

[18] J. Bustoz, A. Feldstein, R. Goodman, and S. Linnainmaa, “Improved trailing digits estimates applied to optimal computer arithmetic,” *Journal of the ACM*, vol. 26, no. 4, pp. 716 – 730, 1979. [Online]. Available: http://doi.acm.org/10.1145/322154.322162

[19] J. Bertoin, “Probabilités,” 2001, cours de licence de mathématiques appliquées. [Online]. Available: http://www.proba.jussieu.fr/cours/bertoin.pdf

[20] J. Neveu, Ed., *Martingales à temps discret*. Masson, 1972.

Deductive Verification of Cryptographic Software

José Bacelar Almeida Manuel Barbosa Jorge Sousa Pinto
Bárbara Vieira

CCTC / Departamento de Informática
Universidade do Minho
Campus de Gualtar, 4710-Braga, Portugal
{jba,mbb,jsp,barbarasv}@di.uminho.pt

Abstract

We report on the application of an off-the-shelf verification platform to the RC4 stream cipher cryptographic software implementation (as available in the openSSL library), and introduce a deductive verification technique based on self-composition for proving the absence of error propagation.

1 Introduction

Software implementations of cryptographic algorithms and protocols are at the core of security functionality in many IT products. They are also vital components in safety-critical systems, namely in the military area. However, the development of this class of software products is understudied as a domain-specific niche in software engineering. This development is clearly distinct from other areas of software engineering due to a combination of factors. Firstly, cryptography is an inherently inter-disciplinary subject. The design and implementation of cryptographic software draws on skills from mathematics, computer science and electrical engineering. Such a rich body of research is difficult to absorb and apply without error, even for the most expert software engineer. Secondly, security is notoriously difficult to sell as a feature in software products, even when clear risks such as identity theft and fraud are evident. An important implication of this fact is that security needs to be as close to invisible as possible in terms of computational and communication load. As a result, it is critical that cryptographic software is optimised aggressively, without altering the security semantics. Finally, typical software engineers develop systems focused on desktop class processors within computers in our offices and homes. Cryptographic software is implemented on a much wider range of devices, from embedded processors with very limited computational power, memory and autonomy, to high-end servers, which demand high-performance and low-latency. Not only must cryptographic software engineers understand each platform and the related security requirements, they must optimise each algorithm with respect to each platform.

CACE (Computer Aided Cryptography Engineering, `http://www.cace-project.eu`) is an European Project that targets the lack of support currently offered to cryptographic software engineers. The central objective is the development of a tool-box of domain-specific languages, compilers and libraries, that supports the production of high quality cryptographic software. Specific components within the tool-box will address particular software development problems and processes; and combined use of the constituent tools is enabled by designed integration between their interfaces. The project started in 2008 and will run for three years.

This paper stems from CACE - Work Package 5, which aims to add formal methods technology to the tool-box, as a means to increase the degree of assurance than can be provided by the development process. We describe promising early results obtained during our exploration of existing verification techniques and tools used to construct high-assurance software implementations for other domains. Specifically, we present our achievements in using an off-the-shelf verification tool to validate security-relevant properties of a `C` implementation of the `RC4` encryption scheme that is included in the well-known open-source library `openSSL` `http://www.openssl.org`). Additionally, we have developed a framework that permits automating most of the steps required to verify non-interference using the self-composition approach introduced in [3]. We believe this verification technique may be of independent interest.

E. Denney, D. Giannakopoulou, C.S. Păsăreanu (eds.); The First NASA Formal Methods Symposium, pp. 146-155

2 Background

Deduction-based Program Verification. The techniques employed in this paper are based on Hoare Logic [9], brought to practice through the use of *contracts* – specifications consisting of preconditions and post-conditions, annotated into the programs. In recent years verification tools based on contracts have become more and more popular, as their scope evolved from toy languages to very realistic fragments of languages like C, C#, or Java.

In a nutshell, a verification infra-structure consists of a verification conditions generator (VCGen for short) and a proof tool, which may be either an automatic theorem prover or an interactive proof assistant. The VCGen reads in the annotated code (which contains contracts and other annotations meant to facilitate the verification, such as loop invariants and variants) and produces a set of proof obligations known as *verification conditions*, that are sent to the proof tool. The correctness of the VCGen guarantees that if all the proof obligations are valid then the program is correct with respect to its specification.

The concrete tools we have used in this work were `Frama-c` [4], a tool for the static analysis of `C` programs annotated using the ANSI-C Specification Language (ACSL [4]). `Frama-c` contains a multi-prover VCGen [8], that we used together with a set of proof tools that included the Coq proof assistant [18], and the Simplify [6] and Ergo [5] automatic theorem provers. Both `Frama-c` and ACSL are very much work in progress; we have used the latest release of `Frama-c` (known as Lithium).

A few features of the `Frama-c` VCGen should be emphasized in the context of our work. The first is the possibility to export individual proof obligations to different proof tools, which allows users to first try discharging them with one or more automatic provers, leaving the harder conditions to be exported to an interactive proof assistant. A second feature is the declaration of lemmata: results that can be used to prove goals, but give themselves origin to new goals. We have used this possibility to clearly isolate the parts of proofs that cannot be handled automatically. It is often the case that the difficulty resides in a single result that has to be proved interactively, but whose existence as a lemma allows for all the remaining conditions to be proved automatically. The possibility to define inductive predicates (passed on to the proof tool as axiomatized predicates, and kept as inductive definitions if the tool supports them, as is the case of Coq), and the *labels* mechanism that allows the value of an expression in a program state to be used explicitly in logical expressions, are two additional features that we have made use of.

Finally, `Frama-c` generates *safety conditions* – special verification conditions (not generated from contracts), whose validity implies that the program will execute safely with respect to, say, memory accesses or integer overflow. Safety verification may be run independently of functional verification.

Information Flow Security. Information flow security refers to a class of security policies that constrain the ways in which information can be manipulated during program execution. These properties can be formulated in terms of non-interference between high-confidentiality input variables and low-confidentiality output variables, and are usually verified using a special extended type system. A dual formulation permits capturing security policies which constrain information flow from non-trustworthy (or low-integrity) inputs, to trusted (or high-integrity) outputs. We provide a short overview of developments in this area related to the work in this paper in Section 6.

Consider the more common case of secure information-flow that aims to preserve data confidentiality. Information may flow from high-security to low-security either directly via assignment instructions, or indirectly. The following code from Terauchi and Aiken [17] computes in f_1 the n^{th} Fibonacci number and then assigns a value to l that depends on the value of f_1.

```
while (n > 0) { f1 = f1 + f2; f2 = f1 - f2; n--; }
if (f1 > k) l = 1; else l = 0;
```

Let l be low security and n high security; then clearly there is an indirect information leakage from n to l, since the assignment `l = 1` is guarded by a condition that depends on the value of f_1, and assignments to the latter variable are performed inside a loop that is controlled by the high security condition $n > 0$. The program is thus insecure. If n were not high security, the program would be secure.

Type-based analyses would address the problem by tracking assignments to low security variables. Observe, however, that this fails to capture subtle situations where an apparently insecure program is in fact secure. If the last line of the program were changed to `if (f1 > k) l = E1; else l = E2;` where `E1`, `E2` are two expressions that evaluate to the same value, then the program should be classified as high security, since there is no way to tell from the final value of l anything about f_1. Type-based analyses would typically fail to distinguish this from the previous program: both would be conservatively classified as insecure. An alternative approach is to define a program as secure if different terminating executions, starting from states that differ only in the values of high-security variables, result in final states that are equivalent with respect to the values of low-security variables. This approach, based on the language semantics, avoids the excessively conservative behaviour of the previous method.

More formally, let $V_\uparrow$ and $V_\downarrow$ denote respectively the sets of high-security and low-security variables of C, and $V_\downarrow{}' = \mathit{Vars}(C) \setminus V_\uparrow$. We write $(C,\sigma) \Downarrow \tau$ to denote the fact that when executed in state σ, C stops in state τ (i.e. $\Downarrow$ is the evaluation relation in a big-step semantics of the underlying language). Then the program C is secure if $\sigma \overset{V_\downarrow{}'}{=} \tau \;\wedge\; (C,\sigma) \Downarrow \sigma' \;\wedge\; (C,\tau) \Downarrow \tau' \implies \sigma' \overset{V_\downarrow}{=} \tau'$ for arbitrary states σ, τ, where $\sigma \overset{X}{=} \tau$ denotes the fact that $\sigma(x) = \tau(x)$ for all $x \in X$, i.e. σ and τ are X-indistinguishable.

Self-composition. The operational definition of non-interference involves two executions of the program, but it can be reformulated so that a single execution (of a transformed program) is considered, using the *self-composition* technique [3]. Given some program C, let C^s be the program that is equal to C except that every variable x is renamed to a fresh variable x^s. Non-interference can be formulated considering a single execution of the self-composed program $C;C^s$. Note that any state σ of $C;C^s$ can be partitioned into two states with disjoint domains $\sigma = \sigma^o \cup \sigma^s$ where $\mathit{dom}(\sigma^o) = \mathit{Vars}(C)$ and $\mathit{dom}(\sigma^s) = \{x^s \mid x \in \mathit{Vars}(C)\}$. C is information-flow secure if any terminating execution of the self-composed program $C;C^s$, starting from a state σ such that σ^o and σ^s differ only in the values of high-security variables, results in a final state σ' such that σ'^o and σ'^s are equivalent with respect to the values of low-security variables. This can be formulated without referring explicitly to the state partition: if $\sigma(x) = \sigma(x^s)$ for all $x \in V_\downarrow{}'$ and $(C;C^s,\sigma) \Downarrow \sigma'$, then $\sigma(x) = \sigma(x^s)$ for all $x \in V_\downarrow$.

Self-composition allows for a shift from an operational semantics-based to an axiomatic semantics-based definition, since the former can be written as the Hoare logic partial correctness specification $\{\bigwedge_{x \in V_\downarrow{}'} x = x^s\}\, C;C^s\, \{\bigwedge_{x \in V_\downarrow} x = x^s\}$ which can be verified, for instance, as an ACSL contract.

An ACSL annotated program can easily be written for this example. Note that ACSL contracts pertain to functions, and as such a C function must be created whose body is the self-composed program. The contract contains a precondition stating that initial values of all non-high security variables are pairwise equal (for both composed copies of the program), and a post-condition stating that the values of low security variables are pairwise equal. Running `Frama-c` on this code, with the obvious 'control' invariants for each loop (regarding the minimum value of the variables n and n^s) also annotated, produces (with safety verification turned off) a total of 10 proof obligations, 2 of which cannot be automatically discharged. Admittedly, the control invariants do not sufficiently describe what the loops do (in particular, the fact that they are calculating Fibonacci numbers), and for this reason the post-condition cannot be proved, whether `n==ns` is included in the precondition (meaning that n is not considered high-security) or not. The verification thus fails to recognize a secure program. The difficulties of applying self-composition in practice have been previously identified (see Section 6); in this paper we introduce a technique that can help in successfully applying it with the help of a deductive verification system.

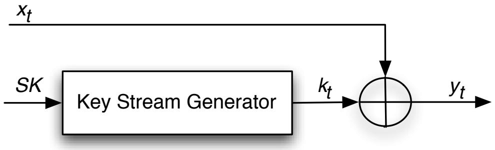


Figure 1: Block diagram of the RC4 cipher

3 RC4 and Verification of its Safety Properties in openSSL

RC4 is a symmetric cipher designed by Ron Rivest at RSA labs in 1987. It is a proprietary algorithm, and its definition was never officially released. Source code that allegedly implements the RC4 cipher was leaked on the Internet in 1994, and this is commonly known as ARC4 due to trademark restrictions. In this work we will use the RC4 denomination to denote the definition adopted in literature [16]. RC4 is widely used in commercial products, as it is included as one of the recommended encryption schemes in standards such as TLS, WEP and WPA. In particular, an implementation of RC4 is provided in the pervasively used open-source library openSSL, which we selected as the case study for this paper.

In cryptographic terms, RC4 is a synchronous stream cipher, which means that it is structured as two independent blocks, as shown in Figure 1. The security of the RC4 cipher resides in the strength of the key stream generator, which is initialized with a secret key *SK*. The key stream output is a byte sequence k_t that approximates a perfectly random bit string, and is independent of plaintext and ciphertext (we adopt the most widely used version of RC4, implemented in openSSL, which operates over byte-sized words). The encryption operation consists simply of XOR-ing each plaintext byte x_t with a fresh key stream byte k_t. Decryption operates in an identical way. The key stream generator operates over a state which includes a permutation table $S = (S[l])_{l=0}^{l=255}$ of (unsigned) byte-sized values, and two (unsigned) byte-sized indices i and j. We denote the values of these variables at time t by S_t, i_t and j_t. The state and output of the key stream generator at time t (for $t \geq 1$) are calculated according to the following recurrence, in which all additions are carried out modulo 256.

$$i_t = i_{t-1} + 1 \ , \ j_t = j_{t-1} + S_{t-1}[i_t] \ , \ S_t[i_t] = S_{t-1}[j_t] \ , \ S_t[j_t] = S_{t-1}[i_t] \ , \ k_t = S_t[S_t[i_t] + S_t[j_t]]$$

The initial values of the indices i_0 and j_0 are set to 0, and the initial value of the permutation table S_0 is derived from the secret key *SK*. The details of this initialisation are immaterial for the purpose of this paper, as they are excluded from the analysis.

We present in the full version of this paper [1] the C implementation of RC4 included in the openSSL open-source. The function receives the current state of the RC4 key stream generator (key), and two arrays whose length is provided in parameter len. The first array contains the plaintext (indata), and the second array will be used to return the ciphertext (outdata). The same function can be used for decryption by providing the ciphertext in the indata buffer. We note that this implementation is much less readable than the concise description provided above, as it has been optimised for speed using various tricks, including macro inlining and loop unrolling. We have added ACSL annotations to this RC4 implementation in order to facilitate the verification of a set of safety properties. These comprise memory safety, including the absence of buffer overflows, and also the absence of numeric errors due to overflows in integer calculations. This annotated version of RC4 was processed using Frama-c, which generated 869 verification conditions. All of these verification conditions could be automatically proved using tools such as Alt-Ergo and Z3. We highlight some important points in this verification work.

Frama-C interprets C primitive types (e.g. char, int, etc.) as integers with different precisions, based on the number of bits of each type. This means that a number of proof obligations must be automatically generated to ensure the validity of each arithmetic operation, by imposing range limits on the corresponding results. Proof obligations that ensure that every memory access is safe are also automatically generated. Note that, even though these proof obligations do not result from explicit assertions made by the programmer, it is usually necessary to annotate the code with preconditions. These limit the analysis to function executions for which the caller has provided valid inputs. For example, in RC4 one must assume that the `indata` and `outdata` arrays have a valid addressable range between `0` and `len-1` for the proof obligations to be valid. The required preconditions also include the validity of the memory region in which the RC4 key stream generator state (`key`) is passed to the function, and bounding the length of the input and output buffers (`len`).

The verification of the proof obligations generated by `Frama-C` also required the annotation of the RC4 code with loop invariants. These invariants are critical to enable the deductive verification process to reason about the program state before, during and after each loop execution. For example, in the first loop, the invariant permits establishing that index `i` lies between `0` and `len>>3L`, and also keeping track of how the `indata` and `outdata` pointer values change during the loop execution. Given the high number of proof obligations to be proved, and to guide the automatic provers in the process of establishing the validity of some of these conditions, additional assertions were introduced in the code. For example, at the end of the first loop, one assertion is introduced to *force* the provers to pinpoint the condition that must be valid at the end of the loop execution. Finally, and given that cryptographic code tends to make use of some arithmetic operators that are not commonly used in other application domains, we noted that the proof tools lacked appropriate support in some cases, namely for bit-wise operators. To overcome this difficulty we added some very simple axioms to the annotated RC4 code which express bounds on the outputs of these operators.

4 Towards Automating Proofs by Self-Composition.

Let us consider again the Fibonacci example of Section 2. The success of being able to prove automatically the security of programs using a Hoare logic-based tool such as `Frama-c` depends totally on adequately annotating the program. The ACSL program could be annotated with loop invariants describing the functional behaviour of each loop, which would require a formalisation of Fibonacci numbers. This kind of functional property would allow for the post-condition to be proved (both loops would be annotated as calculating Fibonacci numbers); however, such properties are hard to produce and not adequate for an automatic approach that should be applicable independently of what the code does.

A more feasible alternative is to use an abstract invariant that captures in logical form the state transformation associated with the loop, written as a formula that uses an inductively defined predicate (as supported by `Frama-c`). We will refer to these as the loop's *natural invariant*. Let $\vec{v}$ denote the vector of variables used inside a given loop. The predicate $\mathit{natinv}(\vec{v_i},\vec{v})$ will be defined with the meaning that execution of the loop started with initial values of the variables given by $\vec{v_i}$; and the current values of the variables are given by $\vec{v}$. The inductive definition of this predicate has in general a base case of the form $\mathit{natinv}(\vec{v_i},\vec{v_i})$ (corresponding to the loop initialization) and an inductive case of the form $\mathit{natinv}(\vec{v_i},\vec{v_t}) \rightarrow B \rightarrow R(\vec{v_t},\vec{v}) \rightarrow \mathit{natinv}(\vec{v_i},\vec{v})$, where B is the boolean condition of the loop, and the formula $R(\vec{v_t},\vec{v})$ relates the values of the variables in two successive iterations.

Turning back to our example, the predicate that gives rise to the natural invariant of the loop, written $\mathit{natinv}(f_{1i},f_{2i},n_i,f_1,f_2,n)$, is defined inductively by the following two cases:

$$\begin{array}{rl}
\text{Base case:} & \forall f_{1i},f_{2i},n_i.\ \mathit{natinv}(f_{1i},f_{2i},n_i,f_{1i},f_{2i},n_i) \\
\text{Inductive case:} & \forall f_{1i},f_{2i},n_i,f_{1t},f_{2t},n_t,f_1,f_2,n.\ \mathit{natinv}(f_{1i},f_{2i},n_i,f_{1t},f_{2t},n_t) \rightarrow \\
 & \qquad n>0 \rightarrow (f_1 = f_{1t}+f_{2t} \wedge f_2 = f_{1t} \wedge n = n_t - 1) \rightarrow \mathit{natinv}(f_{1i},f_{2i},n_i,f_1,f_2,n)
\end{array}$$

The natural invariant of the loop is then written simply as $\mathit{natinv}(f_1@\mathbf{Init},f_2@\mathbf{Init},n@\mathbf{Init},f_1,f_2,n)$, where x@**Init** denotes the value of x in the initial state of the loop execution. We remark that this is quite an atypical use of loop invariants, since it relates the values of variables in different states. For such a small example the definition of the natural predicate can be written by hand, but the point here is that for more realistic programs a symbolic evaluation algorithm can be used to synthesize the natural invariants.

The ACSL self-composed program annotated with the natural invariant and the inductive predicate definition is included in the full version of this paper [1]. Of the 16 VCs generated by `Frama-c`, 2 cannot be discharged automatically, and require interactive proof.

A General Framework. The hard part in the proof above is the fact that the inductive definition is deterministic in the sense that in $\mathit{natinv}(f_{1i}, f_{2i}, n_i, f_1, f_2, n)$ the values of f_1 and f_2 are uniquely determined by the values of f_{1i}, f_{2i}, n_i, and n (i.e. the current state of the loop is uniquely determined by the initial state and the current iteration). We isolate this as the following lemma:

$$\forall f_{1i}, f_{2i}, n_i, f_{1a}, f_{2a}, n_a, f_{1b}, f_{2b}, n_b.\ \mathit{natinv}(f_{1i}, f_{2i}, n_i, f_{1a}, f_{2a}, n_a) \rightarrow \mathit{natinv}(f_{1i}, f_{2i}, n_i, f_{1b}, f_{2b}, n_b) \rightarrow \\ n_a = n_b \rightarrow (f_{1a} = f_{1b} \wedge f_{2a} = f_{2b})$$

The lemma clearly implies that two executions of the loop starting from the same initial conditions are synchronized (in fact a weaker lemma would be sufficient for our purposes, since only the final state of the loop is relevant). Once this lemma is added to the ACSL specification, all the VCs are automatically discharged (the program is proved secure). Hoping for an automatic proof of the lemma would be too ambitious. However, we found that an interactive proof in Coq is straightforward and can be conveniently decomposed. Furthermore, such lemmata are fairly easy to write and can also be generated automatically. Based on this observation, we have developed a method to construct self-composition proofs that aims to maximise the degree of automation and essentially eliminate the need to interactively use the prover. For this, we isolate the small parts of the process that clearly require user intervention, allowing the user to just *fill-in the blanks*. In this direction, rather than stand-alone proofs for particular applications of self-composition, we have produced a Coq development that formalises natural invariants in general, and can be used to generate proofs of desired lemmata. This development is presented in the full version of this paper [1]. In a nutshell, a relational specification is first extracted from the annotated program, which is then completed with a small set of facts provided by the user. The catch is that, since these facts are kept very simple (non-inductive), they can be checked automatically. This specification is then used to instantiate Coq modules containing definitions and basic facts, from which Coq can then generate and automatically prove the desired lemma.

Reasoning with Arrays. In Section 5 we will employ the self-composition technique to the RC4 algorithm. In that context, some of the variables that must be checked to be equal at the end of the self-composed program execution are array variables (e.g. the high-integrity outputs). The appropriate notion of array equality that must be used in preconditions and post-conditions when applying the self-composition technique is of course *extensional*. Equality of array variables corresponds to equality of memory addresses, which is unsuitable to capture equality of inputs and outputs as required. Moreover, the renaming operation C^s used by the self-composition technique must now allocate a new array for every array variable in the program C. Our Coq formalisation contemplates these aspects. An aspect of `Frama-c` that is required for natural invariants involving arrays is the possibility to have *state parameters* in predicates. The $\mathit{natinv}(\vec{v_i}, \vec{v})$ predicate was described as having as parameters the values of variables in two different states; for array variables a single array parameter is used, together with two state parameters. It is then possible to refer to the contents of array positions in those states. Having two array parameters would be wrong, since both would be passed the same memory address. Determinism lemmata as described above must also parametrised by states, and basically express properties like “if an execution of a loop starts in state S_1 and ends in S_2; a second execution of the same loop starts in state S_3 and ends in S_4; and moreover the contents of a given array is the same in states S_1 and S_3, then those contents will also be the same in states S_2 and S_4".

5 Using Information Flow to Capture Error Propagation

An important property of stream ciphers such as RC4 is how they behave when used to transfer data over channels which may introduce transmission errors. In particular, it is relevant how the decryption process reflects a wrong ciphertext symbol in the resulting plaintext: depending on the cipher construction, a ciphertext error may simply lead to a corresponding flip in a plaintext symbol, or it may affect a significant number of subsequent symbols. This property, sometimes called error propagation, is usually taken as a criterion to select ciphers for noisy communication media, where the absence of error propagation can greatly increase throughput. Note that error propagation can sometimes be seen as a desirable feature, as it amplifies errors that may be introduced maliciously, and which are therefore more easily detected.

In this section, we show how a generic technique for verifying secure information flow in software implementations can be used to check that the RC4 implementation in `openSSL` does not introduce error propagation. Given that this property is a known feature of synchronous ciphers such as RC4, it is clear that a complete proof of functional correctness with respect to a reference specification (such as the one introduced in the next section) would imply this result. However, the idea here is to demonstrate that general purpose verification tools and techniques developed for other software domains can be usefully adapted to verify relevant (and to the best of our knowledge never before addressed in literature) security properties in cryptographic software implementations.

Formalising Error Propagation as Non-Interference. The goal is to capture the error propagation property using a well studied formalism, so that we can take advantage of existing verification techniques to check this property in stream cipher implementations. To do this, we recall the notion of non-interference that was introduced in Section 2 as a possible formalisation of secure information flow. The intuition underlying this formalisation is that secure information flow can be guaranteed by checking that arbitrary changes in high-security (or low-integrity) input variables cannot be detected by observing the values of low-security (or high-integrity) output variables. Observe that the notion of a low-integrity input variable can be naturally associated with that of a transmission error over a communications channel. Hence, we map the $i^{\mathtt{th}}$ possibly erroneous ciphertext symbol to a non-trusted low-integrity input (we are looking at the decryption algorithm that, in the case of RC4, is identical to the one used for encryption). The non-interference definition can then conveniently be used to naturally capture the absence of error propagation. For this, we associate the output plaintext symbols starting at position $i+1$ to trusted high-integrity outputs. More precisely, our formulation captures the following idea: if an arbitrary change in the $i^{\mathtt{th}}$ input ciphertext symbol cannot be observed in the output plaintext symbols following position i, this implies that the stream cipher does not introduce error propagation. In other words, we want to verify that an erroneous (possibly tampered) input symbol, which will unavoidably result in a corresponding erroneous output symbol in the same position, will not affect subsequent outputs. Formally, following the notation introduced in Section 2 that associates $V_\uparrow$ with the set of low-integrity input variables and $V_\downarrow$ with the set of high-integrity outputs, we have for some $i \in [0, len[$:

$$V_\uparrow = \{\mathtt{indata}[i]\}, \quad (1)$$

$$V_\downarrow = \{\mathtt{outdata}[j] \,|\, i < j < \mathtt{len}\}. \quad (2)$$

```
unsigned char RC4NextKeySymbol(RC4_KEY *key) {
  unsigned char *d,x,y,tx,ty;

  x=key->x; y=key->y; d=key->data;
  x=((x+1)&0xff);     tx=d[x];
  y=(tx+y)&0xff;      d[x]=ty=d[y];
  d[y]=tx;  key->x=x; key->y=y;
  return d[(tx+ty)&0xff];
}

void RC4(RC4_KEY *key, const unsigned long len,
     const unsigned char *indata, unsigned char *outdata) {
  int i=0;
  while(i<len) { outdata[i]=indata[i] ^ RC4NextKeySymbol(key); i++; }
}
```

Figure 2: Simplified RC4 implementation

Verifying The Absence of Error Propagation in RC4. We are now in a position to apply self-composition to verify whether the RC4 implementation in `openSSL` indeed satisfies the error propagation property. To do this, we use the approach introduced in Section 4. As has been pointed out in literature [17], this is a non-trivial exercise even for such a small example: the self-composition technique for the verification of non-interference is very attractive from a conceptual point of view, but its applicability to real-world applications is yet to be demonstrated. We believe the results in this section are an important contribution towards answering this open question. However, given that we have used an off-the-shelf, general-purpose, verification tool, which is still under development, problems were to be expected. The limitations that we encountered were essentially due to the growth of the problem size due to the aggressive optimisations that were used in the RC4 implementation in `openSSL`: (1) the use of macros rather than (inline) function calls leads to a source code size expansion; (2) the use of loop unrolling leads to intricate control flow inside the function (namely the extensive use of the `break` statement); and (3) the use of pointer arithmetic greatly increases the complexity of the generated proof obligations. For this reason, a refactoring of the code was required in order to achieve the goals that we set out in the beginning of this section (more on this in Section 7). Figure 2 shows the version of the RC4 we used.

The full version of this paper [1] contains the `Frama-c` input file used to show that the `RC4` function above does not introduce error propagation. The function is composed with itself and disjoint sets of variables are created for the two copies of the function, which is parametrised with the position i in which a transmission error could occur. The preconditions imposed on the composed function establish the equality of the high-integrity and unspecified-security input variables for both copies of the function: all input variables except position i in the `indata` buffers. The post-condition on the composed function requires that the high-integrity output variables have equal values upon termination: $i+1$ to `len` in the `outdata` buffer. The verification of this code with `Frama-c` resulted in the generation of 17 proof obligations, all of which are automatically discharged by the prover `Simplify`. This is made possible by the inclusion of a helper lemma (proved offline with Coq following Section 4) in the ACSL annotations.

6 Related Work

Language Based Information Flow Security is surveyed in [15]. Leino and Joshi [12] were the first to propose a semantic approach to checking secure information flow, with several desirable features: it gives a more precise characterisation of security; it applies to all programming constructs whose semantics are well-defined; and it can be used to reason about indirect information leakage through variations in program behaviour (e.g., whether or not the program terminates). An attempt to capture this property in program logics using the *Java Modelling Language*(JML)[11] was presented by Warnier and Oostdijk[19]. They proposed an algorithm, based on the strongest post-condition calculus, that generates an annotated source file with specification patterns for confidentiality in JML. Dufay et al. [7] proposed an extension to JML to enforce non-interference through self-composition. This extended annotation language allows a simple definition of non-interference in Java programs. However, the generated proof obligations are complex, which limits the general applicability of the approach.

Terauchi and Aiken [17] identified problems in the self-composition approach, arguing that automatic tools are not powerful enough to verify this property over programs of realistic size. To compensate for this, the authors propose a program transformation technique based on an extended version of the self-composition approach, which incorporates the notion of security level downgrading using *relaxed non-interference* [13]. Rather than replicating the original code, the renamed version is interleaved and partially merged with it. Naumann[14] extended Terauchi and Aiken's work to encompass heap objects, presented a systematic method to validate the transformations proposed in [17], and reported on the experience of using these techniques with the ESC/JAVA2[10] and Spec# [2] tools.

7 Conclusions

The main contribution of this paper is to report on the application of the off-the-shelf Frama-c verification platform to a real-world example of a cryptographic software implementation: the widely used C implementation of the RC4 stream cipher available in the openSSL library. Our results focus on two security-relevant properties of this implementation: (1) safety properties such as the absence of numeric errors and memory safety, and (2) the absence of error propagation.

We take advantage of the built-in Frama-c functionality that automatically generates proof obligations for these two common types of secure coding safety properties. Concretely, we use Frama-c to prove that RC4 implementation does not cause null pointer de-referencing exceptions, and always performs array accesses with valid indices. In other words, the implementation is secure against *buffer overflow* attacks. Additionally, we demonstrate that the limited ranges of numeric variables used in the RC4 implementation are guaranteed not to introduce calculation errors for particular input values.

An important property of stream ciphers such as RC4 is their behaviour when a bit in the ciphertext is flipped over a communication channel. The change may be due to a transmission error, or maliciously introduced by an attacker. The behaviour of RC4 is common to other *synchronous* ciphers: bit errors are not propagated in any way, i.e. if a ciphertext bit is flipped during transmission, then only the corresponding plaintext bit is affected. We formalise this property as a novel application of the *non-interference* concept, widely used in the formalisation and verification of secure information flow properties. Subsequently, we proved that the RC4 implementation indeed has this property.

Finally, our work answers some open questions raised by previous work, which seemed to indicate that self-composition was not directly applicable to real-world cases. Our results are promising in that we have been able to achieve our goal using an off-the-shelf verification tool and a technique with a high potential for automation. Our use of natural invariants is part of a bigger effort: the same technique facilitates the development of *equivalence proofs*, which we have been applying in the cryptographic context to prove the correctness of real implementations with respect to reference implementations. In particular, the validity of the refactoring used to produce the version of RC4 in Figure 2 can be addressed using these techniques. The full version of this work [1] covers also these aspects of our research.

References

[1] José Bacelar Almeida, Manuel Barbosa, Jorge Sousa Pinto, and Bárbara Vieira. Deductive verification of cryptographic software. Technical Report DI-CCTC-09-03, CCTC, Univ. Minho, 2009. Available from `http://cctc.uminho.pt/`.

[2] Mike Barnett, K. Rustan, M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, pages 49–69. Springer, 2004.

[3] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. Secure information flow by self-composition. In *CSFW*, pages 100–114. IEEE Computer Society, 2004.

[4] Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI/ISO C Specfication Language*. CEA LIST and INRIA, 2008. Preliminary design (version 1.4, December 12, 2008).

[5] Sylvain Conchon, Evelyne Contejean, and Johannes Kanig. Ergo: a theorem prover for polymorphic first-order logic modulo theories, 2006.

[6] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[7] Guillaume Dufay, Amy Felty, and Stan Matwin. Privacy-sensitive information flow with JML. In *Automated Deduction - CADE-20*, pages 116–130. Springer Berlin / Heidelberg, August 2005.

[8] Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In Werner Damm and Holger Hermanns, editors, *CAV*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177. Springer, 2007.

[9] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969.

[10] B. P. F. Jacobs, J. R. Kiniry, M. E. Warnier, Bart Jacobs, Joseph Kiniry, and Martijn Warnier. Java program verification challenges. In *FMCO 2002: Formal Methods for Component Objects, Proceedings, volume 2852 of Lecture Notes in Computer Science*, pages 202–219. Springer, 2003.

[11] Gary T. Leavens, Clyde Ruby, K. Rustan M. Leino, Erik Poll, and Bart Jacobs. JML (poster session): notations and tools supporting detailed design in Java. In *OOPSLA '00: Addendum to the 2000 proceedings of the conference on Object-oriented programming, systems, languages, and applications (Addendum)*, pages 105–106, New York, NY, USA, 2000. ACM.

[12] K. Rustan M. Leino and Rajeev Joshi. A semantic approach to secure information flow. *Lecture Notes in Computer Science*, 1422:254–271, 1998.

[13] Peng Li and Steve Zdancewic. Downgrading policies and relaxed noninterference. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 158–170, New York, NY, USA, 2005. ACM.

[14] David A. Naumann. From coupling relations to mated invariants for checking information flow. In *Computer Security - ESORICS 2006*, volume 4189 of LNCS, pages 279– 296, 2006.

[15] A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), 2003.

[16] Bruce Schneier. *Applied cryptography: protocols, algorithms, and source code in C*. Wiley, New York, 2nd edition, 1996.

[17] Tachio Terauchi and Alexander Aiken. Secure information flow as a safety problem. In Chris Hankin and Igor Siveroni, editors, *SAS*, volume 3672 of *Lecture Notes in Computer Science*, pages 352–367. Springer, 2005.

[18] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.2*, 2008. `http://coq.inria.fr`.

[19] Martijn Warnier and Martijn Oostdijk. Non-interference in JML. Technical Report ICIS-R05034, Nijmegen Institute for Computing and Information Sciences, 2005.

Coloured Petri net refinement specification and correctness proof with Coq

Christine Choppy, Micaela Mayero and Laure Petrucci *
LIPN, UMR CNRS 7030, Université Paris 13
FRANCE
`firstname.lastname@lipn.univ-paris13.fr`

Abstract

In this work, we address the formalisation of symmetric nets, a subclass of coloured Petri nets, refinement in COQ. We first provide a formalisation of the net models, and of their type refinement in COQ. Then the COQ proof assistant is used to prove the refinement correctness lemma. An example adapted from a protocol example illustrates our work.

1 Introduction

Modelling and analysing large and complex systems requires elaborate techniques and support. To harness the problems inherent to designing and model-checking a large system (such as the state space explosion problem), a specification is often developed step by step. First, an abstract model is designed, and its properties are verified. Once it is proven correct, a refinement step takes place, introducing further detail. Such an addition can either be enhancing the description of the actual functioning of part of the system, or introducing an additional part. This new refined model is then verified, and another refinement step can take place. This process is applied until an adequate level of description is obtained.

There are several advantages to using refinement and hence to start with a more abstract model. It gives a better and more structured view of the system under study. The components within the system are clearly identified and the modelling process is eased. The modeller does not have to bother with spurious details. The validation process also becomes easier: the system properties are checked at each step. Thus, abstract models are validated before new details are added. Moreover, when model-checking is used, the analysis of a full concrete model may not be amenable, due to its very large state space. Refinement helps in coping with this problem since it may preserve some properties, or analysis results obtained at an earlier step for a more abstract model may be used for the analysis of the refined model. For example, Lakos and Lewis [9] use the state space of the abstract model to compute the state space of the refined one: some tests for enabledness of transitions are avoided, as well as the construction of partial markings that have already been computed. Moreover, the state space can be structured. Hence, this approach saves both space and time in the analysis, countering the state space explosion problem.

In this paper, we consider specifications written as symmetric nets, a subclass of coloured Petri nets [7]. Lakos and Lewis [8, 9] consider three kinds of Petri nets refinements for coloured Petri nets, node (place or transition) refinement, subnet refinement, and type refinement. Our work provides a formalisation in COQ [5] of both the abstract Petri net and the refined net, as well as refinement correction lemmas, together with the refinement correction proof.

In a previous work [4], we considered mainly place/transition Petri nets (sketching how coloured nets might be taken into account), and the subnet refinement (the node refinement processing being similar).

In this paper, we address coloured nets, which are more complex in nature, and the formalisation in COQ had to be significantly changed to take the typing issues into account. The *type refinement* formalisation and correction lemma are also addressed, thus pursuing the initial work of [4]. A protocol example adapted from [7] illustrates our work.

E. Denney, D. Giannakopoulou, C.S. Păsăreanu (eds.); The First NASA Formal Methods Symposium, pp. 156-165

*`http://www-lipn.univ-paris13.fr/~lastname`

This type refinement is interesting as it allows for specification of concrete and useful properties in practice. It requires a more complete formalisation, since colours (seen as types) are necessary. When compared to places/transitions nets, the use of colours lead to decrease the size of nets, leading to more amenable models.

The paper is structured as follows. Section 2 recalls the definitions of coloured Petri nets. Section 3 recalls the different refinements. Then, section 4 describes the case study (a protocol example) and formalisations for proving the type refinement of the net in this example. Conclusions and future work are finally discussed in section 5.

2 Coloured Petri nets definition

The definition of coloured Petri nets [8, 9] used in this paper is the following:

Definition 2.1 (Coloured Petri net)**.** *A Coloured Petri net $\mathscr{R}$ is an 8-tuple $\mathscr{R} = \langle P, T, A, C, E, \mathbb{M}, \mathbb{Y}, M_0 \rangle$ where:*

1. *P is a set of places*
2. *T is a set of transitions, such that $P \cap T = \emptyset$*
3. *A is a set of arcs, such that $A \subseteq (P \times T) \cup (T \times P)$*
4. *C: $P \cup T \to \Sigma$ where Σ is a universe of non-empty colour sets (or types), determines the colours of places and the transition modes.*
5. *E: $A \to \Phi\Sigma$ yields the arc inscriptions, such that $E(p,t), E(t,p)$: $C(t) \to \mu C(p)$*
6. *$\mathbb{M} = \mu\{(p,c) | p \in P, c \in C(p)\}$ is a set of markings, that associate a value c with a place p of P.*
7. *$\mathbb{Y} = \mu\{(t,c) | t \in T, c \in C(t)\}$ is a set of steps (multisets of transitions with their firing mode).*
8. *M_0 is the initial marking, $M_0 \in \mathbb{M}$.*

where $\Phi\Sigma$ is a function over Σ defined by $\Phi\Sigma = \{X \to Y \mid X, Y \in \Sigma\}$ and $\mu X = \{X \to \mathbb{N}\}$ are multisets over a set X, where $\mathbb{N}$ is the set of natural numbers.

In the example in Figure 1, the marking of place `PacketsToSend` is the multiset 1‘1++1‘2++1‘3++1‘4++1‘5++1‘6, where 1‘6 denotes one occurrence of value 6, and ++ denotes the multiset addition operator.

Definition 2.2. *[8] The incremental effects $E^+, E^- : \mathbb{Y} \to \mathbb{M}$ of the occurrence of a step Y are given by:*

1. *$E^-(Y) = \sum_{(t,m) \in Y} \sum_{(p,t) \in A} \{p\} \times E(p,t)(m)$*
2. *$E^+(Y) = \sum_{(t,m) \in Y} \sum_{(t,p) \in A} \{p\} \times E(t,p)(m)$*

E^- defines the input arc inscriptions while E^+ defines the output arc inscriptions.

Type refinement modifies the information carried by the tokens (a colour is a value of a token) while the net structure is unchanged. Type refinement brings additional information, which may be done e.g. by adding components in a tuple, or by representing an abstract data type by a more concrete one. The properties of the refined type should be preserved, that is if type A is refined by type B, then type B should satisfy the properties of A after an adequate syntactic translation. As for nets, it should always be possible to associate a behaviour of the abstract net with a behaviour of the refined one. Type refinement issue is associated with the issue of abstraction and implementation in the context of formal specifications (e.g. algebraic specifications [12]), and with studies on subtyping in the context of object-oriented programming languages [11, 3]. In this work (as in the work of Lakos [8, 9]), the type refinement considered is adding components in a tuple.

Since coloured Petri nets can use very general types and functions over these types which are thus not amenable, we here restrict ourselves to the *symmetric Petri nets* subclass. Symmetric nets are defined as coloured Petri nets that allow only the use of particular types and functions: enumerated types, booleans, integer intervals, tuples and combinations of these, as well as the associated functions. We actually also handle lists of such types that can easily be manipulated by the COQ theorem prover.

3 Definitions of refinements

As mentioned previously, Lakos and Lewis [8, 9] consider three kinds of Petri nets refinements, node (place or transition) refinement, subnet refinement, and type refinement. *Node refinement* consists in replacing a place (transition) by a place- (transition-) bordered subnet. *Subnet refinement* consists in adding net components (places, transitions and arcs or even additional tokens). In this section the definition of type refinement is recalled, and we give the corresponding correctness lemma. The lemmas for subnet and node refinements can be found in [4]. Very little work as been achieved concerning type refinement.

In the following definition of *type refinement* [8], $\mathcal{N}_a = \langle P, T, A, C, E, \mathbb{M}, \mathbb{Y}, M_0 \rangle$ is the abstract net and $\mathcal{N}_r$ is the refined net.

Definition 3.1 (Type refinement)**.** *A morphism $\phi : \mathcal{N}_a \rightarrow \mathcal{N}_r$ is a* type refinement *if:*

1. *ϕ is the identity function on P, T, A, i.e. $\forall p \in P$: $\phi(p) = p$, etc.*

2. $\forall x \in P \cup T$: $C(x) <_{:} \phi(C)(x)$

3. $\forall x \in P \cup T$: $\forall c \in C(x)$: $\phi(1` \ (x,c)) = 1` \ (x, \Pi_{\phi(C)(x)}(c))$

4. $\forall (p,t) \in A$: $\forall (t,c) \in \mathbb{Y}$:
   $\phi(E^{-}(1` \ (t,c)))(p) = \Pi_{\phi(C)(p)}(E(p,t)(c)) = \phi(E)(p,t)(\Pi_{\phi(C)(t)}(c))$
   $\forall (t,p) \in A$: $\forall (t,c) \in \mathbb{Y}$:
   $\phi(E^{+}(1` \ (t,c)))(p) = \Pi_{\phi(C)(p)}(E(t,p)(c)) = \phi(E)(t,p)(\Pi_{\phi(C)(t)}(c))$

The following interpretation will be used in section 4.2,

Lemma 3.2.

1. *The network structure (places, transitions and arcs) is kept unchanged, i.e. $P = P'$, $T = T'$, $uA = uA'$ where P', T' et uA' are resp. the sets of places, transitions and arcs (without their associated type) of the refined net while P, T and uA are those of the abstract net.*

2. *For any token* $1`(x',c')$*, of value* $x'$ *for colour* $c'$ *in the initial marking of the refined net, there exists a corresponding token* $1`(x,c)$ *in the initial marking of the abstract net. They must be such that both the sub-typing and projection relations (resp. denoted* $<_{:}$ *and* $\prod$*) are satisfied:* $c <_{:} c'$ *and* $c = \prod_c(c')$*.*

3. *The arc expressions are refined according to the token refinement:* $\prod_{C_r(p)}(C_a(arc)) = C_r(arc)(C_r(t))$*.*

According to the formal definition of type refinement in [10], the net structure is unchanged. Since type refinement consists in incorporating additional information in token values, a token type in the refined net is a subtype of the one in the abstract net.

4 Case study: the simple protocol

4.1 Description

In this section, the correction lemma is illustrated by a simple protocol example adapted from [7].

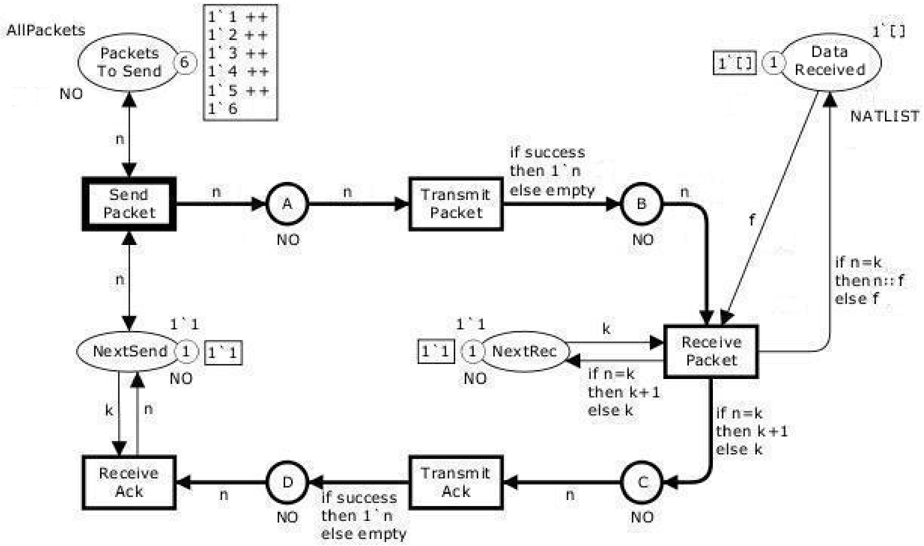


Figure 1: Example of a simple protocol

Figure 1 describes this simple protocol. The left-hand side part models the sender, the right-hand side the receiver while the middle part represents the network. The sender state is modelled by two places: *PacketsToSend* and *NextSend*. The receiver state is modelled by the *DataReceived* place. Places *A*, *B*, *C* and *D* constitute the network.

Note that place *PacketsToSend* is initially marked by six tokens with integer values. The textual inscriptions under a place is called "the colour set" of this place, which represents the available set of token colours. For example, the tokens in place *NextSend* always have an integer value. Here, the colour set NO is used to model sequence numbers. The inscription at the right top of place *NextSend* specifies that the initial marking of this place contains a single token with colour (value) 1. Informally, 1‘1 means that the data packet number 1 is to be sent. Finally, we will eventually obtain in place *DataReceived* a list of natural numbers: $[6,5,4,3,2,1]$. Let us note that arc expressions yield token values together with their multiplicity. However, when the multiplicity is 1, it is omitted, thus n denotes 1‘n.

This example is refined by associating additional information with tokens (while the net structure in terms of places and transitions is unchanged). The refined net is presented in Figure 2.

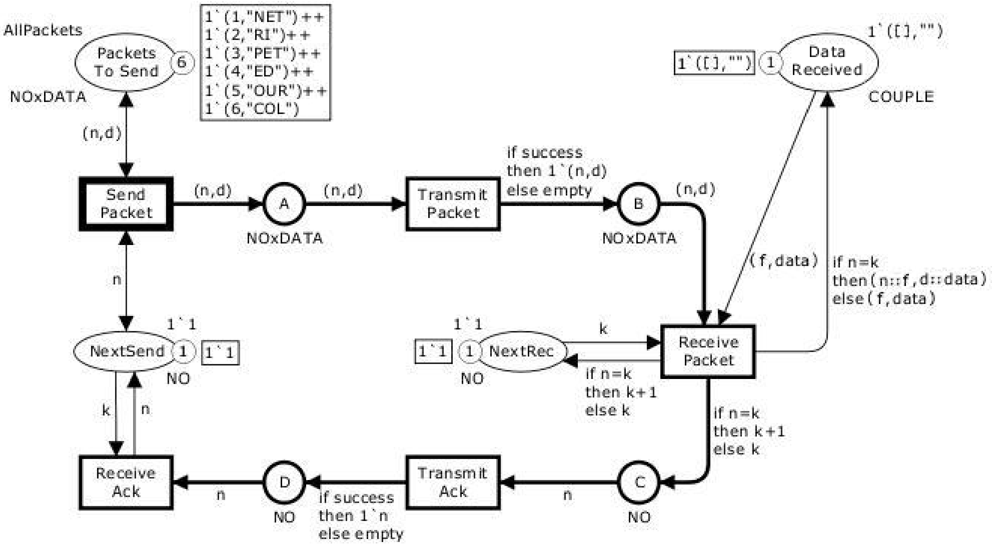


Figure 2: Refined protocol example

The colour sets of places *PacketsToSend*, *A*, *B* and *DataReceived* are extended from `NO` to `NO`×`DATA` which is defined as the cartesian product of the sets describing types `NO` and `DATA`. Note that here some places are not refined, e.g. *NextSend* is still of type `NO`.

The type refinement achieved in Figure 2 not only changes the type of tokens and places but also modifies the arcs expressions accordingly. Note that there exists a subtyping relation between e.g. (x) and $(x,1)$.

4.2 Formalisation and proof in Coq

The simple protocol example is now formalised. Tokens carry complex information, and several functions are required to certify the system. In this example, the arc expressions are of four possible types (all with an integer multiplicity).

nat ∗	(nat)
	(nat ∗ string)
	(list nat)
	(list nat ∗ string)

In Coq, these kinds of arcs are defined by an inductive type:

```
Definition nat_Tuple := list nat.

Inductive arc_type : Type :=
  |bi_types: nat*nat -> arc_type
  |tri_types: nat*(nat*string) -> arc_type
  |bi_n_tuples: nat*nat_Tuple -> arc_type
  |tri_tuples: nat*(nat_Tuple*string) -> arc_type.
```

Then, places and transitions are indexed by natural numbers:

```
Record Place : Type := mkPlace
        { Pr : nat }.

Record Transition : Type := mkTransition
        { Tr : nat }.
```

We now present an excerpt of the definitions of places, transitions and arcs for our example. Sets of places, transitions or arcs (both the untyped arc, i.e. the edge in the graph, and the arc expression) are represented by lists:

```
Definition P1_PackToSend := mkPlace 1.
...
Definition list_P:= P1_PackToSend::P2_A::P3_B::P4_NextRec::
                    P5_DataRec::P6_C::P7_D::P8_NextSend::nil.

Definition T1_SendPack := mkTransition 1.
...
Definition list_T:= T1_SendPack::T2_TransPack::T3_RecPack::
                    T4_TransAck::T5_RecAck::nil.

Definition uAP1T1 := (P1_PackToSend,T1_SendPack).
...
Definition AP1T1 := (P1_PackToSend,T1_SendPack,bi_types (1,1)).
...
Definition list_APT := AP1T1::AP2T2::AP3T3::AP4T3::AP5T3::
                       AP6T4::AP7T5::AP8T5::AP8T1::nil.
Definition list_ATP := AT1P2 ::AT2P3::AT3P4::AT3P5::AT3P6::
                       AT4P7::AT5P8::AT1P8::nil.
Definition list_ATP := uAT1P2...
...
Definition list_APT’ := A’P1T1::A’P2T2::A’P3T3::A’P4T3::A’P5T3::A’P6T4::
                        A’P7T5::A’P8T5::A’P8T1::nil.
Definition list_ATP’ := A’T1P2 ::A’T2P3::A’T3P4::A’T3P5::A’T3P6::
                        A’T4P7::A’T5P8::A’T1P8::nil.
```

The most interesting aspect of type refinement is due to arc expressions. Type refinement can be seen as a relation between types, which is subtyping. For example, the following table presents subtyping relations involved in our refinement of the simple protocol (note that the first line is unchanged by the refinement, and this still needs to be checked):

ARC EXPRESSIONS	EXAMPLE OF ARC VALUES	VALUE TYPE	COQ arc_type
if n=k then k+1 else k	1‘6	nat*nat	bi_types
n (n, d)	1‘6 1‘(6, ”COL”)	nat*nat nat*(nat*string)	bi_types tri_types

if n=k then n::f else f
if n=k then (n::f, d::data) else (f, data)
	1‘[6]
	1‘([6], ”COL”)
		nat*list nat
		nat*(list nat*string)
			bi_n_tuples
			tri_tuples

f (f, data)	1‘[6]) 1‘(6, ”COL”)	nat*list nat nat*(list nat*string)	bi_n_tuples tri_tuples

The subtyping relation must be formalised for this example. We begin by defining a function is_sub which gives a relation between types of arc_type. This relation is then extended to tuples (is_sub_tupl_apt) and lists of tuples (is_sub_l_apt) for describing arcs from places to transitions. Similar extensions are defined for arcs from transitions to places.

```
Definition is_sub (subtyp:arc_type)(typ:arc_type) : Prop :=
  match subtyp, typ with
  | (bi_types _), (bi_types _) => True
  | (tri_types _), (bi_types _) => True
  | (tri_tuples _), (bi_n_tuples _) => True
  | _, _ => False
  end.

Definition is_sub_tupl_apt (subtupl: Place * Transition * arc_type)
          (tupl: Place * Transition * arc_type) : Prop :=
          (is_sub (snd subtupl) (snd tupl)).

Fixpoint is_sub_l_apt (subl: list (Place * Transition * arc_type))
         (l: list (Place * Transition * arc_type)) {struct subl} : Prop :=
    match subl, l with
     | nil, nil => True
     | (cons a tla), (cons b tlb) =>
            (is_sub_tupl_apt a b) /\ (is_sub_l_apt tla tlb)
     | _, _ => False
    end.
```

The type refinement correctness lemma can now be written, with the help of lemma 3.2 (where, for the sake of readability, we indicate in parenthesis to which item the COQ code relates):

```
Lemma type_colour_refined:
      eqlist Place list_P list_P' /\                      (1.)
      eqlist Transition list_T list_T' /\                 (1.)
      eqlist (Place*Transition) list_uAPT list_uAPT' /\    (1.)
      eqlist (Transition*Place) list_uATP list_uATP' /\    (1.)
      eqlist (list (nat*nat)) list_MP (hd_list list_MP')/\ (2.)
      is_sub_l_apt list_APT' list_APT /\                  (3.)
      is_sub_l_atp list_ATP' list_ATP.                    (3.)
```

where `eqlist` is an equality between lists and $l = l'$ is equivalent to $l \subseteq l'$ and $l' \subseteq l$, `list_MP` and `list_MP'` define the initial markings, and function `hd_list` is defined as follows:

```
Fixpoint hd_list_couple (l:list (nat*(nat*string))):=
    match l with
     | nil=>nil
     |(a,(b,c))::tl=>(a,b)::(hd_list_couple tl)
    end.

Fixpoint hd_list (l:list (list (nat*(nat*string)))):=
     match l with
      |nil=>nil
      |a::tl=>(hd_list_couple a)::(hd_list tl)
     end.
```

Thanks to our simple and general formalisation, the formal correctness proof is almost automatic.

```
Proof.
repeat split;unfold incl;tauto.
Qed.
```

Note that this simple formalisation was obtained after carefully studying different possibilities for encoding Petri net elements in COQ. These are detailed in [4]. Moreover, the proof could be simplified using powerful constructs such as the `split` tactic, which is particularly well-suited for our purposes. This tactic applies to inductive types with a single constructor, which is the case for the `/\` operator in the lemma `type_colour_refined`.

When the proof fails, it still gives valuable information w.r.t. the refinement to be proven: either the lists representing the net graph elements (places, transitions, or edges) do not match, and the refinement relation does not hold ; or the error occurs when examining arc expressions. It may then be the case that refinement does not hold, but also that the type refinement between the supposedly refined and abstract arc expression cannot be automatically proven.

The full development is available at `http://www-lipn.univ-paris13.fr/~mayero/CPNCoq/Jensen_protocol_NFM.v`.

5 Conclusion

When modelling and validating critical systems, one often proceeds in a step-by-step fashion: a first abstract model is designed and validated ; it is then refined so as to take into account additional details ; and this process is repeated as many times as necessary. In order to guarantee that the behaviour of the system is preserved by refinement, it should obey some rules. Three kinds of refinements of coloured Petri nets were formally defined in [9]. Our aim here was to show that the proof of refinement — i.e. that a refined net actually is a refinement of an abstract net — can be automated using theorem-proving techniques, thus avoiding error-prone and lengthy manual proofs.

Previous work focussed on two kinds of refinements: node refinement and subnet refinement, while the third one was scarcely mentioned. This paper has shown that when restricting coloured Petri nets to an appropriate subclass, type refinement can also be handled.

This work confirms that our choices of formalisation, made in [4], are suitable. The prerequisite to the refinements is the formalisation of a given Petri net. This formalisation is probably the most tedious part of our work and requires a significant automation. Since the refinement issues we tackle are meant to be integrated within a step-by-step modelling process, the refined net should be designed by the user starting from the abstract net. Therefore, places and transitions that are in both nets should remain exactly the same and can be identified by their name. Hence, we do not address the problem of proving that a net is a refinement of another one, starting from arbitrary nets.

The possibility to easily integrate automation at a later stage was a key issue in the work presented in this paper. For example, as seen in section 4, to define all the places, all the transitions and all the arcs manually is certainly not efficient, especially if the net has more than 50 places/transitions.

We plan to solve this problem using an interface to PNML (Petri Net Markup Language, [2]). PNML is currently being standardised within ISO/IEC 15909-2. It aims at becoming the common language for Petri nets tools, e.g. CPN-AMI [1], CPN-Tools [6] or other tools supporting Petri nets. Such files can be directly translated into COQ to generate the places, transitions and arcs.

We think that our method scales up rather well. Indeed, the proof is generic and does not change with the nets considered. The only modifications are sub-typing relations and type definitions. Moreover, when proceeding step-by-step, refinements are applied one at a time. Therefore, the nets to be considered are only slightly different.

To complete this work, we should consider refinement as part of a modular design process. In such a framework, a type refinement can affect several modules which could be checked separately for refinement, and one must ensure that type refinement has been applied consistently in all modules.

Acknowledgments Implementation of this work in COQ was achieved with the help of Yibei Yu, a trainee supervised by the authors.

References

[1] CPN-AMI*: Home Page*. `http://www-src.lip6.fr/logiciels/mars/CPNAMI/`.

[2] J. Billington, S. Christensen, K. van Hee, E. Kindler, O. Kummer, L. Petrucci, R. Post, C. Stehno, and M. Weber. The Petri Net Markup Language: Concepts, technology and tools. In *Proc. 24th Int. Conf. Application and Theory of Petri Nets (ICATPN'2003), Eindhoven, The Netherlands, June 2003*, volume 2679 of *Lecture Notes in Computer Science*, pages 483–505. Springer, 2003.

[3] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17(4):471–522, 1985.

[4] Christine Choppy, Micaela Mayero, and Laure Petrucci. Experimenting Formal Proofs of Petri Nets Refinements. *Electr. Notes Theor. Comput. Sci.*, 214:231–254, 2008.

[5] *The Coq proof assistant*. `http://coq.inria.fr`.

[6] *cpntools*. `http://wiki.daimi.au.dk/cpntools/cpntools.wiki`.

[7] Kurt Jensen and Lars M. Kristensen. *Coloured Petri Nets, Modelling and Validation of Concurrent Systems*. Monograph to be published by Springer Verlag, 2008.

[8] Charles Lakos. Composing abstractions of coloured Petri nets. In Nielsen, M. and Simpson, D., editors, *Lecture Notes in Computer Science: 21st International Conference on Application and Theory of Petri Nets (ICATPN 2000), Aarhus, Denmark, June 2000*, volume 1825, pages 323–345. Springer-Verlag, 2000.

[9] Charles Lakos and Glen Lewis. Incremental state space construction of coloured Petri nets. In *Proc. 22nd Int. Conf. Application and Theory of Petri Nets (ICATPN'01), Newcastle, UK, June 2001*, volume 2075 of *Lecture Notes in Computer Science*, pages 263–282. Springer, 2001.

[10] Glen Lewis. *Incremental specification and analysis in the context of coloured Petri nets*. PhD thesis, University of Hobart, Tasmania, 2002.

[11] Barbara Liskov and Jeannette M. Wing. A new definition of the subtype relation. In *ECOOP '93: Proceedings of the 7th European Conference on Object-Oriented Programming*, pages 118–141, London, UK, 1993. Springer-Verlag.

[12] Fernando Orejas, Marisa Navarro, and Ana Sanchez. Algebraic implementation of abstract data types: a survey of concepts and new compositionality results. *Mathematical Structures in Computer Science*, pages 33–67, 1996.

Modeling Guidelines for Code Generation in the Railway Signaling Context

Alessio Ferrari
General Electric Transportation Systems
Florence, Italy
alessio.ferrari@ge.com

Alessandro Fantechi
University of Florence
D.S.I., Florence, Italy
fantechi@dsi.unifi.it

Stefano Bacherini
General Electric Transportation Systems
Florence, Italy
stefano.bacherini@ge.com

Niccoló Zingoni
General Electric Transportation Systems
Florence, Italy
niccolo.zingoni@ge.com

Abstract

Modeling guidelines constitute one of the fundamental cornerstones for Model Based Development. Their relevance is essential when dealing with code generation in the safety-critical domain. This article presents the experience of a railway signaling systems manufacturer on this issue.

1 Introduction

Introduction of Model-Based Development (MBD) and code generation in the industrial safety-critical sector created a crucial paradigm shift in the development process of dependable systems. While traditional software development focuses on the code, with MBD practices the focus shifts to model abstractions. The change has fundamental implications for safety-critical systems, which still need to guarantee a high degree of confidence also at code level. Usage of the Simulink/Stateflow platform for modeling, which is a *de facto* standard in control software development, does not ensure by itself production of high-quality dependable code. This issue has been addressed by companies through the definition of modeling rules imposing restrictions on the usage of design tools components, in order to enable production of qualified code.

The MAAB Control Algorithm Modeling Guidelines (MathWorks Automotive Advisory Board)[3] is a well established set of publicly available rules for modeling with Simulink/Stateflow. This set of recommendations has been developed by a group of OEMs and suppliers of the automotive sector with the objective of enforcing and easing the usage of the MathWorks tools within the automotive industry. The guidelines have been published in 2001 and afterwords revisited in 2007 in order to integrate some additional rules developed by the Japanese division of MAAB [5].

The scope of the current edition of the guidelines ranges from model maintainability and readability to code generation issues. The rules are conceived as a reference baseline and therefore they need to be tailored to comply with the characteristics of each industrial context. Customization of these recommendations has been performed for the automotive control systems domain in order to enforce code generation [7]. The MAAB guidelines have been found profitable also in the aerospace/avionics sector [1] and they have been adopted by the MathWorks Aerospace Leadership Council (MALC).

General Electric Transportation Systems (GETS) is a well known railway signaling systems manufacturer leading in Automatic Train Protection (ATP) systems technology. Inside an effort of adopting formal methods within its own development process, GETS decided to introduce system modeling by means of the MathWorks tools [2], and in 2008 chose to move to code generation. This article reports the experience performed by GETS in developing its own modeling standard through customizing the MAAB rules for the railway signaling domain and shows the result of this experience with a successful product development story.

E. Denney, D. Giannakopoulou, C.S. Păsăreanu (eds.); The First NASA Formal Methods Symposium, pp. 166-170

2 Models for Railway Signaling Systems Rationale

MAAB guidelines have been developed to address the need of the automotive industries for a common language in MBD using the MathWorks tools. GETS decided to adopt these rules as a baseline for defining its own modeling standard according to the needs of the railway signaling systems domain.

Railway signaling systems software, and in particular the code of Automatic Train Protection (ATP) systems, is characterized by the extensive usage of control modes logic and message analysis algorithms. These kind of features can be easily modeled through Stateflow state machines, while Simulink blocks are more suitable for numerical algorithms [3]. This observation suggested the possibility of adopting Stateflow as the only tool for modeling the software, and use Simulink as a framework for allowing Stateflow Charts integration and to simulate the external environment.

Any railway signaling application software developed for Europe shall comply with the CENELEC standard [4]. The norm does not cover code generation issues, but relies just on source code quality, requiring structuring, readability, traceability and testability of the code. Two code generators have been evaluated: Real Time Workshop Embedded Coder (RTW) and Stateflow Coder. The first one allows code production for complete Simulink systems, while the second one generates code for only Stateflow Charts. Stateflow Coder has been chosen as the code generation tool since the code produced through RTW Embedded Coder was barely readable, not compliant with CENELEC quality requirements, and the development was focused on Stateflow modeling. With Stateflow Coder, integration among different generated software units was easily addressed by hand.

The MAAB guidelines are mainly composed by Naming Conventions, Model Architecture, Simulink and Stateflow recommendations. The same structure has been followed by GETS while defining its own guidelines. The majority of the work has been focused on Model Architecture and Stateflow rules. GETS already provided its own Naming Conventions for software and these conventions have been translated into modeling rules according to specific analysis of the generated code. Since only Stateflow blocks were used for the purpose of code generation, only MAAB Simulink rules dealing with model structure have been adopted and included in the Model Architecture rules set.

MAAB recommendations for Model Architecture and Stateflow have been analyzed and enhanced to produce the currently used GETS modeling standard.

3 Model Architecture Guidelines

Guidelines on Model Architecture concern the hierarchical development of Simulink/Stateflow models. The MAAB recommendations belonging to this set are quite general and do not give much support in developing the architecture of a system and moreover do not cover issues related to code generation. More detailed architectural guidelines have been developed to fit the railway signaling domain and address the code generation problem. The most important ones are reported below.

Rule GE_A_1: View Partitioning *The system shall be partitioned into three hierarchical view: Interface View, Architecture View and Design View.*

The Interface View describes the interaction with external systems. It is implemented through Simulink and it is linked to the Architecture View through a Model Reference block. The Architecture View describes the interaction between functional units. It is implemented in a Simulink model and it is constituted by interacting Stateflow Charts. The Design View represents the single functional units and it is implemented through Stateflow.

Rule GE_A_2: Interface View *The overall system shall define its Interface View in terms of just variables and buses. No direct connection with custom driver code is allowed at any level of the model.*

This rule enables the possibility of generating the code from the system and afterwords connecting the input variables with the sensor drivers (e.g., odometer, operator dashboard) and the output variables with the actuators drivers (e.g., braking system, operator display), facilitating the integration phase.

Rule GE_A_3: Design View *Each functional unit shall define input and output data and shall be implemented through a single Stateflow Chart.*
This rule helps concurrent development, since developers can implement Stateflow Charts separately, once the interface with the other components is given. At code generation level this enables easy unit testing: each Stateflow Chart is translated into a single file with a unique interface function having the same input and outputs of the Stateflow chart from which it has been generated.

4 Stateflow Guidelines

MAAB recommendations given for Stateflow modeling concern Stateflow constructs usage and patterns implementation. Each one of these rules is rated with a priority label (Recommended (R), Strongly Recommended (SR) and Mandatory (M)), issuing the level of importance of the recommendation. Even though the guidelines are rather detailed, they need some priority restriction and modification to be suitable for the railway signaling context. GETS, as many other companies, has its own coding standard addressing the quality requirements of the CENELEC norm. A strong effort has been devoted to identify Stateflow modeling rules ensuring compliance of the generated code with the existing software coding standard. Table 1 summarizes the rules that have been selected from the Stateflow recommendation set of MAAB and the priority restrictions or possible modifications issued.

Rule ID	Title	Priority	Priority Restriction
db_0129	Stateflow transition appearance	SR	Modified
db_0133	Use of patterns for Flowcharts	SR	M
db_0132	Transitions in Flowcharts	SR	M
jc_501	Format of entries in a State block	R	SR
jc_0521	Use of the return value from graphical functions	R	SR
na_0001	Bitwise Stateflow operators	SR	M
jc_0451	Use of unary minus on unsigned integers in Stateflow	R	M
na_0013	Comparison operation in Stateflow	R	M
db_0122	Stateflow and Simulink interface signals and parameters	SR	SR
db_0125	Scope of internal signals and local auxiliary variables	SR	M
jc_0491	Reuse of variables within a single Stateflow scope	R	M
jm_0011	Pointers in Stateflow	SR	M
db_0151	State machine patterns for transition actions	SR	Modified
db_0148	Flowchart patterns for conditions	SR	M
db_0149	Flowchart patterns for condition actions	SR	M
db_0134	Flowchart patterns for If constructs	SR	M
db_0159	Flowchart patterns for case constructs	SR	M
db_0135	Flowchart patterns for loop constructs	R	M

Table 1: MAAB Guidelines Restrictions

Priority restrictions are in general given for those rules having some impact on code generation. For example, rule db_0125 states that every local data in a Stateflow Chart shall be defined at Chart level and not at Machine level. Since every local data that is defined at Machine level is generated as global data by Stateflow Coder, this rule has to be set as mandatory: global data is forbidden by the GETS software coding standard. Another example is the rule jc_0451 which regulates the usage of unary minus on unsigned integers. Unary minus shall be forbidden for unsigned integers to avoid possible overflow errors even if no warning is issued by the code generator.

Some rules needed to be replaced or modified when considered in the GETS context. For example rule db_0132 states that transitions entering states are allowed if they are directed to sub-states. This is not permitted by GETS modeling rules (see GE_S_04 in figure 1) since allowing a transition between states at different hierarchical level reduces the separation of concerns and moreover the generated code results in a *switch/case* statement without a *default* condition, violating one of the pre-existing GETS coding rules.

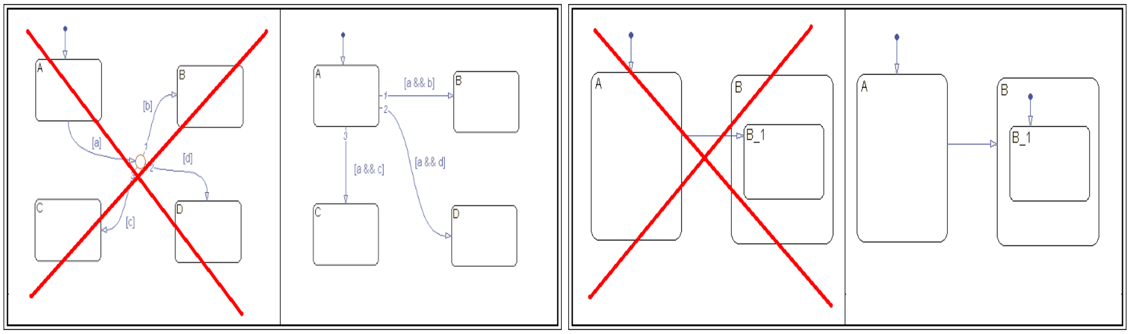
Other GETS specific guidelines have been added to address the need for GETS coding rules compliance. These additional recommendations range from limitation in the number of states for each Stateflow Chart to restriction in Function objects usage. Examples of additional rules are reported below.

Rule GE_S_01: States and Junctions *States and Junctions shall not be used jointly*

Rule GE_S_02: Path Connections *Avoid path connections in Function objects*

Rule GE_S_03: Maximum Number of States *Each state machine shall be composed of maximum 10 states having the same hierarchical level*

Violation of the first two rules generates code with additional local variables with names depending on the code generator, decreasing the readability degree of the software. The third rule is used to limit the maximum number of *case* statements for each *switch/case* block.



(a) GE_S_01

(b) GE_S_04

Figure 1: Two examples of the GETS modeling rules

5 SCMT/SSC Baseline 3 Story

GETS experience with Stateflow modeling started in 2002 with the specification of the SSC ATP system [2], a platform developed by GETS for the Railway Italian Network (RFI) and currently deployed over 2000 kilometers of Italian secondary lines. The model of SSC was developed during the requirements definition phase, to support interaction with the costumer and without considering the issues related to code generation. At the end of 2007 GETS started refactoring the previously existing SSC model to enable code generation. The refactoring experience led to the definition of a preliminary version of the GETS modeling rules, bound to modify and enhance the MAAB guidelines.

In 2008 GETS was involved in a bid for the development of a new platform, called SCMT/SSC Baseline

3. The system was deemed to integrate into a single on-board platform the functionality of SSC and SCMT, the ATP system deployed over the main Italian lines. In order to speed-up development time and to deal with the complexity of the new platform, GETS decided to build a model of the SCMT system targeted for code generation, according to the previously defined rules. A third model, the SCMT/SSC Manager, was developed to let the two systems cooperate in an integrated environment. After code generation the software units of the three models have been integrated with the driver code of the system and software/hardware integration testing has been successfully performed. Along with the development of the new platform the GETS modeling rules have been updated and formalized to address new issues related to the large scale of the project.

The final on-board platform is composed by about 150K LOC of generated application level code and 40K LOC of hand-crafted driver and operating system code. This means that the majority of the software can be directly simulated and tested through Simulink. Development of the entire system, including testing activities, took about one year with a team of 15 people. Since SSC/SCMT Baseline 3 was a larger scale project compared to previous ones, it is not possible to quantify the productivity enhancement achieved. What can be stated is that the adoption of the modeling tools together with the MAAB rules tailoring allowed GETS to manage a complex platform in a structured yet flexible manner without increasing the number of developers. This would have been rather hard to address through the traditional GETS process based on hand-crafted code.

6 Conclusion

In this article we presented the successful experience of a railway signaling manufacturer in developing its own modeling guidelines for code generation, through proper tailoring of the MAAB recommendations. The experience demonstrated how automatic production of qualified code can be addressed through a rigorous set of rules, and showed how the peculiarities of the railway signaling context can be handled through suitable modeling choices. GETS is currently considering customizable tools that can provide automatic verification of these rules, such as MATE [6] and Simulink Model Advisor.

References

[1] National Aeronautics and Space Administration. *Report on the Utility of the MAAB Style Guide for V&V/IV&V of NASA Simulink/Stateflow Models*. N.A.S.A, May 1st, 2004. `http://sarpresults.ivv.nasa.gov/`.

[2] S. Bacherini, A. Fantechi, M. Tempestini, and N. Zingoni. *A Story about Formal Methods Adoption by a Railway Signaling Manufacturer*. 2006. FM 2006, Hamilton, Canada, Lecture Notes in Computer Science, 4025.

[3] Mathworks Automotive Advisory Board. *Control Algorithm Modeling Guidelines Using Matlab, Simulink and Stateflow, Version 2.0*. July 27th, 2007. `http://www.mathworks.com/industries/auto/maab.html`.

[4] European Committee for Electrotechnical Standardization. *CENELEC EN 50128, Railway Applications, Software for Railway Control and Protection Systems*. June, 1997.

[5] Japan MathWorks Automotive Advisory Board (J-MAAB). *Simulink StyleGuide*. March, 2003. `http://www.embeddedcmmi.at/fileadmin/docs/reports/J-MAAB_SimulinkStyleGuide_Eng.pdf`.

[6] D. Sturmer and I. Travkin. *Automated Transformation of MATLAB Simulink and Stateflow Models*. 2007. Proceedings of 4th Workshop on Object-oriented Modeling of Embedded Real-time Systems, pp 57-62.

[7] dSPACE GmbH TargetLink Product Management. *Modeling Guidelines for MATLAB/Simulink/Stateflow and TargetLink*. May 21st, 2008. `https://www.e-guidelines.de`.

Tactical Synthesis Of Efficient Global Search Algorithms

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

Algorithm synthesis transforms a formal specification into an efficient algorithm to solve a problem. Algorithm synthesis in Specware combines the formal speicifcation of a problem with a high-level algorithm strategy. To derive an efficient algorithm, a developer must define operators that refine the algorithm by combining the generic operators in the algorithm with the details of the problem specification. This derivation requires skill and a deep understanding of the problem and the algorithmic strategy. In this paper we introduce two tactics to ease this process. The tactics serve a similar purpose to tactics used for determining indefinite integrals in calculus, that is suggesting possible ways to attack the problem.

1 Background

There have been a variety of approaches to program synthesis (e.g. see [Kre98] for a survey). The focus of this paper is an algorithm class called Global Search (GS) [Smi88]. Using this algorithm class, Smith and his colleagues have successfully synthesized a number of practical algorithms, including, in one case, a scheduler that ran several orders of magnitude faster than comparable hand-written ones [SPW95]. The starting point is a specification $\langle D, R, O \rangle$, where D is an input type, R an output type, and $O : D \times R \rightarrow Boolean$ is an output or correctness condition, along with a global search theory extension (described below). Then the following program, given an input x, returns a solution $z : R$ satisfying the output condition, if one exists (there are some additional conditions on R which will be explained shortly):

```
f(x:D) : R =
    if propagate(x,$r_0$(x))=None then None else gs(x,r)
gs (x:D, r:R) : R =
    let z=Extract(r) in if z/=None && O(x,z) then z else gsAux(x,Subspaces(r))
gsAux (x:D, subs:{R}) :  R =
        if subs={} then None
        else let (s, rest) = arbsplit(subs) in
             if propagate(s) = None then gsAux(x, rest)
             else let  z= gs(x,s) in
                  if z = None  then gsAux(x,rest) else z
propagate(x, r) =  if $\Phi$(x,r) then iterateToFixedPoint($\psi$, x, r)  else None
iterateToFixedPoint (f, x, r) =
    let fr = f(x,r) in if FP?(fr,r) then fr else iterateToFixedPoint(f, x, fr)
```

The program is a classic search algorithm. It works by taking an initial space of possible solutions (corresponding to the root node of a search tree), and unless it can immediately extract a feasible solution, partitioning it into subspaces (corresponding to child nodes), each of which is searched in turn until a feasible solution is found. In this paper we provide tactics for synthesizing the operators Φ and ψ.

The remaining functions are defined in the global search theory extension, *GS-ext,* supplied by the developer, which is an algebra over R with the following operators: $r_0 : D \rightarrow R$ returns a descriptor of the initial search space, $Extract$:$R \rightarrow R$ determines whether the given space is terminal and if so, returns a solution (otherwise the distinguished element None, denoting an empty space). $Subspaces : R \rightarrow \{R\}$

E. Denney, D. Giannakopoulou, C.S. Păsăreanu (eds.); The First NASA Formal Methods Symposium, pp. 171-175

returns a set of subspaces of the current space,$\Phi : D \times R \rightarrow Boolean$ is a necessary filter - those spaces that do not pass Φ need not be examined. It can be any predicate over R satisfying $r \sqsubseteq z \wedge O(x,z) \Rightarrow \Phi(x,r)$. $\sqsubseteq$ is a refinement relation over R. The intent of $\sqsubseteq$ is that if $r \sqsubseteq s$ then s is is a subspace of r (any solution contained in s is contained in r) and is "more defined" than r . $\psi : D \times R \rightarrow R$ is called a necessary propagator. It "tightens" a given space to eliminate infeasible solutions and can be any predicate satisfying $\forall z. r \sqsubseteq z \wedge O(x,z) \Rightarrow \psi(x,r) \sqsubseteq z$. When $\langle R, \sqsubseteq \rangle$ forms a lattice, Smith et al. [SPW95] show how a monotone inflationary ψ can be iterated from any starting space to a fixpoint which is the tightest possible space that still preserves all the original feasible solutions. That is what the *propagate* function in the abstract program above does. An axiomatic definition of GS theory and proof of correctness of the abstract program can be found in [Smi88].

1.1 A Constraint Satisfaction Theory

We are developing a specialization of Global Search to solve problems that involve multi-variable Constraint Satisfaction (CS) [Dec03]. Unlike generic constraint solvers [San94], which accept constraints as input and find a solution, in our approach the constraint is the output condition of the problem to be solved. This constraint is the starting point of algorithm synthesis, not dynamic constraint solving. In this way, many of the problems we will look at can be solved by constraint satisfaction,[Dec03]. For this reason, it is useful to have a specialization of the GS class for Constraint Satisfaction (CS) problems, which we can later extend to each specific problem as needed.

In a nutshell, constraint satisfaction does the following:: given a set of variables, $\{1..maxVar\}$, assign a value, drawn from some domain D_v, to each variable, in a manner that satisfies some set of constraints. The theory which does this, we call CST, is defined below. All other domain specific theories we will use will monotonically extend this theory.

$R \mapsto m : Map(Nat \rightarrow D_v) \times tbd : \{Nat\} \times ch : Map(Nat \mapsto \{D_v\})$
$D \mapsto maxVar : Nat \times vals : \{D_v\}$
$O \mapsto \lambda x, z.\ dom(z.m) = \{1..x.maxVar\}$
$r_0 \mapsto \lambda x. \{m = \emptyset, tbd = \{1..x.maxVar\}, ch = \{(v \mapsto x.vals) | v \in \{1..x.maxVar\}\})$
$Subspaces \mapsto \lambda x, \widehat{z}.\ \{\widehat{z}' : v = pick(\widehat{z}.tbd) \wedge a \in \widehat{z}.ch(v) \wedge \widehat{z}'.m = \widehat{z}.m \oplus \{v \mapsto a\} \wedge \widehat{z}'.tbd = \widehat{z}.tbd - \{v\}\}$
$Extract \mapsto \lambda z.$ if $z.tbd = \emptyset$ then z else $None$
$\sqsubseteq \mapsto \{(\widehat{z}, \widehat{z}') | \widehat{z}.m \subseteq \widehat{z}'.m\}$
$\Phi \mapsto \lambda x, \widehat{z}.\ True$
$\psi \mapsto \lambda x, \widehat{z}.\ \widehat{z}$

In this theory, branching occurs via the *subspaces* function. The *subspaces* function, after picking a variable from the set of variables not yet assigned a value (*tbd*), returns the subspaces formed by assigning to v each of the possible values (drawn from ch*(v)*), adding each pair to the map m, and removing v from tbd. The initial space r_o makes all the values in $x.vals$ available to every variable. The choice of which variable to pick does not matter functionally, but can have a significant impact on the efficiency of the actual program. We will often abbreviate $\widehat{z}.m(i)$ as $\widehat{z}_i$. Now with a definition for D_v, and whatever conditions are appropriate added to O, the abstract program given earlier becomes a working constraint satisfaction solver. The key to making it efficient are appropriate definitions for Φ and ψ[1] . This is what the next section examines.

[1] Often, further optimizations such as context-dependent simplification, finite differencing, and data structure selection have to be carried out before arriving at a final efficient algorithm . However, these latter operations are not the focus of this paper.

2 Tactics

In order to get an efficient final algorithm, the developer must typically find good instantiations of the operators Φ and ψ. The question of where to begin often arises. For this reason we propose to formulate a library of *tactics* that can be used by a developer attempting to instantiate one of the operators. The analogy is with tactics used for integration in calculus. Unlike differentiation, integration has no straightforward algorithm. Rather, there are a number of (some 7 or 8) tactics such as "integration by parts", "integration by partial fractions", "integration by change of variable", etc., that can be tried in order to try and determine the integral of a given formula. There are of course differences. Unlike integration, there is often no one "correct" answer. Also our tactics are often inspired by techniques used in algorithms in computer science and operations research, rather than calculus. But the basic principle is the same: to package up a number of tedious calculations into a pattern-matching rule. Furthermore, by expressing the technique in more abstract form as a tactic, it can be applied to other problem areas, *without requiring the developer be familiar with the implicit assumptions and notations when the technique is buried inside a specific algorithm*, The ultimate goal is that a competent developer will be able to use the approach we propose here to investigate a variety of solutions to their problem.

2.1 A Tactic for Calculating Φ

This tactic helps in constructing Φ (necessary) filters when the feasibility constraint takes a certain form.

TACTIC 1: *If a conjunct from O matches the form $\bigotimes_{i\in I} F_i(z_i) \preceq K$ where $\bigotimes$ is a monotone associative operator, and $\preceq$ forms a meet semi-lattice over range(F), then a possible Φ is one in which the combination of value assignments in the partial solution combined ($\otimes$) with the least possible value assignments for the remaining variables is $\preceq K$.*

The tactic is backed by the following theorem. Note, $\oplus$ denotes extending a partial solution, that is $\widehat{z} \oplus e$ means $\widehat{z}.m \cup e.m$ (unless otherwise stated, we will always be assuming $dom(\widehat{z}.m) \cap dom(e.m) = \emptyset$)

Theorem 1. If $O(x,z) \Rightarrow \bigotimes_{i\in I} F_i(z_i) \preceq K$ for some K, some family of functions $\{F_i\}$, $\bigotimes$ a monotone associative operator, and $\preceq$ forms a meet semi-lattice over $rng(F)$, *then*

$$O(x,\widehat{z} \oplus e) \Rightarrow \Big(\bigotimes_{1\leq i\leq \#\widehat{z}} F_i(\widehat{z}_i) \otimes \bigotimes_{1\leq i\leq \#e} f_i \Big) \preceq K \textit{ where } f_i = \sqcap_{a\in x.vals} F_i(a)$$

Proof.

$$\begin{array}{l}
O(x,z) \\
\Rightarrow \{\text{assumption}\} \\
\bigotimes_{1\leq i\leq \#z} F_i(z_i) \preceq K \\
= \{z = \widehat{z} \oplus e \text{ and use associativity of } \otimes\} \\
\bigotimes_{1\leq i\leq \#\widehat{z}} F_i(\widehat{z}_i) \otimes \bigotimes_{1\leq i\leq \#e} F_i(e_i) \preceq K \\
\Rightarrow \{\text{replace every } F_i(e_i) \text{ with } f_i = \sqcap_{a\in x.vals} F_i(a) \text{ and use polarity}\} \\
\bigotimes_{1\leq i\leq \#\widehat{z}} F(\widehat{z}_i) \otimes \bigotimes_{1\leq i\leq \#e} f_i \preceq K
\end{array}$$

□

Additionally, if F_i is monotone, and $x.vals$ has a least element, then, using the following Quantifier Elimination law: $\sqcap_{\breve{a}\preceq a} F_i(a) = F_i(\breve{a})$, we can rewrite the last line above as:$\bigotimes_{1\leq i\leq \#\widehat{z}} F_i(\widehat{z}_i) \otimes \bigotimes_{1\leq i\leq \#e} F_i(\breve{a}) \preceq K$ where $\breve{a}$ is the least value of $a \in x.vals$.

A symmetrical result is obtained by replacing $\preceq$ with $\succeq$, "meet semi-lattice" with "join semi-lattice", $\breve{a} \preceq a$ with $\hat{a} \succeq a$, and $\sqcap$ with $\sqcup$ everywhere in the above theorem.

Example 2. 0-1 Integer Linear Programming (01-ILP) [2]

A GSO theory for 01-ILP is obtained by extending CST as follows (only the components that differ from CST are shown): $D_v \mapsto \{0,1\}$, $D \mapsto CST.D \times l : Nat \times \mathbf{A} : Map(\{1..l\} \times \{1..n\} \mapsto Real) \times \mathbf{b} : Map(\{1..n\} \mapsto Real)$, $O \mapsto \lambda(x,z).\, CST.O(x,z) \wedge (x.\mathbf{A}) \cdot (z.m) \leq x.\mathbf{b}$

To apply the tactic above, the operator $\otimes$ is interpreted as $\sum$ and F_i as $(\mathbf{A}_{hi}\cdot)$ for appropriate h , over the lattice $\langle Real, \leq, \min, \max\rangle$. Applying the tactic (but not the final simplification since $(\mathbf{A}_{hi}\cdot)$ is not monotone) gives the following filter Φ:$\forall h.\, 1 \leq h \leq 1.\, \sum_{i \in dom(\widehat{z}.m)} \mathbf{A}_{hi} \cdot z_i + \sum_{i \in dom(e.m)} (\min_{a \in \{0,1\}} \{\mathbf{A}_{hi} \cdot a\}) \leq \mathbf{b}_h$ which is by case analysis: $\forall h.\, 1 \leq h \leq l.\, \sum_{i \in dom(\widehat{z}.m)} \mathbf{A}_{hi} \cdot z_i + \sum_{j \in dom(e.m)} (\min\{\mathbf{A}_{hi} \cdot 0, \mathbf{A}_{hi} \cdot 1\}) \leq \mathbf{b}_h$, or after simplifying:

$$\forall h.\, 1 \leq h \leq l.\, \sum_{i \in dom(\widehat{z}.m)} \mathbf{A}_{hi} \cdot z_i + \sum_{i \in dom(e.m)} (\min\{0, \mathbf{A}_{hi}\}) \leq \mathbf{b}_h$$

Using the same tactic we have obtained a filter for the Vehicle Routing Problem (VRP) equivalent to one used in algorithm textbooks. The next example shows that the generalization offered by the tactic is indeed useful enough to carry over to other qualitatively differet problems.

Example 3. The Set Covering Problem (SCP)

Suppose we are given a collection of subsets of a set S, each of which has a certain cost. The SCP is the problem of determining the minimum cost collection of subsets that "covers" the original set, ie. every element in S is in at least one subset in the resulting collection. The problem has many practical applications including airline crew scheduling, facility location, and logic circuit minimization. A GSO theory for SCP is obtained by extending CST as follows (only the components that differ from the base theory are shown): $D_v \mapsto \{False, True\}$, $D \mapsto CST.D \times ss : Map(Nat \mapsto \{Id\})$, $O \mapsto \lambda(x,z).\, CST.O \wedge \bigcup_{i|z_i} S_i = S$

Id is some user defined set element type. *x.ss* returns the actual subset given a variable from $\{1..x.maxVar\}$. S_i stands for the subset $x.ss(i)$, and S stands for $\bigcup_{i \in \{1..x.maxVar\}} S_i$. To apply the tactic, we instantiate $\otimes$ as $\cup$, F_i as $\lambda z_i.\, z_i \rightarrow S_i \,|\, \{\}$, over the join semi-lattice $\langle \{S\}, \subseteq, \{\}, \{S\}\rangle$. Certainly, $\bigcup_{i|z_i} S_i = S$ implies $\bigcup_{i|z_i} S_i \supseteq S$ that is, $\bigcup_i F_i(z_i) \supseteq S$. Applying the tactic gives us a filter $\bigcup_i F_i(\widehat{z}_i) \cup \bigcup_i F_i(S_i) \sqcup \{\}) \supseteq S$ = $\bigcup_i F_i(\widehat{z}_i) \cup \bigcup_i F_i(S_i) \supseteq S$, that is if at any point, the union of the selected sets in $\widehat{z}$ along with all the remaining sets is not at least S, then the space $\widehat{z}$ can be eliminated.

2.2 A Tactic for Calculating ψ

Observe that in the initial space r_o all value choices (from D_v) are available to every variable. The intent, though, is that propagation will narrow this set to only those that would lead to feasible solutions (analogous to hyper-arc consistency in CSP). If at any point a choice set becomes empty, then that space can be abandoned. This is the idea behind the following tactic for ψ . The tactic applies when the variables (*vars*) of the input can be viewed as, or represent, nodes in some kind of graph structure, so we can talk about the "neighborhood" around a variable.

TACTIC 2: *If one of the conjuncts of O matches the form* $\forall j \in N_i.\, z_i \neq z_j$ *where* N_i *is some neighborhood of points around i then a possible* ψ *is* one in which the choice of values available to variable j does not contain the value assigned to variable i.

The tactic is backed by the following theorem

[2] To simplify the presentation we have omitted the optimization aspect of many of the examples we discuss since none of our tactics pertain to optimization. In our actual implementation we use a generalization of GS that incorporates optimization.

Theorem 4. *If $O(x,z) \Rightarrow \forall j \in N_i . z_i \neq z_j$for some set $N_i \subseteq x.vars$ then*

$$\widehat{z} \sqsubseteq z \wedge O(x,z) \Rightarrow \psi(x,\widehat{z}) \sqsubseteq z \textit{ where } \psi(x,\widehat{z}) = \widehat{z}\{ch(j) = \widehat{z}.ch(j) - \widehat{z}_i \,|\, j \in N_i\}$$

where the notation $o\{f(i) = v \,|\, P(i)\}$ denotes the object obtained by replacing the value of the ith index of field f of object o with v when $P(i)$ holds. The value is unchanged otherwise.

Example 5. Maximum Independent Segment Sum Problem (MISS), [SHT00]

This is a variant of the well-known maximum segment sum problem (MSS) in which the goal is to maximize the sum of a selection of elements from a given array, with the restriction that no two adjacent elements can be selected. The specification of the problem is as follows: $D_v \mapsto \{False, True\}$, $D \mapsto CST.D \times data : [Int]$, $O \mapsto \lambda(x,z).\, CST.O \wedge \forall i : 1 \leq i < \#z.m. \ : z_i \Rightarrow \neg z_{i+1}$

Now let N_i be the left and right neighbors of i, i.e. $i-1$ and $i+1$, if z_i and $\{\}$ otherwise. Then in the case where z_i holds, $\psi(\widehat{z}) = \widehat{z}\{ch(i+1) = ch(i+1) - \{True\}\}$ which is just $\widehat{z}\{ch(i+1) = \{True\}\}$.

Using this tactic we have also derived a ψ function for the Graph Coloring Problem and a variety of puzzles including n-Queens and Sudoku.

2.3 Summary and Future Work

We have shown how for certain problem types, calculation of the operators Φ and ψ can be replaced by pattern matching and substitution. The lesson here for program synthesis is that narrowing down the range of problem types can lead to much faster program design. We have developed a number of other such tactics, which space does not permit us to describe here. We can also handle optimization problems by incorporating dominance relations [Smi88] and bounds tests into our approach, and have developed a number of tactics for their calculation. Using one such tactic, we have synthesized a previously unpublished greedy solution to the Unbounded Knapsack Problem, and another tactic for dominance relations led us to fast solutions to variants of the Maximum Segment Sum problem that improve on the work of Sasano et al.,[SHT00]. Our eventual goal is to have a library of tactics sufficient to tackle significant Global Search problems such as synthesizing fast planners and efficiently mapping platform independent models to platform specific models.

Acknowledgment

This material is based upon work supported by the National Science Foundation under Grant CCF-0724979

References

[Dec03] R Dechter. *Constraint Processing*. Morgan Kauffman, 2003.

[Kre98] Christoph Kreitz. Program synthesis. In W. Bibel and P. Schmitt, editors, *Automated Deduction – A Basis for Applications*, volume III, chapter III.2.5, pages 105–134. Kluwer, 1998.

[San94] Michael Sannella. The skyblue constraint solver and its applications. In *Proceedings of the 1993 Workshop on Principles and Practice of Constraint Programming*, pages 385–406. MIT Press, 1994.

[SHT00] Isao Sasano, Zhenjiang Hu, and Masato Takeichi. Make it practical: A generic linear-time algorithm for solving maximum-weightsum problems. In *Proc. Intl. Conf. on Functional Prog.(ICFP)*, 2000.

[Smi88] D R Smith. Structure and design of global search algorithms. Technical Report Kes.U.87.12, Kestrel Institute, 1988.

[SPW95] Douglas R. Smith, Eduardo A. Parra, and Stephen J. Westfold. Synthesis of high-performance transportation schedulers. Technical report, Kestrel Institute, 1995.

Towards Co-Engineering Communicating Autonomous Cyber-physical Systems

Marius C. Bujorianu and Manuela L. Bujorianu*
[Marius,Manuela].Bujorianu@manchester.ac.uk
School of Mathematics, University of Manchester, UK

Abstract

In this paper, we sketch a framework for interdisciplinary modeling of space systems, by proposing a holistic view. We consider different system dimensions and their interaction. Specifically, we study the interactions between computation, physics, communication, uncertainty and autonomy. The most comprehensive computational paradigm that supports a holistic perspective on autonomous space systems is given by cyber-physical systems. For these, the state of art consists of collaborating multi-engineering efforts that prompt for an adequate formal foundation. To achieve this, we propose a leveraging of the traditional content of formal modeling by a co-engineering process.

Keywords: *cyber-physical systems, uncertainty, degrees of autonomy, communication, formal modeling, structured operational semantics, system co-engineering.*

1 The need for holistic modeling

Many systems from aerospace engineering can be characterized as been *cyber-physical*, i.e. their dynamics is based on the interaction between *physics* and *computation* and they are networked. Satellites, aircraft, planetary rovers are instants of cyber-physical systems (CPS). In the process of formal modeling of these systems, a developer should consider actually consider the interactions between *communication*, physics and computation. We sketch a reference framework, where the subtleties of these interactions can be captured. Moreover, we add a further dimension to these complex interactions by adding *uncertainty*. However, the special conditions in which the space systems are deployed require also a high degree of *autonomy*, adding an extra-dimension to system modeling. We approach the issue of mastering the interactions of many system dimensions for complex space systems by integrating formal modeling into a larger system development process called *system co-engineering*. Instances of this process are the *Hilbertean formal methods* [2] and the *multidimensional system co-engineering* (MScE) framework [1].

Autonomous systems can be modeled from two perspectives: *black box* and *white box*. The black box view is specific to the approaches based on *hybrid dynamical systems*. In these approaches, the system behavior is described as seen by an *external observer*. This observer records a sequence of continuous behaviors, each one triggered by a discrete transition (event). In our model, there are two types of discrete transitions: *controlled*, which are the transitions of a discrete automaton, and *spontaneous* (or *autonomous*), which are transitions that can not be explained using only the elements of the model. The systems with spontaneous transitions are called *uncertain*, and they are usually modeled as random processes. Using *structured operational semantics* (SOS), some spontaneous transitions can be defined as a special class of controlled transitions that are triggered by communication. In this way, communication reduces the randomness of the model. In the white box view, some of the internal system structure is revealed. In our framework, we explain the interaction between communication, autonomy and control using the concept of *nested feedback*. The feedback is the fundamental structure of the controlled systems, constructed by connecting some input and output channels. A nested feedback results from adding a feedback loop to a system that has already another feedback in its structure. When applying this concept to CPS, one can easily distinguish a subclass known as *hierarchical hybrid systems* [6]. We define and use nested feedbacks to explain the autonomy and its interaction with communication. Specifically, as more nested feedbacks are added, the system autonomy increases. A system with four nested feedbacks

E. Denney, D. Giannakopoulou, C.S. Păsăreanu (eds.); The First NASA Formal Methods Symposium, pp. 176-180

*This work was funded by the CICADA project under the EPSRC grant EP/E050441/1.

can be considered as fully autonomous because it has enough *information structure* to partially control itself. Systems with five nested feedbacks or more have additional features like *concurrency* and *self-* properties* (self-reconfiguration, self-healing, etc). In a hierarchy of nested feedbacks, communication is defined as the top loop. In this way, the whole behavior of the cyber-physical system is controlled via communication. A rigorous study of autonomy and communication in a cyber-physical context goes beyond the traditional content of formal methods, in a form of an interdisciplinary paradigm that we call *system co-engineering* (see [1] and the references therein). Co-engineering is a creative process combining concepts and techniques from two different scientific disciplines. In our approach, the system co-engineering is *multi-dimensional*, integrating *formal engineering*, *control engineering*, and *mathematical engineering*. The integration process departs from a mathematical model of complex systems called *stochastic hybrid processes* (SHS) [3].

2 Cyber-physical systems: autonomy and communication

Designing safe autonomous space systems requires accurate and holistic models, where all interactions between orthogonal system features can be understood. In a formal approach, the first step will be to define formal models for which these interactions can be mathematically studied. Such systems would involve digital control of some devices with continuous dynamics and embedded in a physical environment. They are also uncertain in the sense that they are subject to some random perturbations from the physical environment. Moreover, we adopt a holistic view by studying each device in its deployment context and by proposing a concurrent model. For example, in the case of an outerspace aircraft there can be communications with the ground control or/and with the ISS. Another example is that of two extra-terrestrial rovers that co-ordinate their activities by communicating complex data (position, etc.).

2.1 A formal model for cyber-physical systems

First, we need to model the physical environment. For simplicity, we consider the system state space to be a subset X of the Euclidean space of dimension n (the number of relevant parameters). A system will evolve within a set Q of regions (that we call formally *modes* or *locations*) defined as a sort of topological sets (that could be open/closed/compact sets, and so on). Each mode $q \in Q$ is characterized by a predicate β_q. The random perturbations are modeled as a "white" noise, i.e. in each region there is defined a Wiener process $(W_t, t \geq 0)$. Note that, in different regions, the system can be subject to different types of perturbation.

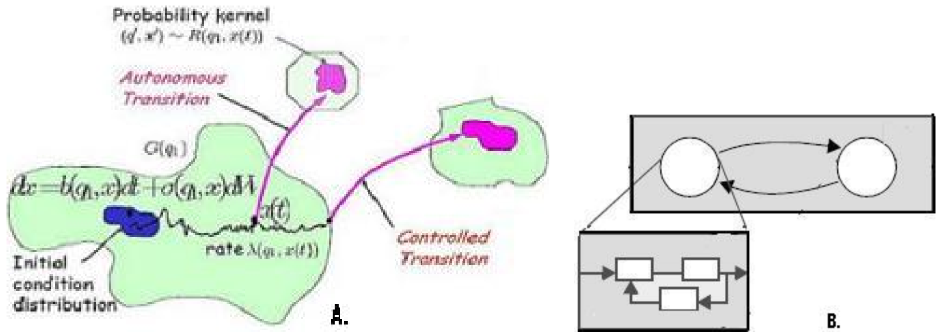

Probability kernel
$(q', x') \sim R(q_1, x(t))$
Autonomous Transition
$G(q_1)$
$dx = b(q_1, x)dt + \sigma(q_1, x)dW$
$x(t)$
rate $\lambda(q_1, x(t))$
Initial condition distribution
Controlled Transition
A.
B.

Figure 1: Simple and hierarchical cyber-physical systems

In each location, the system dynamics is described by a system of deterministic differential equations (usually a first order moving equations), called the designed behavior. In practice, because the system behavior is affected, in each location, by a white noise, the resulted dynamics is described by a stochastic differential equation (SDE): $dx(t) = f^q(x(t))dt + \sigma^q(x(t))dW_t$ and we call that the *physical dynamics*.

The controller transitions are discrete transitions between locations that are triggered by Boolean guards B. We call these *controlled transitions*. However, there is also a class of discrete transitions that take place because of the system autonomy and that are called *spontaneous (or autonomous) transitions*.

Some controlled transitions have communication labels l, usually denoting a communication channel. These are called *communication triggered transitions*. The data types that can be transmitted throughout communication channels are specific to a mode. We denote by γ_q the predicate that state the correctness of communicated data.

In order to predict, evaluate and control the physical dynamics on long time, we need to associate probabilities to all discrete transitions. We call these *jump probabilities* and we denote their rates by λ. Using the jump probabilities, a discrete transition can be formalised by means of a stochastic kernel $R : \overline{X} \times \mathscr{B}(X) \to [0,1]$, where $\mathscr{B}(X)$ represents a class of universal measurable subsets of X. This is a special function, which is measurable in the first argument and a probability measure in the second.

The full formal model is described in [1]. The jump probabilities can be defined using the stochastic kernel and various parameters of the physical dynamics. This is the key to define many sorts of stochastic dependencies between the physical behaviors (*physics*) and discrete transitions (the *computation*).

Each execution path is a Markov string [3]. As result, the global dynamics can be formalised as a Markov process (more specific, a stochastic hybrid process).

2.2 Degrees of autonomy

Considering the very harsh and highly unpredictable environments, in which space systems are deployed, a large autonomy and high reliability are desirable. In this subsection we follow the white box view by defining an original structural classification of autonomous system and indicate how communication can be added by a top feedback to CPS.

The *autonomy degree* of a system is

• (*no feedback*) of level zero (i.e. the system is non-autonomous) if the system is without any feedback connection between or within its components.

• (*nested feedback*) of level n if is obtained by a feedback connection between a system of $(n-1)$ degree with a system with a degree inferior to $(n-1)$ (see *Fig. 2 A.*).

Let us denote the class of n degree systems by $DS(n)$. A *balanced feedback* coupling is a feedback connection for which $n-k \leq 2$. The control of an un-balanced autonomous system is more difficult.

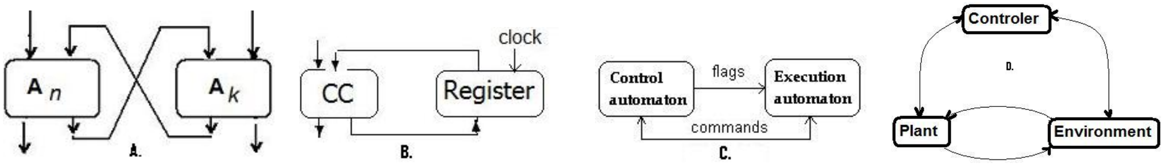


Figure 2: Nested feedbacks: latch, automaton, processor, cyber-physical system

Let us consider in the DS(0) class all *combinational circuits and systems*. Then the DS(1) class contains the *elementary memory* functions (consider NOR gates for the components from Fig. 2 A.). The DS(2) class contains execution (i.e. combinatorial) elements coupled with a memory, i.e. the automata

(Fig. 2. B). The processors are in the DS(3) class (Fig. 2. C), and in the DS(4) class the computer (the von Neumann architecture) can be defined (as a un-blanced couple between a processor and a memory). We can make a correspondence between this hierarchy and the formal languages hierarchy: DS(2) class -> *regular languages*; DS(3) class -> *context-free languages*; DS(4) class -> *context-sensitive languages*.

When considering for level zero a continuous dynamical system we get a hierarchical hybrid system [6] (Fig. 1. B). A CPS contains at least three nested feedbacks (Fig. 2. D), its autonomy degree being thus higher then five (the controller is supposed to be a DS(2) automaton).

In [5], a feedback is modeled as a generalised relation (a span $\swarrow\searrow$ as in category theory) between succesive instances of the plant, modeled as objects in a suitable category. The message passing communication in a CCS style is defined as relations (i.e. spans, i.e. feedback) between systems. This construction can be straight forward applied to the category of SHS defined in [4]. Because we have shown in the previous section that the behaviors of CPS is a SHS, the construction can be also applied to CPS. A network of communicating autonomous CPS has an autonomy degree higher then six.

2.3 Communicating cyber-physical systems

The study of communication in computer science produced a large number of formal models. Not surprisingly, there is wealth of formal models for hybrid systems, and even more for probabilistic systems. However, for more complex systems like SHS communication is less studied [7, 8]. A reason might be the lack of interdisciplinarity, i.e. the communication is not studied in relationship with control and stochastic modeling. This observation is the departing point of our approach in defining communicating CPS in black box style.

The message passing is defined as an one-way communication that takes place only when the sender executes a communication triggered transition.

We denote by $q \overset{l,B,R,a}{\mapsto} q'$, $q \overset{l,B,R,a}{\rightarrow} q'$ and $q \overset{l,R,\lambda}{\rhd} q'$ respectively the communication triggered, controlled, and autonomous transitions from q to q' with label l, guard B, reset map R and communicated data stored in the variable a. The appearance of B, a and λ is optional.

A communication label has two forms: $l = !a$, meaning the value of a is send throughout the channel l or $?b$, meaning the value received the channel l is stored in the variable b. If the label l has the form $!a$ then the label $\bar{l}$ is like $?b$ and viceversa.

The parallel composition of two CPS (with components indexed as 1 and 2) has the parameters:

- $Q = Q_1 \times Q_2$;
- its state space is embedded in the product of the Euclidean spaces corresponding to the two CPS;
- the perturbation is modeled by the product of the two Wiener processes that describe the perturbations corresponding to the two CPS;
- its modes (locations) are obtained by means of tensor products of the component locations;
- $f^{(q_1,q_2)} = \begin{pmatrix} f^{q_1} \\ f^{q_2} \end{pmatrix}$ and $\sigma^{(q_1,q_2)} = \begin{pmatrix} \sigma^{q_1} \\ \sigma^{q_2} \end{pmatrix}$;

The concurrent composition adds the following transition rules to a parallel composition

$$\frac{q_1 \overset{l,B_1,R_1,a}{\mapsto} q_1', q_2 \overset{\bar{l},B_2,R_2,b}{\rightarrow} q_2'}{(q_1,q_2) \overset{l,B,R,c}{\rightarrow} (q_1',q_2')} \quad , \quad \frac{q_1 \overset{l,B_1,R_1,a}{\mapsto} q_1', q_2 \overset{l,R_2,\lambda,b}{\rightsquigarrow} q_2'}{(q_1,q_2) \overset{l,B',R,c}{\rightarrow} (q_1',q_2')} \; l = !a,\, B = B_1 \times B_2,\, B' = B_1 \times \beta_{q_2'}, R = R_1 \times R_2$$

$$\frac{q_1 \overset{l,B_1,R_1,a_1}{\mapsto} q_1', q_2 \overset{\bar{l},B_2,R_2,a_2}{\rightarrow} q_2'}{(q_1,q_2) \overset{l,B,R,a_1}{\rightarrow} (q_1',q_2')} \quad , \quad \frac{q_1 \overset{l,B_1,R_1,a_1}{\mapsto} q_1', q_2 \overset{l,R_2,\lambda,a_2}{\rightsquigarrow} q_2'}{(q_1,q_2) \overset{l,B',R,a_1}{\rightarrow} (q_1',q_2')} \text{ where } B = B_1 \wedge B_2 \wedge \beta_{q'}$$

$$\frac{q_1 \overset{l,R_1}{\mapsto} q_1', q_2 \overset{l}{\nrightarrow}}{(q_1,q_2) \overset{l,R}{\mapsto} (q_1',q_2)} \quad , \quad \frac{q_1 \overset{l,R_1}{\mapsto} q_1', q_2 \overset{l}{\ntriangleright}}{(q_1,q_2) \overset{l,R}{\mapsto} (q_1',q_2)} \; , \; \frac{q_1 \overset{l,B_1,R_1,a1}{\mapsto} q_1'}{(q_1,q_2) \overset{l,B_1,R_1,a1}{\mapsto} (q_1',q_2)}$$

$$\frac{q_1 \overset{l,R_1,\lambda_1}{\rhd} q_1', q_2 \overset{l}{\nrightarrow}}{(q_1,q_2) \overset{l,R,\lambda}{\rhd} (q_1',q_2)}, \quad \frac{q_1 \overset{l}{\nrightarrow}, q_2 \overset{l,R_2,\lambda_2}{\rhd} q_2'}{(q_1,q_2) \overset{l,R,\lambda}{\rhd} (q_1,q_2')} \quad, \frac{q_1 \xrightarrow{l,R_1} q_1', q_2 \xrightarrow{l,R_2} q_2'}{(q_1,q_2) \xrightarrow{l,R} (q_1',q_2')} \quad , \quad \frac{q_1 \xrightarrow{l,B_1,R_1} q_1', q_2 \xrightarrow{l,B_2,R_2} q_2'}{(q_1,q_2) \xrightarrow{l,R} (q_1',q_2')}$$

$$\frac{q_1 \xrightarrow{l,B_1,R_1} q_1', q_2 \overset{l}{\nrightarrow}}{(q_1,q_2) \xrightarrow{l,B,R} (q_1',q_2)} \text{ where } B = B_1 \times \gamma_{q_2} \text{ and } R = (R_1 \times 1)\left((x_1,x_2),\cdot\right) = R_1(x_1,\cdot) \otimes 1_{x_2},$$

$$\frac{q_1 \overset{l}{\nrightarrow}, q_2 \xrightarrow{l,B_2,R_2} q_2'}{(q_1,q_2) \xrightarrow{l,B,R} (q_1',q_2)} \quad \text{where } B = \gamma_{q_1} \times B_2 \text{ and } R = 1 \times R_2$$

$$\frac{q_1 \xrightarrow{l,R_1} q_1', q_2 \overset{l}{\nrightarrow}}{(q_1,q_2) \xrightarrow{l,R} (q_1',q_2)}, \frac{q_1 \overset{l}{\nrightarrow}, q_2 \xrightarrow{l,R_2} q_2'}{(q_1,q_2) \xrightarrow{l,R} (q_1,q_2')} \text{ where } R\left((x_1,x_2),\cdot\right) = R_1(x_1,\cdot) \otimes R_2(x_2,\cdot)\ x_1 \in \partial X^{q_1},\ x_2 \in X^{q_1'}$$

The transition map λ is given by $\lambda(x_1,x_2) = \lambda_1(x_1)$ for all $x_1 \in X^{q_1}$ and $x_2 \in X^{q_1'}$;

If one CPS agent is able to execute a send event and the other CPS agent does not have a matching receive event, then the first agent executes the transition while the second agent stays in the same location. If contrary, the first agent can execute a controlled transition and the second agent has a matching communication triggered transition, then both agents execute respectively the send and communication triggered transitions at the same time. If the first agent has a communication triggered transition with label l and the second agent has no communication triggered transition with label l, then the composed system has a communication triggered transition with label l outgoing from the joint location, which gives the possibility to interact with other CPS agent, in an other composition context. If both agents have a communication triggered transition with the same label, then the composed system also has a communication triggered transition with this label. The implication of this fact is that both agents can execute the communication triggered transitions at the same time in another composition context where a third CPS agent executes a communication transition with the same label.

The main advantage of the reference model described in this paper is that it allows combinations of verification techniques from different disciplines. For example reachability analysis can be carried out using computational methods from statistics and optimal control.

References

[1] Marius C. Bujorianu., Manuela L. Bujorianu, and Howard Barringer: *A Formal Framework for User Centric Control of Probabilistic Multi-Agent Cyber-Physical Systems.* Proc. of the 9th CLIMA workshop, Springer LNCS, (2008), in press.

[2] Marius C. Bujorianu and Manuela L. Bujorianu: *Towards Hilbertian Formal Methods* Proc. of Conf. on Application of Concurrency to System Design ACSD, IEEE Press (2007): 240-241.

[3] Manuela L. Bujorianu and John Lygeros: *Towards Modelling of General Stochastic Hybrid Systems*. In "*Stochastic Hybrid Systems: Theory and Safety Critical Applications"* LNCIS **337** (2006).: 3-30.

[4] Manuela L. Bujorianu, John Lygeros and Marius C. Bujorianu: *Bisimulation for General Stochastic Hybrid Systems*. In Proc. of HSCC, Springer LNCS 3414 (2005):198-216.

[5] Marius C. Bujorianu: *Integration of Specification Languages Using Viewpoints*. In Proc. of Integrated Formal Methods, Springer LNCS 2999, (2004): 421-440.

[6] John Lygeros: *Hierarchical, Hybrid Control of Large Scale Systems*, Ph.D. Thesis, University of California, Berkeley (1996).

[7] Jose Meseguer and R. Sharykin: *Specification and Analysis of Distributed Object-Based Stochastic Hybrid Systems.* Springer LNCS 3927 (2006): 460-475.

[8] Stefan S. Strubbe, Agung Julius and Arjan van der Schaft: *Communicating Piecewise Deterministic Markov Processes*. Conf. on Analysis and Design of Hybrid Systems (2003): 349-354.

Formal Methods for Automated Diagnosis of Autosub 6000

Juhan Ernits
School of Computer Science
University of Birmingham
Birmingham, UK
ernitsj@cs.bham.ac.uk

Richard Dearden
School of Computer Science
University of Birmingham
Birmingham, UK
rwd@cs.bham.ac.uk

Miles Pebody
National Oceanography Centre
University of Southampton
Southampton, UK
M.Pebody@noc.soton.ac.uk

Abstract

This is a progress report on applying formal methods in the context of building an automated diagnosis and recovery system for Autosub 6000, an Autonomous Underwater Vehicle (AUV). The diagnosis task involves building abstract models of the control system of the AUV. The diagnosis engine is based on Livingstone 2, a model-based diagnoser originally built for aerospace applications. Large parts of the diagnosis model can be built without concrete knowledge about each mission, but actual mission scripts and configuration parameters that carry important information for diagnosis are changed for every mission. Thus we use formal methods for generating the mission control part of the diagnosis model automatically from the mission script and perform a number of invariant checks to validate the configuration. After the diagnosis model is augmented with the generated mission control component model, it needs to be validated using verification techniques.

1 Introduction

There are vast areas of the Earth's seabed that have yet to be explored or studied. Recent discoveries of hot hydrothermal vents deep on mid oceanic ridges have revealed whole new ecosystems, many existing independently of energy from the sun. However the global extent and variety of these features is not known. The extensive and efficient exploration of these areas by the oceanographic science community is one example of the use of Autonomous Underwater Vehicles (AUVs). These vehicles are able to operate independently for several days with future developments extending this time span to several months. The Autosub 6000 project of the National Oceanography Centre in Southampton is a continuation of a successful series of projects, that takes the Autosub AUV to depths of up to 6000 m.

During more than 400 previous scientific missions, the predecessors of Autosub 6000 have suffered both near losses and one actual loss. In two cases the AUV has been recovered with a remotely operated underwater vehicle at significant expense. In once case the Autosub2 AUV was permantly lost 17Km under the 200m thick Fimbul Ice Shelf in the Antarctic. There are numerous cases of missions that have had to be aborted but where recovery was possible by the operations team and the attending support ship. Based on the experience of operating the Autosub AUVs a project to apply automated diagnosis and recovery methods for Autosub 6000 was initiated with the primary focus being on the detection of faults that may result in collisions with the seabed. Collision with the seabed is undesirable because it is has been demonstrated to have been one of the primary causes of vehicle loss. The approach we are taking is to use the Livingstone 2 (L2) diagnosis engine on Autosub. L2 is a discrete, model-based diagnosis system that is compositional, allowing models of individual comonents to be plugged together relatively easy to build larger models. We describe L2 in more detail in the next section.

Autosub 6000 [5] provides a configurable payload space of 0.5 m^3 that enables scientists to explore deep water with a range of sensor equipment. Typical missions have included temperature and salinity profiling, water sampling, seafloor mapping, seafloor photography and chemical analyses in areas ranging from tropical waters around Bermuda to under ice in the Arctic and Antarctic. The AUV needs to be reconfigured for each mission, meaning that the sensory equipment may be replaced and depth, position, mission control and abort parameters are changed. Furthermore, for each mission a custom mission script that details the waypoints which the AUV must pass and the actions to be performed must be produced.

E. Denney, D. Giannakopoulou, C.S. Păsăreanu (eds.); The First NASA Formal Methods Symposium, pp. 181-185

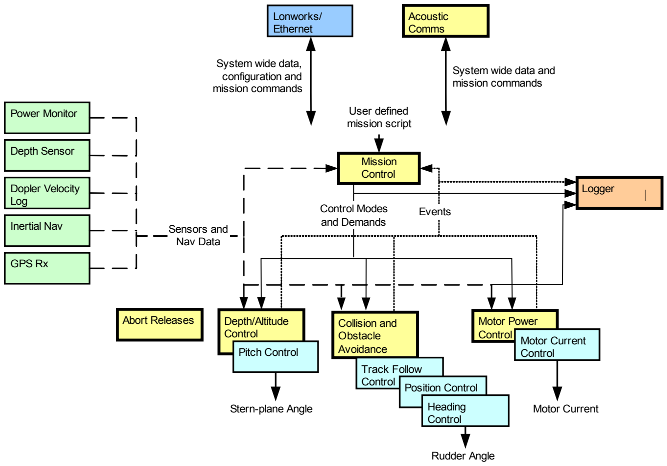


Figure 1: Architecture of the control system of Autosub 6000.

The fact that Autosub is frequently reconfigured between missions means that it is fundamentally different from the space missions that L2 has previously been used for. As well as changes to the payload of the vehicle, each mission has its own configuration script that sets a number of parameters such as maximum depth and limits of the dive plane angles, and the activities carried out will also vary from mission to mission. This leads to an important challenge: How can we update the diagnosis model quickly and correctly between missions? We believe that formal methods provide part of the solution to this challenge by allowing the mission script and configuration script to be sanity checked before use and allowing diagnosis models to be automatically generated from the scripts that can be integrated with the hand-built models of the components.

Generating parts of the diagnosis model automatically either requires proving the model generation correct or applying some verification techniques to the resultant model to raise the level of confidence that integration of the automatically created components with the rest of the model will be reliable.

It has been suggested by Kurien and R-Moreno [3] that the risks and costs of applying model-based diagnosis outweigh the expected value. We believe that while this may be true for space missions, applications such as Autosub are much more compelling domains for model-based diagnosis and that the application of formal methods for automatic generation and validation of the diagnosis model can significantly improve the diagnostic power of the system.

2 Automated diagnosis for Autosub 6000

Our diagnosis approach is based on Livingstone 2 [2, 9], a model-based diagnosis tool developed at NASA's Ames Research Center that has been deployed in several applications. Livingstone works using a discrete event model of the system, and a constraint solver to detect inconsistencies between the model and the system and finds explanations for them.

A Livingstone model is composed of components (which may map onto physical components), connections between components and constraints. A component is specified by variables, with a set of discrete, qualitative values for each variable in its local nominal and failure modes. For each mode, the model specifies the components' behavior and transitions.

Most of the subsystems and components of the Autosub AUV will be modelled ahead of time, i.e. before the AUV is sent to carry out concrete tasks. Fig. 1 illustrates the network variable based control system used in the AUV, based on the principles explained in [6]. The Mission Control component communicates with a number of other components in the system and could thus be a valuable resource for improving the precision of diagnosis. The challenge is to automatically create models representing the information in the mission script and the configuration script and to integrate them with the hand-built component models. This has two advantages which can substantially improve the reliability of the system. First, the mission and configuration scripts constrain the behaviour of the vehicle so diagnosis is easier since the diagnosis engine knows more precisely what should be happening. This means that, for example, the diagnosis engine can detect when the vehicle differs from its allowed behaviour, perhaps by diving too deep. The second advantage is that the diagnosis model can be used as an abstract simulation of the system, so it can be used to perform a number of tests on the mission script, for example to check whether the vehicle can actually carry out the mission successfully if all systems remain nominal. This is similar to the approach taken by Livingstone PathFinder [4].

3 Mission Control

The mission control node [7] is the Autosub component that executes the mission script. The script is an event driven sequence of commands in a text file that is compiled and then downloaded to the mission control node as part of the mission preparation procedure. When a scripted mission event occurs the next commands are issued to the various motor, actuator and sensor control nodes. The mission control then iterates to the next element in the mission script and waits for the next specified event. In addition to the main mission script the operator may specify up to two alternative endings to the mission in the form of mission termination scripts. These can be triggered by a configurable set of events and provide an alternative ending to a mission.

A mission script contains a list of event triggered actuator mode and demand settings contained in a basic structure format called a mission element:

```
when( event$_0$, event$_1$, event$_2$)
    mode$_0$( demand$_0$),
    mode$_1$( demand$_1$),
    ...
    mode$_n$( demand$_n$);
)
```

With actual events modes and demands this might look like:

```
when( GotPosition, ElementTimeout),
    SetElementTimer( 0:0:30:0),            // Time argument 0 days, 0 hours, 30 minutes, 0 seconds.
    MotorPower( 320),                      // Set motor to run at 320W
    PositionP( N:52:30.0, W:15:0.0),       // Position control navigation
    Depth( 2000);                          // Depth control 2000m
)
```

Built into the mission controller are a number of *event processing tasks*. These range from simple input signal polling to the maintenance of internal timers and arbitration between inputs for starting and stopping a mission. Mission control events in general may be divided into events that are generated within the mission control node itself and those that are received from other nodes as network variable updates. A disjunction of up to three events may be specified to trigger a mission element.

The position, depth and motor controllers in the Autosub take commands in the form of an *operation mode setting* and a *demand*. For example the position controller has a mode called ”heading” that is given a demand in degrees of the compass. The mission control outputs a data structure that communicates the required mode and demand to the respective controller. In addition auxiliary modes and demands can be sent to payload sensors. The controllers in the Autosub act on the last received mode and demand. There are a small number of modes and demands that affect the behaviour of the mission control. These enable the setting of parameters that define the internal generation of mission events, in particular depth thresholds and timeout values for mission elements.

A number of features including obstacle avoidance, track following and beacon homing do not form part of the Mission Script. Their effect is at a level below the operator programmer interface and event states provide a means of programming mission script responses to them.

4 Construction and validation of the diagnosis model

The first step in the automatic construction and validation of the diagnosis model is to formalise the general requirements that apply to all mission scripts and configuration. For example, we have identified a number of requirements such as ”the configuration variable `ncSafeMaxDepth` has to be less than abort weight release max depth” and ”the variable `ncMinDepth` has to be less than the maximum depth” that are domain specific but should hold globally for all mission configurations. Previous experience shows that the configuration files that accompany mission scripts can have errors and we expect even such simple checks to improve the overall robustness of the system.

While constructing the diagnosis model from the mission script we apply analysis to the mission script to make sure that demands set in the mission script do not violate any of the general domain specific requirements or configuration parameters. The mission script is comprised of two different kinds of statements, those which block awaiting some particular event (a `when` statement), and those which set operation modes and demands after an event triggering the next step of the mission is observed. So, an example of a safety check performed during the model generation is to make sure a mission script does not have a demand exceeding the maximum allowed depth for the mission.

Each of the `when` statements is converted into two states, the state where the `when` statement is waiting for the triggering event and an intermediate state where the body of the `when` statement is executed. Such a setup allows the diagnosis model to monitor mission progress and detect if the mission control component has omitted broadcasting any demands listed in the mission script. After the demands have been seen by the diagnosis model, the model moves on to a state waiting for an event that triggers the next `when` statement. The mission control component of the diagnosis model is a linear string of such pairs of states. Those states where alternative termination scenarios are allowed can branch to similar pairs of states specified by appropriate termination scripts.

Automatic generation of parts of the diagnosis model will necessarily raise questions about the correctness of the resultant model. While proving the model generation correct would be one option, we will initially use techniques such as Livingstone PathFinder [4] and Livingstone model verification [8] to verify the properties of the resultant diangosis model.

One possible way to validate the resultant model after the mission control component of the diagnosis model has been generated is to create an example mission scenario from the excerpts of the data of previous missions matching the constraints and demands of the appropriate step of the mission script. This provides a way to have dry runs of successful scenarios and also various error scenarios and use an approach like Livingstone Pathfinder to check if the diagnoses of L2 subsume the faults that have been introduced into the scenario script. One research challenge is to see if we can build a useful abstract model of the behaviour of the AUV that encapsulates a large number of concrete scenarios that can be used to prove that no false positives can occur.

The diagnosis model is coupled with the actual variables of the system via monitors which convert continuous variables into discrete values. While the discussion of how exacly the continuous variables are tracked is out of scope of this paper, another of our research challenges is how to model such monitors mathematically for establishing the correctness of their implementations and to further establish the adequacy of the diagnosis model given the properties of the input signals.

Automation of verification and proof procedures in such applications is of vital importance as mission scripts are built and modified during research cruises and it is likely that there is neither time nor expertise for manually validating or building the diagnosis models at sea between subsequent missions.

Another of the open challenges is the *verification of the diagnosis engine* itself. Ideally all code that runs on an autonomous platform should be at least proved correct for safety policies such as memory safety. There are technologies and tools that support such proofs based on Hoare style annotations of pre- and postconditions and loop invariants for establishing various safety policies [1]. In addition, with recent advances in verification tools, an engine written in C++, such as Livingstone 2, with additional support for continuous variables is a verification challenge. The L2 code has been empirically tested to be flight safe, but modifying it will quite likely introduce new issues which can be discovered with verification techniques. We invite interested parties to contact us if there is interest for pursuing this path.

5 Conclusions

We have presented a work in progress on applying formal methods in the context of automated diagnosis and recovery to Autosub 6000 AUV. The work includes several research challenges that are relevant to the formal methods community. One of the challenges is automatic generation of the mission control component of the diagnosis model from mission scripts and configuration. Success of such generation can only be guaranteed by proving the resultant model to satisfy formalised safety (and liveness) policies using verification techniques. Another open challenge is proving the Livingstone 2 engine and the hybrid monitors that we add to the engine to be correct regarding a number of safety and liveness policies.

Acknowledgements. This work was funded by the UK Natural Environment Research Council as part of grant NE/F01256X/1, Automated Diagnosis for Fault Detection, Identification and Recovery in Autosub 6000.

References

[1] E. Denney and B. Fischer. Correctness of source-level safety policies. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME*, volume 2805 of *Lecture Notes in Computer Science*, pages 894–913. Springer, 2003.

[2] S. C. Hayden, A. J. Sweet, and S. Shulman. Lessons learned in the Livingstone 2 on Earth Observing One flight experiment. In *Proc. AIAA 1st Intelligent Systems Tech. Conf., Am. Inst. Aeronautics and Astronautics*, 2004.

[3] J. Kurien and M. D. R-Moreno. Costs and benefits of model-based diagnosis. *Aerospace Conference, 2008 IEEE*, pages 1–14, March 2008.

[4] A. E. Lindsey and C. Pecheur. Simulation-based verification of autonomous controllers via Livingstone PathFinder. In K. Jensen and A. Podelski, editors, *TACAS*, volume 2988 of *Lecture Notes in Computer Science*, pages 357–371. Springer, 2004.

[5] S. McPhail, M. Furlong, V. Huvenne, and P. Stevenson. Autosub6000: Results of its engineering trials and first science missions, 2008. presented at the 10th Unmanned Underwater Vehicle Showcase, The National Oceanography Centre, Southampton, UK.

[6] S. D. McPhail and M. Pebody. Navigation and control of an autonomous underwater vehicle using a distributed, networked control architecture. *The International Journal of the Society for Underwater Technology*, 23:19–30, 1998.

[7] M. Pebody. The contribution of scripted command sequences and low level control behaviours to autonomous underwater vehicle control systems and their impact on reliability and mission success. *OCEANS 2007 - Europe*, pages 1–5, June 2007.

[8] C. Pecheur and R. G. Simmons. From Livingstone to SMV. In *FAABS '00: Proceedings of the First International Workshop on Formal Approaches to Agent-Based Systems-Revised Papers*, pages 103–113, London, UK, 2001. Springer-Verlag.

[9] B. C. Williams and P. P. Nayak. A model-based approach to reactive self-configuring systems. In *Proceedings of AAAI-96*, pages 971–978, 1996.